T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

題目(和文)	バージョンを言語要素に持つプログラミング言語の研究
Title(English)	A Programming Language with Versions
著者(和文)	田辺裕大
Author(English)	Yudai Tanabe
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第12329号, 授与年月日:2023年3月26日, 学位の種別:課程博士, 審査員:増原 英彦,鹿島 亮,南出 靖彦,脇田 建,西崎 真也,五十嵐 淳
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第12329号, Conferred date:2023/3/26, Degree Type:Course doctor, Examiner:,,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Powered by T2R2 (Science Tokyo Research Repository)

TOKYO INSTITUTE OF TECHNOLOGY

DOCTORAL THESIS

A Programming Language with Versions

Author: Yudai TANABE

Supervisor: Dr. Hidehiko MASUHARA

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Science

in the

Programming Research Group
Department of Mathematical and Computing Science

March 1, 2023

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

School of Computing
Department of Mathematical and Computing Science

Doctor of Science

A Programming Language with Versions

by Yudai TANABE

While modern software development heavily relies on versioned packages, the concept of versions is rarely supported in the semantics of programming languages, resulting in bulky and unsafe software updates. The dissertation proposes a programming language that intrinsically supports versions. To establish a basis of finer-grained version control in language semantics, the author proposes a language called VL, with core calculus for supporting multiple versions, a compilation method to the core, and an inference algorithm determining the version of each expression. The author proved the type safety of the core calculus to guarantee consistent versions in a program. The author also implements VL, a minimal but adequate functional language that supports data structures and a module system, and conducts a case study involving the simultaneous use of multiple versions.

Acknowledgements

I would like to thank Professor Hidehiko Masuhara, who guided me for six years from undergraduate to master and doctoral programs. Under his guidance, I can pursue my research theme freely.

From 2018 to 2022, I worked with Tomoyuki Aotani and Luthfan Anshar Lubis on programming with versions. I want to thank them for their continuous advice and cooperation in my research.

Finally, financial support from the Japan Society for the Promotion of Science (JSPS) and the Tokyo Tech Tsubame Scholarship allowed me to devote myself to research. The contribution of this financial support to my research is immeasurable. Therefore, I am grateful for their financial support for my research proposal.

List of Publications

Chapter 2 to chapter 4, and chapter 7 and chapter 8 consists of:

- Yudai Tanabe, Tomoyuki Aotani, and Hidehiko Masuhara. A Context-Oriented Programming Approach to Dependency Hell. In *Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Runtime Composition (COP '18)*. Association for Computing Machinery, New York, NY, USA, 8–14. 2018.

- Yudai Tanabe, Luthfan Anshar Lubis, Tomoyuki Aotani, and Hidehiko Masuhara. – A Functional Programming Language with Versions. In *The Art, Science, and Engineering of Programming*, Vol. 6, No. 1, Article 5. 2022.

Chapter 2, chapter 3 and chapter 5 consists of:

- Yudai Tanabe, Luthfan Anshar Lubis, Tomoyuki Aotani, and Hidehiko Masuhara. A Step toward Programming with Versions in Real-World Functional Languages. In *Proceedings of the 14th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-time Composition (COP '22).* Association for Computing Machinery, New York, NY, USA, 44-51. 2022.

Contents

1 Introduction 1

2 Problem Statements 5
- 2.1 Component-based Development and Updates . . . 5
  - 2.1.1 Introduction . . . 5
  - 2.1.2 Motivating Example . . . 6
- 2.2 External Tool Support for Dependency Conflict . . . 8
- 2.3 Problems caused by Dangling Dependency . . . 10
  - 2.3.1 Situations Where Incompatibility Problems Occur . . . 10
  - 2.3.2 Declarative Dependency Requirements . . . 11
  - 2.3.3 Name Mangling when Dependency Conflicts . . . 12
  - 2.3.4 Problems Caused by Name Mangling . . . 12
- 2.4 Indirect Dependencies Complicate Incompatibility Problems . . . 13

3 Programming with Versions 15
- 3.1 Overview . . . 15
- 3.2 Advantages of Language-Based Versioning . . . 15
- 3.3 Language Features in Programming with Versions . . . 16
- 3.4 Intuition to Language-based Approach . . . 17

4 Lambda VL 19
- 4.1 Versions within a Language Semantics . . . 19
  - 4.1.1 Versioned values . . . 19
  - 4.1.2 Type System of λ_{VL} . . . 20
- 4.2 A Taste of λ_{VL} . . . 20
  - 4.2.1 Versioned Values . . . 20
  - 4.2.2 Versioned Function Application . . . 21
  - 4.2.3 Versioned-Independent Programs . . . 21
  - 4.2.4 Types of Versioned Values . . . 22
  - 4.2.5 Ensuring a Consistent Version of the Computation . . . 22
- 4.3 Syntax of λ_{VL} . . . 23
- 4.4 Version Resources . . . 23
- 4.5 The Declarative Type System of λ_{VL} . . . 24
- 4.6 Dynamic Semantics . . . 27
- 4.7 Metatheory . . . 29

5 Programming with Versions on Ordinary Functional Languages 31
- 5.1 Overview . . . 31
  - 5.1.1 Motivation . . . 32
  - 5.1.2 Intuition to the Compilation . . . 33
- 5.2 An Intermediate Language, VLMini . . . 33
  - 5.2.1 Syntax . . . 33
  - 5.2.2 Types and Kinds . . . 34

5.2.3 Constraints 34
5.3 Girard's Translation for VLMini 35
5.4 Bundling 36
5.4.1 Intuition to the Bundling 36
5.4.2 Constraint-based Bundling 38
5.5 Algorithmic Type Inference 38
5.5.1 Type Unification and Substitutions 40
5.5.2 Pattern Type Synthesis 41
5.5.3 Type Inference 42
5.6 Extend with Data Structures 44

6 Implementation and Case Study **47**
6.1 Implementation 47
6.1.1 Overview 47
6.1.2 Frontend 47
6.1.3 Ad-hoc Version Polymorphism via Duplication 50
6.1.4 Constraints Solving with z3 50
6.1.5 Code Generation 51
6.2 Case Study 52
6.2.1 Settings 52
`Matrix` Package Update 52
`Main` Program with Two Versions 54
6.2.2 `Main` Program in VL Programming 54
Detecting Inconsistent Version 54
Simultaneous Use of Multiple Versions 55
6.3 Limitations of the Current VL 56
6.3.1 Lack of Support for Structural Incompatibility 56
Types of Incompatibilities 56
Extending VL to Support Structural Incompatibility 57
6.3.2 Inadequate Version Polymorphism 58

7 Related Work **61**
7.1 Software Product Lines 61
7.1.1 Delta-Oriented Programming 61
7.1.2 Variational Programming 62
7.2 Adaptation Techniques 62
7.3 Container 62
7.4 Monorepository 63
7.5 Coeffect Calculus 63

8 Conclusion **65**
8.1 Towards a Per-expression Dependency Analysis 65
8.2 Conclusion 65

Bibliography **67**

A Definitions and Proofs **73**
A.1 Resource Properties 73
A.2 Context Properties 75
A.3 Substituions Properties 76
A.4 Type Safety 91

List of Figures

2.1 Minimal configuration before (top) and after (bottom) the update that causes dependency hell. 6
2.2 *App*, *Dir*, and *Hash* modules *before* update. 7
2.3 *Hash* module *after* update. 7
2.4 Minimal configuration before (top) and after (bottom) an update that causes dependency hell. 10
2.5 Minimal package configuration in cargo to accept multiple versions of Package *B* by name mangling. 12
2.6 Package configuration before (top) and after (bottom) an update cause dependency hell, with no direct dependencies on Package *B*. 14

4.1 λ_{VL}typing rules 25
4.2 λ_{VL}dynamic semantics 28

5.1 Translation Overview. 31
5.2 Simple example program written in VL (Haskell subset). 36
5.3 VLMini type unification 40
5.4 VLMini pattern type synthesis 41
5.5 VLMini algorithmic typing 42
5.6 VLMini context grading 43
5.7 VLMini constraints generation 44
5.8 Extension for algorithmic typing inference, pattern type synthesis, and type unification for data strutcures. 45

6.1 The pipeline of VL Language System 48
6.2 Module configuration with two mixed versions for the both module `A` and `B`. 49
6.3 The snipet of module `Matrix` version 1.0.0. 53
6.4 The snipet of module `Matrix` version 2.0.0. 54
6.5 `Main` module *before* rewriting. 55
6.6 `Main` module *after* rewriting. 55
6.7 The `main` function with all definitions dispatched (`join` and `sortVector` excerpted) 56
6.8 The well-typed (left) and ill-typed (right) programs illustrate the difference between the treatment of local and external variables. 58
6.9 The well-typed (top) and ill-typed (bottom) programs indicate that only one version of each module's top-level symbol is allowed. 59

List of Tables

6.1 Availability of functions in hmatrix before and afte tha update in version 0.16. 52
6.2 Structural change: availability of functions in GDK 3 56
6.3 Behavioral change: reliability of an alarm time in Android API 57

Chapter 1

Introduction

One of the problems currently confronting software developers is the cost of maintaining dependent packages up-to-date. While updates bring many improvements, they can also trigger problems in the client software. Despite the prevalence of these incompatible updates, the differences between the old and new versions are often implicitly described, making them difficult to debug. This problem has been exacerbated over the past few years owing to the escalating quantity of transitive package dependencies.

One of the main factors making updates difficult is that most software systems insist on making only a single version of a package available anywhere. This fact leads to a lack of support for the simultaneous use of multiple versions in most programming languages. Therefore, even when only a small portion of a program requires updating, developers are obliged to carry out additional work, which often delays the adoption of new versions.

We aim to develop a *programming language with versions* and motivates developers to update their software. Introducing expression-level version control as a fundamental aspect of programming language enables the concurrent utilization of multiple versions of a particular package. By allowing developers to use previous versions of packages along with newer ones, they can implement updates incrementally and avoid additional work on unaffected codes from the update. This results in a more streamlined process of updating software, as developers can focus their efforts on their tasks, without being impeded by the need to ensure compatibility across the entire codebase.

Thesis Statement *Establish the basis of programming language with versions, which allows simultaneous use of multiple versions*

This dissertation presents a programming language called VL, which is a functional language with versions that incorporates versions as an intrinsic language element. Unlike most existing programming languages that assume only one version of each package, VL allows an access to multiple versions of external expressions to refer to an individual version of external modules. This dissertation demonstrates the feasibility of safe programming with multiple versions in a general-purpose functional language. To this end, we investigate (a) a core calculus with a type system for programming with versions, (b) a compilation method between lambda-based functional languages and the core, and (c) an inference algorithm for determining the version of each expression.

Core Calculus λ_{VL} is an extension of the linear lambda calculus, which allows a single value to have multiple variations of versions. Such values are called versioned

values, and they store version-specific definitions in record-like entities, along with version labels indicating which version the definition exists. Function application of versioned values represents multiple possible computations of a function and an argument. The internal computation is extracted by specifying the label in the same way as in a normal record. The type safety of λ_{VL} programs is explained by version consistency, which ensures that every subterm has a consistent version definition, thereby preventing λ_{VL} programs from being evaluated with versions where no definition exists. Furthermore, the type system is designed as an instance of coeffect calculus, which is a substructural calculus capable of analyzing various computational resources.

Girard's Translation and Bundling We developed a compilation method that translates modules of lambda-calculus-based surface language, developed in multiple versions, to a λ_{VL} program that behaves in a version-crossing manner. The entire process involves two compilations: (1) Girard's translation from a single version of the surface language program to VLMini, a subset of λ_{VL}. We define the translation as a generalization of Girard's translation for linear lambda calculus. The compilation replaces lambda abstractions with cotextual-let and dually inserts a promotion for the argument of function application. The extended Girard's translation allows functions to capture version constraints of their argument value in the type system. (2) The second translation, called bundling, bundles the top-level declarations of each version of a VLMini program into a single versioned value. Bundling allows a top-level symbol with the same name across versions to behave as a versioned value representing multiple possibilities of computations.

Algorithmic Type Inference for λ_{VL} We develop algorithmic type inference rules based on the declarative type system following Hindley-Milner type inference. The constraints about versions are generated from either (1) bundling that generates a dependency on a particular version for a top-level symbol or (2) type inference that generates version consistency among variables. These constraints will be resolved after all modules' type inference/bundling is completed.

Implementation Finally, the author implements a VL language system with all the above concepts of programming with versions. The VL system is implemented by Haskell (GHC 9.2) and uses z3 as the constraint solver. Code generation specializes in a λ_{VL} program into a version-specialized Haskell program using the solution from z3-solver. We observe how the proposed language system guarantees safety in multiple versions through some examples. Furthermore, since the initial implementation strictly guarantees version consistency, we introduce a syntax extension that can communicate the user's intent to the type inference system and enable the simultaneous use of multiple versions.

Summary In this dissertation, we show that for traditional functional languages such as Haskell and the ML language family, multiple versions of a single package can be used without compromising type safety. Although this dissertation focuses on Haskell-like minimal languages, it will be possible to support the higher-level language constructs such as user-defined data types and version polymorphisms. The basic idea that values have versions also provides a theoretical basis for more granular dependency checking, such as link-time dependency checking and semantic versioning into language semantics.

Outline

- Chapter 2 provides a detailed account of incompatibility issues. It introduces version consistency through examples of small incompatibility issues and demonstrates how existing technologies have been used to ensure version consistency.
- Chapter 3 presents the advantages of language-based versioning in contrast to existing technologies. Then, we introduce fundamental concepts in programming with versions.
- Chapter 4 introduces λ_{VL}. We describe the syntax and dynamic and static semantics of λ_{VL}, as well as a type-safe type system to ensure version consistency at the expression level.
- Chapter 5 presents ideas for programming with versions on the Haskell-like functional language VL. First, we introduce Bundling and Girard's transformations, with version labels elaborated as identifiers to differentiate versions in expressions. Moreover, we present a version inference algorithm for VL.
- Chapter 6 features an implementation of the VL language and a case study that simulates an incompatible update made in a Haskell library. Additionally, we explore the potential for further development of the language by identifying issues with the current language implementation.

Finally, we conclude the dissertation by presenting a related work in chapter 7 and conclusion in chapter 8.

Chapter 2

Problem Statements

Contents

2.1 Component-based Development and Updates **5**
2.1.1 Introduction 5
2.1.2 Motivating Example 6
2.2 External Tool Support for Dependency Conflict **8**
2.3 Problems caused by Dangling Dependency **10**
2.3.1 Situations Where Incompatibility Problems Occur 10
2.3.2 Declarative Dependency Requirements 11
2.3.3 Name Mangling when Dependency Conflicts 12
2.3.4 Problems Caused by Name Mangling 12
2.4 Indirect Dependencies Complicate Incompatibility Problems . . **13**

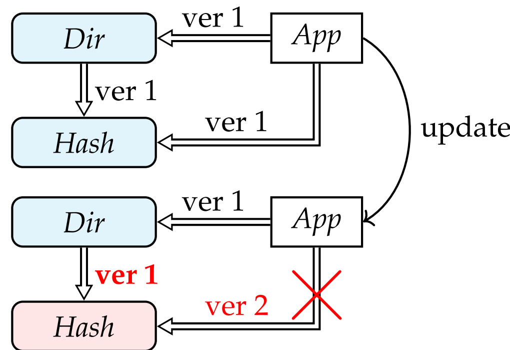
2.1 Component-based Development and Updates

2.1.1 Introduction

Current software systems are composed of various software components. These software components are usually called *packages*, a collection of several modules (which, unlike packages, are provided by programmatic language features). Since packages are responsible for the means of code distribution, continuously evolving packages' implementations are distinguished by their *versions*. A newer version usually improves the older ones by adding features, improving performance, or fixing bugs. Developers can notice the availability of newer implementations of externally developed packages by checking their version numbers and deciding whether they should replace the packages currently in use with new ones.

Updates are essential parts of modern software development with the growing popularity of component-based development. Upstream developers provide new implementations of software packages as a new version, and downstream developers incorporate improvements of packages (e.g., security fixes, added features, or performance improvements) through updates. In recent years, package management for many programming languages has moved to a model where packages are stored in package repositories, and specific versions are provided to client software as required. Using such tools has simplified the update process compared to previous ad-hoc manual efforts. [Die+19a]

One of the problems software developers face today is the cost of updating dependent packages. While new versions bring many improvements, they can also



After the update of *Dir* to version 2, the two direct/indirect dependencies to *Hash* cause a conflict.

FIGURE 2.1: Minimal configuration before (top) and after (bottom) the update that causes dependency hell.

cause problems for client software. A new version of a package can either be *compatible* or *incompatible* (or with "breaking changes") from older versions. When it is compatible, we can replace an older version with the new one without a problem. Otherwise, we would need to modify our program to use the newer version. [DJB14; RvV17] Even though such incompatible updates are common, [RDV14] the differences between old and new versions are often only implicitly described, making debugging difficult to debug. [Beu+99] This problem has been getting worse recently as packages have increasingly become transitive dependencies. Some recent studies have observed that dependency-related errors are among the most frequently encountered. [Seo+14]

In this chapter, we focus on incompatibilities and the existing methods to manage them. Each package continually evolves for projects that rely on frequently updated open-source packages, so compatibility issues are inevitable. [DJB14; RvV17; Art+12; Die+19b; Bav+15; Bog+16]

2.1.2 Motivating Example

We explain a small example to illustrate the essence of incompatibility problems. Here, we consider a scenario in which a breaking change is introduced to a dependent package during the development of an application *App*.

The top of Figure 2.1 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 2.2. 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 function `match` 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 a file with a file name equal to the given hash exists.

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 a file exists; otherwise, it results in an error `Not Found`. Assuming that there is a file with the required name in the directory, the executable of *App* with the dependency shown on top of Figure 1 will print `Found` on the standard output.

```
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 -> Bool
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 -> Bool
6  match str hash = (makeHash str) == hash
```

FIGURE 2.2: *App*, *Dir*, and *Hash* modules *before* update.

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

FIGURE 2.3: *Hash* module *after* update.

Now, the *App* developer updated *Hash* from version 1 to version 2 due to security concerns, as shown in Figure 2.3. Figure 2.3 shows the updated version of *Hash*. Version 2 of *Hash* uses SHA-3 as the new hashing algorithm.

After the update, the dependencies of *App* are changed, as shown at the bottom of Figure 2.1. 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 versions of *Hash*. The situation in *Dir* can occur for various 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 does not change the *App* code but causes problems with *App*. The build systems reject the program when multiple versions of the same package are required. Even if the programs were successfully compiled with a technique that allows 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*: *App* uses version 2 of `makeHash` in line 7 of *App*. On 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 the hashes generated by two completely different algorithms, SHA-3 and MD-5. As a result, on line 8 of *App*, `exists digest` evaluates to `false` against the expected behavior.

This example suggests considering the consistency of versions using 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 these results were compared.

Therefore, developers use buildtools 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 2.1.

2.2 External Tool Support for Dependency Conflict

As in the file explorer example in the previous section, incompatible versions can introduce bugs into the application due to incompatibilities between versions. Many systems, programming languages, and execution environments allow programs to use only *one-version-at-a-time* for each package to avoid the unintended use of dependent-package versions. The limitation of such systems stems from the fact that the maintenance cost can be huge when a program simultaneously requires a new and an older version. [Bav+15] These systems can be classified according to which units the constraints of one package version are imposed.

- *Per device*: Traditional package managers provided by Unix-like OS distributions, such as apt[1], rpm[2], pacman[3], or FreeBSD Ports[4] are classified in this category. Since these tools do not allow the use of different versions across multiple projects, each time we install a package, it can break the behavior

[1] `https://tracker.debian.org/pkg/apt` (January 11, 2023)
[2] `http://rpm.org/index.html` (January 11, 2023)
[3] `https://archlinux.org/pacman/` (January 11, 2023)
[4] `https://www.freebsd.org/ports/` (January 11, 2023)

of other software or cause dependency conflicts. These tools require ad hoc efforts to install multiple versions of the same package simultaneously. For example, rpm clarifies that it does not care about the installed package version and recommends that developers give each version a unique name to manage a particular version if they must care about its version. [5]

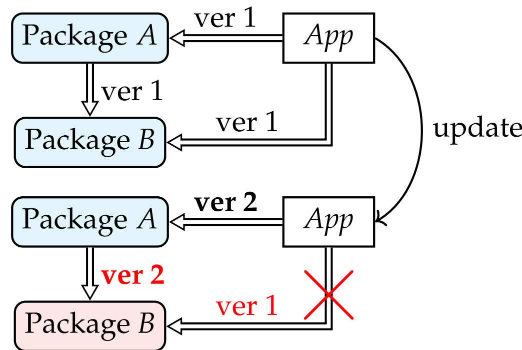
- *Per project*: Most package managers specialized in specific programming languages, such as cabal [6], stack [7], opam [8], PyPI [9], and maven [10] are classified in this category. [Aba+20] The main idea in these packages is to sandbox dependencies. The idea of sandboxing is adopted by cabal and stack and solves the problem of dependency conflicts between projects. Sandboxing ensures that every project has an isolated location for its dependencies. In other words, installing a package in one project does not affect other sandboxes. For example, even though package A depends on version 1.0.0 of package C and package B depends on 2.0.0 of package C, package A and B can be installed simultaneously. However, using multiple versions of a package in a single project build is impossible. For example, when maven detects a dependency conflict, it will only adopt the version closest to that package on the dependency graph. [11] As a different approach for unifying versions, stack provides a curated set of packages called resolver, which guarantees no dependency conflicts between the versions of a package registered in the resolver. If developers stick to the packages within a resolver, they can avoid the tedious task of finding a compatible version of a package.

However, it is difficult to permanently fix versions of every package in a device or project, so many techniques have been developed recently to mitigate one-version-at-a-time limitation for more flexibility. We can classify those techniques in the unit of software components that allow different versions.

- *Device*: For packages that are only allowed to exist in one version on a device (e.g., operating system standard libraries like the C standard library (libc)), OS-level virtualization software enables one to use different versions in the virtualized environment. For example, traditional UNIX-like operating systems only allow one version of the OS standard library; OS-level virtualization software, such as Docker [Mer14] and QEMU [Bel05], can provide environments with different versions.

- *Process*: For a package (or a library) linked to a program, a dynamic loading mechanism can provide a different version for a different process of the program. For example, with the shared library mechanism in UNIX-like operating systems, a compiled program can run with a different library version by providing a different load path. More advanced mechanisms, e.g., OSGi [All18], automatically load the appropriate versions.

[5] https://rpm.org/user_doc/multiple_versions.html (January 11, 2023)
[6] https://cabal.readthedocs.io/en/stable/ (January 11, 2023)
[7] https://docs.haskellstack.org/en/stable/ (January 11, 2023)
[8] https://opam.ocaml.org/ (January 11, 2023)
[9] https://pypi.org/ (January 11, 2023)
[10] https://maven.apache.org/ (January 11, 2023)
[11] https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Transitive_Dependencies (January 11, 2023)



After an update of Package *A* to version 2, the two direct/indirect dependencies to Package *B* cause a dependency conflict.

FIGURE 2.4: Minimal configuration before (top) and after (bottom) an update that causes dependency hell.

- *Package*: Some modern and sophisticated package managers allow loading two versions of the required package. Such features can be found in npm [12] for JavaScript, cargo [13] for Rust, and Maven with the shade plugin [14] for Java. For example, npm handles multiple versions of the same package through nested dependencies [15]. npm manages all packages with version numbers and limits the scope of a package to its direct dependencies. When a developer needs different package versions, npm places the later required versions in the dependency graph as nested dependencies. This approach allows multiple versions to coexist in the same project, but using two or more objects simultaneously derived from different versions can lead to confusing runtime errors because the javascript runtime does not recognize the difference between objects of two versions. The other two tools utilize a similar technique to that of npm with *name mangling* that will be described in section 2.3.3.

2.3 Problems caused by Dangling Dependency

While the npm and cargo method may seem like solutions to compatibility problems, they are not silver bullets. The cause of the corner cases of both approaches is a *dangling dependency*. A dangling dependency is an indirect dependency that includes a transitive dependency in its interface. When development software depends on a package with a dangling dependency, the package's interface in the transitive dependency is exposed to the development software as a direct dependency, resulting in a direct dependency on multiple versions simultaneously. All of the techniques in section 2.2 will fail if multiple versions are required as direct source dependencies. This section details the reason by introducing a problem in name mangling in cargo.

2.3.1 Situations Where Incompatibility Problems Occur

Figure 2.4 shows a situation where the developer updates the software called *App*. The upper and lower halves show the configurations before and after the update.

[12] `https://www.npmjs.com/` (February 1, 2021)
[13] `https://doc.rust-lang.org/cargo/` (February 1, 2021)
[14] `https://maven.apache.org/plugins/maven-shade-plugin/` (February 1, 2021)
[15] `https://npm.github.io/how-npm-works-docs/npm3/how-npm3-works.html` (January 11, 2023)

On the upper half, *App* depends on Package *A* version 1 and Package *B* version 1, and also, the package *A* version 1 depends on the package *B* version 1. In this configuration, there is no incompatibility problem.

Suppose that the developer of *App* decided to switch the version of Package *A* from 1 to 2. The two versions 1 and 2 of Package *A* are compatible, but the dependency on Package *B* has been changed from version 1 to 2 that are incompatible. Since types exposed in Package *B* are often re-exported by Package *A*, the update of Package *A* can introduce unintended direct dependencies on Package *B* into *App*. Since *App* itself requires Package *B* version 1, it is impossible to use Package *A* version 2 without modifying *App* to support Package *B* version 2.

2.3.2 Declarative Dependency Requirements

Many package managers, including npm and cargo, employ semantic versioning or similar conventions so that package users can automatically update dependent packages while maintaining compatibility. For this purpose, all package developers need to define the version requirements of their dependent package properly.

The semantic versioning recommendation [Pre13] specifies that version numbers are a sequence that consists of major, minor, and patch versions separated by dots as in `MAJOR.MINOR.PATCH`. The semantic versioning strategy can explain to users of a package what types of change will occur in the new release, and users can use version numbers as a guide to decide whether to accept the new release. For example, when incrementing the minor version of a package (e.g., `1.2.1` ⇒ `1.3.0`), all changes should be backward compatible with previous versions. On the contrary, incompatible code changes are only permitted when the `MAJOR` level is incremented (e.g., `1.2.1` ⇒ `2.0.0`).

Dependencies are usually described in terms of the range of dependent package versions that the client software allows. For example, Figure 2.4 shows a situation where the developer updates software called *App*. On the upper half, *App* depends on Package *A* version 1.0.0 and Package *B* version 1.0.0, and also the package *A* version 1 depends on the package *B* version 1.0.0.

The cargo user can specify the dependencies of Figure 2.4 in manifest files of package *App* and Package *A* as follows:

```
1 [package]
2 name = "App"
3 [dependencies]
4 A = "1.0.0"
5 B = "1.0.0"
```

```
1 [package]
2 name = "A"
3 version = "1.0.0"
4 [dependencies]
5 B = "1.0.0"
```

In cargo, `1.0.0` means `>=1.0.0 && <2.0.0`. This requirement shows the range of backward compatible versions with `1.0.0` based on SemVer. The dependency resolver in Cargo collects these two manifest files and automatically retrieves the latest version that meets all requirements for each package. Assuming that all packages only have `1.0.0` and `2.0.0` for simplicity, the requirements above are satisfied by getting `1.0.0` for all packages.

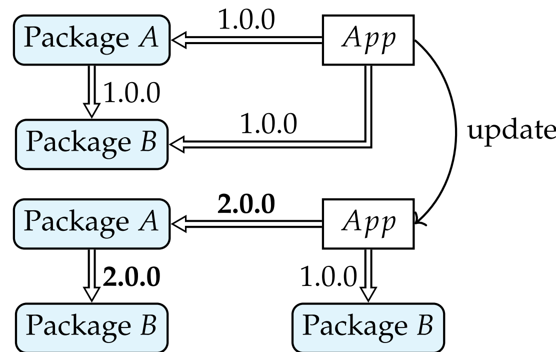


FIGURE 2.5: Minimal package configuration in cargo to accept multiple versions of Package *B* by name mangling.

2.3.3 Name Mangling when Dependency Conflicts

The Rust compiler's name mangling scheme is defined in RFC 2603, [16] it assigns a unique mangled name to each identifier for every combination of crate, module, type, and crate version, without collisions. Name mangling is a mechanism to alleviate the one-version-at-a-time limitation by mangling (or shading) their package names. When an *App* developer decides to switch Package *A* from version 1 to version 2 that requires Package *B* version 2, then the manifest file is modified as on the left below. Due to the modification, cargo refers to the manifest file of Package *A* version 2 as on the right below.

```
1 [package]
2 name = "App"
3 [dependencies]
4 A = "2.0.0" // modified
5 B = "1.0.0"
```

```
1 [package]
2 name = "A"
3 version = "2.0.0" // updated
4 [dependencies]
5 B = "2.0.0" // updated
```

In this example, both *App* and Package *A* have a dependency on Package *B*, but *App* and Package *A* require versions `>=1.0.0 && <2.0.0` and `>=2.0.0 && <3.0.0` of Package *B*, respectively. Since there is no version of Package *B* that satisfies both constraints simultaneously, the traditional dependency analyzer reports this situation as a dependency error.

Cargo, on the other hand, allows the two versions to coexist by creating two copies of each version of Package *B* for *App* and Package *A*, as shown in the lower half of Figure 2.5. In this way, even if the same function name exists in different versions of a package, it is possible to determine the correct version of the function needed for each package. This solution merely replicates Package *B* into two completely different packages. However, it allows different versions of the same package to coexist in a dependency graph, mitigating the limitations of the one-version-at-a-time policy in a sense, but cause some limitations described in the next section.

2.3.4 Problems Caused by Name Mangling

Name mangling is one reasonable mitigation measure, but it leads to type-level incompatibilities [Tol17; Coa19]. This approach collapses when values derived from different package versions are inevitably mixed in the same code, due to a dangling dependency. For example, consider a situation where Package *B* is a framework that

[16] https://rust-lang.github.io/rfcs/2603-rust-symbol-name-mangling-v0.html

provides type `Key` representing a cryptographic hash key, and Package *A* is a library that includes `Key` in its interface. Suppose that the definitions of `Key` are the same in v1.0.0 and v2.0.0, and some breaking changes in other parts lead to the increment of the major version as follows.

```
1 // package B v1.0.0
2 pub type Key = /* complicated */
```

```
1 // package B v2.0.0
2 pub type Key = /* Same as v1.0.0 */
```

Here, we simplify the code syntax, including function definitions, extern, and semicolons, as we want to avoid addressing Rust-specific issues. Using the `Key` defined in Package *B*, the APIs in Package *A* and *App* are written as follows.

```
1 // package A v2.0.0
2 fn gen_key() -> Key // from v2.0.0
```

```
1 // App
2 let x: (A::Key) = gen_key() // error
```

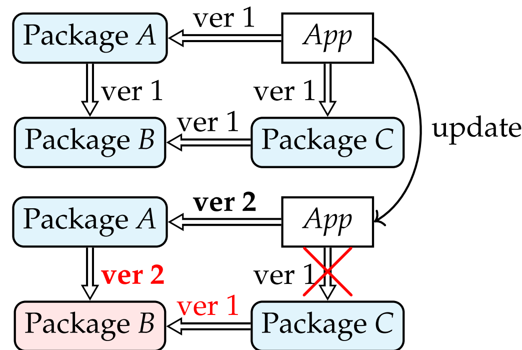
Package *A* provides the `gen_key` function for generating hash keys. Note that the return type of `gen_key` is `Key` defined in Package *B* v2.0.0.

In this example, the *App* program above will be rejected by the Rust type system. The expected version for the type `Key` is v1.0.0, but Package *A* v2.0.0 relies on Package *B* v2.0.0, so the `gen_key` function only returns the `Key` object in v2.0.0. The Rust compiler gives v1.0.0 and v2.0.0 of type `Key` completely different identifiers so that `A::Key` in the type declaration of `x` and `Key` in the return type of `gen_key` are completely different ones. Therefore, the programmer is eventually notified that "`Key` is not equal to `Key`" (which may look strange, but error reporting does not use internally mangled names) even if the definitions provided by both versions of Package *B* do not change at all.

2.4 Indirect Dependencies Complicate Incompatibility Problems

Despite the advancement of techniques to isolate and analyze dependencies in section 2.2, many developers are still plagued with compatibility problems. Incompatibility problems are more likely to occur when there are transitive dependencies between subcomponents. [Art+12; Bog+16] For more complex software with many dependencies, a huge amount of work is required to update a single package. Since it is common to develop with multiple packages with the growing popularity of development with centralized package registries, programmers often encounter this problem. [DMG19]

Even worse, if a program indirectly depends on two incompatible versions of a package, the only way to solve the problem is to wait for the developers of the intermediate package to catch up the downstream updates. As shown in Figure 2.6, suppose that *App* used Package *C*, which in turn depends on Package *B* version 1. In the situation, resolving conflicts between Packages *A* and *C* requires the *App* developer to contact the maintainer of Package *C*, and multiple developers to work together to coordinate their dependencies. Otherwise the developer abandons Package *C* and rewrite the same functionality from scratch by yourself. If Package *C* were



After an update of Package *A* to version 2, the two indirect dependencies from *App* to Package *B* cause a dependency conflict.

FIGURE 2.6: Package configuration before (top) and after (bottom) an update cause dependency hell, with no direct dependencies on Package *B*.

outdated or closed source, the *App* developer would have no way of resolving the conflict on their own.

The nature of the problem is that all inter-operators must use compatible versions of the package. This problem, called "*version-lock*" [Pre13] i.e., the inability to upgrade a package without having to release new versions of every dependent package, is a threat to software reuse. Indeed, many developers are unwilling to update dependencies unless there is a significant update [Bav+15]. This fact has a tremendous impact on the ecosystem when Package *B* is widely used, such as a wrapper of libc and openssl [Tol17].

Chapter 3

Programming with Versions

Contents

3.1 Overview 15
3.2 Advantages of Language-Based Versioning 15
3.3 Language Features in Programming with Versions 16
3.4 Intuition to Language-based Approach 17

3.1 Overview

We investigate programming languages that support multiple versions inside a module by following this technology trend that supports multiple versions with a finer computation unit. In contrast to previous technical efforts that focused on avoiding the simultaneous use of multiple versions as direct dependencies, we focus on programming to simultaneously use multiple versions of a package. As there are few attempts to develop such languages, it needs to be clear what language abstractions are suitable to represent multiple versions and what kind of safety we can guarantee.

We establish a foundation for more freely combining and controlling different versions through a language-based approach. We develop a programming language called VL, with a notion of versions. A VL program can depend on multiple versions of a package, and static checks ensure that each value is dispatched to the appropriate implementation. We develop λ_{VL} as a core calculus for such surface languages. λ_{VL} has the terms for combining multiple versions of definitions and the type system for identifying version-safe programs in multiple versions.

This chapter first describes the advantages of a language-based approach in section 3.2 and then illustrates core features through programming with λ_{VL} in section 4.2.

3.2 Advantages of Language-Based Versioning

The language-based versioning here is to perform dependency analysis at the term level as an alternative to dependency analysis on package managers of traditional languages. Language-based versioning assumes a core language that supports the version as an intrinsic language element. The difference between language-based and package-based versioning is that an individual version can be assigned to every expression. In language-based versioning, all expressions are implicitly expected to have multiple versions of their definitions and are evaluated statically or dynamically in the appropriate version. Since package-based versioning requires a single

version of a dependent package to be defined outside of the language semantics, it is impossible to use multiple versions of a program in a naive way. This language feature of language-based versioning allows the following features.

- *Expression-level version selection.*
  The language-based approach allows different implementations to be referenced for the same symbol in a program. This feature allows programmers to write programs with a mixture of values from multiple versions, such as the program in section 2.3.4. This feature allows programmers to incremental update dependent packages. With package-based versioning, the dependent versions must be consistent throughout the entire program, which entailed heavy implementation costs and discouraged programmers from updating a dependent package. With language-based versioning, programmers do not have to make the extra effort to unify dependent package versions, even if they are in the middle of an update. It also allows partially updated development programs to be verified with existing unit tests.

- *Exhaustive static analysis on multiple version combinations.*
  In the language-based approach, every value implicitly has multiple versions of the definition, so the language system considers multiple combinations of versions. With package-based versioning, on the other hand, a static analysis of the program is performed only on specific version combinations. For example, a program for multiple package versions of C uses a preprocessor macro (`#ifdef`) to obtain the package version number from the system and dispatch the appropriate implementation before compilation. Since the version of the dependent package is determined outside of the language semantics, only programs with fixed versions in advance are subject to type checking.

3.3 Language Features in Programming with Versions

The research question is, *"What are the essential features in a language with programming with versions?"* For the sake of discussion here, we assume a module-based development, which has become the standard today. Here we do not consider a module system with a high-level abstraction that ML-family languages have. Still, a typical module system that handles namespace and module dependencies, such as that Haskell has. We will also identify packages with modules. In other words, all packages consist of a single module, and versions are held in the units of modules instead of packages.

In order to allow the simultaneous use of multiple versions, the VL should have the following features.

- *Module interfaces, written in a version-crossing manner.*
  Unlike existing language interfaces, symbols from external modules in VL are expected to behave as programs with multiple versions of their implementations. Following this principle, a module interface of VL should provide the interface for multiple module versions.

- *Semantic analysis to analyze the availability of a program.*
  The VL program can have multiple interpretations, depending on which version is specified for each subterm. Some of these versions may differ from the programmer's assumptions, and some may not have subterms with definitions. Semantic analysis is needed to record and analyze the availability of

each program so that we preclude incorrect versions from enormous combinations of multiple versions.

3.4 Intuition to Language-based Approach

To elaborate on this idea, we will rewrite the codes in section 2.3.4 in VL. The current VL does not have user-defined data types and records, but we use these here to illustrate the basic idea of language-based approach. We give a definition of the type `Key` in both versions of Module *B* as follows.

```
1 -- Module B v1.0.0
2 data Key = Key { ... }
```

```
1 -- Module B v2.0.0
2 data Key = Key { ... }
```

Given the above two versions of Module *B*, we write its interface of VL in a version-crossing manner, where individual symbols are given version information. For example, the interface of Module *B* is the following.

```
1 -- Interface of Module B
2 data Key @ [(B, 1.0.0)], [(B, 2.0.0)]
```

The module interface indicates available versions and the type of each symbol. In the interface of Module *B*, we denote the module name and version number pair after `@` that records which version of the type is provided. Here, the type `Key` is annotated by `[(B, 1.0.0)]` and `[(B, 2.0.0)]`. These annotations enable the user of Package *B* to access both versions of definitions with the type *Key*. Which version of the computation is evaluated will be determined later by the semantic analysis as the available versions.

Similarly to the codes in section 2.3.4, we can write `gen_key` in VL as follows.

```
1 -- Module A version 1.0.0
2 import B
3 -- (gen_key does not exist)
4 ...
```

```
1 -- Module A version 2.0.0
2 import B
3 gen_key :: Key
4 gen_key = ...
```

```
1 -- Interface of Module A
2 import B
3 gen_key :: Key @ [(A, 2.0.0)]
```

Since `gen_key` is implemented only in version 2 of Module *A*, the module interface indicates that it is available only in `(A, 2.0.0)`.

```
1 -- Module App
2 import A
3 import B
4 main :: Key = gen_key
```

Consequently, the program in Module *App* will be interpreted in VL as follows. The type system calculates the version shared by all the values in the data flow by using type checking and rejects them if they do not exist. In this example, the *Key* provided in Package *B* has definitions for both versions, and the `gen_key` provided in Package *A* has a definition for the *Key* in version 2.0.0 of Package *A*. As a result, the *App* code is available in the combination of version 2.0.0 of Package *A* and both versions of Package *B*.

The VL language system works as described above by default. However, programmers often want to write programs that depend on a specific version of a dependent package as in existing languages. If we want to describe a user-defined version dependency in VL program as in section 2.3.4, rewrite version 2.0.0 of Module *A* as follows.

```
1 -- Module A version 2.0.0
2 import B version 2.0.0 as Bv2
3 gen_key :: Bv2.Key
4 gen_key = ...
```

```
1 -- Interface of Module A
2 import B
3 gen_key :: Key @ [ (A, 2.0.0), (B, 2.0.0) ]
```

The **`import`** statement of version 2.0.0 of Module *B* as `Bv2` expresses a dependency on the specific version of Module *B*, and the interface incorporates this implementation by adding `(B, 2.0.0)` to the annotation.

In the same way, the *App* program is changed to specify version 1.0.0 of Module *B*.

```
1 -- Module App
2 import A
3 import B version 1 as Bv1
4 main : Bv1.Key = gen_key // type error
```

In this case, the VL type system outputs a type error to the `main` function because the type of `gen_key` is `Key` of version 2.0.0 of Module *B*, even though the `main` function expects a that of version 1.0.0 of Module *B*.

This approach is different from the name mangling. While name mangling treats the two versions as completely different packages, the language-based approach assumes a kind of identity for the two versions, and the difference between the two versions is captured by the constraints. The type system keeps track of which version all variables depend on, so it is possible to report to the programmer what is causing the version mismatch.

Chapter 4

Lambda VL

Contents

4.1 Versions within a Language Semantics **19**
4.1.1 Versioned values 19
4.1.2 Type System of λ_{VL} 20
4.2 A Taste of λ_{VL} **20**
4.2.1 Versioned Values 20
4.2.2 Versioned Function Application 21
4.2.3 Versioned-Independent Programs 21
4.2.4 Types of Versioned Values 22
4.2.5 Ensuring a Consistent Version of the Computation 22
4.3 Syntax of λ_{VL} **23**
4.4 Version Resources **23**
4.5 The Declarative Type System of λ_{VL} **24**
4.6 Dynamic Semantics **27**
4.7 Metatheory **29**

4.1 Versions within a Language Semantics

A language-based approach requires a version-crossing interface and a mechanism to know in which version each symbol is available. Therefore, the core language must have (1) a notation that expresses a value that consists of sub-values of multiple versions and (2) a mechanism for analyzing which versions of a program may be available in more than one version.

We developed a core calculus λ_{VL} to achieve them. It realizes (1) and (2) by:

1. *versioned values*: records of multiple values distinguished by their version, and
2. *type systems*: statically checks the version to which all subterms in the program agree.

4.1.1 Versioned values

Versioned values represent multiple versions of computation and bundle them as a single value. Versioned values allow us to bundle multiple versions of values. For example, the versioned value $\{v_1 = \lambda x.x, v_2 = \lambda x.x + 1\}$ represents a versioned function whose initial implementation (v_1) is an identity function, and its next implementation (v_2) is a successor function.

A function application of a versioned function to a versioned value is also a versioned value. This versioned value consists of version-specific terms obtained by applying version-specific functions to the corresponding values in the versioned value in a version-wise manner. For example, we obtain $\{v_1 = 1, v_2 = 3\}$ if we apply $\{v_1 = \lambda x.x, v_2 = \lambda x.x + 1\}$ to $\{v_1 = 1, v_2 = 2\}$. If a function and its arguments have a different set of versions, the application is calculated on the common part of each version set. For example, we obtain $\{v_1 = 1\}$ if we apply $\{v_1 = \lambda x.x, v_2 = \lambda x.x + 1\}$ to $\{v_1 = 1\}$.

4.1.2 Type System of λ_{VL}

We develop a calculus called λ_{VL} based on the *coeffect calculus* [Bru+14] to guarantee type safety. A coeffect calculus is a type system derived from linear type systems [Gir87; Wad90] and a type system scheme for analyzing the usage of various computational resources, not just the number of times a variable is used. Just as other coeffect calculi track their computational resources, λ_{VL} attaches version numbers to types, such as $x : \Box_{\{v1,v2\}}T$, meaning x is a variable of type T and computable under versions v_1 and v_2. The λ_{VL} type system collects the annotated type information for the program and calculates the set of versions needed to run the program. The type system ensures at least one consistent version where the program can be evaluated.

In this chapter, we will demonstrate how these features are achieved by introducing the core features of λ_{VL}.

4.2 A Taste of λ_{VL}

4.2.1 Versioned Values

λ_{VL} is an extension of the coeffect calculus with versioned values that have multiple components tagged with versions. One way to construct versioned values is through *versioned records* $\{\overline{l_i = t_i}\}$[1,2]. We denote *labels* (l_i) to distinguish the different versions of values, and the values inside the versioned record are called *version-specific terms*. Versioned records provide a mechanism to write programs independent of a specific version. For example, we can write as follows to denote a default key length parameter 1024 and 4096 in versions 1 and 2, respectively.

$$\{l_1 = 1024, l_2 = 4096\}.$$

Another way to construct a versioned value is through *suspensions* $[t]$. The suspension $[t]$ promotes the term t to a versioned value such as $[1]$ and $[\lambda x.x]$. The two constructors for versioned values delay the inside computation until a specific version is later determined.

To conduct a suspended computation, programmers can use *extractions* $t.l$. The extraction $t.l$ extracts the version-specific term according to the label l from the versioned value returned by t. For example, consider the case where version 1 generates

[1] We will sometimes abbreviate a sequence as $\overline{*}$, i.e. $\overline{l_i}$ denotes $l_1, ..., l_n$ and $\overline{l_i = t_i}$ denotes $l_1 = t_1, ..., l_n = t_n$.

[2] Although our system, which we will describe in detail in section 4, explicitly states the default label such as $\{\overline{l = t} \mid l_k\}$, we omit it here for simplicity.

a key with a bit length of 1024, but version 2 generates one of length 4096. To generate a key with the appropriate bit length for each version and then retrieve the version-specific term in version 2, we can write as follows.

$$\{l_1 = \text{gen_key } 1024, l_2 = \text{gen_key } 4096\}.l_2.$$

4.2.2 Versioned Function Application

Functions with different version-specific values across different versions are also represented as versioned values, called *versioned functions*. For example, $v3.20$ and $v3.22$ of GDK 3 provide different-named functions with the same functionality. GDK versions before 3.22 provide `gdk_screen_get_n_monitors` that tells the number of connected physical monitors. However, versions 3.22 later provide the same functionality function `gdk_display_get_n_monitors` and deprecate `gdk_screen_get_n_monitors`.

We can write as follow to define a new function that can retrieve the number of connected monitors in both versions.

$$\begin{aligned}\{l_1 &= \text{gdk_screen_get_n_monitors},\\ l_2 &= \text{gdk_display_get_n_monitors}\}.\end{aligned}$$

Hereafter, we call this versioned function get_n_monitors.

We need to pass a versioned value for the argument to apply a versioned function. Here, we can use *contextual let-binding* $\textbf{let } [x] = t_1 \textbf{ in } t_2$ to apply a versioned function. For example, we apply the versioned function get_n_monitors to the versioned value $\{l_1 = ()\}$ as follows.

$$\textbf{let } [f] = \text{get_n_monitors} \textbf{ in let } [x] = \{l_1 = ()\} \textbf{ in } [f\ x] \tag{4.1}$$

This program first extracts the function `gdk_screen_get_n_monitors` from get_n_monitors and binds it to f; then it extracts the value $()$ from $\{l_1 = ()\}$ and binds it to x.

4.2.3 Versioned-Independent Programs

In the previous example, $\{l_1 = ()\}$ was bound to x and had only one definition with l_1. However, if both the versioned function and versioned value have multiple definitions, the language should evaluate the functional application in multi-version contexts. We achieve this by using the suspension $[t]$. For example, we apply get_n_monitors to $\{l_1 = (), l_2 = ()\}$, both of which have two definitions with l_1 and l_2 as follows.

$$\textbf{let } [f] = \text{get_n_monitors} \textbf{ in let } [x] = \{l_1 = (), l_2 = ()\} \textbf{ in } [f\ x]$$

This program returns a suspended computation that can return an integer value available in l_1 and l_2. Since the two version-specific terms in $\{l_1 = (), l_2 = ()\}$ are the same, and we can rewrite the above program as follows.

$$\textbf{let } [f] = \text{get_n_monitors} \textbf{ in let } [x] = [()] \textbf{ in } [f\ x].$$

4.2.4 Types of Versioned Values

The type of a versioned value is denoted as $\Box_r T$. The index r, called the *version resources*, indicates which version-specific terms are available in the versioned value. This notion of type comes from coeffect calculus, and the exact method of calculating r is based on the version resource semiring described later in section 4.

For example, assuming that both version-specific functions in get_n_monitors have type $\mathsf{Unit} \to \mathsf{Int}$, the above example programs are typed as follows.

$$\{l_1 = (), l_2 = ()\} : \Box_{\{l_1,l_2\}}\mathsf{Unit}$$
$$\mathrm{get_n_monitors} : \Box_{\{l_1,l_2\}}(\mathsf{Unit} \to \mathsf{Int})$$

The type $\Box_{\{l_1,l_2\}}\mathsf{Unit}$ denotes that this versioned value has version-specific terms of type Unit and they are available in both versions l_1 and l_2.

The contextual let-binding $\mathbf{let}\ [x] = t_1\ \mathbf{in}\ t_2$ propagates the version requirements through the captured variable x. For example, the Eq. 4.1 program is typed as follows.

$$\mathbf{let}\ [f] = \mathrm{get_n_monitors}\ \mathbf{in}\ \mathbf{let}\ [x] = \{l_1 = ()\}\ \mathbf{in}\ [f\ x] : \Box_{\{l_1\}}\mathsf{Int} \tag{4.2}$$

where the result type has resource l_1 because get_n_monitors and $\{l_1 = ()\}$ only have l_1 as their shared labels.

Note that the extraction $t.l$ makes the type of t lose the version resource. Once extracted, a version-specific term can be used with terms from other versions. For example, the type of the following program no longer has a version resource.

$$\mathbf{let}\ [f] = \mathrm{get_n_monitors}\ \mathbf{in}\ \mathbf{let}\ [x] = \{l_1 = ()\}\ \mathbf{in}\ [f\ x].l_1 : \mathsf{Int} \tag{4.3}$$

4.2.5 Ensuring a Consistent Version of the Computation

The λ_{VL} type system ensures that all necessary implementation versions exist. In other words, if a program extracts a specific version of a value even though all the version values in the program do not have a shared label, the type system will reject such a program. The first example is a variant of Eq. 4.2.

$$\mathbf{let}\ [f] = \mathrm{get_n_monitors}\ \mathbf{in}\ \mathbf{let}\ [x] = \{l_3 = ()\}\ \mathbf{in}\ [f\ x] : \Box_{\emptyset}\mathsf{Int}$$

The type system keeps track of the available versions of each variable by a set of labels in the context. In this example, it records $\{l_1, l_2\}$ for f, and $\{l_3\}$ for x. For each promotion, the type system calculates the shared version resource in the context that should be multiplied by the term. In the program type above, $[f\ x]$ is given version resource $\emptyset = \{l_1, l_2\} \cap \{l_3\}$, which indicates that there is no shared version available. It means no longer possible to extract any version-specific computation from this program, and the type system will reject such extractions. The type system can report the reason for the ill-versioned extraction by using the version information recorded in the context.

$\mathbf{let}\ [f] = \mathrm{get_n_monitors}\ \mathbf{in}\ \mathbf{let}\ [x] = \{l_3 = ()\}\ \mathbf{in}\ [f\ x].l_3 : (\textit{rejected})$

– ERROR: f and x are expected to be available in l_3, but f is not available in l_3.

The second example is a variation of Eq. 4.3. The following program is rejected because $\{l_1 = ()\}$ has no definition of l_2.

$\textbf{let}\ [f] = \text{get_n_monitors}\ \textbf{in}\ \textbf{let}\ [x] = \{l_1 = ()\}\ \textbf{in}\ [f\ x].l_2 : (\textit{rejected})$
– ERROR: f and x are expected to be available in l_2, but x is not available in l_2.

4.3 Syntax of λ_{VL}

λ_{VL} is an extension of the coeffect calculus $\ell\mathcal{R}$PCF [Bru+14] and GrMini [OLE19]. We defined the *version resource algebra* and extended the coeffect calculus by adding *versioned terms*. The terms and types of λ_{VL} are as follows:

Definition 4.3.1 (Lambda VL Syntax)**.**

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

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

Most of the terms in λ_{VL} derive from linear λ-calculus. Additional terms introduce and eliminate versioned values. Versioned values can be declared through promotions $[t]$ and versioned records $\{\overline{l = t} \mid l_i\}$. The type of versioned values $\Box_r A$ are indexed by a *version resource* r, where r ranges over the elements of the version resource semiring $\mathcal{R}$ described in section 4.4. The *versioned records* $\{\overline{l = t} \mid l_i\}$ has a *default label* along with a pair of labels and version-specific definitions. In the current design, programmers can note a default label $l_i \in \{\overline{l}\}$ for the case of multiple versions of calculation results. The default label is overridden in the dynamic semantics described in section 4.6. The term $[t]$ is a promotion of version necessity and allows t to be used to track the use of version resources in a program. The term $\textbf{let}\ [x] = t_1\ \textbf{in}\ t_2$ provides an elimination for version necessity and provides version-aware let-binding. Finally, the versioned computations $\langle \overline{l = t} \mid l_i\rangle$ represent intermediate terms whose evaluation in the default version is postponed. We assume that versioned computations appear only in intermediate terms during evaluation and not in the user's code.

4.4 Version Resources

The λ_{VL} type system is parameterized by the version resource semiring $\mathcal{R}$. It captures how a program depends on its context by tracking version information on the variables used in the program. Version resources $r \in \mathcal{R}$ appear in types with $\Box$-constructors and contexts with $[*]_r$-notions to denote the sets of versions on which the programs implicitly depend.

The version resources r are given by the following.

Definition 4.4.1 (Version Resouces)**.**

$$r ::= \bot \mid \emptyset \mid \{l_i\} \mid r_1 \cup r_2 \qquad \text{(version resources)}$$

Intuitively, an element of $\mathcal{R}$ is a set of labels such as $\{l_1\}$ and $\{l_1, l_2\}$. The language produced by this grammar is equivalent to the elements of the version resource semiring $\mathcal{R}$.

Definition 4.4.2. [Version resource semiring] The version resource semiring is given by the structural semiring (semiring with preorder) $(\mathcal{R}, \oplus, 0, \otimes, 1, \sqsubseteq)$, defined as follows.

$$0 = \bot \quad 1 = \emptyset \quad \bot \sqsubseteq \mathtt{r} \quad \frac{\mathtt{r}_1 \subseteq \mathtt{r}_2}{\mathtt{r}_1 \sqsubseteq \mathtt{r}_2}$$

$$r_1 \oplus r_2 = \begin{cases} r_1 & r_2 = \bot \\ r_2 & r_1 = \bot \\ r_1 \cup r_2 & \text{otherwise} \end{cases} \qquad r_1 \otimes r_2 = \begin{cases} \bot & r_1 = \bot \\ \bot & r_2 = \bot \\ r_1 \cup r_2 & \text{otherwise} \end{cases}$$

where $\bot$ is the smallest element of $\mathcal{R}$, and $r_1 \subseteq r_2$ is the standard subset relation over sets defined only when both r_1 and r_2 are not $\bot$.

The fact that version resource semiring is structural semiring with pre-order [Bru+14] is proven in appendix A.1.1. The multiplication $\otimes$ represents that if a value is used in version l_i, then all values in that data flow must also be available in version l_i; we can apply the versioned function with resource $\{l_1\}$ to both a versioned value with resource $\{l_1\}$ and $\{l_1, l_2\}$. The addition $\oplus$ represents splitting the data flow of a value in a typing context. Values with resource $\{l_i, lj\}$ are also allowed to be used in the context of l_i, and likewise for l_j; thus, $\{l_i\} \cup \{l_j\} = \{l_i, l_j\}$.

$0 = \bot$ is the smallest element indicating an irrelevant resource. Conversely, $1 = \emptyset$ explicitly indicates that the value has no version restrictions. A 1-indexed versioned value can be used as any versioned value unless it is a 0-indexed versioned value. These intuitive explanations will be detailed later in the typing rules.

4.5 The Declarative Type System of λ_{VL}

Typing judgments of the type system are of the form $\Gamma \vdash t : A$ with typing contexts Γ. A typing context Γ (or we sometimes note Δ) is a set of typed variables defined as follows.

Definition 4.5.1 (Typing Contexts)**.**

$$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A]_r \qquad \text{(contexts)}$$

Typing contexts are either empty $\emptyset$ or extended with a linear variable assumption $x : A$ or a *versioned assumption* $x : [A]_r$. For a versioned assumption, x can behave non-linearly, with substructural behavior captured by the semiring element $r \in \mathcal{R}$, which describes x's use in a term. We denote a *versioned context* by $[\Gamma]$, the typing context in which every assumption is a versioned assumption.

Figure 4.1 shows the typing rules for λ_{VL}. The typing rules for λ-terms are (INT), (VAR), (APP), and (ABS). (VAR) shows that linear variables can only be typed in a single context including themselves. Note that the typing rules for splitting a data flow, such as (APP), include context concatenation $+$, which permits the splitting version resources as defined in Def. 4.5.2.

λ_{VL} **typing rules** $\boxed{\Gamma \vdash t : A}$

$$\frac{}{\varnothing \vdash n : \mathsf{Int}}\ (\textsc{int}) \qquad \frac{}{x : A \vdash x : A}\ (\textsc{var}) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\ (\textsc{abs})$$

$$\frac{\Gamma_1 \vdash t_1 : A \to B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B}\ (\textsc{app})$$

$$\frac{\Gamma_1 \vdash t_1 : \Box_r A \quad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ [x] = t_1\ \mathbf{in}\ t_2 : B}\ (\textsc{let})$$

$$\frac{\Gamma \vdash t : A}{\Gamma, [\Delta]_0 \vdash t : A}\ (\textsc{weak}) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ (\textsc{der}) \qquad \frac{[\Gamma] \vdash t : A}{r \cdot [\Gamma] \vdash [t] : \Box_r A}\ (\textsc{pr})$$

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : B \quad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : B}\ (\textsc{sub}) \qquad \frac{\Gamma \vdash t : \Box_r A \quad l \in r}{\Gamma \vdash t.l : A}\ (\textsc{extr})$$

$$\frac{[\Gamma_i] \vdash t_i : A}{\bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash \{\overline{l = t} \mid l_i\} : \Box_{\{\overline{l}\}} A}\ (\textsc{ver})$$

$$\frac{[\Gamma_i] \vdash t_i : A}{\bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash \langle \overline{l = t} \mid l_i \rangle : A}\ (\textsc{veri})$$

FIGURE 4.1: λ_{VL}typing rules

Definition 4.5.2. [Context concatenation , & +] Two typing contexts can be concatenated by "," if they contain disjoint assumptions. Furthermore, the versioned assumptions appearing in both typing contexts can be combined using the context concatenation $+$ defined with the addition $\oplus$ in the version resource semiring as follows.

$$\begin{aligned} \varnothing + \Gamma &= \Gamma \\ (\Gamma, x : A) + \Gamma' &= (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin \mathrm{dom}(\Gamma') \\ \Gamma + \varnothing &= \Gamma \\ \Gamma + (\Gamma', x : A) &= (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin \mathrm{dom}(\Gamma) \\ (\Gamma, x : [A]_r) + (\Gamma', x : [A]_s) &= (\Gamma + \Gamma'), x : [A]_{(r \oplus s)} \end{aligned}$$

The (WEAK) rule provides weakening only for version assumptions indexed by 0. Since $0 = \bot$ is defined as an irrelevant resource in $\mathcal{R}$, this rule indicates that adding unneeded versioned assumptions to the typing context does not prevent the term from type checking, just as in linear type systems. The (DER) rule converts a linear assumption into a versioned assumption indexed by 1. This rule indicates the intuition that the linear assumption does not have any restrictions on versions. The (PR) rule introduces a version necessity indexed by r to a term and propagates the assumption into the context using the context multiplication $\cdot$ defined in Def. 4.5.3.

Definition 4.5.3. [Context multiplication $\cdot$ by a resource] Assuming that a context

contains only version assumptions, denoted $[\Gamma]$ in typing rules, then Γ can be multiplied by a version resource $r \in \mathcal{R}$ by using the product $\otimes$ in the version resource semiring, as follows.

$$r \cdot \emptyset = \emptyset \qquad r \cdot (\Gamma, x : [A]_s) = (r \cdot \Gamma), x : [A]_{(r \otimes s)}$$

Informally, $r \cdot \Gamma$ requires all assumptions in Γ to be available in that version. This property is the cornerstone of the type system and will be illustrated in Example 4.5.6 with examples.

The (SUB) rule weakens the version assumption based on the order defined in the version resource semiring. For example, suppose a value is typable in a context where a variable is only available in version 1. In that case, the value should also be typable, even if the variable is available in both versions 1 and 2. The (SUB) rule formalizes this intuition using the preorder in the version resource semiring. This rule is detailed in Example 4.5.5. The (LET) rule provides a way to remove the versioned necessity assigned to a term. From the perspective of term reuse, the version necessity assigned to a term ($\Box_r A$) is converted to a version assumption ($[A]_r$) and add it to the context of the body typing. Note that the contexts of the subterms are combined by context concatenation $+$ likewise in the (APP) rule.

The last two rules (VER) and (VERI) are for version records. The context of a version record is the sum of typing contexts multiplied by the version resource corresponding to each version-specific term. The summation of typing contexts is defined as follows:

Definition 4.5.4. [Context summation $\biguplus$] Using the context concatenation $+$, summation of typing contexts is defined as follows:

$$\biguplus_{i=1}^{n} \Gamma_i = \Gamma_1 + \cdots + \Gamma_n$$

Note that the type of the versioned computation and the extraction in (VERI) and (EXTR) have lost their version resource. In our current design, once a versioned value is evaluated in a particular version, it becomes a common value that can be used with other versioned values. The concluding types of these two rules illustrate this feature.

To aid in understanding the type system, we show some important facts in the following example.

Example 4.5.5 (Weakening version resources)**.**

Any linear assumption in the environment can be regarded as an arbitrary versioned assumption. This fact can be obtained by a combination of the rules (VAR), (DER), and (SUB) as follows.

$$\dfrac{\dfrac{\dfrac{}{f : \mathsf{Int} \to \mathsf{Int} \vdash f : \mathsf{Int} \to \mathsf{Int}}\,(\text{VAR})}{f : [\mathsf{Int} \to \mathsf{Int}]_1 \vdash f : \mathsf{Int} \to \mathsf{Int}}\,(\text{DER}) \qquad 1 \sqsubseteq r}{f : [\mathsf{Int} \to \mathsf{Int}]_r \vdash f : (\mathsf{Int} \to \mathsf{Int})}\,(\text{SUB})$$

The example describes the intuition that a linear assumption has no constraints on variable use for its versions.

As shown above, the (SUB) rule allows the versioned resources in a context can be increased. This fact supports the intuition that a term that is typed with versioned

assumptions only available in a particular version will still be typed with versioned assumptions that are available in more versions.

Example 4.5.6 (Ensuring the existence of consistent versions)**.**

The purpose of the type system is to ensure that all versions needed to evaluate a given program exist. The type system ensures this property by allocating resources with consisten and shared labels. The following program, a simplified version of the program in 4.2.5, is an example of a faulty program.

$$\textbf{let}\ [f] = \{l_1 = \text{id}, l_2 = \text{succ} \mid l_1\}\ \textbf{in let}\ [y] = \{l_1 = 1 \mid l_1\}\ \textbf{in}\ [f\ y].l_2$$

Type-checking this program halfway yields the following derivation tree.

$$\dfrac{\vdots \quad \dfrac{\vdots \quad \dfrac{\dfrac{\text{ERROR: } \{l_2\} \cup r \text{ cannot be a subset of } \{l_1\}}{\{l_1\}\cdot(f : [\mathsf{Int} \to \mathsf{Int}]_{\{l1,l2\}}, y : [\mathsf{Int}]_{\{l1\}}) \vdash [f\ y] : \Box_{\{l_2\} \cup r} X}\ (\text{PR})}{\{l_1\}\cdot(f : [\mathsf{Int} \to \mathsf{Int}]_{\{l1,l2\}}, y : [\mathsf{Int}]_{\{l1\}}) \vdash [f\ y].l_2 : X}\ (\text{EXTR})}{f : [\mathsf{Int} \to \mathsf{Int}]_{\{l1,l2\}} \vdash \textbf{let}\ [y] = \{l_1 = 1 \mid l_1\}\ \textbf{in}\ [f\ y].l_2 : X}\ (\text{LET})}{\varnothing \vdash \textbf{let}\ [f] = \{l_1 = \text{id}, l_2 = \text{succ} \mid l_1\}\ \textbf{in let}\ [y] = \{l_1 = 1 \mid l_1\}\ \textbf{in}\ [f\ y].l_2 : X}\ (\text{LET})$$

For clarity, the largest shared version resource is specified outside the context in this derivation tree such as $\{l_1\}\cdot(f : [\mathsf{Int} \to \mathsf{Int}]_{\{l1,l2\}}, y : [\mathsf{Int}]_{\{l1\}})$. Now recall that (PR) requires the same version resources for the entire context as introduced in the term. The largest shared resource in the context is $\{l_1\}$, but the resource in the term must have $\{l_2\}$ as a subset. As a result, the type checker reports this discrepancy. In this way, we use the nature of $\otimes$ to guarantee that each version value has a consistent version.

4.6 Dynamic Semantics

We give the small-step operational semantics of λ_{VL} in Figure 4.2. The rules follow the lazy-evaluation strategy; i.e., only functions t are evaluated to values to evaluate applications $t\ t'$. The operational semantics of λ_{VL} consists of two main parts – *evaluation* and *default version overwriting*. The λ_{VL} evaluation proceeds by alternating between reduction and default version overwriting.

We define *values* and *evaluation context* of λ_{VL} as follows.

Definition 4.6.1 (Values)**.**

$$v ::= n \mid \lambda x.t \mid [t] \mid \{\overline{l = t} \mid l_i\} \qquad \text{(values)}$$

Definition 4.6.2 (Evaluation contexts)**.**

$$E ::= [\cdot] \mid E\,t \mid E.l \mid \textbf{let}\ [x] = E\ \textbf{in}\ t \qquad \text{(evaluation contexts)}$$

Figure 4.2 shows the dynamic semantics for λ_{VL}. The λ_{VL} has five reduction rules. The (E-ABS) rule is the β-reduction rule for the lazy evaluation strategy, and (E-CLET) is a rule for contextual let-bindings. Each uses the captured x to assign a value according to the substitutions. Substitutions are given by a partial function of forms $(t \triangleright x)t$ or $(t \triangleright x)t$ and it remove versioned value constructors with both variable and term together in the $(\triangleright_\Box)$ and $(\triangleright_{\text{ver}})$ rules. A well-typed versioned value bound to a contextual-let binding will have its outer versioned value constructors

Evaluation rule

$$\frac{t \rightsquigarrow t'}{E[t] \longrightarrow E[t']}$$

Reduction rules

$$\frac{}{(\lambda x.t)\, t' \rightsquigarrow (t' \rhd x)t}\ (\text{E-ABS}) \qquad \frac{}{\textbf{let}\ [x] = v\ \textbf{in}\ t \rightsquigarrow (v \rhd [x])t}\ (\text{E-CLET})$$

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

Substitutions

$$\frac{}{(t' \rhd x)t = [t'/x]t}\ (\rhd_{\text{var}}) \qquad \frac{(t \rhd x)t' = t''}{([t] \rhd [x])t' = t''}\ (\rhd_{\square})$$

$$\frac{}{(\{\overline{l = t} \mid l_i\} \rhd [x])t = [\langle \overline{l = t} \mid l_i \rangle / x]t}\ (\rhd_{\text{ver}})$$

Default version overwriting

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

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

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

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

FIGURE 4.2: λ_{VL}dynamic semantics

removed along with the variable. A term that will eventually be substituted into x loses its outermost version resource.

The next three reduction rules are for extracting versioned values. As explained above, a versioned value can only be evaluated when it is extracted. The two versioned values – promotions and versioned records – are evaluated to terms that lose their version resources along with the @-notation by extraction. The @-notation is a function for overriding a default version; it scans through the sub terms recursively and overwrites default versions of all intermediate terms $\langle \overline{l = t} \mid l_i \rangle$ with its label l. Eventually, these intermediate terms are evaluated into version-specific terms using the (E-VERI) rule.

Example 4.6.3 (Evaluation process of a versioned function application).

To aid in understanding the dynamic semantics, we show an evaluation process of the versioned function application presented in the introduction.

$$\textbf{let}\ [f] = \{l_1 = \lambda x.x, l_2 = \lambda x.x+1 \,|\, l_1\}\ \textbf{in}\ \textbf{let}\ [y] = \{l_1 = 1, l_2 = 2 \,|\, l_1\}\ \textbf{in}\ [f\,y]$$

Both version function f and version variable have different definitions with labels l_1 and l_2. This program intuitively evaluates to $\{l_1 = 1, l_2 = 3\}$ – precisely, to a suspended versioned computation that is expected to evaluate to 1 and 3 in each version. Hereafter, we abbreviate $\lambda x.x$ as id and $\lambda x.x+1$ as succ.

And next, since $\{l_1 = 1, l_2 = 2 \,|\, l_1\}$ is a value, the program is evaluated as follows.

$$\begin{aligned}
&\longrightarrow \underline{[\{l_1 = \mathrm{id}, l_2 = \mathrm{succ} \,|\, l_1\} \triangleright [f]]}\ (\textbf{let}\ [y] = \{l_1 = 1, l_2 = 2 \,|\, l_1\}\ \textbf{in}\ [f\,y]) && (\text{E-CLET}) \\
&= \underline{[\langle l_1 = \mathrm{id}, l_2 = \mathrm{succ} \,|\, l_1\rangle / f]}\ (\textbf{let}\ [y] = \{l_1 = 1, l_2 = 2 \,|\, l_1\}\ \textbf{in}\ [f\,y]) && (\triangleright_{\mathrm{ver}}) \\
&= (\textbf{let}\ [y] = \{l_1 = 1, l_2 = 2 \,|\, l_1\}\ \textbf{in}\ [\langle l_1 = \mathrm{id}, l_2 = \mathrm{succ} \,|\, l_1\rangle\, y]) && (\text{substitution})
\end{aligned}$$

Note that in the first two lines, the ($\triangleright_{\mathrm{ver}}$) rule simultaneously removes the versioned constructors of $[x]$ and $\{l_1 = \mathrm{id}, l_2 = \mathrm{succ} \,|\, l_1\}$. The term eventually assigned to f is a versioned computation that inherits the default version l_1.

The program is evaluated as well for y.

$$\longrightarrow^* [\langle l_1 = \mathrm{id}, l_2 = \mathrm{succ} \,|\, l_1\rangle\ \langle l_1 = 1, l_2 = 2 \,|\, l_1\rangle] \qquad (\text{E-CLET}, \triangleright_{\mathrm{ver}}, \text{substitution})$$

The result suspended versioned computation contains the respective computations for labels l_1 and l_2; thus, we can obtain the result value by extraction, as shown below.

$$\begin{aligned}
&\textbf{let}\ [f] = \{l_1 = \mathrm{id}, l_2 = \mathrm{succ} \,|\, l_1\}\ \textbf{in}\ \textbf{let}\ [y] = \{l_1 = 1, l_2 = 2 \,|\, l_1\}\ \textbf{in}\ [f\,y].l_1 \\
\longrightarrow^*\ & [\langle l_1 = \mathrm{id}, l_2 = \mathrm{succ} \,|\, l_1\rangle\ \langle l_1 = 1, l_2 = 2 \,|\, l_1\rangle].l_1 && (\text{E-CLET}, \triangleright_{\mathrm{ver}}, \text{substitution}) \\
\longrightarrow\ & (\langle l_1 = \mathrm{id}, l_2 = \mathrm{succ} \,|\, l_1\rangle\ \langle l_1 = 1, l_2 = 2 \,|\, l_1\rangle)@l_1 && (\text{E-EX1}) \\
\equiv\ & \langle l_1 = \mathrm{id}, l_2 = \mathrm{succ} \,|\, l_1\rangle@l_1\ \langle l_1 = 1, l_2 = 2 \,|\, l_1\rangle@l_1 && (\text{def-ver overwriting}) \\
\equiv^*\ & 1
\end{aligned}$$

We can perform the same extraction for l_2 and obtain 3.

4.7 Metatheory

We give a precise formalization to some properties of λ_{VL}. Appendix A provides collected rules in this section, further auxiliary definitions, supplemental lemmas, and proofs.

As with other coeffect calculi, there are two variants of the substitution lemmas, one through linear assumptions and the other through versioned assumptions. We give the proofs by structural induction on the typing derivation, which is somewhat tricky. We must carefully manage how version resources are divided in the typing context; thus, we would like to adopt a generalized form of the versioned substitution lemma.

Lemma 4.7.1. [Well-typed linear substitution]

$$\left.\begin{array}{r}\Delta \vdash t' : A \\ \Gamma, x : A, \Gamma' \vdash t : B\end{array}\right\} \implies \Gamma + \Delta + \Gamma' \vdash [t'/x]t : B$$

Lemma 4.7.2. [Well-typed versioned substitution]

$$\left.\begin{array}{r}[\Delta] \vdash t' : A \\ \Gamma, x : [A]_r, \Gamma' \vdash t : B\end{array}\right\} \implies \Gamma + r \cdot \Delta + \Gamma' \vdash [t'/x]t : B$$

We further claim that for well-typed versioned values in λ_{VL}, an extraction for $l \in r$ always succeeds. Here we need type preservation for the default version overwrite function @.

Lemma 4.7.3. [Type safety for default version overwriting @]
For any version label l:

$$\Gamma \vdash t : A \implies \Gamma \vdash t@l : A$$

Lemma 4.7.4. [Type-safe extraction for versioned values]

$$[\Gamma] \vdash v : \Box_r A \implies \forall l_k \in r.\ \exists t'. \begin{cases} v.l_k \longrightarrow t' & (\textit{progress}) \\ [\Gamma] \vdash t' : A & (\textit{preservation}) \end{cases}$$

Using the above lemmas, we give a notion of λ_{VL} type safety as follows.

Theorem 4.7.5. [Type preservation for reductions]

$$\left.\begin{array}{l}\Gamma \vdash t : A \\ t \leadsto t'\end{array}\right\} \implies \Gamma \vdash t' : A$$

Theorem 4.7.6. [Type preservation for evaluations]

$$\left.\begin{array}{l}\Gamma \vdash t : A \\ t \longrightarrow t'\end{array}\right\} \implies \Gamma \vdash t' : A$$

Theorem 4.7.7. [λ_{VL} progress]

$$\emptyset \vdash t : A \implies (\mathsf{value}\ t) \vee (\exists t'. t \longrightarrow t')$$

Chapter 5

Programming with Versions on Ordinary Functional Languages

Contents

5.1 Overview **31**
5.1.1 Motivation 32
5.1.2 Intuition to the Compilation 33
5.2 An Intermediate Language, VLMini **33**
5.2.1 Syntax 33
5.2.2 Types and Kinds 34
5.2.3 Constraints 34
5.3 Girard's Translation for VLMini **35**
5.4 Bundling **36**
5.4.1 Intuition to the Bundling 36
5.4.2 Constraint-based Bundling 38
5.5 Algorithmic Type Inference **38**
5.5.1 Type Unification and Substitutions 40
5.5.2 Pattern Type Synthesis 41
5.5.3 Type Inference 42
5.6 Extend with Data Structures **44**

5.1 Overview

This chapter presents techniques for realizing programming with versions in an ordinary functional program developed per package. We elaborate this approach by compiling a functional language with a Haskell-like syntax, called VL, into VLMini, a subset of λ_{VL} excluding the label-dependent terms. The entire translation consists of three parts: *Girard's translation*, *Bundling*, and an *algorithmic type inference* in Figure 5.1.

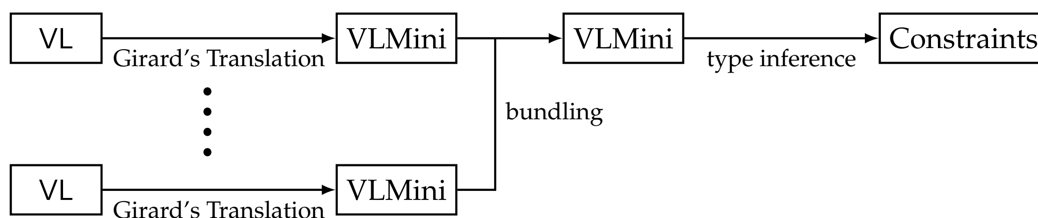


FIGURE 5.1: Translation Overview.

Girard's Translation for VLMini An abstract syntax tree of a VL program for each module and version is translated into a VLMini program with extended Girard's transformation. Girard's transformation is originally a translation of the simply-typed λ-calculus into an intuitionistic linear calculus. [Gir87] The idea is to replace every intuitionistic arrow $A \rightarrow B$ with $!A \multimap B$, and subsequently unbox via let in abstraction and promote when applying. [OLE19] Following the Orchard notation [OLE19], we extend Girard's translation for applying the translation between VL and VLMini.

Bundling Bundling is a translation to treat top-level definitions in multiple module versions as versioned values. Intuitively, bundling is the automatic generation of a versioned record. It records the availability of top-level symbols in each version and treats them as a single versioned record. The bundling also generates constraints on the version label from the meta-information of each module as a straightforward correspondence of the type inference rules of versioned records to conform to the constraint-based type inference rules described below. This method allows programmers to automatically generate information equivalent to that contained in λ_{VL} label-dependent terms without incurring superfluous implementation costs related to term-level versioning.

Algorithmic Type Inference We present version inference rules for VLMini programs. Intuitively, the inference system synthesizes the available versions of the program from the term structure. First, the inference system assigns a *version resource variable* to every subterm of the VLMini program, just like a type variable in traditional Hindley-Milner type inference. Version resource variables are lifted into the type environment by contextual-let and pattern matching, and constraints are generated between each version resource variable by promotion.

5.1.1 Motivation

This compilation approach is motivated by the following λ_{VL}-specific difficulties.

- *Complex syntax derived from substructural language.* λ_{VL} described in the previous chapter is a core calculus based on the concept of programming with versions. However, since λ_{VL} was not designed as a surface language, its complex syntax and semantics only provide primitive constructs to manipulate versioned values. 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 the program.* The λ_{VL} program includes versions as part of label-dependent terms such as versioned records $\{\overline{l_i = t_i}\}$ and extractions $t.l$. 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.

5.1.2 Intuition to the Compilation

Our solution to mitigate the two difficulties listed above is translating a functional language based on λ-calculus using externally defined version information to construct specific terms of λ_{VL}.

This idea 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 2.2 and 2.3 provide functions `makeHash` and `match`. These programs are distinct, and no program belongs to both versions. Therefore, it is possible to assign each version of the definition to a label mechanically. Therefore, it would be possible to obtain a program with sufficient information for a versioned record by mechanically assigning the definition of each version to a version label for constructing a versioned record.

Second, 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, a program $\{l_1 = ..., l_2 = ... \,|\, ...\}$ will be indicated to be available with labels l_1 and l_2 by the type system as with a type $\square_{\{l_1,l_2\}}A$. Although we need to prioritize these two labels, it should be possible to design type inference and unification algorithms to determine the appropriate version label from these alternatives.

5.2 An Intermediate Language, VLMini

Following the above intuition, the target language addressed in this chapter, called VLMini, is the language in which the label-dependent terms are removed from λ_{VL}. It has the same terms as $\ell\mathcal{R}$PCF and GrMini, but unlike those two calculi, VLMini is specialized to take version resources as coeffects.

5.2.1 Syntax

The syntax of VLMini is defined as follows.

Definition 5.2.1 (VLMini terms)**.**

$$t ::= \underbrace{x \mid t_1\, t_2 \mid \lambda p.t}_{\lambda\text{-terms}} \mid \underbrace{[t]}_{\text{promotion}} \mid \underbrace{C t_0, ..., t_n}_{\text{constructors}} \mid \underbrace{\mathbf{case}\; t\; \mathbf{of}\; \overline{p_i \mapsto t_i}}_{\text{case}} \qquad \text{(terms)}$$

$$p ::= \underbrace{x}_{\text{variables}} \mid \underbrace{_}_{\text{wildcard}} \mid \underbrace{[p]}_{\text{promoted patterns}} \mid \underbrace{n \mid C p_0, ..., p_n}_{\text{constructors}} \qquad \text{(patterns)}$$

$$C ::= \underbrace{(,)}_{\text{pairs}} \mid \underbrace{[,]}_{\text{lists}} \qquad \text{(data constructors)}$$

VLMini has all the terms except for versioned records $\{\overline{l_i = t_i} \,|\, l_k\}$, intermediate term $\langle \overline{l_i = t_i} \,|\, l_k \rangle$, and extractions $t.l_k$. Here, lambda abstraction applied to a promoted pattern is a syntax sugar of cotextual-let in λ_{VL}.

Definition 5.2.2 (Syntax sugar for contextual-let)**.**

$$\mathbf{let}\; [p] = t_1\; \mathbf{in}\; t_2 \triangleq (\lambda[p].t_2)\, t_1 \qquad \text{(syntax sugar)}$$

Another important change is data constructors introduction $Ct_1, ..., t_n$ and elimination $\mathbf{case}\; t\; \mathbf{of}\; \overline{p_i \mapsto t_i}$. VLMini allows lists and pairs as data constructors. The data

constructors must account for nontrivial operations in algorithmic type inference for VLMini. The data structures are provided as pairs and lists.

5.2.2 Types and Kinds

Types and version resources are almost the same as λ_{VL} and are defined as follows.

Definition 5.2.3 (VLMini types)**.**

$$
\begin{aligned}
A, B &::= \mathsf{Int} \mid K A_1, ..., A_n \mid \alpha \mid A \to B \mid \Box_r A && \text{(types)} \\
K &::= (,) \mid [,] && \text{(type constructos)} \\
r &::= \bot \mid \emptyset \mid \{l_i\} \mid \alpha \mid r_1 \oplus r_2 \mid r_1 \otimes r_2 && \text{(version resources)}
\end{aligned}
$$

The $\oplus$ and $\otimes$ here are the sum and product defined in version resource semiring as well as λ_{VL}. Type constructors are also added to the type in response to the VLMini term having a data constructor.

Also, assuming a typical development that utilizes multiple modules, we clarify the definition of version labels.

Definition 5.2.4 (Version labels)**.**

$$
l ::= [\overline{M_i \mapsto V_i}] \qquad \text{(version labels)}
$$

where M is a metavariable over module names, and V is a metavariable over version numbers.

A version label is a vector of modules and their corresponding versions, in which M_i is unique. Given modules and their versions, the corresponding set of version labels characterizes the variation of programs of a version value. The size of a version label set is proportional to the product of the number of modules and the number of versions. The translations and inferences in this chapter aim to use constraints to represent the appropriate version labels for all type variables since the version label space is huge for large projects.

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 as follows.

Definition 5.2.5 (VLMini kinds)**.**

$$
\kappa ::= \mathsf{Type} \mid \mathsf{Labels} \qquad \text{(kinds)}
$$

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 type variables of Type during algorithmic type inference.

5.2.3 Constraints

Constraints generated by bundling and algorithmic type inference are of the following form.

Definition 5.2.6 (Constraints).

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

$$\mathcal{L} ::= \langle\langle \overline{M_i \mapsto V_i} \rangle\rangle \qquad \text{(dependent versions)}$$

where M is a metavariable over module names, and V is a metavariable over version numbers.

Constraints comprise *propositional formulae*, *variable dependencies* ($\alpha \preceq \alpha'$), and *label dependencies* ($\alpha \preceq \mathcal{L}$). Constraints represent dependencies between variables and on specific versions, and a type variable α that occurs in constraints is a type variable that is expected to be classified as Labels. Hence after we will abbreviate $(\mathcal{C}_1 \wedge ... \wedge \mathcal{C}_n)$ as $\bigwedge_{i=1}^{n} \mathcal{C}_i$.

A variable dependency intuitively requires that $\alpha \sqsubseteq \alpha'$, i.e., that the resource variable α be less than α' under the partial order defined by version resource semiring. Similarly, A label dependency is represented by a dependent version $\mathcal{L}$, a sequence of pairs of a module name M_i and a corresponding version number V_i, which implies that "a version label expected for the type variable α must be V_i for module M_i." For example, assuming that versions 1.0.0 and 2.0.0 exist for modules A and B, respectively, the minimal upper bound set of version labels satisfying $\alpha \preceq \langle\langle A \rightarrow 1.0.0 \rangle\rangle$ is $\alpha = \{\{A = 1.0.0, B = 1.0.0\}, \{A = 1.0.0, B = 2.0.0\}\}$, and if the constraint resolution is successful, α will be specialized with either of two elements.

5.3 Girard's Translation for VLMini

This compilation is based on Girard's translation [Gir87] of simply-typed lambda calculus to intuitionistic linear calculus. Orchard [OLE19] note that any term and type derivation of a simply-typed lambda calculus can be translated into GrMini. Following Orchard's approach, we extend Girard's translation as a translation between VL and VLMini.

Girard's translation for VLMini from the simply-typed lambda calculus to VLMini is described as follows.

Definition 5.3.1 (Girard's translation).

$$\begin{aligned} [\![n]\!] &\equiv n \\ [\![x]\!] &\equiv x \\ [\![\lambda p.t]\!] &\equiv \lambda[p].[\![t]\!] \\ [\![t\ s]\!] &\equiv [\![t]\!]\ [[\![s]\!]] \end{aligned}$$

The idea is to replace all occurrences of $A \rightarrow B$ with $\Box_r A \rightarrow B$ using the appropriate 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 5.2. Suppose there are two modules called `Main` and `M`. Then, the `main` function defined in the `Main` module is translated to the following VLMini program.

```
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 5.2: Simple example program written in VL (Haskell subset).

Example 5.3.2 (`main` after applying Girard's translation)**.**

$$
\begin{aligned}
&main : \mathsf{Int} \\
&main = id\ [n]
\end{aligned}
$$

Here, the argument n of the function application is promoted. Similarly, the `id` in `M` version 1 is translated into the VLMini program as follows.

Example 5.3.3 (`id` and `n` after applying Girard's translation)**.**

$$
\begin{aligned}
&id : \Box_r \mathsf{Int} \to \mathsf{Int} \\
&id = \lambda[n].n \\
&n : \mathsf{Int} \\
&n = 1
\end{aligned}
$$

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.

5.4 Bundling

In order to refer to variables from external modules as versioned values in VLMini programs, the bundling treats the top-level symbols of external modules as version records. This section describes the intuition behind the bundling and then discusses the techniques to achieve this in constraint generation.

5.4.1 Intuition to the Bundling

The bundling uses externally defined version information to generate version labels. Using the simple example in Figure 5.2 again, we will explain the purpose of the bundling.

We first consider the variation of external module versions needed to compile the program in the `Main` module. In this example, the modules and versions to be considered are `Main`, `M version 1`, and `M version 2`. Since the `Main` module itself is not an external module of `Main` and is therefore not a candidate for variation, we have two version labels to be generated for `M version 1` and `M version 2`.

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

We then 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. Although λ_{VL} does not support type annotation, the types to be inferred are explicitly indicated here for illustrative purposes.

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

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.

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

$$\begin{aligned}
&main : \Box_s \mathsf{Int} \\
&main = [id\,[n]]
\end{aligned}$$

Here we use the version resource variable s that 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.

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

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.

$$\begin{aligned}
&main.l_1 \to^* 1 \\
&main.l_1 \to^* 2
\end{aligned}$$

These results are equivalent to those obtained by selecting versions 1 and 2 of module `M` in Figure 5.2 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.

5.4.2 Constraint-based Bundling

We propose a constraint-based method for achieving the abovementioned operations in VLMini with no label-dependent terms. The idea is to record the collected label as a label dependency $\langle\langle\overline{M_i \to V_i}\rangle\rangle$ and generate a constraint between the newly generated type variable and its label dependency.

For example, the types *id* and *n* in Figure 5.2 can be represented by the following types with constraints.

Example 5.4.1 (`id` and `n` bundled with constraints)**.**

$$id : \Box_r(\Box_{r'}\mathsf{Int} \to \mathsf{Int}) \mid r \preceq \langle\langle\overline{\mathtt{M} \mapsto 1.0.0}\rangle\rangle \lor r \preceq \langle\langle\overline{\mathtt{M} \mapsto 2.0.0}\rangle\rangle$$
$$n : \Box_s\mathsf{Int} \mid s \preceq \langle\langle\overline{\mathtt{M} \mapsto 1.0.0}\rangle\rangle \lor s \preceq \langle\langle\overline{\mathtt{M} \mapsto 2.0.0}\rangle\rangle$$

where r and s are newly added by this conversion, and the constraints marked to the right side of the type annotation indicate the constraints that r and s will satisfy.

A label dependency indicates the version of the external module that the type variable must satisfy as if it were a kind in a record calculus. [Oho95] For example, the type declarations indicate that both *id* and *n* are versioned values that are expected to be available only in either `version 1` or `version 2` of the module `M`.

Label dependencies are generated only from bundling, while variable dependencies are generated only from type inference. The constraint from the bundling is used for constraint resolution, along with constraints from type inference.

5.5 Algorithmic Type Inference

The previous chapter gave the declarative type system of λ_{VL}, but we need algorithmic type inference rules for implementation. Therefore, this section defines the type inference system for λ_{VL} as an extension of the traditional Hindley-Milner type inference.

The type inference aims to infer a *type and version label* of every occurrence of variables in a λ_{VL} program with no type annotations. What makes it different from normal Hindley-Milner type inference is that the constraints contain not only type information but also constraints on version labels. This is because resource variables are expected to eventually become a set of versions of each module. The type inference does infer and collect constraints on these version resource variables from the program's structure.

The type inference rules of VLMini consist of three judgments: type inference, pattern type synthesis, and type unification. These three judgments are almost common to the Gr [OLE19] language, which is similarly based on coeffect calculus. The difference between Gr and VLMini is that Gr provides type-checking rules in a bidirectional approach [DK13; DK19] to describe complex resource constraint annotations described in the Gr program. In contrast, VLMini provides algorithmic type inference rules that assume support for automatic constraint generation by bundling.

In addition, Gr supports multiple version resources, while VLMini is specialized for version resources.

The type inference is defined by the function *synthesis*, which has the form of:

Definition 5.5.1 (VLMini type inference)**.**

$$\Sigma; \Gamma \vdash t \Rightarrow A; \Sigma'; \theta; \mathcal{C} \qquad \text{(synthesis)}$$

where the inputs to synthesis are the type variable kinds Σ, the environment Γ of term variables, and the term t. As an output, the synthesis produces a type A, output type variable kinds Σ', output substitution θ, and a constraint C for the next phase of type inference. The input and output type variable kinds Σ and Σ' always satisfy $\Sigma \subseteq \Sigma'$ due to the additional type variables added in this phase.

Here, the environments for types (Γ) and type variable kinds (Σ) are defined below. The typing context is the same as λ_{VL}, and a type variable kinds are added to reflect the introduction of the unification variable.

Definition 5.5.2 (VLMini typing contexts)**.**

$$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A]_r \qquad \text{(typing contexts)}$$

Definition 5.5.3 (VLMini type variable kinds)**.**

$$\Sigma ::= \emptyset \mid \Sigma, \alpha : \kappa \qquad \text{(type variable kinds)}$$

The pattern type synthesis is defined as follows.

Definition 5.5.4 (VLMini pattern type synthesis)**.**

$$\Sigma, R \vdash p : A \rhd \Gamma; \Sigma'; \theta \qquad \text{(pattern type synthesis)}$$

Pattern type synthesis takes a pattern p, type variables Σ, and resource environment R as input and synthesizes an output typing context Γ, output type variables Σ', and substitutions θ. 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.

Here, the resource context R is defined as follows.

Definition 5.5.5 (Resource contexts)**.**

$$R ::= - \mid r \qquad \text{(resource contexts)}$$

When we check a nested promoted pattern, the resource context captures version resources inside a pattern.

The following judgment defines type unification.

Definition 5.5.6 (VLMini type unification)**.**

$$\Sigma \vdash A \sim B \rhd \theta \qquad \text{(type unification)}$$

Type unification produces a substitution θ under structural type congruence under typing context Σ. It is triggered to unify the unification type variable during type inference or pattern type synthesis.

VLMini type unification $\boxed{\Sigma \vdash A \sim B \rhd \theta}$

$$\frac{\Sigma \vdash A' \sim A \rhd \theta_1 \quad \Sigma \vdash \theta_1 B \sim \theta_1 B' \rhd \theta_2}{\Sigma \vdash A \to B \sim A' \to B' \rhd \theta_1 \uplus \theta_2} \quad (U_{\to})$$

$$\frac{\Sigma \vdash A \sim A' \rhd \theta_1 \quad \Sigma \vdash \theta_1 r \sim \theta_1 r' \rhd \theta_2}{\Sigma \vdash \Box_r A \sim \Box_{r'} A' \rhd \theta_1 \uplus \theta_2} \quad (U_{\Box})$$

$$\frac{(\alpha : \kappa) \in \Sigma \quad \Sigma \vdash A : \kappa}{\Sigma \vdash \alpha \sim A \rhd \alpha \mapsto A} \quad (U_{\mathrm{VAR}\exists})$$

$$\frac{(\alpha : \kappa) \in \Sigma}{\Sigma \vdash \alpha \sim \alpha \rhd \emptyset} \quad (U_{\mathrm{VAR}=}) \qquad \frac{\Sigma \vdash A : \kappa}{\Sigma \vdash A \sim A \rhd \emptyset} \quad (U_{=})$$

FIGURE 5.3: VLMini type unification

5.5.1 Type Unification and Substitutions

Throughout type inference, we use *type substitutions* θ, a map from a type variable α to type A. The type substitutions defined here are standard, similar to those in many polymorphic systems, but the rule includes substitutions for version resources.

Definition 5.5.7 (Type substitutions θ)**.** Let a θ be a type substitution of the form $\theta = \alpha \mapsto A$. The application of type substitution has the form θB and is defined as follows.

$$\begin{array}{rcll} \theta K &=& K & \\ \theta\alpha &=& A & (\theta(\alpha) = A) \\ \theta\alpha &=& \alpha & (\text{otherwise}) \\ \theta(A \to B) &=& \theta A \to \theta B & \\ \theta(\Box_r A) &=& \Box_{(\theta r)}(\theta A) & \end{array} \qquad \begin{array}{rcll} \theta 0 &=& 0 & \\ \theta 1 &=& 1 & \\ \theta\alpha &=& A & (\theta(\alpha) = A) \\ \theta\alpha &=& \alpha & (\text{otherwise}) \\ \theta(r_1 \otimes r_2) &=& (\theta r_1) \otimes (\theta r_2) & \\ \theta(r_1 \oplus r_2) &=& (\theta r_1) \oplus (\theta r_2) & \end{array}$$

Type substitution can be applied to any type and version resource in which the type variable can appear to traverse an argument type B structurally recursively and substitute the matching α for the type A.

Such substitutions are produced by type unification during type inference. Type unification rules are listed in Figure 5.3. In a rule with multiple premises, the substitution output from the first premise is applied to the type of the next premise.

Substitutions can also be composed, written as $\theta \uplus \theta_2$. We define substitution compositions as follows.

Definition 5.5.8 (Substitution compositions)**.** Let type substitutions θ_1 and θ_2 . We define substitution compositions by induction on θ_1.

$$\emptyset \uplus \theta_2 = \theta_2$$

$$(\theta_1, \alpha \mapsto A) \uplus \theta_2 = \begin{cases} (\theta_1 \uplus (\theta_2 \backslash \alpha) \uplus \theta), \alpha \mapsto \theta A & \theta_2(\alpha) = B \wedge \Sigma \vdash A \sim B \rhd \theta \\ (\theta_1 \uplus \theta_2), \alpha \mapsto A & \alpha \notin \mathrm{dom}(\theta_2) \end{cases}$$

Type assignment may fail because it is a partial operation that depends on unification judgment. Hence after we will abbreviate $(\theta_1 \uplus ... \uplus \theta_n)$ as $\biguplus_{i=1}^{n} \theta_i$.

VLMini pattern type inference $\boxed{\Sigma, R \vdash p : A \rhd \Gamma; \Sigma'; \theta}$

$$\frac{}{\Sigma; - \vdash n : \mathsf{Int} \rhd \emptyset; \Sigma; \emptyset}\ (\text{PINT}) \qquad \frac{\Sigma \vdash r : \mathsf{Labels}}{\Sigma; r \vdash n : \mathsf{Int} \rhd \emptyset; \Sigma; \emptyset}\ ([\text{PINT}])$$

$$\frac{\Sigma \vdash A : \mathsf{Type}}{\Sigma; - \vdash x : A \rhd x : A; \Sigma; \emptyset}\ (\text{PVAR}) \qquad \frac{\Sigma \vdash A : \mathsf{Type} \quad \Sigma \vdash r : \mathsf{Labels}}{\Sigma; r \vdash x : A \rhd x : [A]_r; \Sigma; \emptyset}\ ([\text{PVAR}])$$

$$\frac{\begin{array}{c}\Sigma' = \Sigma, \alpha :_\exists \mathsf{Labels}, \beta :_\exists \mathsf{Type} \quad \Sigma' \vdash \alpha : \mathsf{Labels} \\ \Sigma'; \alpha \vdash p : \beta \rhd \Delta; \Sigma''; \theta \quad \Sigma' \vdash A \sim \Box_\alpha \beta \rhd \theta'\end{array}}{\Sigma; - \vdash [p] : A \rhd \Delta; \Sigma''; \theta \uplus \theta'}\ (\text{P}\Box)$$

$$\frac{\begin{array}{c}\Sigma' = \Sigma, \alpha :_\exists \mathsf{Labels}, \beta :_\exists \mathsf{Type} \quad \Sigma' \vdash \alpha : \mathsf{Labels} \\ \Sigma'; r \otimes \alpha \vdash p : \beta \rhd \Delta; \Sigma''; \theta \quad \Sigma' \vdash A \sim \Box_\alpha \beta \rhd \theta'\end{array}}{\Sigma; r \vdash [p] : A \rhd \Delta; \Sigma''; \theta \uplus \theta'}\ ([\text{P}\Box])$$

FIGURE 5.4: VLMini pattern type synthesis

Example 5.5.9 (Successful composition)**.**

$$(\alpha_0 \mapsto (\mathsf{Int}, \alpha_1)) \uplus (\alpha_0 \mapsto (\alpha_2, \mathsf{Int})) = (\alpha_0 \mapsto (\mathsf{Int}, \mathsf{Int}), \alpha_1 \mapsto \mathsf{Int}, \alpha_2 \mapsto \mathsf{Int})$$

Here (Int, α_1) and (α_2, Int) are structurally unifiable, so the composition succeeds and produces a composed substitution $\alpha_0 \mapsto (\mathsf{Int}, \mathsf{Int}), \alpha_1 \mapsto \mathsf{Int}, \alpha_2 \mapsto \mathsf{Int}$.

5.5.2 Pattern Type Synthesis

Pattern type synthesis rules are classified into two categories whether or not it has resources in the input resource context R. The base rules are PINT, PVAR, P□, and PCON, while the other rules are resource-aware versions of the corresponding rules. The resource-aware rules assume that they are triggered within the promoted pattern and collect version resource r in the resource context. The purpose of pattern type synthesis is to properly convey the version resources captured by promoted patterns to the output typing context.

We provide the following two rules for variables, whether the variable pattern occurs within a promoted pattern or not.

$$\frac{\Sigma \vdash A : \mathsf{Type}}{\Sigma; - \vdash x : A \rhd x : A; \Sigma; \emptyset}\ (\text{PVAR}) \qquad \frac{\Sigma \vdash A : \mathsf{Type} \quad \Sigma \vdash r : \mathsf{Labels}}{\Sigma; r \vdash x : A \rhd x : [A]_r; \Sigma; \emptyset}\ ([\text{PVAR}])$$

The rule on the left has no resources in the resource context because the variable pattern is not inside a promoted pattern. Therefore, this pattern produces typing context $x : A$ just like a normal linear lambda calculus. The right rule is for a variable pattern within the promoted pattern, and a resource r is lifted 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 the promoted pattern propagate the information of version resources to the pattern type synthesis of a subpattern.

VLMini algorithmic type synthesis $\boxed{\Sigma; \Gamma \vdash t \Rightarrow A; \Sigma'; \theta; \mathcal{C}}$

$$\frac{}{\Sigma; \Gamma \vdash n \Rightarrow \mathsf{Int}; \Sigma; \emptyset; \top} \quad (\Rightarrow_{\text{INT}})$$

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

$$\frac{\begin{array}{c}\Sigma_1, \alpha :_{\exists} \mathsf{Type}; - \vdash p : \alpha \rhd \Gamma'; \Sigma_2; \theta \\ \Sigma_2; \Gamma, \Gamma' \vdash t \Rightarrow B; \Sigma_3; \theta'; \mathcal{C}\end{array}}{\Sigma_1; \Gamma \vdash \lambda p.t \Rightarrow \theta(\alpha \to B); \Sigma_3; \theta \uplus \theta'; \mathcal{C}} \quad (\Rightarrow_{\text{ABS}})$$

$$\frac{\begin{array}{c}\Sigma_1; \Gamma \vdash t_1 \Rightarrow A; \Sigma_2; \theta_1; \mathcal{C}_1 \quad \Sigma_2; \Gamma \vdash t_2 \Rightarrow A'; \Sigma_3; \theta_2; \mathcal{C}_2 \\ \Sigma_4 = \Sigma_3, \beta :_{\exists} \mathsf{Type} \quad \Sigma_4 \vdash A \sim A' \to \beta \rhd \theta_3 \\ \theta_4 = \theta_1 \uplus \theta_2 \uplus \theta_3\end{array}}{\Sigma_1; \Gamma \vdash t_1\, t_2 \Rightarrow \theta_4\beta; \Sigma_4; \theta_4; \mathcal{C}_1 \wedge \mathcal{C}_2} \quad (\Rightarrow_{\text{APP}})$$

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

FIGURE 5.5: VLMini algorithmic typing

$$\frac{\begin{array}{c}\Sigma' = \Sigma, \alpha :_{\exists} \mathsf{Labels}, \beta :_{\exists} \mathsf{Type} \quad \Sigma' \vdash \alpha : \mathsf{Labels} \\ \Sigma'; \alpha \vdash p : \beta \rhd \Delta; \Sigma''; \theta \quad \Sigma' \vdash A \sim \square_\alpha \beta \rhd \theta'\end{array}}{\Sigma; - \vdash [p] : A \rhd \Delta; \Sigma''; \theta \uplus \theta'} \quad (\text{P}\square)$$

The input type A is expected to be versioned type, so the rule generates type variables α and β and then performs pattern type synthesis of the subpattern considering A as $\square_\alpha \beta$. Here, the resource α captured by the promoted pattern is lifted to the resource context in the subpattern synthesis. Finally, the rule unifies A and $\square_\alpha \beta$ and produces a substitution for type refinement after pattern type synthesis. The only change for the nested promoted pattern is that α interacts with the resource r captured by the outer promoted pattern.

5.5.3 Type Inference

We describe each type inference rule in turn. The algorithmic type rules for VLMini are listed in Figure 5.5, and they are algorithmic variants of λ_{VL}'s declarative typing rules.

The inference rules synthesize their types by looking at the input typing context.

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

In $\Rightarrow_{\text{GR}}$, the variable in the input typing contexts has a versioned type, so the r is checked for having kind Labels. Since both $\Rightarrow_{\text{LIN}}$ and $\Rightarrow_{\text{GR}}$ do not create new type variables, the output type variable kinds are the same as that of the input, and hereafter the same will be assumed unless otherwise described.

VLMini context grading $\boxed{\Sigma \vdash [\Gamma]_{\mathsf{Labels}} \rhd \Gamma'}$

$$\frac{}{\Sigma \vdash [\emptyset]_{\mathsf{Labels}} \rhd \emptyset} \quad (\emptyset)$$

$$\frac{\Sigma \vdash [\Gamma]_{\mathsf{Labels}} \rhd \Gamma'}{\Sigma \vdash [\Gamma, x : A]_{\mathsf{Labels}} \rhd \Gamma', x : [A]_{\emptyset}} \quad ([\text{LIN}])$$

$$\frac{\Sigma \vdash [\Gamma]_{\mathsf{Labels}} \rhd \Gamma'}{\Sigma \vdash [\Gamma, x : [A]_r]_{\mathsf{Labels}} \rhd \Gamma', x : [A]_r} \quad ([\text{GR}])$$

FIGURE 5.6: VLMini context grading

With the introduction of syntax sugar, the type rules of the contextual-let are integrated into the type inference rules of the abstract. Instead, abstract 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.

$$\frac{\begin{array}{c}\Sigma_1, \alpha :_{\exists} \mathsf{Type}; - \vdash p : \alpha \rhd \Gamma'; \Sigma_2; \theta \\ \Sigma_2; \Gamma, \Gamma' \vdash t \Rightarrow B; \Sigma_3; \theta'; \mathcal{C}\end{array}}{\Sigma_1; \Gamma \vdash \lambda p.t \Rightarrow \theta(\alpha \to B); \Sigma_3, \alpha :_{\exists} \mathsf{Type}; \theta \uplus \theta'; \mathcal{C}} \quad (\Rightarrow_{\text{ABS}})$$

The rule $\Rightarrow_{\text{ABS}}$ tentatively generates a unification type variable α, along with the binding pattern p of the lambda abstraction generates the typing context Γ'. Then the rule synthesizes a type B for the lambda body under Γ', and the resulting type of the lambda abstraction is $\theta(\alpha \to B)$, where the tentatively generated α is further refined by the substitution θ produced by the pattern type synthesis.

The rule $\Rightarrow_{\text{APP}}$ has multiple inferences as premises, unlike the other rules.

$$\frac{\begin{array}{c}\Sigma_1; \Gamma \vdash t_1 \Rightarrow A; \Sigma_2; \theta_1; \mathcal{C}_1 \quad \Sigma_2; \Gamma \vdash t_2 \Rightarrow A'; \Sigma_3; \theta_2; \mathcal{C}_2 \\ \Sigma_4 = \Sigma_3, \beta :_{\exists} \mathsf{Type} \quad \Sigma_4 \vdash A \sim A' \to \beta \rhd \theta_3 \\ \theta_4 = \theta_1 \uplus \theta_2 \uplus \theta_3\end{array}}{\Sigma_1; \Gamma \vdash t_1\, t_2 \Rightarrow \theta_4\beta; \Sigma_4; \theta_4; \mathcal{C}_1 \wedge \mathcal{C}_2} \quad (\Rightarrow_{\text{APP}})$$

The rule $\Rightarrow_{\text{APP}}$ synthesizes types A and A' for t_1 and t_2, respectively. It then generates a unification type variable β tentatively, and applies unification to $A' \to \beta$ and A (expected to have a function type), then generates a substitution θ_3 under the congruence between the synthesized types. The resulting type is $\theta_4\beta$, where the tentatively generated type variable β is refined by the substitution generated from the unification. The output constraint is generated by concatenating the constraints from the premise with $\wedge$.

The last rule $\Rightarrow_{\text{PR}}$ is the only rule that introduces constraints in the entire type inference algorithm.

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

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

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

FIGURE 5.7: VLMini constraints generation

This rule intuitively infers a consistent version resources in the environment. Since we implicitly allow for weakening here, we generate a constraint from an environment Γ' that contains only the free variables in t, produced by *context grading* defined as follows.

Definition 5.5.10 (Context grading)**.**

$$\Sigma \vdash [\Gamma]_{\mathsf{Labels}} \qquad \text{(context grading)}$$

The rules for context grading are listed in Figure 5.6. Context grading converts all assumptions in the input environment into versioned assumptions by assigning the empty set (= 1 of version resource semiring) for the linear variable assumption.

Finally, the rule generates a type variable $alpha$ to capture the resource propagated to $[t]$ and generates constraints from Γ' and α as defined below.

Definition 5.5.11 (Constraint generation)**.**

$$\Sigma \vdash \alpha \sqsubseteq_c [\Gamma] \rhd \mathcal{C} \qquad \text{(constraint generation)}$$

The rules for constraint generation are listed in Figure 5.7. This rule asserts that the input type variable α is a subset of all the resources of the versioned assumptions in the input environment $[\Gamma]$.

The simplest example is below, which is triggered by the type inference of $[f\ x]$.

Example 5.5.12 (Constraint Generation for a function application)**.**

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

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

5.6 Extend with Data Structures

Finally, we extend the rules so far to support data structures. Unfortunately, supporting data structures is not straightforward. This problem is motivated by the following program.

$$\begin{aligned} p &:: (\Box_r \mathsf{Int}, \Box_s \mathsf{Int}) \\ p &= ([1], [2]) \\ fst &:: \Box_{r'} (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int} \\ fst &= \lambda p.\ \mathsf{case}\ p\ \mathsf{of}\ [(x, y)] \to x \end{aligned}$$

VLMini algorithmic type inference $\boxed{\Sigma; \Gamma \vdash t \Rightarrow A; \Sigma'; \theta; \mathcal{C}}$

$$\frac{\Sigma_{i-1}; \Gamma \vdash t_i \Rightarrow A_i; \Sigma_i; \theta_i; \mathcal{C}_i}{\Sigma_0; \Gamma \vdash (t_1, .., t_n) \Rightarrow (A_1, .., A_n); \Sigma_n; \biguplus_{i=1}^{n} \theta_i; \bigwedge_{i=1}^{n} \mathcal{C}_i} \quad (\Rightarrow_{(,)})$$

$$\frac{\Sigma_{i-1}; \Gamma \vdash t_i \Rightarrow A; \Sigma_i; \theta_i; \mathcal{C}_i}{\Sigma_0; \Gamma \vdash [t_1, .., t_n] \Rightarrow [A]; \Sigma_n; \biguplus_{i=1}^{n} \theta_i; \bigwedge_{i=1}^{n} \mathcal{C}_i} \quad (\Rightarrow_{[,]})$$

$$\frac{\begin{array}{c}\Sigma_0; \Gamma \vdash t \Rightarrow A; \Sigma_1; \theta_0; C_0 \\ \Sigma_{i-1}; - \vdash p_i : A \rhd \Delta_i; \Sigma'_i; \theta'_i \quad \Sigma'_i; \Gamma, \Delta_i \vdash t_i \Rightarrow B; \Sigma_i; \theta_i; \mathcal{C}_i\end{array}}{\Sigma_0; \Gamma \vdash \textbf{case } t \textbf{ of } \overline{p_i \mapsto t_i} \Rightarrow (\biguplus_{i=0}^{n} \theta_i) B; \Sigma_n; \biguplus_{i=0}^{n} \theta_i; \bigwedge_{i=0}^{n} \mathcal{C}_i} \quad (\Rightarrow_{\textsc{case}})$$

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

$$\frac{\begin{array}{c}\Sigma'_{i-1} = \Sigma_{i-1}, \alpha_i : \mathsf{Type} \quad \Sigma'_{i-1}; - \vdash p_i : \alpha_i \rhd \Gamma_i; \Sigma_i; \theta_i \\ \Sigma_n \vdash K \alpha_1, .. \alpha_n \sim A \rhd \theta'\end{array}}{\Sigma_0; - \vdash C\, p_1 .. p_n : A \rhd \Gamma_i, .., \Gamma_n; \Sigma_n; \theta' \uplus \biguplus_{i=1}^{n} \theta_i} \quad (\textsc{pCon})$$

$$\frac{\begin{array}{c}\Sigma'_{i-1} = \Sigma_{i-1}, \alpha_i : \mathsf{Type} \quad \Sigma'_{i-1}; r \vdash p_i : \alpha_i \rhd \Gamma_i; \Sigma_i; \theta_i \\ \Sigma_n \vdash K \alpha_1, .. \alpha_n \sim A \rhd \theta' \quad \Sigma'_i \vdash r : \mathsf{Labels}\end{array}}{\Sigma_0; r \vdash C\, p_1 .. p_n : A \rhd \Gamma_i, .., \Gamma_n; \Sigma_n; \theta' \uplus \biguplus_{i=1}^{n} \theta_i} \quad ([\textsc{pCon}])$$

VLMini type unification $\boxed{\Sigma \vdash A \sim B \rhd \theta}$

$$\frac{\Sigma \vdash A_1 \sim B_1 \rhd \theta_1 \quad \Sigma \vdash \theta_{i-1} A_i \sim \theta_{i-1} B_i \rhd \theta_i}{\Sigma \vdash K\, A_1 .. A_n \sim K\, B_1 .. B_n \rhd \biguplus_{i=1}^{n} \theta_i} \quad (U_{Con})$$

FIGURE 5.8: Extension for algorithmic typing inference, pattern type synthesis, and type unification for data strutcures.

The example program is written in a hypothetical language, almost the same as VLMini but with type annotations and top-level bindings. The variable p binds a pair whose elements are versioned values that promote the value of Int, and the function fst is a function that takes a versioned value of a pair and returns its left element.

All values in VLMini are expected to be versioned values, so if we naively try to construct a tuple value, we get a value of a type like p. On the other hand, fst expects to receive a versioned value of a tuple as an argument. Therefore, a function application $fst\ p$ will be rejected by type inference.

Huges et al. [HVO21] discuss this problem in general coeffect systems and distributive laws for coeffectful data types. They show that it is possible to automatically derive *push* and *pull* for any type constructor K. For example, the push and pull functions for tuples are defined in the Granule language as follows.

Example 5.6.1 (push and pull for $(,)$ from [HVO21]).

```
1 push : ∀{a b: Type, r: Labels}. (a,b)[r] -> (a[r],b[r])
2 push [(x, y)] = ([x], [y])
```

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

Resources are annotated with brackets in Granule programs. The `push` function "pushes" the external resource `r` into the pair and distributes it to each subcomponent. The `pull` function "pulls" the inner resources of each subcomponent outside a data constructor. The inner elements resources `n` and `m` are pulled out as `n ⊓ m`. In VLMini, s is the version resource semiring, so `n ⊓ m` represents the common part of label sets `n` and `m`.

Following their approach, we extend the rules with pairs and lists. When a data structure value p is applied to a function f, $f\ p$ is implicitly interpreted to be $f\ (pull\ p)$. As a dual, a pattern match of a data structure value $\mathbf{case}\ p\ \mathbf{of}\ \overline{p_i \mapsto t_i}$ is interpreted to be $\mathbf{case}\ (push\ p)\ \mathbf{of}\ \overline{p_i \mapsto t_i}$.

Girard's translation is extended to data constructors and case expressions as follows.

Definition 5.6.2 (Girard's translation for case expressions and data constructors)**.**

$$\llbracket \mathbf{case}\ t\ \mathbf{of}\ \overline{p_i \mapsto t_i} \rrbracket \equiv \mathbf{case}\ [t]\ \mathbf{of}\ \overline{[p_i] \mapsto \llbracket t_i \rrbracket}$$
$$\llbracket C\ t_1 \mathinner{..} t_n \rrbracket \equiv C\ \llbracket t_1 \rrbracket \mathinner{..} \llbracket t_n \rrbracket$$

Similarly to lambda abstraction, the matching term t in the case expression is promoted as a versioned value, and the pattern p_i is translated into a promoted pattern. For data constructors, all subcomponents are translated recursively.

The extended rules are listed in Figure 5.8. Type inference rules for tuples and lists are straightforward. The rule infers types for each subcomponent, and the inferred types are used to synthesize the types of the data constructor. Like the other rules, the unification rule U_{CON} recursively compares the types of two data constructors to produce a substitution.

$$\frac{\Sigma_0; \Gamma \vdash t \Rightarrow A; \Sigma_1; \theta_0; C_0 \qquad \Sigma_{i-1}; - \vdash p_i : A \rhd \Delta_i; \Sigma_i'; \theta_i' \quad \Sigma_i'; \Gamma, \Delta_i \vdash t_i \Rightarrow B; \Sigma_i; \theta_i; \mathcal{C}_i}{\Sigma_0; \Gamma \vdash \mathbf{case}\ t\ \mathbf{of}\ \overline{p_i \mapsto t_i} \Rightarrow (\biguplus_{i=0}^{n} \theta_i) B; \Sigma_n; \biguplus_{i=0}^{n} \theta_i; \bigwedge_{i=0}^{n} \mathcal{C}_i} \quad (\Rightarrow_{\text{CASE}})$$

The inference rule for the case expression first infers the type of t, then use the resulting type A to generate the environment Δ_i under which each pattern p_i has the type A, and then infers a type of each body t_i under the context concatenated with Δ_i.

$$\frac{\Sigma_{i-1}' = \Sigma_{i-1}, \alpha_i : \mathsf{Type} \quad \Sigma_{i-1}'; - \vdash p_i : \alpha_i \rhd \Gamma_i; \Sigma_i; \theta_i \qquad \Sigma_n \vdash K\,\alpha_1, \mathinner{..} \alpha_n \sim A \rhd \theta'}{\Sigma_0; - \vdash C\ p_1 \mathinner{..} p_n : A \rhd \Gamma_i, .., \Gamma_n; \Sigma_n; \theta' \uplus \biguplus_{i=1}^{n} \theta_i} \quad (\text{pCon})$$

The pattern type synthesis rule $pCon$ is also similar to other pattern type synthesis rules. First, the rule chooses a type constructor K according to the input data constructor C and generates as many type variables α_i as the number of subpatterns. Next, it generates the environment Γ_i under the assumption that each subpattern p_i has type α_i. The resulting environment is the summation of Γ_i.

Chapter 6

Implementation and Case Study

Contents

6.1 Implementation **47**
6.1.1 Overview 47
6.1.2 Frontend 47
6.1.3 Ad-hoc Version Polymorphism via Duplication 50
6.1.4 Constraints Solving with z3 50
6.1.5 Code Generation 51
6.2 Case Study **52**
6.2.1 Settings 52
6.2.2 `Main` Program in VL Programming 54
6.3 Limitations of the Current VL **56**
6.3.1 Lack of Support for Structural Incompatibility 56
6.3.2 Inadequate Version Polymorphism 58

6.1 Implementation

6.1.1 Overview

We implement a VL language system with all the concepts introduced so far. The VL language system is implemented on the latest stable version (9.2.4) of GHC[1] as of October 2022, and the readers can find all its source code on Github[2]. The implementation use haskell-src-exts[3], a Haskell parser library with some extension (described below), and z3 [MB08][4] as a constraint solver. The implementation of the VL language system has the structure as shown in Figure 6.1. For clarity, the program elements are shown with thick borders for multi-module programs and thin borders for single-module programs of single-module programs.

6.1.2 Frontend

The surface language VL is mostly Haskell except for some syntax extension, which supports only minimal terms and literals, no type classes, and is monomorphic. We translate it into `Lang.Absyn`, the abstract syntax tree of the VL language. This phase implicitly includes name resolution and desugaring, although they are not shown in

[1] https://gitlab.haskell.org/ghc/ghc
[2] https://github.com/yudaitnb/vl
[3] https://github.com/yudaitnb/vl-haskell-src-exts
[4] https://github.com/Z3Prover/z3

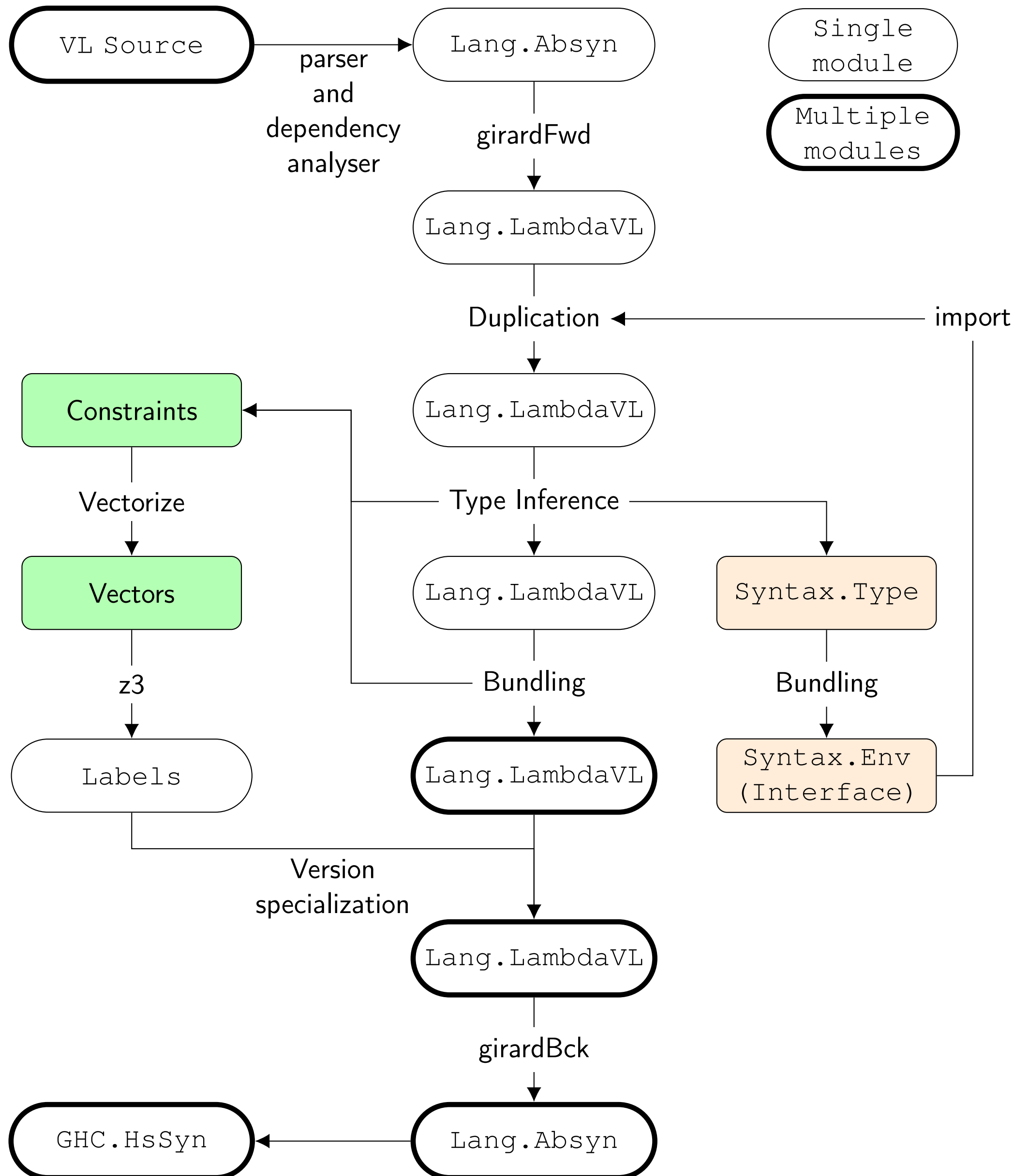


FIGURE 6.1: The pipeline of VL Language System

Figure 6.1. `Lang.Absyn` is an abstract syntax tree of the VL language whose definitions are provided from haskell-src-exts. The Haskell AST provided by haskell-src-exts follows Haskell2010 except for a few language extensions and is used in developing many programming tools of Haskell.

The surface language is extended from Haskell AST with a syntax extension `version L of t` and `unversion t` to convey user-defined version constraints to the type inference system. `version L of t` is useful when a programmer wants to impose version constraints on a particular term. This term takes a version constraint L and a term t and requires that all the subterms of `t` depend on that version. `L` is a list of module and version pairs that are expected to be compiled into the dependent versions $\mathcal{L}$ defined in chapter 5.

In contrast, `unversion t` is useful when a programmer wants to remove version constraints on a particular term. This term takes a term t and does not propagate any version constraints to the super terms of t. In the type checking of programs

```
-- version 1.0.0
module A where
import B
a = f x
```

```
-- version 2.0.0
module A where
import B
a = f (f x)
```

```
-- version 1.0.0
module B where
x = 1
f x = x * 1
g x = x + 1
```

```
-- version 2.0.0
module B where
x = 2
f x = x * 2
h x = x + 2
```

FIGURE 6.2: Module configuration with two mixed versions for the both module A and B.

that include t as a subterm, the type of `unversion t` is considered a versioned type with unconstrained resource variables; the programmer can use the value of `unversion t` at any program location.

`unversion` plays an important role in simultaneously using multiple versions. Since the type inference system requires version consistency for all data flows in the program by default, the programmer tells the type inference system by `unversion` that mixing multiple versions of the program is to the programmer's responsibility.

For example, suppose we have the programs in Figure 6.2. There are two modules A and B, and g is available only in version 1.0.0 of module B, and h is available only in version 2.0.0 of B. Under this module configuration, will the following program be well-typed?

Example 6.1.1 (A `main` function that depends on multiple versions (ill-typed))**.**

```
-- version 1.0.0
module Main where
import B
main = g 1 + h 1 -- type error
```

The `main` function sums the int values of the result of the program using g and h. It seems that `main`, which is an addition between int values, would be well-typed, but the current VL type inference takes the most conservative stand, and this program produces a version-incosistent error. This situation arises when a programmer adopts a new version of a dependent package for only part of a program, and the calculations of the existing and new programs are mixed.

To allow this program, insert `unversion` as follows.

Example 6.1.2 (A `main` function that depends on multiple versions (well-typed))**.**

```
-- version 1.0.0
module Main where
import B
main = g 1 + unversion (h 1)
```

Here, the function application `h 1`, which depends on the new version (2.0.0) of B, is annotated `unversion` by the programmer. The inferred type of `unversion (h 1)` is $\Box_r$Int, and its resource variable r has no constraints. As a result, the old and

new versions can be used simultaneously without giving up version consistency or type soundness of both the new and old programs.

6.1.3 Ad-hoc Version Polymorphism via Duplication

The VL language system duplicates external variables to assign individual versions to a single external variable. Duplication is necessary because the core language VLMini is monomorphic. 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 the VLMini type inference generates constraints by referring only to identifier names in the type environment. The goal is to assign a different version to each of identifier. This requires assigning a unique resource variable to each identifier in the preliminary step of constraint generation. Unfortunately, We cannot accomplish this without duplication because simple constraint generation results in the identical resource variable being assigned to all homonymous variables.

For example, given the version `2.0.0` of module `A` of Figure 6.2, duplication is performed as follows.

```
1 -- version 1.0.0
2 module Main where
3 import B
4 main = f_0 (f_1 x_0)
```

In this program, the symbols provided to the external module (`B`) are `f` and `x`. The duplication translation assigns unique symbols by 0-based indexing for `f` and `x`. Constraints containing resource variables of type `f` and the type of `f` that exist in the constraint and type environments are duplicated simulteneously with the same indexed name. This conversion is performed globally, and the unique symbol name will not be shared with any same-name symbols in other modules.

6.1.4 Constraints Solving with z3

Constraints collected by the type inference algorithm and bundling are resolved using z3-solver [MB08]. The VL language system uses the sbv [5] library as the binding of z3-solver.

The sbv library is a library for constructing first-order formulas in Haskell and providing an interactive interface to support multiple SMT solvers including ABC [6] CVC4, CVC5 [7] Boolector [8], MathSAT [9], Yices [10], DReal [11], and z3 [12]. The sbv library internally converts constraints into SMT-LIB scripts [BST+10] and supplies the resulting scripts to z3-solver. SMT-Lib script is a Lisp-like language developed as a common input/output language for SMT solvers. SMT-Lib scripts consist of commands for instructing a solver such as `(assert ...)` and `(check-sat)`, and *symbolic values* including numericals and string literals.

[5] https://hackage.haskell.org/package/sbv-9.0
[6] http://www.eecs.berkeley.edu/~alanmi/abc/
[7] https://cvc4.github.io/andhttps://cvc5.github.io
[8] http://fmv.jku.at/boolector/
[9] http://mathsat.fbk.eu/
[10] http://yices.csl.sri.com/
[11] http://dreal.github.io/
[12] http://github.com/Z3Prover/z3/wiki

We represent a dependency constraint as a vector of symbolic int values. The vector length is equal to the number of external modules, and the elements are unique int values corresponding to the version number of each module. Constraint resolution identifies the expected vectors for symbolic variables. This solution vector corresponds to the label on which some identifier in the VL should depend. If more than one labels satisfy the given constraints, select a newer one by default.

The VL language system transforms constraints into symbolic vectors by a transformation called vectorization as a pre-/postprocess for constraint resolution. We define vectorization of version labels as follows.

Definition 6.1.3 (Vectorize version labels)**.** Given a conversion var from a type variable to a symbolic variable, a conversion id_{mod} from a module to a vector index, and a conversion id_{ver} from a version number to the corresponding symbolic int value, the function vectorize is defined recursively as follows.

$$
\begin{aligned}
\mathrm{vectorize}(\top) &= \texttt{sTrue} \\
\mathrm{vectorize}(\alpha \preceq \langle\langle \overline{M_i \mapsto V_i} \rangle\rangle) &= \bigwedge_{i=1}^{n} \mathrm{var}(\alpha).\mathrm{id}_{mod}(M_i)\ \texttt{.==}\ \mathrm{id}_{ver}(V_i) \\
\mathrm{vectorize}(\alpha \preceq \alpha') &= \bigwedge_{i=1}^{n} \mathrm{var}(\alpha).i\ \texttt{.==}\ \mathrm{var}(\alpha').i \\
\mathrm{vectorize}(\mathcal{C}_1 \wedge \mathcal{C}_2) &= \mathrm{vectorize}(\mathcal{C}_1)\ \texttt{.\&\&}\ \mathrm{vectorize}(\mathcal{C}_2) \\
\mathrm{vectorize}(\mathcal{C}_1 \wedge \mathcal{C}_2) &= \mathrm{vectorize}(\mathcal{C}_1)\ \texttt{.||}\ \mathrm{vectorize}(\mathcal{C}_2)
\end{aligned}
$$

where `sTrue`, `.&&`, `.||`, and `.==` given by sbv library are the symbolic versions of True, &&, `||`, and == of a standard Haskell value/function.

6.1.5 Code Generation

The VL language system compiles VLMini programs back to Haskell ASTs during the code generation phase, using the labels from constraint resolution. The code generation consists of version specializations and Girard's translation. Since the VL type system keeps track of which symbols type variables were generated, the label can be used to determine which version of implementation the program refers to.

Example 6.1.4 (Code generation)**.**

```
1 -- Input VL program
2 module Main where
3 import B
4 main = g a + unversion (h 1)
```

```
1 -- Generated Haskell AST (prettyprint)
2 module Main where
3 main = (+)
4   ((let g = \x -> (*) x 1 in g)
5     (let a = (let f = x -> (+) x 1 in f)
6                  ((let f = \x -> (+) x 1 in f)
7                     (let y = 1 in y))
8      in a))
9     ((let h = \x -> (*) x 2 in h) 1)
```

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

TABLE 6.1: Availability of functions in hmatrix before and afte tha update in version 0.16.

```
10 -- g@[A=2.0.0,B=1.0.0] -->* \x -> (*) x 1
11 -- a@[A=2.0.0,B=1.0.0] -->* (f f x)@[A=2.0.0,B=1.0.0]
12 --                     -->* (x -> (+) x 1) (x -> (+) x 1)
    (1)
13 -- h@[B=2.0.0] -->* \x -> (*) x 2
```

The code generation takes the top VL program and generates the bottom program. Here, since `unversion` is inserted, it is possible to choose different labels for `g a` and `h 1`. The code generation expands the program with `A` version `2.0.0` and `B` version `1.0.0` for `g a` and `B` version `2.0.0` for `h a`.

6.2 Case Study

In this section, we will confirm that the two main benefits of programming with versions are achieved in VL programming. For this purpose, we implemented two packages in VL: the package `List`, a partial port of `Data.List` in the Haskell standard library, [13] and `Matrix`, a simple linear algebra library that emulates hmatrix, [14] one of the major Haskell library for numeric linear algebra and matrix computations.

Due to a limitation of the VL language, `Matrix` differs from hmatrix in several ways. hmaterix represents a vector or matrix of element a as type `Vector a` or `Matrix a`. In contrast, the VL language does not provide user-defined data structures, so vectors are represented as `List Int`, and matrices are rewritten as `List (List Int)`. Also, since the VL language does not provide error processing, we ignore cases that do not satisfy pre-conditions, such as `head []`, and implicitly assume that all arguments satisfy the appropriate pre-conditions. Furthermore, since our goal is to observe hmatrix incompatibilities in the VL language, we have yet to implement all functions.

6.2.1 Settings

Table 6.1 shows the changes in version 0.16 of hmatrix. [15] Before version 0.15, the hmatrix library provided a function named `join` to combine multiple vectors. However, in the update from version 0.15 to 0.16, `join` was removed and reimplemented as `vjoin`. Also, several new functions were added, such as `udot`, `sortVector`, and `roundVector`.

We first implement `Matrix` in VL that simulates the backward incompatible changes of hmatrix in the version 0.16 update and a program that works with two contradictory versions of `Matrix` at the same time.

`Matrix` Package Update

[13] https://hackage.haskell.org/package/base (January 11, 2023)
[14] http://dis.um.es/~alberto/hmatrix/hmatrix.html (January 11, 2023)
[15] https://github.com/haskell-numerics/hmatrix/blob/master/packages/base/CHANGELOG

```
1  -- Matrix version 1.0.0
2  module Matrix where
3  import List
4
5  join xs ys = case xs of
6    []    -> ys
7    x:xs -> x : (join xs ys)
8
9  determinant mx = sum
10       (map
11           (\xs -> (product (pick 1 xs mx)) * (sign xs))
12           (mkPerm (length mx)))
13
14 pick c xs mx = case xs of
15   [] -> []
16   _  -> (index (head xs) (index c mx))
17          : (pick (c+1) (tail xs) mx)
18
19 sign xs = if (mod2 (inversion xs)) > 0 then -1 else 1
20
21 inversion xs = case xs of
22   x:[] -> 0
23   x:xs -> sum (map (\y -> if x > y then 1 else 0) xs)
24           + inversion xs
25
26 index c xs = last (take c xs)
27
28 mkPerm n = permutations (mkLst n)
29
30 mkLst n = reverse (mkLst' n)
31 mkLst' n = case n of
32   0 -> []
33   _ -> n : mkLst' (n-1)
```

FIGURE 6.3: The snipet of module `Matrix` version 1.0.0.

Figure 6.3 shows a snippet of version 1.0.0 of the `Matrix` package. Version 1.0.0 provides a function `join`; since vectors are values of a type `List Int`, `join` is equivalent to `(++)` for `[Int]` in Haskell standard library. It also provides another function, called `determinant`, which computes determinants. The function `determinant` is implemented using several auxiliary functions and the list operations provided by `List` and an algorithm based on the common definition of determinants.

Figure 6.4 shows a snippet of version 2.0.0 of the `Matrix` package. In the update of version 2.0.0, the `join` function is removed and replaced by `vjoin`, which has the same definition of `join`. Other changes include a new function, `sortVector`, which computes the sorted vector of an argument and is implemented using some functions defined in `List`. Also, the function `determinant` is still available in this version.

```
1  -- Matrix version 2.0.0
2  module Matrix where
3  import List
4
5  determinant = -- (same as version 1)
6
7  -- join is replaced by vjoin.
8  vjoin = case xs of
9    []    -> ys
10   x:xs -> x : (vjoin xs ys)
11
12 -- newly added function.
13 sortVector xs = case xs of
14   []  -> []
15   [x] -> [x]
16   xs  -> let r = bubble xs
17          in vjoin (sortVector (init r)) [last r]
18
19 bubble xs = case xs of
20   []  -> []
21   [x] -> [x]
22   (x:y:xs) -> if x < y
23               then x : bubble (y:xs)
24               else y : bubble (x:xs)
```

FIGURE 6.4: The snipet of module `Matrix` version 2.0.0.

`Main` Program with Two Versions

Figure 6.5 shows a snippet of the `Main` module. The `Main` module simulates a work-in-progress situation where the developer is updating `Matrix` from version 1.0.0 to version 2.0.0, and the function `main` uses functions from both versions of `Matrix` together. The `main` function uses the `join` function, available only in version 1.0.0, and the `sortVector` function, available only in version 2.0.0. Such a program has direct dependencies on both versions of `Matrix` at the same time, no matter what existing package manager the developer uses. Therefore, it will be impossible to build this program in an existing language unless the developer gives up using either `join` or `sortVector`.

6.2.2 `Main` Program in VL Programming

On the other hand, the VL language can provide a way to control and accept this program in two stages, described below.

Detecting Inconsistent Version

First, we are notified of a version consistency error if we compile the `Main` module in the VL system without any changes. This is because even though all programs with `join` as a subterm need to be consistent with version 1.0.0, they also require consistency to version 2.0.0 from `tranpose`, so it is impossible to determine on which version of `Matrix` the `main` function should depend on. As explained in section

```
1  module Main where
2
3  import Matrix
4  import List
5
6  main = let vec = [2, 1]
7             sorted = sortVector vec
8             m22 = join -- [[1,2],[2,1]]
9                     (singleton sorted)
10                    (singleton vec)
11        in determinant m22 -- error: version inconsistent
```

FIGURE 6.5: `Main` module *before* rewriting.

```
1  module Main where
2
3  import Matrix
4  import List
5
6  main = let vec = [2, 1]
7             sorted = unversion (sortVector vec)
8             m22 = join -- [[1,2],[2,1]]
9                     (singleton sorted)
10                    (singleton vec)
11        in determinant m22 -- -->* -3 (= 1 * 1 - 2 * 2)
```

FIGURE 6.6: `Main` module *after* rewriting.

2.1.2, combining different versions of dependent packages can cause unexpected bugs and is generally incorrect. The version consistency error output by the VL language system prevents the use of such incorrect version combinations of programs.

Simultaneous Use of Multiple Versions

In this case, the return values of both join and sortVector are vectors and matrices. We know that it is acceptable to use each version of join and sortVector simultaneously. Therefore, we used `unversion t` for the Main program to accept the simultaneous use of the two versions.

Figure 6.6 shows a snippet of the Main module after rewriting: `sortVector xs`, the newer function, is replaced by `unversion (sortVector xs)`. Assuming that we do not use a program that depends on a specific version in other parts of the program, we successfully compile this program.

Figure 6.7 is a prettyprint of the Haskell code output by the VL system. Here, some of the code in the output has been omitted because of the very large code size. In lines n and m, `join` and `sortVector` are each dispatched to a specific version of the implementation, so we have successfully used the functions provided by multiple versions of `Matrix` simultaneously in the `Main` module.

```
1  module Main where
2
3  main = ...
4      -- sortVector
5      (let sortVector = \xs ->
6          case xs of
7              []  -> []
8              [x] -> [x]
9              xs  -> (\r -> (let vjoin = ...  in vjoin)
10                                (sortVector
11                                  ((let init = ... in init) r))
12                                [(let last = ... in last) r])
13                      ((let bubble = ... in bubble) xs)
14     in sortVector)
15   ...
16     -- join
17     (let join = \xs -> \ys ->
18         case xs of
19             [] -> ys
20             x : xs -> (:) x (join xs ys)
21     in join)
22   ...
```

FIGURE 6.7: The `main` function with all definitions dispatched (`join` and `sortVector` excerpted)

version	`gdk_screen_get_n_monitors`	`gdk_display_get_n_monitors`
< 3.22	available	inavailable
≥ 3.22	deprecated	available

TABLE 6.2: Structural change: availability of functions in GDK 3

6.3 Limitations of the Current VL

This section discusses the limitations of the current VL language and possible solutions.

6.3.1 Lack of Support for Structural Incompatibility

One of the apparent problems with the current VL system is that it does not support *type incompatibilities*, a key element of structural incompatibilities. We will first analyze the types of incompatibilities and then discuss ways to extend the current VL system.

Types of Incompatibilities

Incompatibilities between old and new versions of a package caused by updates can be broadly classified into two categories *structural incompatibilities* and *behavioral incompatibilities*.

version	set	setExact
< 19	exact	inavailable
≥ 19	inexact	exact

TABLE 6.3: Behavioral change: reliability of an alarm time in Android API

Structural Incompatibilities A structural incompatibility occurs when multiple versions of a package provide different set of definitions including function names and data structures. Structural incompatibilities are caused by adding and removing definitions, internal changes to data structures, and renaming. Table 6.2 shows an example of structural incompatibility in GIMP Drawing Kit (GDK). GDK is a C library for creating graphical user interfaces and is used by many projects, including GNOME.

If the deprecated functions are not available, version 3.22 is structurally incompatible with version 3.20 because the former lacks `gdk_screen_get_n_monitors` that is available in the latter. GDK versions before 3.22 provide `gdk_screen_get_n_monitors` that tells the number of connected physical monitors. However, versions 3.22 later provide the same functionality function `gdk_display_get_n_monitors` and deprecate `gdk_screen_get_n_monitors`. When we upgrade GDK to version 3.22 and build software that uses this function without modifying anything, the build system will give us an undefined reference error. With a static type check, the programmer will be informed of the incompatibility problem as a compilation error.

Behavioral Incompatibilities A behavioral incompatibility is a situation where multiple versions of a package provide the same definition but differ in their behavior. Code changes may also cause behavioral incompatibilities that include additions, removals, and changes in side effects, even if there is no change in name or type. Table 6.3 shows an example of behavioral incompatibility in the Android Platform API (henceforth Android API). The Android API is the standard library written mostly in Java, and its version synchronizes with Android OS.

Before version 19[16], the Android API provided the `set` method in the `AlarmManager` class that schedules an alarm at a specified time. However, since version 19, the Android API has changed its behavior for power management. Despite having the same name and type definitions, `set` no longer guarantees accurate alarm delivery. For developers who require accurate delivery, the method `setExact` is provided instead.

Extending VL to Support Structural Incompatibility

The current VL language system 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. In λ_{VL}, definitions of the same type can be combined as a versioned record (even if the programmer has given them different names), while terms with different types cannot be in a versioned record. Also, the VL language system will stop compilation if it finds a definition with the same name but a different type in more than one version of the same module.

[16]The Android API uses *levels* instead of versions as identifiers for API revisions, but we will call them versions for consistency.

```
1 module Main where
2
3 import A
4 import B
5
6 main =
7   let x = unversion (g a)
8       y = unversion (h a)
9   in x + y
10 -- well-typed
```

```
1 module Main where
2
3 import A
4 import B
5
6 main =
7   let a' = a
8       x = unversion (g a')
9       y = unversion (h a')
10  in x + y -- ill-typed
```

FIGURE 6.8: The well-typed (left) and ill-typed (right) programs illustrate the difference between the treatment of local and external variables.

However, more feature is needed to deal with broader incompatibilities. Raemaekers et al. conducted a comprehensive analysis of the seven-year history of library releases in Maven Central. They found that about one-third of all releases introduced at least one structural incompatibility change. The top three causes of structural incompatibilities were class, method, or field deletions, and the remaining seven were type changes. [RvV17] It seems an important step to extend the language system to support a wider variety of type incompatibilities and to help programmers improve dependencies.

The current λ_{VL} design is motivated by the basic design that "the type of a versioned record is similar to the type $\Box_r A$, a type with a resource in coeffect calculli." In the current λ_{VL}, the type of versioned record $\{\overline{l = t} \,|\, l_k\}$ is $\Box_r A (r = \{\bar{l}\})$, and no difference exists between a type of versioned records and promotions of a term of type A. This design has the advantage that versioned records and promotions could be treated in a unified manner, making it easier to formalize dynamic and static semantics.

One useful idea to address this problem is to decouple version inference from the type inference of coeffect calculus and implement a type system that guarantees version consistency on top of the polymorphic record calculus. [Oho95] The idea stems from the fact that the type $\Box_{\{l_1,l_2\}} A$ is structurally similar to the variant type $\langle l_1 : A, l_2 : A\rangle$ of $\Lambda^{\forall,\bullet}$. It is no longer required with variants that types be the same, so terms with different types can be stored as a single value, such as $\langle l_1 = true, l_2 : 100\rangle : \langle l_1 : Bool, l_2 : Int\rangle$. Although the current version inference is uniformly defined with type inference, we believe it is possible to separate its algorithm and implement it in another calculus because the type and version inference in the type system of VLMini is orthogonal to each other. In the current VL system, constraints generated from type inference and constraints generated from version inference are completely independent, and all constraints passed to z3 are version constraints.

6.3.2 Inadequate Version Polymorphism

As we attempt to scale VL programming to a realistically sized development, incomplete version polymorphism via duplication described in section 6.1.3 becomes an obstacle. The following examples are VL programs that depend on modules A and B in Figure 6.2. Both use functions g and h provided by module B and the variable a provided by module A.

```
1 module Main where
2
3 import A
4 import B
5
6 main =
7   let xx = (unversion (g (version {A=2.0.0} of a)))
8       yy = (unversion (h (version {A=1.0.0} of a)))
9   in xx + yy -- well-typed
```

```
1 module Main where
2
3 import A
4 import B
5
6 main =
7   let xx = (unversion (g (version {A=2.0.0} of a)))
8       yy = (unversion (h (version {A=2.0.0} of a)))
9   in xx + yy -- ill-typed
```

FIGURE 6.9: The well-typed (top) and ill-typed (bottom) programs indicate that only one version of each module's top-level symbol is allowed.

The first problem is the difference between the treatment of local and external variables. The two programs in Figure 6.8 illustrate this problem. The only difference between the two programs is that the program on the left is written to apply functions without local variables, whereas the program on the right binds `a` to `a'`. However, the left one succeeds, while the right fails in version inference.

The reason for this problem is the type inference system assigns the only resource variable to the local variable `a'`. The applications `g a'` and `h a'` generate constraints that require `a'` to depend on versions 1 and 2 of module `B`, respectively, but there is no version label that satisfies both. All external variables are given unique names by duplication, but local variables are not. Therefore, the type inference results differ in the two programs in Figure 6.8.

The second problem is that there is only one version on which each version of the top-level symbol can depend. The programs in Figure 6.9 illustrate this problem.

The top program requires `a` of `A` versions 2.0.0 and 1.0.0 as arguments of `g` and `h`, respectively, whereas the bottom program requires `A` version 2.0.0 for both arguments. The result of type inference is that the top program has a label that satisfies this requirement, while the bottom program does not.

The cause of this problem is that the inference system produces a variable dependency on one of the versions of the original top-level symbol. The current VL type inference creates a variable dependency on either version of the source when creating a resource variable with the same constraints as the source of the duplication. In this example, the two copies of `a`, `a_0` (for `g (version A=2.0.0 of a)`) and `a_1` (for `h (version A=2.0.0 of a)`), are expected to select either version of `a`. Furthermore, the generated constraints constrain the selected version of `a`. In line 7, `g` requires `a_0` to have a dependency on version 1.0.0 of `B`, and version A=2.0.0 of `a_0` requires that `a_0` is equal to the label selected for version 2.0.0 of a, resulting

in version 2.0.0 of `a`. Similarly, line 8 generates a constraint that requires that the label for version 2.0.0 of `a` must contain version 1.0.0 of `B`, so no label satisfies both simultaneously.

It is necessary to introduce full-resource polymorphism in the core calculus instead of duplication to solve this irrational problem,. The idea is to store external variables and constraints that behave in a version polymorphic manner in a top-level definition environment and instantiate them with a new resource variable for each symbol occurrence. This kind of resource polymorphism is similar to that already implemented in the Gr language [OLE19]. 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.

Chapter 7

Related Work

Contents

7.1 Software Product Lines 61
7.1.1 Delta-Oriented Programming 61
7.1.2 Variational Programming 62
7.2 Adaptation Techniques 62
7.3 Container 62
7.4 Monorepository 63
7.5 Coeffect Calculus 63

7.1 Software Product Lines

Software Product Lines (SPLs) [Par76; PBL05] are methods for creating a collection of similar software from a shared set of programs. Since program updates can be considered a kind of program extension, some programming techniques in SPLs [Bos01; Pre97; Sch+10] are closely related to this work.

7.1.1 Delta-Oriented Programming

Delta-oriented programming (DOP) [SD10; Sch+10; SBD11] provides a mechanism called *delta-modules* for modularizing program modifications. The delta-module language allows the addition and overriding of classes and methods and their removal. Each delta module contains the application conditions for modifications, and delta modules can be combined to form complex constraints on the features of a product.

The DOP approach is complementary to our research. For the implementation of version analysis by type checking in the future, it is essential to have a packaging system with expression-level dependency information. Given that the evolution of packages today is basically linear over versions, it is possible to develop a package system modularized by program diffs. Patrick Lam *et al.* [LDP20] point out that the lack of tool support for package changes requires developers to pay a great deal of attention to compatibility and discuss the implications of calculating compatibility information in the context of program analysis. For the implementation of version analysis by type checking in the future, it is essential to have a packaging system with expression-level dependency information. Such tools will lead to the development of a novel package system with more detailed compatibility information.

7.1.2 Variational Programming

Variational Programming [Wal13; EW11; Wal+14] is a language paradigm with syntactic support for data variation. For example, the Variational Programming Calculus [CEW16] (VPC) represents differences as binary choices called *dimensions*, such as `A<1,true>`, which can be passed as function arguments. In addition, we can extract and manipulate choices. For example, a program that applies the identity function to either 1 or `true` and extracts left one can be written as `sel A.L (λx.x A<1, true>)`.

At first glance, the main part of the λ_{VL} can be encoded by VPC, but this is impossible. The key difference is semantics: VPC doesn't have a mechanism to deal with computations that lack definitions like versioned records in λ_{VL}. In λ_{VL}, functions and arguments with different dimensions are applied according to a smaller dimension: the calculation is allowed by the type system only if they have common versions, i.e., **let** $[x] = \{v1 = 1, v2 = 2\}$ **in let** $[y] = \{v1 = 1, v2 = 2, v3 = 3\}$ **in** $[x + y]$ is well-typed and interpreted as $\{v1 = 1 + 1, v2 = 2 + 2\}$ defined only in versions $v1$ and $v2$. In contrast, in VPC, such computations are interpreted according to a larger dimension: functions/values with smaller dimensions are interpreted in a distributed manner, i.e. `A1<1, 2> + A1<1, A2<2, 3> >` is interpreted as `A1<1 + 1, A2<2 + 2, 3 + 2> >` that have three variations. Since our primary interest is in disallowing programs to run in versions for which no definition exists, the semantics of VPC don't meet our motivations.

7.2 Adaptation Techniques

Adaptation techniques help client code connect to a version of a library incompatible with the older one. The simplest approach is to compare the old and new versions of a source code and find substitution patterns. Other approaches generate replacement rules based on structural similarities [CWC14; Wu11] and extract API replacement patterns from a code base with been migrated. [SJM08]

Another approach lets the library maintainer generate replacement rules. Some techniques [DJ06; HD05] require the library maintainer to record refactorings made to the source code and generate refactoring scripts for providing it to library users. Another technique [CN96] requires the library maintainer provides annotations that describe how to update client code. These techniques are reported to provide correct code recommendations on average in only less than 20% of cases. [CW12]

7.3 Container

Docker [Mer14] makes it clear that it is used as a solution to dependency hell. Docker project aims to package an application per environment with all its dependencies and to run smoothly in further development, testing, and working environments. Modern applications are often a combination of existing applications. However, it has been difficult to develop all related applications in a single environment because each application has separate dependencies that may have conflicting dependency paths. Docker is a powerful tool in that case. Docker makes it possible to isolate the environment as a container for each application, making it possible for different versions of libraries to coexist on a single physical machine. In addition, since all dependencies are bundled within the Docker container, no dependency libraries will be missing, even if the Docker container is ported to a different physical machine.

This feature makes it extremely easy to migrate the development environment. On the other hand, however, the Docker project is not interested in the dependencies inside the container. That is, if there are conflicting dependency paths in the dependency graph of an application, dependency hell will still occur.

7.4 Monorepository

Monorepository is a method of versioning two or more logical libraries that normally reside in different repositories in a version control system but are versioned in a single repository. By managing in a single repository, it is possible to have a single version control for all the libraries that it encompasses. The monorepository approach has the advantage that updates across multiple libraries can be managed with a single commit, and the dependencies between libraries can be centrally understood, making it easier to test and find bugs across libraries.

On the other hand, it is pointed out that the system's modularity is reduced, resulting in a huge number of commits that are essentially irrelevant to some functions. Furthermore, the performance of the version control system is reduced due to the explosive number of managed files.

The monorepository method is particularly useful for the large-scale development, and Google [PL16] and Facebook [Dur14] have publicly announced that they manage their libraries in a monorepository.

However, the monorepository method is only a technique to help understand dependencies, not a technology to solve the dependency hell. The legitimacy of dependencies among many libraries in a monorepository must be verified by other methods, and dependency hell still exists for libraries outside the monorepository's control.

7.5 Coeffect Calculus

Coeffect calculus simultaneously emerged from several contexts in the literature [Bru+14; GS14; POM14] since the 2010s. The common denominator of these formalizations is that they annotate the assumptions in the typing context with usage information derived from a semiring. Recent studies use coeffect calculus to track bounded reuse [Bru+14; POM14], deconstructor use [POM13], security levels [OLE19], and scheduling constraints in hardware synthesis [GS14].

Granule [OLE19] is a fully-fledged functional language focused on coeffect calculus. Its core Gr demonstrates a good combination of coeffect calculus and standard language features, *i.e.*, data types, pattern matching, and recursion. Since the core subset of Granule, GrMini, has almost the same structure as λ_{VL}, we expect that most of the language extensions from GrMini to Gr can be applied to λ_{VL}. On the other hand, the difference is that the only thing that affects resources in these languages is the availability of each function, whereas λ_{VL} provides a means to manipulate resources such as version records and extractions.

Chapter 8

Conclusion

Contents

8.1 Towards a Per-expression Dependency Analysis 65
8.2 Conclusion 65

8.1 Towards a Per-expression Dependency Analysis

The current λ_{VL} type system cannot express the range of compatibility that cargo and npm do in packages. For example, even if the versions 1.0.0 and 1.0.1 of package P are compatible and we apply the value of type $\Box_{1.0.1}P.T$ to the function that expects an argument of type $\Box_{1.0.0}P.T$, this function application is rejected. This is because there is no relationship between types annotated with version resource 1.0.0 and 1.0.1 as the current type system focuses on preventing computation on versions for which a definition does not exist. Considering that many updates change only a small part of the package code and remain backward compatible for the most part, the current type system is too strict.

The next step of our research is to track the range of compatibility in the type system. Incorporating semantic versioning into types is a promising idea. Semantic versioning can also be seen as a compatibility contract from the package provider to the package users. For example, a 1.0.1 package is guaranteed backward compatible with 1.0.0. From the point of view of Liskov's substitution principle [LW94], emulating the semantic versioning strategy at the expression level, it is possible to regard $\Box_{1.0.1}A$ as a subtype of $\Box_{1.0.0}A$. Such a type system paves the way for type-safe casts between objects derived from different-version packages.

8.2 Conclusion

Even though dependency hell has been considered a problem for many years, it is still difficult for most programmers to solve. Most recent build tools use compatibility maintenance strategies like semantic versioning, but name mangling may burden the development community in programming languages with sophisticated type systems.

Our research aims to enable programmers to freely combine and control programs of different versions in a single code. This research brings versions, which used to be simply identifiers of packages, into a programming language, and provides a new perspective on handling multiple versions of programs. As a first step toward our goal, we discussed type safety, a compilation method, the implementation, and a case study with programs with multiple versions in VL. We hope this

research will stimulate a discussion in the research community on compatibility in the context of program analysis.

Bibliography

[Aba+20] Pietro Abate et al. "Dependency Solving Is Still Hard, but We Are Getting Better at It". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2020, pp. 547–551. DOI: 10.1109/SANER48275.2020.9054837.

[All18] OSGi Alliance. *Osgi core release 7 specification*. 2018. URL: https://www.%20osgi.org/release-7-1/.

[Art+12] Cyrille Artho et al. "Why Do Software Packages Conflict?" In: *IEEE International Working Conference on Mining Software Repositories* (June 2012), pp. 141–150. DOI: 10.1109/MSR.2012.6224274.

[Bav+15] Gabriele Bavota et al. "How the Apache Community Upgrades Dependencies: An Evolutionary Study". In: *Empirical Software Engineering* 20.5 (Oct. 2015), pp. 1275–1317. ISSN: 1382-3256. DOI: 10.1007/s10664-014-9325-9. URL: https://doi.org/10.1007/s10664-014-9325-9.

[Bel05] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41.

[Beu+99] A. Beugnard et al. "Making components contract aware". In: *Computer* 32.7 (1999), pp. 38–45. DOI: 10.1109/2.774917.

[Bog+16] Christopher Bogart et al. "How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 109–120. ISBN: 9781450342186. DOI: 10.1145/2950290.2950325. URL: https://doi.org/10.1145/2950290.2950325.

[Bos01] Jan Bosch. "Software Product Lines: Organizational Alternatives". In: *Proceedings of the 23rd International Conference on Software Engineering*. ICSE '01. Toronto, Ontario, Canada: IEEE Computer Society, 2001, pp. 91–100. ISBN: 0769510507.

[Bru+14] Aloïs Brunel et al. "A Core Quantitative Coeffect Calculus". In: *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 351–370. ISBN: 9783642548321. DOI: 10.1007/978-3-642-54833-8_19. URL: https://doi.org/10.1007/978-3-642-54833-8_19.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The smt-lib standard: Version 2.0". In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*. Vol. 13. 2010, p. 14.

[CEW16] Sheng Chen, Martin Erwig, and Eric Walkingshaw. "A Calculus for Variational Programming". In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Ed. by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 6:1–6:28. ISBN: 978-3-95977-014-9. DOI: 10.4230/LIPIcs.ECOOP.2016.6. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6100.

[CN96] Chow and Notkin. "Semi-automatic update of applications in response to library changes". In: *1996 Proceedings of International Conference on Software Maintenance*. New York, USA: IEEE, 1996, pp. 359–368. DOI: 10.1109/ICSM.1996.565039.

[Coa19] Stephen Coakley. *How Rust Solved Dependency Hell*. https://stephencoakley.com/2019/04/24/how-rust-solved-dependency-hell. Apr. 2019.

[CW12] Bradley E. Cossette and Robert J. Walker. "Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. Cary, North Carolina: Association for Computing Machinery, 2012. ISBN: 9781450316149. DOI: 10.1145/2393596.2393661. URL: https://doi.org/10.1145/2393596.2393661.

[CWC14] Bradley Cossette, Robert Walker, and Rylan Cottrell. *Using Structural Generalization to Discover Replacement Functionality for API Evolution*. 2014. DOI: 10.11575/PRISM/10182. URL: https://prism.ucalgary.ca/handle/1880/49996.

[Die+19a] Jens Dietrich et al. "Dependency Versioning in the Wild". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 349–359. DOI: 10.1109/MSR.2019.00061.

[Die+19b] Jens Dietrich et al. "Dependency Versioning in the Wild". In: *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 349–359. DOI: 10.1109/MSR.2019.00061. URL: https://doi.org/10.1109/MSR.2019.00061.

[DJ06] Danny Dig and Ralph Johnson. "How do APIs evolve? A story of refactoring". In: *Journal of Software Maintenance and Evolution: Research and Practice* 18.2 (2006), pp. 83–107. DOI: https://doi.org/10.1002/smr.328. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.328. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.328.

[DJB14] J. Dietrich, K. Jezek, and P. Brada. "Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades". In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 2014, pp. 64–73. DOI: 10.1109/CSMR-WCRE.2014.6747226.

[DK13] Jana Dunfield and Neelakantan R. Krishnaswami. "Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism". In: *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 429–442. ISSN: 0362-1340. DOI: 10.1145/2544174.2500582. URL: https://doi.org/10.1145/2544174.2500582.

[DK19] Jana Dunfield and Neelakantan R. Krishnaswami. "Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290322. URL: https://doi.org/10.1145/3290322.

[DMG19] Alexandre Decan, Tom Mens, and Philippe Grosjean. "An empirical comparison of dependency network evolution in seven software packaging ecosystems". In: *Empirical Software Engineering* 24.1 (2019), pp. 381–416. ISSN: 1573-7616. DOI: 10.1007/s10664-017-9589-y. URL: https://doi.org/10.1007/s10664-017-9589-y.

[Dur14] Durham Goode. *Facebook Engineering: Scaling Mercurial at Facebook*. https://code.fb.com/core-data/scaling-mercurial-at-facebook/. Jan. 2014.

[EW11] Martin Erwig and Eric Walkingshaw. "The Choice Calculus: A Representation for Software Variation". In: *ACM Trans. Softw. Eng. Methodol.* 21.1 (Dec. 2011). ISSN: 1049-331X. DOI: 10.1145/2063239.2063245. URL: https://doi.org/10.1145/2063239.2063245.

[Gir87] Jean-Yves Girard. "Linear Logic". In: *Theor. Comput. Sci.* 50.1 (Jan. 1987), pp. 1–102. ISSN: 0304-3975. DOI: 10.1016/0304-3975(87)90045-4. URL: https://doi.org/10.1016/0304-3975(87)90045-4.

[GS14] Dan R. Ghica and Alex I. Smith. "Bounded Linear Types in a Resource Semiring". In: *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 331–350. ISBN: 9783642548321. DOI: 10.1007/978-3-642-54833-8_18. URL: https://doi.org/10.1007/978-3-642-54833-8_18.

[HD05] J. Henkel and A. Diwan. "CatchUp! Capturing and replaying refactorings to support API evolution". In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* New York, USA: IEEE, 2005, pp. 274–283. DOI: 10.1109/ICSE.2005.1553570.

[HVO21] Jack Hughes, Michael Vollmer, and Dominic Orchard. "Deriving Distributive Laws for Graded Linear Types". In: Proceedings Second Joint International Workshop on *Linearity & Trends in Linear Logic and Applications*, Online, 29-30 June 2020. Ed. by Ugo Dal Lago and Valeria de Paiva. Vol. 353. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2021, pp. 109–131. DOI: 10.4204/EPTCS.353.6.

[LDP20] Patrick Lam, Jens Dietrich, and David J. Pearce. "Putting the Semantics into Semantic Versioning". In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 157–179. ISBN: 9781450381789. URL: https://doi.org/10.1145/3426428.3426922.

[LW94] Barbara H. Liskov and Jeannette M. Wing. "A Behavioral Notion of Subtyping". In: *ACM Trans. Program. Lang. Syst.* 16.6 (Nov. 1994), pp. 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383. URL: https://doi.org/10.1145/197320.197383.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[Mer14] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583.

[Oho95] Atsushi Ohori. "A Polymorphic Record Calculus and Its Compilation". In: *ACM Trans. Program. Lang. Syst.* 17.6 (Nov. 1995), pp. 844–895. ISSN: 0164-0925. DOI: 10.1145/218570.218572. URL: https://doi.org/10.1145/218570.218572.

[OLE19] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. "Quantitative Program Reasoning with Graded Modal Types". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341714. URL: https://doi.org/10.1145/3341714.

[Par76] D. L. Parnas. "On the Design and Development of Program Families". In: *IEEE Trans. Softw. Eng.* 2.1 (Jan. 1976), pp. 1–9. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233797. URL: https://doi.org/10.1109/TSE.1976.233797.

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Heidelberg: Springer-Verlag, 2005. ISBN: 3540243720.

[PL16] Rachel Potvin and Josh Levenberg. "Why Google Stores Billions of Lines of Code in a Single Repository". In: *Commun. ACM* 59.7 (June 2016), pp. 78–87. ISSN: 0001-0782. DOI: 10.1145/2854146. URL: https://doi.org/10.1145/2854146.

[POM13] Tomas Petricek, Dominic Orchard, and Alan Mycroft. "Coeffects: Unified Static Analysis of Context-Dependence". In: *Automata, Languages, and Programming*. Ed. by Rūsiņš Fomin Fedor V.and Freivalds, Marta Kwiatkowska, and David Peleg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 385–397. ISBN: 978-3-642-39212-2.

[POM14] Tomas Petricek, Dominic Orchard, and Alan Mycroft. "Coeffects: A Calculus of Context-dependent Computation". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. New York, NY, USA: ACM, 2014, pp. 123–135. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628160. URL: http://doi.acm.org/10.1145/2628136.2628160 (visited on 05/23/2018).

[Pre13] Tom Preston-Werner. *Semantic Versioning 2.0.0*. 2013. URL: http://semver.org.

[Pre97] Christian Prehofer. "Feature-Oriented Programming: A Fresh Look at Objects". In: *Proceedings of the 11th European Conference on Object-Oriented Programming*. Vol. 1241. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 1997, pp. 419–443. ISBN: 3-540-63089-9. DOI: 10.1007/BFb0053389.

[RDV14] Steven Raemaekers, Arie van Deursen, and Joost Visser. "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository". In: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. New York, USA: IEEE, 2014, pp. 215–224. DOI: 10.1109/SCAM.2014.30.

[RvV17] S. Raemaekers, A. van Deursen, and J. Visser. "Semantic versioning and impact of breaking changes in the Maven repository". In: *Journal of Systems and Software* 129 (2017), pp. 140–158. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.04.008. URL: http://www.sciencedirect.com/science/article/pii/S0164121216300243.

[SBD11] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. "Compositional Type-Checking for Delta-Oriented Programming". In: *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*. AOSD '11. Porto de Galinhas, Brazil: Association for Computing Machinery, 2011, pp. 43–56. ISBN: 9781450306058. DOI: 10.1145/1960275.1960283. URL: https://doi.org/10.1145/1960275.1960283.

[Sch+10] Ina Schaefer et al. "Delta-Oriented Programming of Software Product Lines". In: *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*. SPLC'10. Jeju Island, South Korea: Springer-Verlag, 2010, pp. 77–91. ISBN: 3642155782.

[SD10] Ina Schaefer and Ferruccio Damiani. "Pure Delta-Oriented Programming". In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. FOSD '10. Eindhoven, The Netherlands: Association for Computing Machinery, 2010, pp. 49–56. ISBN: 9781450302081. DOI: 10.1145/1868688.1868696. URL: https://doi.org/10.1145/1868688.1868696.

[Seo+14] Hyunmin Seo et al. "Programmers' Build Errors: A Case Study (at Google)". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 724–734. ISBN: 9781450327565. DOI: 10.1145/2568225.2568255. URL: https://doi.org/10.1145/2568225.2568255.

[SJM08] Thorsten Schäfer, Jan Jonas, and Mira Mezini. "Mining Framework Usage Changes from Instantiation Code". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: Association for Computing Machinery, 2008, pp. 471–480. ISBN: 9781605580791. DOI: 10.1145/1368088.1368153. URL: https://doi.org/10.1145/1368088.1368153.

[Tol17] David Tolnay. *The semver trick*. https://github.com/dtolnay/semver-trick. July 2017.

[Wad90] Philip Wadler. "Linear Types Can Change the World!" In: *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

[Wal+14] Eric Walkingshaw et al. "Variational Data Structures: Exploring Tradeoffs in Computing with Variability". In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2014. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 213–226. ISBN: 9781450332101. DOI: 10.1145/2661136.2661143. URL: https://doi.org/10.1145/2661136.2661143.

[Wal13] Eric Walkingshaw. *The Choice Calculus: A Formal Language of Variation*. Oregon State University, 2013.

[Wu11] Wei Wu. "Modeling Framework API Evolution as a Multi-objective Optimization Problem". In: *2011 IEEE 19th International Conference on Program Comprehension*. New York, USA: IEEE, 2011, pp. 262–265. DOI: 10.1109/ICPC.2011.43.

Appendix A

Definitions and Proofs

Contents

λ_{VL} **syntax**

$$
\begin{aligned}
t &::= x \mid t_1\, t_2 \mid \lambda x.t \mid n \mid [t] \mid \textbf{let}\, [x] = t_1\, \textbf{in}\, t_2 \\
&\quad\mid \{\overline{l = t} \,|\, l_i\} \mid t.l \mid \langle \overline{l = t} \,|\, l_i \rangle && \text{(terms)} \\
v &::= \lambda x.t \mid n \mid [t] \mid \{\overline{l = t} \,|\, l_i\} && \text{(values)} \\
A, B &::= \mathsf{Int} \mid A \to B \mid \Box_r A && \text{(types)} \\
\Gamma, \Delta &::= \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A]_r && \text{(contexts)} \\
r &::= \bot \mid \{\overline{l_i}\} && \text{(version resources)} \\
l &::= [\overline{M_i \mapsto V_i}] && \text{(version labels)} \\
E &::= [\cdot] \mid E\, t \mid E.l \mid \textbf{let}\, [x] = E\, \textbf{in}\, t && \text{(evaluation contexts)}
\end{aligned}
$$

A.1 Resource Properties

Definition 4.4.2. [Version resource semiring] The version resource semiring is given by the structural semiring (semiring with preorder) $(\mathcal{R}, \oplus, 0, \otimes, 1, \sqsubseteq)$, defined as follows.

$$
0 = \bot \quad 1 = \emptyset \quad \bot \sqsubseteq \mathtt{r} \quad \frac{\mathtt{r}_1 \subseteq \mathtt{r}_2}{\mathtt{r}_1 \sqsubseteq \mathtt{r}_2}
$$

$$
r_1 \oplus r_2 = \begin{cases} r_1 & r_2 = \bot \\ r_2 & r_1 = \bot \\ r_1 \cup r_2 & \text{otherwise} \end{cases} \qquad r_1 \otimes r_2 = \begin{cases} \bot & r_1 = \bot \\ \bot & r_2 = \bot \\ r_1 \cup r_2 & \text{otherwise} \end{cases}
$$

where $\bot$ is the smallest element of $\mathcal{R}$, and $r_1 \subseteq r_2$ is the standard subset relation over sets defined only when both r_1 and r_2 are not $\bot$.

Lemma A.1.1 (Version resource semiring is a structural semiring).

Proof. Version resource semiring $(\mathcal{R}, \oplus, \bot, \otimes, \emptyset, \sqsubseteq)$ induces a semilattice with $\oplus$ (join).

- $(\mathcal{R}, \oplus, \bot, \otimes, \emptyset)$ is a semiring, that is:
  - $(\mathcal{R}, \oplus, \bot)$ is a commutative monoid, i.e., for all $p, q, r \in \mathcal{R}$

    * (Associativity) $(p \oplus q) \oplus r = p \oplus (q \oplus r)$ holds since $\oplus$ is defined in associative manner with $\bot$.
    * (Commutativity) $p \oplus q = q \oplus p$ holds since $\oplus$ is defined in commutative manner with $\bot$.
    * (Identity element) $\bot \oplus p = p \oplus \bot = p$
  - $(\mathcal{R}, \otimes, \emptyset)$ is a monoid, i.e., for all $p, q, r \in \mathcal{R}$
    * (Associativity) $(p \otimes q) \otimes r = p \otimes (q \otimes r)$ holds since $\oplus$ is defined in associative manner with $\bot$.
    * (Identity element) $\emptyset \otimes p = p \otimes \emptyset = p$
      · if $p = \bot$ then $\emptyset \otimes \bot = \bot \otimes \emptyset = \bot$
      · otherwise if $p \neq \bot$ then $\emptyset \otimes p = \emptyset \cup p = p$ and $p \otimes \emptyset = p \cup \emptyset = p$
  - multiplication $\otimes$ distributes over addition $\oplus$, i.e., for all $p, q, r \in \mathcal{R}, r \otimes (p \oplus q) = (r \otimes p) \oplus (r \otimes q)$ and $(p \oplus q) \otimes r = (p \otimes r) \oplus (q \otimes r)$
    * if $r = \bot$ then $r \otimes (p \oplus q) = \bot$ and $(r \otimes p) \oplus (r \otimes q) = \bot \oplus \bot = \bot$.
    * otherwise if $r \neq \bot$ and $p = \bot$ and $q \neq \bot$ then $r \otimes (p \oplus q) = r \otimes q = r \cup q = (r \cup r) \cup q = r \cup (r \cup q) = (r \oplus p) \cup (r \cup q) = (r \otimes p) \oplus (r \otimes q)$
    * otherwise if $r \neq \bot$ and $p = \bot$ and $q = \bot$ then $r \otimes (p \oplus q) = r \otimes \bot = \bot$ and $(r \otimes p) \oplus (r \otimes q) = \bot \oplus \bot = \bot$.
    * otherwise if $r \neq \bot$ and $p \neq \bot$ and $q \neq \bot$ then $r \otimes (p \oplus q) = r \cup (p \cup q) = (r \cup p) \cup (r \cup q) = (r \otimes p) \oplus (r \otimes q)$

    The other cases are symmetrical cases.
  - $\bot$ is absorbing for multiplication: $p \otimes \bot = \bot \otimes p = \bot$ for all $p \in \mathcal{R}$

- $(\mathcal{R}, \sqsubseteq)$ is a bounded semilattice, that is
  - $\sqsubseteq$ is a partial order on $\mathcal{R}$ such that the least upper bound of every two elements $p, q \in \mathcal{R}$ exists and is denoted by $p \oplus q$.
  - there is a least element; for all $r \in \mathcal{R}$, $\bot \sqsubseteq r$.

- (Motonicity of $\oplus$) $p \sqsubseteq q$ implies $p \oplus r \sqsubseteq q \oplus r$ for all $p, q, r \in \mathcal{R}$
  - if $r = \bot$ then $p \oplus r \sqsubseteq q \oplus r \Leftrightarrow p \subseteq q$, so this case is trivial.
  - otherwise if $r \neq \bot, p = q = \bot$ then $p \oplus r \sqsubseteq q \oplus r \Leftrightarrow r \subseteq r$, so this case is trivial.
  - otherwise if $r \neq \bot, p = \bot, q \neq \bot$ then $p \oplus r \sqsubseteq q \oplus r \Leftrightarrow r \subseteq q \cup r$, and $r \subseteq q \cup r$ holds in standard subset relation.
  - otherwise if $r \neq \bot, p \neq \bot, q \neq \bot$ then $p \oplus r \sqsubseteq q \oplus r \Leftrightarrow p \cup r \subseteq q \cup r$, and $p \subseteq q$ implies $p \cup r \subseteq q \cup r$.

- (Motonicity of $\otimes$) $p \sqsubseteq q$ implies $p \otimes r \sqsubseteq q \otimes r$ for all $p, q, r \in \mathcal{R}$
  - if $r = \bot$ then $p \otimes r \sqsubseteq q \otimes r \Leftrightarrow \bot \subseteq \bot$, so this case is trivial.
  - otherwise if $r \neq \bot, p = q = \bot$ then $p \otimes r \sqsubseteq q \otimes r \Leftrightarrow \bot \subseteq \bot$, so this case is trivial.
  - otherwise if $r \neq \bot, p = \bot, q \neq \bot$ then $p \otimes r \sqsubseteq q \otimes r \Leftrightarrow \bot \subseteq q \cup r$, so this case is trivial.
  - otherwise if $r \neq \bot, p \neq \bot, q \neq \bot$ then $p \otimes r \sqsubseteq q \otimes r \Leftrightarrow p \cup r \subseteq q \cup r$, and $p \subseteq q$ implies $p \cup r \subseteq q \cup r$.

□

Definition A.1.2 (Version resource summation $\sum$)**.** Using the addition $+$ of version resource semiring, summation of version resouce is defined as follows:

$$\sum_i r_i = r_1 \oplus \cdots \oplus r_n$$

A.2 Context Properties

Definition 4.5.2. [Context concatenation , & +] Two typing contexts can be concatenated by "," if they contain disjoint assumptions. Furthermore, the versioned assumptions appearing in both typing contexts can be combined using the context concatenation $+$ defined with the addition $\oplus$ in the version resource semiring as follows.

$$\begin{aligned} \emptyset + \Gamma &= \Gamma \\ (\Gamma, x : A) + \Gamma' &= (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin \mathrm{dom}(\Gamma') \\ \Gamma + \emptyset &= \Gamma \\ \Gamma + (\Gamma', x : A) &= (\Gamma + \Gamma'), x : A \quad \text{iff } x \notin \mathrm{dom}(\Gamma) \\ (\Gamma, x : [A]_r) + (\Gamma', x : [A]_s) &= (\Gamma + \Gamma'), x : [A]_{(r \oplus s)} \end{aligned}$$

Definition 4.5.3. [Context multiplication $\cdot$ by a resource] Assuming that a context contains only version assumptions, denoted $[\Gamma]$ in typing rules, then Γ can be multiplied by a version resource $r \in \mathcal{R}$ by using the product $\otimes$ in the version resource semiring, as follows.

$$r \cdot \emptyset \ = \ \emptyset \qquad\qquad r \cdot (\Gamma,\ x : [A]_s) \ = \ (r \cdot \Gamma),\ x : [A]_{(r \otimes s)}$$

Definition 4.5.4. [Context summation $\bigcup$] Using the context concatenation $+$, summation of typing contexts is defined as follows:

$$\bigcup_{i=1}^{n} \Gamma_i \ = \ \Gamma_1 + \cdots + \Gamma_n$$

Definition A.2.1 (Context partition)**.** For typing contexts Γ_1 and Γ_2, we define $\Gamma_{1|\Gamma_2}$ and $\Gamma_{1|\overline{\Gamma_2}}$ as follows.

$$\begin{aligned} \Gamma_{1|\Gamma_2} &\triangleq \{x : A \mid x \in \mathrm{dom}(\Gamma_1) \wedge x \in \mathrm{dom}(\Gamma_2)\} \\ \Gamma_{1|\overline{\Gamma_2}} &\triangleq \{x : A \mid x \in \mathrm{dom}(\Gamma_1) \wedge x \notin \mathrm{dom}(\Gamma_2)\} \end{aligned}$$

$\Gamma_{1|\Gamma_2}$ is a subsequence of Γ_1 that contains all the term variables that are *included* in Γ_2, and $\Gamma_{1|\overline{\Gamma_2}}$ is a subsequence of Γ_1 that contains all the term variables that are *not included* in Γ_2.

Using $\Gamma_{1|\Gamma_2}$ and $\Gamma_{1|\overline{\Gamma_2}}$, we state some corollaries about typing contexts. These theorems follow straightforwardly from the definitions of A.2.1.

Lemma A.2.2 (Context collapse)**.** For typing contexts Γ_1 and Γ_2,

$$\Gamma_{1|\Gamma_2} + \Gamma_{1|\overline{\Gamma_2}} = \Gamma_1$$

Lemma A.2.3 (Context shuffle)**.** For typing contexts Γ_1, Γ_2, Γ_3 and Γ_4, and variable x and type A:

$$(\Gamma_1, x : A, \Gamma'_1) + \Gamma_2 = (\Gamma_1 + \Gamma_{2|\Gamma_1}), x : A, (\Gamma'_1 + \Gamma_{2|\overline{\Gamma_1}}) \quad (1)$$

$$\Gamma_1 + (\Gamma_2, x : A, \Gamma'_2) = (\Gamma_{1|\overline{\Gamma'_2}} + \Gamma_2), x : A, (\Gamma_{1|\Gamma'_2} + \Gamma'_2) \quad (2)$$

$$(\Gamma_1, \Gamma_2) + (\Gamma_3, \Gamma_4) = \Big((\Gamma_1 + \Gamma_{3|\Gamma_1} + \Gamma_{4|\Gamma_1}), (\Gamma_2 + \Gamma_{3|\overline{\Gamma_1}} + \Gamma_{4|\overline{\Gamma_1}})\Big) \quad (3)$$

Lemma A.2.4 (Composition of context shuffle)**.** For typing contexts Γ_i and Γ'_i for $i \in \mathbb{N}$, there exixts typing contexts Γ and Γ' such that:

$$\bigcup_i (\Gamma_i, \Gamma'_i) = \Gamma, \Gamma' \;\wedge\; \bigcup_i (\Gamma_i + \Gamma'_i) = \Gamma + \Gamma'$$

Lemma A.2.5 (Distribution of version resouce over context addition)**.** For a typing context Γ and resources $r_i \in R$:

$$(r_1 \cdot \Gamma) + (r_2 \cdot \Gamma) = (r_1 \oplus r_2) \cdot \Gamma$$
$$\bigcup_i (r_i \cdot \Gamma) = (\sum_i r_i) \cdot \Gamma$$

Lemma A.2.6 (Disjoint context collapse)**.** Given typing contexts Γ_1, Δ, and Γ_2 such that Γ_1 and Γ_2 are disjoint, then we can conclude the following.

$$(\Gamma_1 + \Delta + \Gamma_2) = (\Gamma_1 + \Delta_{|\Gamma_1}), \Delta_{|\overline{(\Gamma_1, \Gamma_2)}}, (\Gamma_2 + \Delta_{|\Gamma_2})$$

A.3 Substituions Properties

Lemma 4.7.1. [Well-typed linear substitution]

$$\left.\begin{array}{r} \Delta \vdash t' : A \\ \Gamma, x : A, \Gamma' \vdash t : B \end{array}\right\} \implies \Gamma + \Delta + \Gamma' \vdash [t'/x]t : B$$

Proof. This proof is given by induction on the structure of $\Gamma, x : A, \Gamma' \vdash t : B$ (assumption 2). Consider the cases for the last rule used in the typing derivation of assumption 2.

$$\text{Case.} \quad \frac{}{y : B \vdash y : B}\;(\text{VAR})$$

In this case we know the following:

- $\Gamma = \Gamma' = \emptyset$
- $x = t = y$
- $A = B$

Now the conclusion of the lemma is:

$$\Delta \vdash [t'/y]y : B$$

Since $[t'/y]y = t'$ from the definition of substitution, the conclusion of the lemma is assumption 1 itself.

$$\text{Case.} \quad \frac{\Gamma, x : A, \Gamma', y : B_1 \vdash t : B_2}{\Gamma, x : A, \Gamma' \vdash \lambda y.t : B_1 \to B_2} \text{(ABS)}$$

In this case, by applying the induction hypothesis to the second premise, we know the following:

$$\Gamma + \Delta + (\Gamma', y : B_1) \vdash [t'/x]t : B_2$$

where $y : B$ is disjoint with Γ, Δ, and Γ'. Thus, $\Gamma + \Delta + (\Gamma', y : B_1) = (\Gamma + \Delta + \Gamma'), y : B_1$ from lemma A.2.3 (2), the typing derivation above is equal to the following:

$$(\Gamma + \Delta + \Gamma'), y : B_1 \vdash [t'/x]t : B_2$$

We then reapply (ABS) to obtain the following:

$$\frac{(\Gamma + \Delta + \Gamma'), y : B_1 \vdash [t'/x]t : B_2}{\Gamma + \Delta + \Gamma' \vdash \lambda y.[t'/x]t : B_1 \to B_2} \text{(ABS)}$$

By the definition of substitution $\lambda y.[t'/x]t = [t'/x](\lambda y.t)$, and we obtain the conclusion of the lemma.

$$\text{Case.} \quad \frac{\begin{array}{c}\Gamma_1 \vdash t_1 : B_1 \to B_2 \\ \Gamma_2 \vdash t_2 : B_1\end{array}}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B_2} \text{(APP)}$$

In this case we know the following:

- $t = t_1\, t_2$
- $B = B_2$
- $(\Gamma, x : A, \Gamma') = (\Gamma_1 + \Gamma_2)$

Now focusing on the third equation, since the definition of the context addition $+$, the linear assumption $x : A$ is contained in only one of Γ_1 or Γ_2.

- Suppose $(x : A) \in \Gamma_1$ and $(x : A) \notin \Gamma_2$.
  Let Γ_1' and Γ_1'' be typing contexts such that they satisfy $\Gamma_1 = (\Gamma_1', x : A, \Gamma_1'')$. The last typing derivation of (APP) is rewritten as follows.

$$\frac{\begin{array}{c}\Gamma_1', x : A, \Gamma_1'' \vdash t_1 : B_1 \to B_2 \\ \Gamma_2 \vdash t_2 : B_1\end{array}}{(\Gamma_1', x : A, \Gamma_1'') + \Gamma_2 \vdash t_1\, t_2 : B_2} \text{(APP)}$$

  Now, we compare the typing contexts between the lemma and the above conclusion as follows:

$$\begin{aligned}(\Gamma, x : A, \Gamma') &= (\Gamma_1 + \Gamma_2) \\ &= (\Gamma_1', x : A, \Gamma_1'') + \Gamma_2 && (\because \Gamma_1 = (\Gamma_1', x : A, \Gamma_1'')) \\ &= (\Gamma_1' + \Gamma_{2|\Gamma_1'}), x : A, (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1'}}) && (\because \text{A.2.3 (1)})\end{aligned}$$

By the commutativity of ",", we can take Γ and Γ' arbitrarily so that they satisfy the above equation. So here we know $\Gamma = (\Gamma_1' + \Gamma_{2|\Gamma_1'})$ and $\Gamma' = (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1'}})$.

We then apply the induction hypothesis to each of the two premises and reapply (APP) as follows:

$$\frac{\begin{array}{c}\Gamma_1' + \Delta + \Gamma_1'' \vdash [t'/x]t_1 : B_1 \to B_2 \\ \Gamma_2 \vdash t_2 : B_1\end{array}}{(\Gamma_1' + \Delta + \Gamma_1'') + \Gamma_2 \vdash ([t'/x]t_1)\, t_2 : B_2} \quad (\text{APP})$$

Since $([t'/x]t_1)\, t_2 = [t'/x](t_1\, t_2)$ if $x \notin FV(t_2)$, the conclusion of the above derivation is equivalent to the conclusion of the lemma except for the typing contexts.

Finally, we must show that $(\Gamma + \Delta + \Gamma') = ((\Gamma_1 + \Delta + \Gamma_1'') + \Gamma_2)$. This holds from the following reasoning:

$$\begin{aligned}
(\Gamma + \Delta + \Gamma') &= (\Gamma_1' + \Gamma_{2|\Gamma_1'}) + \Delta + (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1'}}) \\
&\qquad (\because \Gamma = (\Gamma_1' + \Gamma_{2|\Gamma_1'}) \text{ and } \Gamma' = (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1'}})) \\
&= \Gamma_1' + \Gamma_{2|\Gamma_1'} + \Delta + \Gamma_1'' + \Gamma_{2|\overline{\Gamma_1'}} && (\because + \text{ associativity}) \\
&= \Gamma_1' + \Delta + \Gamma_1'' + \Gamma_{2|\Gamma_1'} + \Gamma_{2|\overline{\Gamma_1'}} && (\because + \text{ commutativity}) \\
&= (\Gamma_1' + \Delta + \Gamma_1'') + (\Gamma_{2|\Gamma_1'} + \Gamma_{2|\overline{\Gamma_1'}}) && (\because + \text{ associativity}) \\
&= (\Gamma_1' + \Delta + \Gamma_1'') + \Gamma_2 && (\because \text{A.2.2})
\end{aligned}$$

Thus, we obtain the conclusion of the lemma.

- Suppose $(x : A) \notin \Gamma_1$ and $(x : A) \in \Gamma_2$
  Let Γ_2' and Γ_2'' be typing contexts such that they satisfy $\Gamma_2 = (\Gamma_2', x : A, \Gamma_2'')$. The last typing derivation of (APP) is rewritten as follows.

$$\frac{\begin{array}{c}\Gamma_1 \vdash t_1 : B_1 \to B_2 \\ \Gamma_2', x : A, \Gamma_2'' \vdash t_2 : B_1\end{array}}{\Gamma_1 + (\Gamma_2', x : A, \Gamma_2'') \vdash t_1\, t_2 : B_2} \ (\text{APP})$$

  This case is similar to the case $(x : A) \in \Gamma_1$, but using A.2.3 (2) instead of A.2.3 (1).

$$\text{Case.} \qquad \frac{}{\varnothing \vdash n : \mathsf{Int}} \ (\text{INT})$$

In this case, the above typing context is empty ($= \varnothing$), so this case holds trivially.

$$\text{Case.} \quad \frac{\Gamma_1, x : A, \Gamma_2 \vdash t : B}{(\Gamma_1, x : A, \Gamma_2) + [\Delta']_0 \vdash t : B} \ (\text{WEAK})$$

In this case, the linear assumption $x : A$ is not contained in versioned context $[\Delta']_0$. We then compare the typing contexts between the conclusion of the lemma and that of (WEAK) as follows:

$$\begin{aligned}
(\Gamma, x : A, \Gamma') &= (\Gamma_1, x : A, \Gamma_2) + [\Delta']_0 \\
&= (\Gamma_1 + ([\Delta']_0)_{|\Gamma_1}), x : A, (\Gamma_2 + ([\Delta']_0)_{|\overline{\Gamma_1}}) && (\because \text{A.2.3 (1)})
\end{aligned}$$

By the commutativity of ",", we can take Γ and Γ' arbitrarily so that they satisfy the above equation. So here we obtain $\Gamma = \Gamma_1 + ([\Delta']_0)_{|\Gamma_1}$ and $\Gamma' = \Gamma_2 + ([\Delta']_0)_{|\overline{\Gamma_1}}$. We then apply the induction hypothesis to each of the premise and reapply (WEAK) as follows:

$$\frac{\Gamma_1 + \Delta + \Gamma_2 \vdash [t'/x]t : B}{(\Gamma_1 + \Delta + \Gamma_2) + [\Delta']_0 \vdash [t'/x]t : B}\ (\text{WEAK})$$

Since $([t'/x]t_1)\,([t'/x]t_2) = [t'/x](t_1\,t_2)$, the conclusion of the above derivation is equivalent to the conclusion of the lemma except for typing contexts.

Finally, we must show that $(\Gamma_1 + \Delta + \Gamma_2) + [\Delta']_0 = \Gamma + \Delta + \Gamma'$. This holds from the following reasoning:

$$\begin{aligned}
(\Gamma + \Delta + \Gamma') &= (\Gamma_1 + ([\Delta']_0)_{|\Gamma_1}) + \Delta + (\Gamma_2 + ([\Delta']_0)_{|\overline{\Gamma_1}}) \\
&\qquad (\because \Gamma = \Gamma_1 + ([\Delta']_0)_{|\Gamma_1} \text{ and } \Gamma' = \Gamma_2 + ([\Delta']_0)_{|\overline{\Gamma_1}}) \\
&= \Gamma_1 + ([\Delta']_0)_{|\Gamma_1} + \Delta + \Gamma_2 + ([\Delta']_0)_{|\overline{\Gamma_1}} && (\because + \text{ associativity}) \\
&= \Gamma_1 + \Delta + \Gamma_2 + ([\Delta']_0)_{|\Gamma_1} + ([\Delta']_0)_{|\overline{\Gamma_1}} && (\because + \text{ commutativity}) \\
&= (\Gamma_1 + \Delta + \Gamma_2) + (([\Delta']_0)_{|\Gamma_1} + ([\Delta']_0)_{|\overline{\Gamma_1}}) && (\because + \text{ associativity}) \\
&= (\Gamma_1 + \Delta + \Gamma_2) + [\Delta']_0 && (\because \text{A.2.2})
\end{aligned}$$

Thus, we obtain the conclusion of the lemma.

$$\text{Case.} \quad \frac{\Gamma, x : A, \Gamma'', y : B_1 \vdash t : B_2}{\Gamma, x : A, \Gamma'', y : [B_1]_1 \vdash t : B_2}\ (\text{DER})$$

In this case, a linear assumption $x : A$ cannot be a versioned assumption $y : [B_1]_1$. Applying the induction hypothesis to the premise, we obtain the following:

$$\Gamma + \Delta + (\Gamma'', y : B_1) \vdash [t'/x]t : B_2$$

Note that $\Gamma + \Delta + (\Gamma'', y : B_1) = (\Gamma + \Delta + \Gamma''), y : B_1$ holds because $y : B_1$ is a linear assumption and is disjoint with Γ, Δ, and Γ''. Thus, the above judgement is equivalent to the following:

$$(\Gamma + \Delta + \Gamma''), y : B_1 \vdash [t'/x]t : B_2$$

We then reapply (DER) to obtain the following:

$$\frac{(\Gamma + \Delta + \Gamma''), y : B_1 \vdash [t'/x]t : B_2}{(\Gamma + \Delta + \Gamma''), y : [B_1]_1 \vdash [t'/x]t : B_2}\ (\text{DER})$$

Finally, since $y : [B_1]_1$ is disjoint with $\Gamma + \Delta + \Gamma''$, $((\Gamma + \Delta + \Gamma''), y : [B_1]_1) = \Gamma + \Delta + (\Gamma'', y : [B_1]_1)$ holds. Thus, the conclusion of the above derivation is equivalent to the following:

$$\Gamma + \Delta + (\Gamma'', y : [B_1]_1) \vdash [t'/x]t : B_2$$

Thus, we obtain the conclusion of the lemma.

$$\text{Case.} \quad \frac{[\Gamma] \vdash t : B}{r \cdot [\Gamma] \vdash [t] : \Box_r B}\ (\text{PR})$$

This case holds trivially, because the typing context $[\Gamma]$ contains only versioned assumptions and does not contain any linear assumptions.

$$\text{Case.} \qquad \frac{[\Gamma_i] \vdash t_i : A}{\bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash \{\overline{l = t}|l_i\} : \Box_{\overline{l}} A} \ (\textsc{ver})$$

This case holds trivially, because the typing context of the conclusion contains only versioned assumptions (by $[\Gamma_i]$ in the premise) and does not contain any linear assumptions.

$$\text{Case.} \qquad \frac{[\Gamma_i] \vdash t_i : A}{\bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash \langle \overline{l = t} \,|\, l_i \rangle : A} \ (\textsc{veri})$$

This case holds trivially, because the typing context of the conclusion contains only versioned assumptions (by $[\Gamma_i]$ in the premise) and does not contain any linear assumptions.

$$\text{Case.} \qquad \frac{\Gamma \vdash t : \Box_r A \quad l \in r}{\Gamma \vdash t.l : A} \ (\textsc{extr})$$

In this case, we apply the induction hypothesis to the premise and then reapply (EXTR), we obtain the conclusion of the lemma.

$$\text{Case.} \ \frac{\Gamma, y : [B']_r, \Gamma' \vdash t : B \quad r \sqsubseteq s}{\Gamma, y : [B']_s, \Gamma' \vdash t : B} \ (\textsc{sub})$$

In this case, a linear assumption $x : A$ cannot be a versioned assumption $y : [B_1]_s$, and only one of $(x : A) \in \Gamma$ or $(x : A) \in \Gamma'$ holds. In either case, applying the induction hypothesis to the premise and reappling (SUB), we obtain the conclusion of the lemma. $\Box$

Lemma 4.7.2. [Well-typed versioned substitution]

$$\left.\begin{array}{r}[\Delta] \vdash t' : A \\ \Gamma, x : [A]_r, \Gamma' \vdash t : B\end{array}\right\} \implies \Gamma + r \cdot \Delta + \Gamma' \vdash [t'/x]t : B$$

Proof. This proof is given by induction on structure of $\Gamma, x : [A]_r, \Gamma' \vdash t : B$ (assumption 2). Consider the cases for the last rule used in the typing derivation of assumption 2.

$$\text{Case.} \quad \frac{}{y : B \vdash y : B}\ (\text{VAR})$$

In this case, $x : [A]_r$ is a versioned assumption and $y : B$ is a linear assumption, so $x \neq y$ holds, and yet the typing context besides $y : B$ is empty. Thus, there are no versioned variables to be substituted, so this case holds trivially.

$$\text{Case.} \quad \frac{\Gamma, x : [A]_r, \Gamma', y : B_1 \vdash t : B_2}{\Gamma, x : [A]_r, \Gamma' \vdash \lambda y.t : B_1 \to B_2}\ (\text{ABS})$$

In this case, we know the following by applying induction hypothesis to the partial derivation of (ABS):

$$\Gamma + r \cdot \Delta + (\Gamma', y : B_1) \vdash [t'/x]t : B_2$$

where $y : B_1$ is disjoint with Γ, Δ, and Γ'. Thus, $\Gamma + r \cdot \Delta + (\Gamma', y : B_1) = (\Gamma + r \cdot \Delta + \Gamma'), y : B_1$ from lemma A.2.3 (2), the typing derivation above is equal to the following:

$$(\Gamma + r \cdot \Delta + \Gamma'), y : B_1 \vdash [t'/x]t : B_2$$

We then reapply (ABS) to obtain the following:

$$\frac{(\Gamma + r \cdot \Delta + \Gamma'), y : B_1 \vdash [t'/x]t : B_2}{\Gamma + r \cdot \Delta + \Gamma' \vdash \lambda y.[t'/x]t : B_1 \to B_2}\ (\text{ABS})$$

Since $\lambda y.[t'/x]t = [t'/x](\lambda y.t)$ from the definition of substitution, we obtain the conclusion of the lemma.

$$\text{Case.} \quad \frac{\begin{array}{c}\Gamma_1 \vdash t_1 : B_1 \to B_2 \\ \Gamma_2 \vdash t_2 : B_1\end{array}}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : B_2}\ (\text{APP})$$

In this case, we know the following:

- $t = t_1\, t_2$
- $B = B_2$
- $(\Gamma, x : [A]_r, \Gamma') = (\Gamma_1 + \Gamma_2)$

Now focusing on the third equation, since the definition of the context addition $+$, the linear assumption $x : A$ is contained in either or both of the typing context Γ_1 or Γ_2.

- Suppose $(x : [A]_r) \in \Gamma_1$ and $x \notin \mathrm{dom}(\Gamma_2)$
  Let Γ_1' and Γ_1'' be typing contexts such that they satisfy $\Gamma_1 = (\Gamma_1', x : [A]_r, \Gamma_1'')$. The last derivation of (APP) is rewritten as follows:

$$\frac{\begin{array}{c}\Gamma_1', x : [A]_r, \Gamma_1'' \vdash t_1 : B_1 \to B_2 \\ \Gamma_2 \vdash t_2 : B_1\end{array}}{(\Gamma_1', x : [A]_r, \Gamma_1'') + \Gamma_2 \vdash t_1\, t_2 : B_2}\ (\text{APP})$$

  We compare the typing contexts between the conclusion of the lemma and that of the above derivation to obtain the following:

$$\begin{aligned}(\Gamma, x : [A]_r, \Gamma') &= (\Gamma_1', x : [A]_r, \Gamma_1'') + \Gamma_2 \\ &= (\Gamma_1' + \Gamma_{2|\Gamma_1'}), x : [A]_r, (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1'}}) \qquad (\because \text{A.2.3 (1)})\end{aligned}$$

  By the commutativity of ",", we can take Γ and Γ' arbitrarily so that they satisfy the above equation. So here we know $\Gamma = (\Gamma_1' + \Gamma_{2|\Gamma_1'})$ and $\Gamma' = (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1'}})$.

  We then apply the induction hypothesis to each of the two premises of the last derivation and reapply (APP) as follows:

$$\frac{\begin{array}{c}\Gamma_1' + r \cdot \Delta + \Gamma_1'' \vdash [t'/x]t_1 : B_1 \to B_2 \\ \Gamma_2 \vdash [t'/x]t_2 : B_1\end{array}}{(\Gamma_1' + r \cdot \Delta + \Gamma_1'') + \Gamma_2 \vdash ([t'/x]t_1)\,([t'/x]t_2) : B_2}\ (\text{APP})$$

  Since $([t'/x]t_1)\,([t'/x]t_2) = [t'/x](t_1\, t_2)$, the conclusion of the above derivation is equivalent to the conclusion of the lemma except for the typing contexts. Finally, we must show that $(\Gamma + r \cdot \Delta + \Gamma') = ((\Gamma_1 + r \cdot \Delta + \Gamma_1'') + \Gamma_2)$. This holds from the following reasoning:

$$\begin{aligned}(\Gamma + r \cdot \Delta + \Gamma') &= (\Gamma_1' + \Gamma_{2|\Gamma_1'}) + r \cdot \Delta + (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1'}}) \\ &\qquad (\because \Gamma = (\Gamma_1' + \Gamma_{2|\Gamma_1'})\ \&\ \Gamma' = (\Gamma_1'' + \Gamma_{2|\overline{\Gamma_1'}})) \\ &= \Gamma_1' + \Gamma_{2|\Gamma_1'} + r \cdot \Delta + \Gamma_1'' + \Gamma_{2|\overline{\Gamma_1'}} \qquad (\because + \text{ associativity}) \\ &= \Gamma_1' + r \cdot \Delta + \Gamma_1'' + \Gamma_{2|\Gamma_1'} + \Gamma_{2|\overline{\Gamma_1'}} \qquad (\because + \text{ commutativity}) \\ &= (\Gamma_1' + r \cdot \Delta + \Gamma_1'') + (\Gamma_{2|\Gamma_1'} + \Gamma_{2|\overline{\Gamma_1'}}) \qquad (\because + \text{ associativity}) \\ &= (\Gamma_1' + r \cdot \Delta + \Gamma_1'') + \Gamma_2 \qquad (\because \text{A.2.2})\end{aligned}$$

  Thus, we obtain the conclusion of the lemma.

- Suppose $x \notin \mathrm{dom}(\Gamma_1)$ and $(x : [A]_r) \in \Gamma_2$
  Let Γ_2' and Γ_2'' be typing contexts such that they satisfy $\Gamma_2 = (\Gamma_2', x : [A]_r, \Gamma_2'')$. The last typing derivation of (APP) is rewritten as follows:

$$\frac{\begin{array}{c}\Gamma_1 \vdash t_1 : B_1 \to B_2 \\ \Gamma_2', x : [A]_r, \Gamma_2'' \vdash t_2 : B_1\end{array}}{\Gamma_1 + (\Gamma_2', x : [A]_r, \Gamma_2'') \vdash t_1\, t_2 : B_2}\ (\text{APP})$$

  This case is similar to the case $(x : [A]_r) \in \Gamma_1$ and $x \notin \mathrm{dom}(\Gamma_2)$, but using A.2.3 (2) instead of A.2.3 (1).

- Suppose $(x : [A]_{r_1}) \in \Gamma_1$ and $(x : [A]_{r_2}) \in \Gamma_2$ where $r = r_1 \oplus r_2$.
  Let Γ'_1, Γ''_1, Γ'_2, and Γ''_2 be typing contexts such that they satisfy $\Gamma_1 = (\Gamma'_1, x : [A]_{r_1}, \Gamma''_1)$ and $\Gamma_2 = (\Gamma'_2, x : [A]_{r_2}, \Gamma''_2)$. The last derivation of (APP) is rewritten as follows:

$$\frac{\begin{array}{c}\Gamma'_1, x : [A]_{r_1}, \Gamma''_1 \vdash t_1 : B_1 \to B_2 \\ \Gamma'_2, x : [A]_{r_2}, \Gamma''_2 \vdash t_2 : B_1\end{array}}{(\Gamma'_1, x : [A]_{r_1}, \Gamma''_1) + (\Gamma'_2, x : [A]_{r_2}, \Gamma''_2) \vdash t_1\, t_2 : B_2}\ (\text{APP})$$

  Now, we compare the typing contexts between the lemma and the above conclusion as follows:

$$\begin{aligned}
(\Gamma, x : [A]_r, \Gamma') &= (\Gamma'_1, x : [A]_{r_1}, \Gamma''_1) + (\Gamma'_2, x : [A]_{r_2}, \Gamma''_2) \\
&= (\Gamma'_1, \Gamma''_1, x : [A]_{r_1}) + (\Gamma'_2, \Gamma''_2, x : [A]_{r_2}) \quad (\because \text{, commutativity}) \\
&= ((\Gamma'_1, \Gamma''_1) + (\Gamma'_2, \Gamma''_2)), x : [A]_{r_1 \oplus r_2} \quad (\because + \text{ definiton}) \\
&= ((\Gamma'_1 + \Gamma'_{2|\Gamma'_1} + \Gamma''_{2|\Gamma'_1}), (\Gamma''_1 + \Gamma'_{2|\overline{\Gamma'_1}} + \Gamma''_{2|\overline{\Gamma'_1}})), x : [A]_{r_1 \oplus r_2} \\
&\qquad (\because \text{A.2.3 (3)}) \\
&= (\Gamma'_1 + \Gamma'_{2|\Gamma'_1} + \Gamma''_{2|\Gamma'_1}), (\Gamma''_1 + \Gamma'_{2|\overline{\Gamma'_1}} + \Gamma''_{2|\overline{\Gamma'_1}}), x : [A]_{r_1 \oplus r_2} \\
&\qquad (\because \text{, associativity}) \\
&= (\Gamma'_1 + \Gamma'_{2|\Gamma'_1} + \Gamma''_{2|\Gamma'_1}), x : [A]_{r_1 \oplus r_2}, (\Gamma''_1 + \Gamma'_{2|\overline{\Gamma'_1}} + \Gamma''_{2|\overline{\Gamma'_1}}) \\
&\qquad (\because \text{, commutativity})
\end{aligned}$$

  By the commutativity of ",", we can take Γ and Γ' arbitrarily so that they satisfy the above equation. So here we know $\Gamma = (\Gamma'_1 + \Gamma'_{2|\Gamma'_1} + \Gamma''_{2|\Gamma'_1})$ and $\Gamma' = (\Gamma''_1 + \Gamma'_{2|\overline{\Gamma'_1}} + \Gamma''_{2|\overline{\Gamma'_1}})$.
  We then apply the induction hypothesis to each of the two premises of the last derivation and reapply (APP) as follows:

$$\frac{\begin{array}{c}\Gamma'_1 + r_1 \cdot \Delta + \Gamma''_1 \vdash [t'/x]t_1 : B_1 \to B_2 \\ \Gamma'_2 + r_2 \cdot \Delta + \Gamma''_2 \vdash [t'/x]t_2 : B_1\end{array}}{(\Gamma'_1 + r_1 \cdot \Delta + \Gamma''_1) + (\Gamma'_2 + r_2 \cdot \Delta + \Gamma''_2) \vdash ([t'/x]t_1)\, ([t'/x]t_2) : B_2}\ (\text{APP})$$

  Since $([t'/x]t_1)\, ([t'/x]t_2) = [t'/x](t_1\, t_2)$, the conclusion of the above derivation is equivalent to the conclusion of the lemma except for the typing contexts. Finally, we must show that $\Gamma + r \cdot \Delta + \Gamma' = (\Gamma'_1 + r_1 \cdot \Delta + \Gamma''_1) + (\Gamma'_2 + r_2 \cdot \Delta + \Gamma''_2)$.

$$\begin{aligned}
(\Gamma + r \cdot \Delta + \Gamma') &= (\Gamma'_1 + \Gamma'_{2|\Gamma'_1} + \Gamma''_{2|\Gamma'_1}) + (r_1 \oplus r_2) \cdot \Delta + (\Gamma''_1 + \Gamma'_{2|\overline{\Gamma'_1}} + \Gamma''_{2|\overline{\Gamma'_1}}) \\
&\qquad (\because r = r_1 \oplus r_2 \ \&\ \Gamma = (\Gamma'_1 + \Gamma'_{2|\Gamma'_1} + \Gamma''_{2|\Gamma'_1}) \ \&\ \Gamma' = (\Gamma''_1 + \Gamma'_{2|\overline{\Gamma'_1}} + \Gamma''_{2|\overline{\Gamma'_1}})) \\
&= \Gamma'_1 + (r_1 \oplus r_2) \cdot \Delta + \Gamma''_1 + (\Gamma'_{2|\Gamma'_1} + \Gamma'_{2|\overline{\Gamma'_1}}) + (\Gamma''_{2|\Gamma'_1} + \Gamma''_{2|\overline{\Gamma'_1}}) \\
&\qquad (\because + \text{ associativity \& commutativity}) \\
&= \Gamma'_1 + (r_1 \oplus r_2) \cdot \Delta + \Gamma''_1 + \Gamma'_2 + \Gamma''_2 \quad (\because \text{A.2.2}) \\
&= \Gamma'_1 + r_1 \cdot \Delta + r_2 \cdot \Delta + \Gamma''_1 + \Gamma'_2 + \Gamma''_2 \quad (\because \text{A.2.5}) \\
&= (\Gamma'_1 + r_1 \cdot \Delta + \Gamma''_1) + (\Gamma'_2 + r_2 \cdot \Delta + \Gamma''_2) \\
&\qquad (\because + \text{ associativity and commutativity})
\end{aligned}$$

  Thus, we obtain the conclusion of the lemma.

Case. $\dfrac{}{\varnothing \vdash n : \mathsf{Int}}$ (INT)

This case holds trivially because the typing context of (INT) is empty ($= \varnothing$).

Case. $\dfrac{\Gamma'' \vdash t : B}{\Gamma'' + [\Delta']_0 \vdash t : B}$ (WEAK)

In this case, we know $(\Gamma, x : [A]_r, \Gamma') = \Gamma'' + [\Delta']_0$. There are two cases where the versioned assumption $x : [A]_r$ is contained in $[\Delta']_0$ and not included.

- Suppose $(x : [A]_r) \in [\Delta']_0$
  We know $r = 0$. Let Δ_1 and Δ_2 be typing context such that $\Delta' = (\Delta_1, x : [A]_0, \Delta_2)$. The last derivation is rewritten as follows:

  $$\frac{\Gamma'' \vdash t : B}{\Gamma'' + [\Delta_1, x : [A]_0, \Delta_2]_0 \vdash t : B} \text{(WEAK)}$$

  We compare the typing contexts between the conclusion of the lemma and that of the above derivation to obtain the following:

  $$\begin{aligned}(\Gamma, x : [A]_0, \Gamma') &= \Gamma'' + [\Delta_1, x : [A]_0, \Delta_2]_0 && (\because \Delta' = (\Delta_1, x : [A]_0, \Delta_2))\\ &= \Gamma'' + ([\Delta_1]_0, x : [A]_0, [\Delta_2]_0) && (\because \text{definiton of } [\Gamma]_0)\\ &= (\Gamma''_{|[\Delta_2]_0} + [\Delta_1]_0), x : [A]_0, (\Gamma''_{\overline{|[\Delta_2]_0}} + [\Delta_2]_0) && (\because \text{A.2.3 (2)})\end{aligned}$$

  By the commutativity of ",", we can take Γ and Γ' arbitrarily so that they satisfy the above equation. So here we know $\Gamma = (\Gamma''_{|[\Delta_2]_0} + [\Delta_1]_0)$ and $\Gamma' = (\Gamma''_{\overline{|[\Delta_2]_0}} + [\Delta_2]_0)$.

  We then apply the induction hypothesis to the premise of the last derivation and reapply (WEAK) as follows:

  $$\frac{\Gamma'' \vdash [t'/x]t : B}{\Gamma'' + [\Delta_1 + \Delta + \Delta_2]_0 \vdash [t'/x]t : B} \text{(WEAK)}$$

  where we choose $\Delta_1 + \Delta + \Delta_2$ as the newly added typing context. Since x is unused by t, thus note that $[t'/x]t = t$, the conclusion of the above derivation is equivalent to the conclusion of the lemma except for typing contexts.

  Finally, we must show that $(\Gamma + r \cdot \Delta + \Gamma') = \Gamma'' + [\Delta_1 + \Delta + \Delta_2]_0$.

  $$\begin{aligned}(\Gamma + r \cdot \Delta + \Gamma') &= (\Gamma''_{|[\Delta_2]_0} + [\Delta_1]_0) + [\Delta]_0 + (\Gamma''_{\overline{|[\Delta_2]_0}} + [\Delta_2]_0)\\ &\qquad (\because r = 0 \,\&\, \Gamma = (\Gamma''_{|[\Delta_2]_0} + [\Delta_1]_0) \,\&\, \Gamma' = (\Gamma''_{\overline{|[\Delta_2]_0}} + [\Delta_2]_0))\\ &= (\Gamma''_{|[\Delta_2]_0} + \Gamma''_{\overline{|[\Delta_2]_0}}) + ([\Delta_1]_0 + [\Delta]_0 + [\Delta_2]_0)\\ &\qquad (\because + \text{ associativity and commutativity})\\ &= \Gamma'' + ([\Delta_1]_0 + [\Delta]_0 + [\Delta_2]_0) && (\because \text{A.2.2})\\ &= \Gamma'' + [\Delta_1 + \Delta + \Delta_2]_0 && (\because \text{definition of } [\Gamma]_0)\end{aligned}$$

  Thus, we obtain the conclusion of the lemma.

- Suppose $(x : [A]_r) \notin [\Delta']_0$
  Let Γ_1 and Γ_2 be typing context such that $\Gamma'' = (\Gamma_1, x : [A]_r, \Gamma_2)$. The last typing derivation of (WEAK) is rewritten as follows:

$$\frac{(\Gamma_1, x : [A]_r, \Gamma_2) \vdash t : B}{(\Gamma_1, x : [A]_r, \Gamma_2) + [\Delta']_0 \vdash t : B} \text{(WEAK)}$$

  We then compare the typing context between the conclusion of the lemma and that of the that of above derivation as follows:

$$\begin{aligned}(\Gamma, x : [A]_r, \Gamma') &= (\Gamma_1, x : [A]_r, \Gamma_2) + [\Delta']_0 \\ &= (\Gamma_1 + ([\Delta']_0)_{|\Gamma_1}), x : [A]_r, (\Gamma_2 + ([\Delta']_0)_{|\overline{\Gamma_1}}) \qquad (\because \text{A.2.3 (1)})\end{aligned}$$

  By the commutativity of ",", we can take Γ and Γ' arbitrarily so that they satisfy the above equation. So here we know $\Gamma = (\Gamma_1 + ([\Delta']_0)_{|\Gamma_1})$ and $\Gamma' = (\Gamma_2 + ([\Delta']_0)_{|\overline{\Gamma_1}})$. We then apply the induction hypothesis to the premise of the last derivation and reapply (WEAK) as follows:

$$\frac{\Gamma_1 + r \cdot \Delta + \Gamma_2 \vdash [t'/x]t : B}{(\Gamma_1 + r \cdot \Delta + \Gamma_2) + [\Delta']_0 \vdash [t'/x]t : B} \text{(WEAK)}$$

  The conclusion of the above derivation is equivalent to the conclusion of the lemma except for the typing contexts. Finally, we must show that $(\Gamma + r \cdot \Delta + \Gamma') = (\Gamma_1 + r \cdot \Delta + \Gamma_2) + [\Delta']_0$.

$$\begin{aligned}(\Gamma + r \cdot \Delta + \Gamma') &= (\Gamma_1 + ([\Delta']_0)_{|\Gamma_1}) + r \cdot \Delta + (\Gamma_2 + ([\Delta']_0)_{|\overline{\Gamma_1}}) \\ &\qquad (\because \Gamma = (\Gamma_1 + ([\Delta']_0)_{|\Gamma_1}) \ \&\ \Gamma' = (\Gamma_2 + ([\Delta']_0)_{|\overline{\Gamma_1}})) \\ &= (\Gamma_1 + r \cdot \Delta + \Gamma_2) + ([\Delta']_{0|\Gamma_1} + [\Delta']_{0|\overline{\Gamma_1}}) \\ &\qquad (\because + \text{ associativity and commutativity}) \\ &= (\Gamma_1 + r \cdot \Delta + \Gamma_2) + [\Delta']_0 \qquad (\because \text{A.2.2})\end{aligned}$$

  Thus, we obtain the conclusion of the lemma.

$$\text{Case.} \qquad \frac{\Gamma'', y : B_1 \vdash t : B_2}{\Gamma'', y : [B_1]_1 \vdash t : B_2} \text{(DER)}$$

In this case, we know $(\Gamma'', y : [B_1]_1) = (\Gamma, x : [A]_r, \Gamma')$. There are two cases in which the versioned assumption $x : [A]_r$ is equivalent to $y : [B_1]_1$ and not equivalent to.

- Suppose $x : [A]_r = y : [B_1]_1$
  We know $x = y$, $A = B_1$, $r = 1$, $\Gamma = \Gamma''$, and $\Gamma' = \emptyset$. The last derivation is rewritten as follows:

$$\frac{\Gamma'', x : A \vdash t : B_2}{\Gamma'', x : [A]_1 \vdash t : B_2} \text{(DER)}$$

  We then apply Lemma 4.7.1 to the premise to obtain the following:

$$\Gamma'' + \Delta \vdash [t'/x]t : B_2$$

  Note that Δ is a versioned assumption by the assumption 1 and thus $\Gamma'' + \Delta = \Gamma'' + r \cdot \Delta$ where $r = 1$, we obtain the conclusion of the lemma.

- Suppose $x : [A]_r \neq y : [B_1]_1$
  Let Γ_1 and Γ_2 be typing contexts such that $\Gamma'' = (\Gamma_1, x : [A]_r, \Gamma'_1)$. The last derivation is rewritten as follows:

$$\frac{(\Gamma_1, x : [A]_r, \Gamma'_1), y : B_1 \vdash t : B_2}{(\Gamma_1, x : [A]_r, \Gamma'_1), y : [B_1]_1 \vdash t : B_2} \text{(DER)}$$

  We then apply the induction hypothesis to the premise of the last derivation and reapply (DER) to obtain the following:

$$\frac{(\Gamma + r \cdot \Delta + \Gamma''), y : B_1 \vdash [t'/x]t : B_2}{(\Gamma + r \cdot \Delta + \Gamma''), y : [B_1]_1 \vdash [t'/x]t : B_2} \text{(DER)}$$

  Since $y : [B_1]_1$ is desjoint with $\Gamma + r \cdot \Delta + \Gamma''$ and thus $((\Gamma + r \cdot \Delta + \Gamma''), y : [B_1]_1) = \Gamma + r \cdot \Delta + (\Gamma'', y : [B_1]_1)$, we obtain the conclusion of the lemma.

$$\text{Case.} \quad \frac{[\Gamma_1] \vdash t : B}{r' \cdot [\Gamma_1] \vdash [t] : \Box_{r'} B} \text{(PR)}$$

Let r'' be a version resouce and Γ'_1 and Γ''_1 be typing contexts such that $r'' \sqsubseteq r'$ and $[\Gamma_1] = [\Gamma'_1, x : [A]_{r''}, \Gamma''_1]$. The last derivation is rewritten as follows:

$$\frac{[\Gamma'_1, x : [A]_{r''}, \Gamma''_1] \vdash t : B}{r' \cdot [\Gamma'_1, x : [A]_{r''}, \Gamma''_1] \vdash [t] : \Box_{r'} B} \text{(PR)}$$

We then compare the conclusion of the lemma and the above conclusion.

$$\begin{aligned}
(\Gamma, x : [A]_r, \Gamma') &= r' \cdot [\Gamma_1] \\
&= r' \cdot [\Gamma'_1, x : [A]_{r''}, \Gamma''_1] && (\because [\Gamma_1] = [\Gamma'_1, x : [A]_{r''}, \Gamma''_1]) \\
&= r' \cdot [\Gamma'_1],\ x : [A]_{r'' \otimes r'},\ r' \cdot [\Gamma''_1] && (\because \cdot \text{ definition}) \\
&= r' \cdot [\Gamma'_1],\ x : [A]_{r'},\ r' \cdot [\Gamma''_1] && (\because r'' \sqsubseteq r')
\end{aligned}$$

By the commutativity of ",", we can take Γ and Γ' arbitrarily so that they satisfy the above equation. So here we know $\Gamma = (r' \cdot [\Gamma'_1])$ and $\Gamma' = (r' \cdot [\Gamma''_1])$.

We then apply the induction hypothesis to the premise of the last derivation and reapply (PR) to obtain the following:

$$\frac{[\Gamma'_1 + r'' \cdot \Delta + \Gamma''_1] \vdash [t'/x]t : B}{r' \cdot [\Gamma'_1 + r'' \cdot \Delta + \Gamma''_1] \vdash [[t'/x]t] : \Box_{r'} B} \text{(PR)}$$

where we use $[\Gamma'_1, x : [A]_{r''}, \Gamma''_1] = [\Gamma'_1], x : [A]_{r''}, [\Gamma''_1]$ and $[\Gamma'_1 + r'' \cdot \Delta + \Gamma''_1] = [\Gamma'_1] + r'' \cdot \Delta + [\Gamma''_1]$ before applying (PR).

Since $[[t'/x]t] = [t'/x][t]$ by the definiton of substitution, the above conclusion is equivalent to the conclusion of the lemma except for the typing contexts. Finally, we must show that $(\Gamma + r' \cdot \Delta + \Gamma') = r' \cdot [\Gamma'_1 + r'' \cdot \Delta + \Gamma''_1]$ by the following reasoning:

$$\begin{aligned}
(\Gamma + r' \cdot \Delta + \Gamma') &= r' \cdot [\Gamma'_1] + r' \cdot \Delta + r' \cdot [\Gamma'_1] && (\because \Gamma = (r' \cdot [\Gamma'_1]) \ \&\ \Gamma' = (r' \cdot [\Gamma''_1])) \\
&= r' \cdot [\Gamma'_1] + (r' \otimes r'') \cdot \Delta + r' \cdot [\Gamma'_1] && (\because r'' \sqsubseteq r') \\
&= r' \cdot [\Gamma'_1] + r' \cdot (r'' \cdot \Delta) + r' \cdot [\Gamma'_1] && (\because \otimes \text{ associativity}) \\
&= r' \cdot [\Gamma'_1 + r'' \cdot \Delta + \Gamma'_1] && (\because \cdot \text{ distributive law over } +)
\end{aligned}$$

Thus, we obtain the conclusion of the lemma.

$$\text{Case.} \quad \frac{[\Gamma_i] \vdash t_i : B}{\bigcup_i (\{l_i\} \cdot [\Gamma_i]) \vdash \{\overline{l = t} \mid l_i\} : \Box_{\{\bar{l}\}} B} \text{(VER)}$$

We compare the typing contexts between the lemma and the above conclusion as follows:

$$\begin{aligned}
(\Gamma, x : [A]_r, \Gamma') &= \bigcup_i (\{l_i\} \cdot [\Gamma_i]) \\
&= \bigcup_{i \in I_x} (\{l_i\} \cdot [\Gamma'_i,\ x : [A]_{r_i},\ \Gamma''_i]) + \bigcup_{i \in J_x} (\{l_i\} \cdot [\Gamma'_i,\ \Gamma''_i]) \\
&\qquad (\because I_x = \{i \mid x \in \mathrm{dom}(\Gamma_i)\} \text{ and } J_x = \{i \mid x \notin \mathrm{dom}(\Gamma_i)\})
\end{aligned}$$

We then reorganise the typing context $\bigcup_{i \in I_x} (\{l_i\} \cdot [\Gamma'_i,\ x : [A]_{r_i},\ \Gamma''_i])$ as follows:

$$\begin{aligned}
& \bigcup_{i \in I_x} (\{l_i\} \cdot [\Gamma'_i,\ x : [A]_{r_i},\ \Gamma''_i]) \\
&= \bigcup_{i \in I_x} (\{l_i\} \cdot [x : [A]_{r_i},\ \Gamma'_i,\ \Gamma''_i]) && (\because \text{, associativity}) \\
&= \bigcup_{i \in I_x} (\{l_i\} \cdot (x : [A]_{r_i}),\ \{l_i\} \cdot [\Gamma'_i],\ \{l_i\} \cdot [\Gamma''_i]) && (\because \cdot \text{ distributive law}) \\
&= \bigcup_{i \in I_x} (\{l_i\} \cdot (x : [A]_{r_i})),\ \bigcup_{i \in I_x} (\{l_i\} \cdot [\Gamma'_i], \{l_i\} \cdot [\Gamma''_i]) \\
& && (\because \text{Sum of each disjoint sub context}) \\
&= \bigcup_{i \in I_x} (x : [A]_{\{l_i\} \otimes r_i}),\ \bigcup_{i \in I_x} (\{l_i\} \cdot [\Gamma'_i], \{l_i\} \cdot [\Gamma''_i]) && (\because \cdot \text{ definition}) \\
&= x : [A]_{\sum_{i \in I_x} \{l_i\} \otimes r_i},\ \bigcup_{i \in I_x} (\{l_i\} \cdot [\Gamma'_i], \{l_i\} \cdot [\Gamma''_i]) && (\because \textstyle\bigcup \text{ and } + \text{ definition})
\end{aligned}$$

Thus, we obtain the following:

$$\begin{aligned}
& (\Gamma, x : [A]_r, \Gamma') \\
&= \left(x : [A]_{\sum_{i \in I_x} \{l_i\} \otimes r_i},\ \bigcup_{i \in I_x} (\{l_i\} \cdot [\Gamma'_i], \{l_i\} \cdot [\Gamma''_i]) \right) + \bigcup_{i \in J_x} (\{l_i\} \cdot [\Gamma'_i,\ \Gamma''_i]) \\
&= x : [A]_{\sum_{i \in I_x} \{l_i\} \otimes r_i},\ \bigcup_i (\{l_i\} \cdot [\Gamma'_i], \{l_i\} \cdot [\Gamma''_i]) \\
&\qquad (\because \textstyle\bigcup_{i \in J_x} (\{l_i\} \cdot [\Gamma'_i,\ \Gamma''_i]) \text{ are disjoint with } x : [A]_{\sum_{i \in I_x} \{l_i\} \otimes r_i})
\end{aligned}$$

Therefore, by A.2.4, there exists typing contexts $\Gamma'_{\bar{i}}$ and $\Gamma''_{\bar{i}}$ such that:

$$\begin{aligned}
\Gamma'_{\bar{i}}, \Gamma''_{\bar{i}} &= \bigcup_i (\{l_i\} \cdot [\Gamma'_i], \{l_i\} \cdot [\Gamma''_i]) \\
\Gamma'_{\bar{i}} + \Gamma''_{\bar{i}} &= \bigcup_i (\{l_i\} \cdot [\Gamma'_i] + \{l_i\} \cdot [\Gamma''_i])
\end{aligned}$$

Thus, we obtain the following:

$$\begin{aligned}
(\Gamma, x : [A]_r, \Gamma') &= x : [A]_{\sum_{i \in I_x} \{l_i\} \otimes r_i},\ \Gamma'_{\bar{i}},\ \Gamma''_{\bar{i}} \\
&= \Gamma'_{\bar{i}},\ x : [A]_{\sum_{i \in I_x} \{l_i\} \otimes r_i},\ \Gamma''_{\bar{i}} && (\because \text{, commutativity})
\end{aligned}$$

By the commutativity of ",", we can take Γ and Γ' arbitrarily so that they satisfy the above equation. So here we know $\Gamma = \Gamma'_{\bar{i}}$, $\Gamma' = \Gamma''_{\bar{i}}$, and $r = \sum_{i \in I_x}(\{l_i\} \otimes r_i)$. We then apply the induction hypothesis to the premise whose typing context contains x. Here, we define a typing context Δ_i as follows:

$$\Delta_i = \begin{cases} \Delta & (i \in I_x) \\ \emptyset & (i \in J_x) \end{cases}$$

By using Δ_i, we reapply (VER) as follows:

$$\frac{[\Gamma'_i + r_i \cdot \Delta_i + \Gamma''_i] \vdash [t'/x]t_i : B}{\bigcup_i(\{l_i\} \cdot [\Gamma'_i + r_i \cdot \Delta_i + \Gamma''_i]) \vdash \{\overline{l = [t'/x]t} \,|\, l_i\} : \Box_{\{\bar{l}\}} B} \text{(VER)}$$

Since $\{\overline{l_i = [t'/x]t_i} \,|\, l_k\} = [t'/x]\{\overline{l_i = t_i} \,|\, l_k\}$ by the definition of substitution, the above conclusion is equivalent to the conclusion of the lemma except for typing contexts. Finally, we must show that $(\Gamma + r \cdot \Delta + \Gamma') = \bigcup_i(\{l_i\} \cdot [\Gamma'_i + r_i \cdot \Delta_i + \Gamma''_i])$.

$$\begin{aligned}
(\Gamma + r \cdot \Delta + \Gamma') &= \Gamma'_{\bar{i}} + r \cdot \Delta + \Gamma''_{\bar{i}} && (\because \Gamma = \Gamma'_{\bar{i}} \,\&\, \Gamma' = \Gamma''_{\bar{i}}) \\
&= r \cdot \Delta + (\Gamma'_{\bar{i}} + \Gamma''_{\bar{i}}) && (+\text{ associativity \& commutativity}) \\
&= r \cdot \Delta + \bigcup_i \left(\{l_i\} \cdot [\Gamma'_i] + \{l_i\} \cdot [\Gamma''_i]\right) \\
& && (\because \Gamma'_{\bar{i}} + \Gamma''_{\bar{i}} = \textstyle\bigcup_i \left(\{l_i\} \cdot [\Gamma'_i] + \{l_i\} \cdot [\Gamma''_i]\right)) \\
&= \Big(\sum_{i \in I_x}(\{l_i\} \otimes r_i)\Big) \cdot \Delta + \bigcup_i \left(\{l_i\} \cdot [\Gamma'_i] + \{l_i\} \cdot [\Gamma''_i]\right) \\
& && (\because r = \textstyle\sum_{i \in I_x}(\{l_i\} \otimes r_i)) \\
&= \bigcup_{i \in I_x} (\{l_i\} \cdot (r_i \cdot \Delta)) + \bigcup_i \left(\{l_i\} \cdot [\Gamma'_i] + \{l_i\} \cdot [\Gamma''_i]\right) && (\because \textstyle\bigcup \text{ definition}) \\
&= \bigcup_i (\{l_i\} \cdot (r_i \cdot \Delta_i)) + \bigcup_i \left(\{l_i\} \cdot [\Gamma'_i] + \{l_i\} \cdot [\Gamma''_i]\right) && (\because \Delta_i \text{ definition}) \\
&= \bigcup_i (\{l_i\} \cdot (r_i \cdot \Delta_i) + \{l_i\} \cdot [\Gamma'_i] + \{l_i\} \cdot [\Gamma''_i]) \\
& && (\because +\text{ commutativity \& associativity}) \\
&= \bigcup_i \left(\{l_i\} \cdot \left((r_i \cdot \Delta_i) + [\Gamma'_i] + [\Gamma''_i]\right)\right) && (\because \text{districutive law}) \\
&= \bigcup_i \left(\{l_i\} \cdot \left([\Gamma'_i] + (r_i \cdot \Delta_i) + [\Gamma''_i]\right)\right) && (\because +\text{ commutativity}) \\
&= \bigcup_i \left(\{l_i\} \cdot [\Gamma'_i + r_i \cdot \Delta_i + \Gamma''_i]\right) && (\because [\cdot] \text{ definition})
\end{aligned}$$

Thus, we obtain the conclusion of the lemma.

Case. $$\frac{[\Gamma_i] \vdash t_i : B}{\bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash \langle \overline{l = t} | l_i \rangle : B} \text{(VERI)}$$

This case is similar to the case of (VER).

Case. $$\frac{\Gamma \vdash t : \Box_r A \quad l \in r}{\Gamma \vdash t.l : A} \text{(EXTR)}$$

In this case, we apply the induction hypothesis to the premise and then reapply (EXTR), we obtain the conclusion of the lemma.

$$\text{Case.}\quad \frac{\Gamma_1, y : [B']_{r_1}, \Gamma_2 \vdash t : B \quad r_1 \sqsubseteq r_2}{\Gamma_1, y : [B']_{r_2}, \Gamma_2 \vdash t : B}\ (\text{SUB})$$

In this case, we know $(\Gamma, x : [A]_r, \Gamma') = (\Gamma, y : [B']_{r_2}, \Gamma')$. There are three cases where the versioned assumption $x : [A]_r$ is included in Γ_1, included in Γ_2, or equal to $y : [B']_{r_2}$.

- Suppose $(x : [A]_r) \in \Gamma_1$.
  Let Γ_1' and Γ_1'' be typing contexts such that $\Gamma_1 = (\Gamma_1', x : [A]_r, \Gamma_1'')$. The last derivation is rewritten as follows:

$$\frac{\Gamma_1', x : [A]_r, \Gamma_1'', y : [B']_{r_1}, \Gamma_2 \vdash t : B \quad r_1 \sqsubseteq r_2}{\Gamma_1', x : [A]_r, \Gamma_1'', y : [B']_{r_2}, \Gamma_2 \vdash t : B}\ (\text{SUB})$$

  We then apply the induction hypothesis to the premise of the last derivation to obtain the following:

$$\Gamma_1' + r \cdot \Delta + (\Gamma_1'', y : [B']_{r_1}, \Gamma_2) \vdash [t'/x]t : B \tag{A.1}$$

  The typing context of the above conclusion can be transformed as follows:

$$\begin{aligned}
&\Gamma_1' + r \cdot \Delta + (\Gamma_1'', y : [B']_{r_1}, \Gamma_2) \\
&= (\Gamma_1' + (r \cdot \Delta)_{|\Gamma_1'}), (r \cdot \Delta)_{|\overline{(\Gamma_1', (\Gamma_1'', y:[B']_{r_1}, \Gamma_2))}}, \\
&\qquad \Big((\Gamma_1'', y : [B']_{r_1}, \Gamma_2) + (r \cdot \Delta)_{|(\Gamma_1'', y:[B']_{r_1}, \Gamma_2)}\Big) && (\because \text{A.2.6}) \\
&= (\Gamma_1' + (r \cdot \Delta)_{|\Gamma_1'}), (r \cdot \Delta)_{|\overline{(\Gamma_1', (\Gamma_1'', y:[B']_{r_1}, \Gamma_2))}}, \\
&\qquad \Big((\Gamma_1'', y : [B']_{r_1}, \Gamma_2) + \Big((r \cdot \Delta)_{|\Gamma_1''}, (r \cdot \Delta)_{|(y:[B']_{r_1})}, (r \cdot \Delta)_{|\Gamma_2}\Big)\Big) \\
&&& (\because \text{A.2.1}) \\
&= (\Gamma_1' + (r \cdot \Delta)_{|\Gamma_1'}), (r \cdot \Delta)_{|\overline{(\Gamma_1', (\Gamma_1'', y:[B']_{r_1}, \Gamma_2))}}, \\
&\qquad \Big(\Gamma_1'' + (r \cdot \Delta)_{|\Gamma_1''}\Big), \Big(y : [B']_{r_1} + (r \cdot \Delta)_{|(y:[B']_{r_1})}\Big), (\Gamma_2 + (r \cdot \Delta)_{|\Gamma_2}) \\
&&& (\because \text{A.2.3}) \\
&= \Gamma_3, y : [B']_{r_1 \oplus r_3}, \Gamma_3'
\end{aligned}$$

  The last equational transformation holds by the following equation A.2.

  Let Γ_3 and Γ_3' be typing contexts that satisfy the following:

$$\begin{aligned}
\Gamma_3 &= (\Gamma_1' + (r \cdot \Delta)_{|\Gamma_1'}), (r \cdot \Delta)_{|\overline{(\Gamma_1', (\Gamma_1'', y:[B']_{r_1}, \Gamma_2))}}, \Big(\Gamma_1'' + (r \cdot \Delta)_{|\Gamma_1''}\Big) \\
\Gamma_3' &= \big(\Gamma_2 + (r \cdot \Delta)_{|\Gamma_2}\big)
\end{aligned}$$

  For $(r \cdot \Delta)_{|(y:[B']_{r_1})}$, Let r_3 and r_3' be typing contexts such that $r_3 = r \otimes r_3'$ and stisfy the following:

$$(r \cdot \Delta)_{|(y:[B']_{r_1})} = \begin{cases} r \cdot (y : [B']_{r_3'}) = y : [B']_{r \otimes r_3'} = y : [B']_{r_3} & (y \in \text{dom}(\Delta)) \\ \varnothing & (y \notin \text{dom}(\Delta)) \end{cases}$$

Thus, we obtain the following equation.

$$y : [B']_{r_1} + (r \cdot \Delta)_{|(y:[B']_{r_1})} = \begin{cases} y : [B']_{r_1 \oplus r_3} & (y \in \mathrm{dom}(\Delta)) \\ y : [B']_{r_1 \oplus r_3} = y : [B']_{r_1} & (y \notin \mathrm{dom}(\Delta)) \end{cases} \quad \text{(A.2)}$$

Applying all of the above transformations and reapplying (SUB) to the expression A.1, we obtain the following:

$$\frac{\Gamma_3, y : [B']_{r_1 \oplus r_3}, \Gamma_3' \vdash [t'/x]t : B \quad (r_1 \oplus r_3) \sqsubseteq (r_2 \oplus r_3)}{\Gamma_3, y : [B']_{r_2 \oplus r_3}, \Gamma_3' \vdash [t'/x]t : B} \text{(SUB)}$$

The conclusion of the above derivation is equivalent to the conclusion of the lemma except for the typing contexts.

Finally, we must show that $\Gamma_1' + r \cdot \Delta + (\Gamma_1'', y : [B']_{r_2}, \Gamma_2) = (\Gamma_3, y : [B']_{r_2 \oplus r_3}, \Gamma_3')$.

$$\begin{aligned}
&\Gamma_1' + r \cdot \Delta + (\Gamma_1'', y : [B']_{r_2}, \Gamma_2) \\
&= (\Gamma_1' + (r \cdot \Delta)_{|\Gamma_1'}), (r \cdot \Delta)_{|\overline{(\Gamma_1', (\Gamma_1'', y:[B']_{r_2}, \Gamma_2))}}, \\
&\qquad \Big((\Gamma_1'', y : [B']_{r_2}, \Gamma_2) + (r \cdot \Delta)_{|(\Gamma_1'', y:[B']_{r_2}, \Gamma_2)}\Big) && (\because \text{A.2.6}) \\
&= (\Gamma_1' + (r \cdot \Delta)_{|\Gamma_1'}), (r \cdot \Delta)_{|\overline{(\Gamma_1', (\Gamma_1'', y:[B']_{r_2}, \Gamma_2))}}, \\
&\qquad \Big((\Gamma_1'', y : [B']_{r_2}, \Gamma_2) + \Big((r \cdot \Delta)_{|\Gamma_1''}, (r \cdot \Delta)_{|(y:[B']_{r_2})}, (r \cdot \Delta)_{|\Gamma_2}\Big)\Big) \\
&&& (\because \text{A.2.1}) \\
&= (\Gamma_1' + (r \cdot \Delta)_{|\Gamma_1'}), (r \cdot \Delta)_{|\overline{(\Gamma_1', (\Gamma_1'', y:[B']_{r_2}, \Gamma_2))}}, \\
&\qquad \Big(\Gamma_1'' + (r \cdot \Delta)_{|\Gamma_1''}\Big), \Big(y : [B']_{r_2} + (r \cdot \Delta)_{|(y:[B']_{r_2})}\Big), (\Gamma_2 + (r \cdot \Delta)_{|\Gamma_2}) \\
&&& (\because \text{A.2.3}) \\
&= \Gamma_3, y : [B']_{r_2 \oplus r_3}, \Gamma_3'
\end{aligned}$$

The last transformation is based on the following equation that can be derived from the definition A.2.1.

$$(r \cdot \Delta)_{|\overline{(\Gamma_1', (\Gamma_1'', y:[B']_{r_1}, \Gamma_2))}} = (r \cdot \Delta)_{|\overline{(\Gamma_1', (\Gamma_1'', y:[B']_{r_2}, \Gamma_2))}}$$
$$(r \cdot \Delta)_{|(y:[B']_{r_1})} = (r \cdot \Delta)_{|(y:[B']_{r_2})}$$

Thus, we obtain the conclusion of the lemma.

- Suppose $(x : [A]_r) \in \Gamma_2$.
  This case is similar to the case of $(x : [A]_r) \in \Gamma_1$.
- Suppose $(x : [A]_r) = y : [B']_{r_2}$.
  The last derivation is rewritten as follows:

$$\frac{\Gamma_1, x : [A]_{r'}, \Gamma_2 \vdash t : B \quad r' \sqsubseteq r}{\Gamma_1, x : [A]_r, \Gamma_2 \vdash t : B} \text{(SUB)}$$

We apply the induction hypothesis to the premise and then reapply (SUB), we obtain the conclusion of the lemma.

□

A.4 Type Safety

Lemma A.4.1 (Inversion lemma)**.** Let v be a value such that $\Gamma \vdash v : A$. The followings hold for a type A.

- $A = \mathsf{Int} \Longrightarrow v = n$ for some Int constant n.
- $A = \Box_r B \Longrightarrow v = [t']$ for some term t', or $v = \{\overline{l_i = t_i} \mid l_k\}$ for some terms t_i and some labels $l_i, l_k \in r$.
- $A = B \to B' \Longrightarrow v = \lambda p.t$ for some pattern p and term t.

Lemma 4.7.3. [Type safety for default version overwriting @]
For any version label l:

$$\Gamma \vdash t : A \quad \Longrightarrow \quad \Gamma \vdash t@l : A$$

Proof. The proof is given by induction on the typing derivation of $\Gamma \vdash t : A$. Consider the cases for the last rule used in the typing derivation of assumption.

$$\text{Case.} \quad \frac{}{\varnothing \vdash n : \mathsf{Int}}\ (\textsc{int})$$

This case holds trivially because $n@l \equiv n$ for any labels l.

$$\text{Case.} \quad \frac{}{x : A \vdash x : A}\ (\textsc{var})$$

This case holds trivially because $x@l = x$ for any labels l.

$$\text{Case.} \quad \frac{\Gamma, x : A_1 \ \vdash \ t_1 : A_2}{\Gamma \ \vdash \ \lambda x.t_1 : A_1 \to A_2}\ (\textsc{abs})$$

By induction hypothesis, there exists a term $t_1@l$ such that:

$$\Gamma, x : A_1 \ \vdash \ t_1@l : A_2$$

We then reapply (ABS) to obtain the following:

$$\frac{\Gamma, x : A_1 \ \vdash \ t_1@l : A_2}{\Gamma \ \vdash \ \lambda x.(t_1@l) : A_1 \to A_2}\ (\textsc{abs})$$

Thus, note that $(\lambda x.t_1)@l \equiv \lambda x.(t_1@l)$, we obtain the conclusion of the lemma.

$$\text{Case.} \quad \frac{\begin{array}{c}\Gamma_1 \vdash t_1 : B \to A \\ \Gamma_2 \vdash t_2 : B\end{array}}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : A}\ (\textsc{app})$$

By induction hypothesis, there exists terms $t_1@l$ and $t_2@l$ such that:

$$\Gamma_1 \vdash t_1@l : B \to A$$
$$\Gamma_2 \vdash t_2@l : B$$

We then reapply (APP) to obtain the following:

$$\frac{\begin{array}{c}\Gamma_1 \vdash t_1@l : B \to A \\ \Gamma_2 \vdash t_2@l : B\end{array}}{\Gamma_1 + \Gamma_2 \vdash (t_1@l)\,(t_2@l) : A}\ (\textsc{app})$$

Thus, note that $(t_1\, t_2)@l \equiv (t_1@l)\,(t_2@l)$, we obtain the conclusion of the lemma.

$$\text{Case.}\quad \frac{\Gamma_1 \vdash t_1 : \Box_r A \quad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\,[x] = t_1\ \textbf{in}\ t_2 : B}\ (\textsc{let})$$

By induction hypothesis, there exists terms $t_1@l$ and $2@l$ such that:

$$\Gamma_1 \vdash t_1@l : \Box_r A$$
$$\Gamma_2, x : [A]_r \vdash t_2@l : B$$

We then reapply (LET) to obtain the following:

$$\frac{\Gamma_1 \vdash t_1@l : \Box_r A \quad \Gamma_2, x : [A]_r \vdash t_2@l : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\,[x] = (t_1@l)\ \textbf{in}\ (t_2@l) : B}\ (\textsc{let})$$

Thus, note that $(\textbf{let}\,[x] = t_1\ \textbf{in}\ t_2)@l \equiv \textbf{let}\,[x] = (t_1@l)\ \textbf{in}\ (t_2@l)$, we obtain the conclusion of the lemma.

$$\text{Case.}\quad \frac{\Gamma_1 \vdash t : A}{\Gamma_1 + [\Delta']_0 \vdash t : A}\ (\textsc{weak})$$

By induction hypothesis, we know the following:

$$\Gamma_1 \vdash t@l : A$$

We then reapply (WEAK) to obtain the following:

$$\frac{\Gamma_1 \vdash t@l : A}{\Gamma_1 + [\Delta']_0 \vdash t@l : A}\ (\textsc{weak})$$

Thus, we obtain the conclusion of the lemma.

$$\text{Case.}\quad \frac{\Gamma_1, x : B \vdash t : A}{\Gamma_1, x : [B]_1 \vdash t : A}\ (\textsc{der})$$

By induction hypothesis, there exists terms $t@l$ such that:

$$\Gamma_1, x : B \vdash t@l : A$$

We then reapply (WEAK) to obtain the following:

$$\frac{\Gamma_1, x : B \vdash t@l : A}{\Gamma_1, x : [B]_1 \vdash t@l : A}\ (\textsc{der})$$

Thus, we obtain the conclusion of the lemma.

$$\text{Case.}\quad \frac{[\Gamma] \vdash t : B}{r \cdot [\Gamma] \vdash [t] : \Box_r B}\ (\textsc{pr})$$

This case holds trivially because $[t]@l \equiv [t]$ for any labels l.

$$\text{Case.}\quad \frac{[\Gamma_i] \vdash t_i : A}{\bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash \{\overline{l = t} \,|\, l'\} : \Box_{\{\bar{l}\}} A}\ (\textsc{ver})$$

This case holds trivially because $\{\overline{l = t} \,|\, l'\}@l \equiv \{\overline{l = t} \,|\, l'\}$ for any labels l.

$$\text{Case.}\quad \frac{[\Gamma_i] \vdash t_i : A}{\bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash \langle\overline{l = t} \,|\, l_k\rangle : A}\ (\textsc{veri})$$

In this case, there are two possibilities for the one step evaluation of t.

- Suppose $l \in \{\bar{l}\}$.
  We can apply the default version overwriting as follows:

  $$\frac{l \in \{\bar{l}\}}{\langle\overline{l = t} \,|\, l_k\rangle@l \ \equiv\ \langle\overline{l = t} \,|\, l\rangle}$$

  In this case, we can derive the type of $\langle\overline{l = t} \,|\, l\rangle$ as follows:

  $$\frac{[\Gamma_i] \vdash t_i : A}{\bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash \langle\overline{l = t} \,|\, l\rangle : A}\ (\textsc{veri})$$

  Thus, we obtain the conclusion of the lemma.

- Suppose $l \notin \{\bar{l}\}$.
  We can apply the default version overwriting as follows:

  $$\frac{l \notin \{\bar{l}\}}{\langle\overline{l = t} \,|\, l_k\rangle@l \ \equiv\ \langle\overline{l = t} \,|\, l_k\rangle}$$

  This case holds trivially because $\langle\overline{l = t} \,|\, l_k\rangle@l = \langle\overline{l = t} \,|\, l_k\rangle$.

$$\text{Case.}\quad \frac{\Gamma \vdash t_1 : \Box_r A \quad l_k \in r}{\Gamma \vdash t_1.l_k : A}\ (\textsc{extr})$$

By induction hypothesis, there exists a term $t_1@l$ such that:

$$\Gamma \vdash t_1@l : \Box_r A$$

We then reapply (EXTR) to obtain the following:

$$\frac{\Gamma \vdash t_1@l : \Box_r A \quad l_k \in r}{\Gamma \vdash (t_1@l).l_k : A}\ (\textsc{extr})$$

Thus, note that $(t_1.l_k)@l \equiv (t_1@l).l_k$, we obtain the conclusion of the lemma.

$$\text{Case.}\quad \frac{\Gamma_1, x : [B]_r, \Gamma_2 \vdash t : A \quad r \sqsubseteq s}{\Gamma_1, x : [B]_s, \Gamma_2 \vdash t : A}\ (\textsc{sub})$$

By induction hypothesis, there exists a term $t@l$ such that:

$$\Gamma_1, x : [B]_r, \Gamma_2 \vdash t@l : A$$

We then reapply (SUB) to obtain the following:

$$\frac{\Gamma_1, x : [B]_r, \Gamma_2 \vdash t@l : A \quad r \sqsubseteq s}{\Gamma_1, x : [B]_s, \Gamma_2 \vdash t@l : A} \text{ (SUB)}$$

Thus, we obtain the conclusion of the lemma.

□

Lemma 4.7.4. [Type-safe extraction for versioned values]

$$[\Gamma] \vdash v : \Box_r A \quad \Longrightarrow \quad \forall l_k \in r.\, \exists t'. \begin{cases} v.l_k \longrightarrow t' & (\mathit{progress}) \\ [\Gamma] \vdash t' : A & (\mathit{preservation}) \end{cases}$$

Proof. By inversion lemma (A.4.1), v has either a form $[t'']$ or $\{\overline{l = t} \,|\, l'\}$.

- Suppose $v = [t'']$.
  We can apply (E-EX1) as follows:

$$\frac{}{[t''].l_k \rightsquigarrow t''@l_k}\ (\text{E-EX1})$$

  Also, we get the following derivation for v.

$$\cfrac{\cfrac{\cfrac{[\Gamma'] \vdash t'' : A}{r \cdot [\Gamma'] \vdash [t''] : \Box_r A}\ (\text{PR})}{\vdots}\ (\text{WEAK})\text{ or }(\text{SUB})}{[\Gamma] \vdash [t''] : \Box_r A}\ (\text{WEAK})\text{ or }(\text{SUB})$$

  By 4.7.3, we know the following:

$$[\Gamma'] \vdash t''@l_k : A$$

  Finally, we can rearrange the typing context as follows:

$$\cfrac{\cfrac{[\Gamma'] \vdash t''@l_k : A}{\vdots}\ (\text{WEAK})\text{ or }(\text{SUB})}{[\Gamma] \vdash t''@l_k : A}\ (\text{WEAK})\text{ or }(\text{SUB})$$

  Here, we follow the same manner as for the derivation of $[t'']$ (which may use (WEAK) and (SUB)) to get $[\Gamma]$ from $r \cdot [\Gamma']$.

  Thus, we obtain the conclusion of the lemma.

- Suppose $v = \{\overline{l = t} \,|\, l'\}$.
  We can apply (E-EX2) as follows:

$$\frac{}{\{\overline{l = t} \,|\, l'\}.l_k \rightsquigarrow t_k@l_k}\ (\text{E-EX2})$$

  Also, we get the following derivation for v.

$$\cfrac{\cfrac{\cfrac{[\Gamma'_i] \vdash t_i : A}{\bigcup_i(\{l_i\} \cdot [\Gamma'_i]) \vdash \{\overline{l = t} \,|\, l'\} : \Box_{\{\bar{l}\}} A}\ (\text{VER})}{\vdots}\ (\text{WEAK})\text{ or }(\text{SUB})}{[\Gamma] \vdash \{\overline{l = t} \,|\, l'\} : \Box_{\{\bar{l}\}} A}\ (\text{WEAK})\text{ or }(\text{SUB})$$

By 4.7.3, we know the following:

$$[\Gamma'_k] \vdash t_k@l_k : A$$

Finally, we can rearrange the typing context as follows:

$$\dfrac{[\Gamma'_k] \vdash t_k@l_k : A \qquad r_{kj} \sqsubseteq r_{kj} \otimes \{l_k\}}{\underbrace{\{l_k\} \cdot [\Gamma'_k] \vdash t_k@l_k : A}_{P}} \text{(SUB)} * |\Gamma'_k|$$

$$\dfrac{\dfrac{\dfrac{P \qquad r_{kj} \otimes \{l_k\} \sqsubseteq \sum_i (r_{ij} \otimes \{l_i\})}{\bigcup_i (\{l_i\} \cdot [\Gamma'_i]) \vdash t_k@l_k : A} \text{(SUB)} * |\Gamma'_k|}{\vdots} \text{(WEAK) or (SUB)}}{[\Gamma] \vdash t_k@l_k : A} \text{(WEAK) or (SUB)}$$

Here in the multiple application of (SUB), the second premise compares the resources of j-th versioned assumption between the first premise and conclusion. Also, we follow the same manner as for the derivation of $\{\overline{l = t} \mid l'\}$ (which may use (WEAK) and (SUB)) to get $[\Gamma]$ from $\bigcup_i (\{l_i\} \cdot [\Gamma'_i])$.

Thus, we obtain the conclusion of the lemma.

□

Theorem 4.7.5. [Type preservation for reductions]

$$\left.\begin{array}{l}\Gamma \vdash t : A \\ t \rightsquigarrow t'\end{array}\right\} \implies \Gamma \vdash t' : A$$

Proof. The proof is given by induction on the typing derivation of t. Consider the cases for the last rule used in the typing derivation of the first assumption.

$$\text{Case.} \quad \frac{}{\varnothing \vdash n : \mathsf{Int}}\ (\textsc{int})$$

This case holds trivially because there are no reduction rules that can be applied to n.

$$\text{Case.} \quad \frac{}{x : A \vdash x : A}\ (\textsc{var})$$

This case holds trivially because there are no reduction rules that can be applied to x.

$$\text{Case.} \quad \frac{\Gamma, x : A_1 \vdash t_1 : A_2}{\Gamma \vdash \lambda x.t_1 : A_1 \to A_2}\ (\textsc{abs})$$

This case holds trivially because there are no reduction rules that can be applied to $\lambda x.t_1$.

$$\text{Case.} \quad \frac{\begin{array}{c}\Gamma_1 \vdash t_1 : B \to A \\ \Gamma_2 \vdash t_2 : B\end{array}}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : A}\ (\textsc{app})$$

The only reduction rule we can apply to t is (E-ABS).

$$\frac{}{\underbrace{(\lambda x.t_1')\, t_2}_{t} \rightsquigarrow (t_2 \rhd x)\, t_1'}\ (\text{E-}\textsc{abs})$$

where $t_1 = \lambda x.t_1'$ for a term t_1'. Then we can apply $(\rhd_{\mathrm{var}})$ to obtain the following:

$$\frac{}{(t_2 \rhd x)\, t_1' = [t_2/x]t_1'}\ (\rhd_{\textsc{var}})$$

In this case, we know the typing derivation of t has the following form:

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1', x : B \vdash t_1' : A}{\Gamma_1' \vdash \lambda x.t_1' : B \to A}\ (\textsc{abs})}{}\ (\textsc{weak}), (\textsc{der}), \text{or } (\textsc{sub})}{\vdots}}{\Gamma_1 \vdash \lambda x.t_1' : B \to A}\ (\textsc{weak}), (\textsc{der}), \text{or } (\textsc{sub}) \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash (\lambda x.t_1')\, t_2 : A}\ (\textsc{app})$$

By 4.7.1, we know the following:

$$\left.\begin{array}{r}\Gamma_2 \vdash t_2 : B \\ \Gamma_1', x : B \vdash t_1' : A\end{array}\right\} \implies \Gamma_1' + \Gamma_2 \vdash [t_2/x]t_1' : A$$

Finally, we can rearrange the typing context as follows:

$$\cfrac{\cfrac{\Gamma_1' + \Gamma_2 \vdash [t_2/x]t_1' : A}{\vdots}\ (\text{WEAK}), (\text{DER}), \text{or } (\text{SUB})}{\Gamma_1 + \Gamma_2 \vdash [t_2/x]t_1' : A}\ (\text{WEAK}), (\text{DER}), \text{or } (\text{SUB})$$

Here, there exists a derive tree to get $\Gamma_1 + \Gamma_2$ from $\Gamma_1' + \Gamma_2$ as for the derivation of $\lambda x.t_1'$ which may use (WEAK), (DER) and (SUB).

By choosing $t' = [t_2/x]t_1'$, we obtain the conclusion of the theorem.

$$\text{Case.}\quad \frac{\Gamma_1 \vdash t_1 : \Box_r A \quad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let } [x] = t_1 \textbf{ in } t_2 : B}\ (\text{LET})$$

The only reduction rule we can apply is (E-CLET) with two substitution rules, depending on whether t_1 has the form $[t_1']$ or $\{\overline{l = t''} \mid l_k\}$.

- Suppose $t_1 = [t_1']$.
  We can apply (E-CLET) to obtain the following.

  $$\frac{}{\underbrace{\textbf{let } [x] = [t_1'] \textbf{ in } t_2}_{t} \rightsquigarrow ([t_1'] \rhd [x])t_2}\ (\text{E-CLET})$$

  Thus, we can apply ($\rhd_\Box$) and ($\rhd_{\text{var}}$) to obtain the following.

  $$\cfrac{\cfrac{}{(t_1' \rhd x)t_2 = [t_1'/x]t_2}\ (\rhd_{\text{var}})}{([t_1'] \rhd [x])t_2 = [t_1'/x]t_2}\ (\rhd_\Box)$$

  In this case, we know the typing derivation of t has the following form:

  $$\cfrac{\cfrac{\cfrac{\cfrac{[\Gamma_1'] \vdash t_1' : A}{r \cdot [\Gamma_1'] \vdash [t_1'] : \Box_r A}\ (\text{PR})}{\vdots}\ (\text{WEAK}) \text{ or } (\text{SUB})}{\Gamma_1 \vdash [t_1'] : \Box_r A}\ (\text{WEAK}) \text{ or } (\text{SUB}) \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let } [x] = [t_1'] \textbf{ in } t_2 : B}\ (\text{LET})$$

  By 4.7.2, we know the following:

  $$\left.\begin{array}{r}[\Gamma_1'] \vdash t_1' : A \\ \Gamma_2, x : [A]_r \vdash t_2 : B\end{array}\right\} \implies \Gamma_2 + r \cdot [\Gamma_1'] \vdash [t_1'/x]t_2 : B$$

  Finally, we can rearrange the typing context as follows:

$$\dfrac{\dfrac{\Gamma_2 + r \cdot [\Gamma_1'] \;\vdash\; [t_1'/x]t_2 : B}{\vdots}\;(\text{WEAK})\text{ or }(\text{SUB})}{\Gamma_2 + \Gamma_1 \;\vdash\; [t_1'/x]t_2 : B}\;(\text{WEAK})\text{ or }(\text{SUB})$$

Here, there exists a derive tree to get $\Gamma_2 + \Gamma_1$ from $\Gamma_2 + r \cdot [\Gamma_1']$ as for the derivation of $[t_1']$ which may use (WEAK) and (SUB).

Thus, by choosing $t' = [t_1'/x]t_2$, we obtain the conclusion of the theorem.

- Suppose $t_1 = \{\overline{l = t''} \mid l_k\}$.
  We can apply (E-CLET) to obtain the following:

$$\dfrac{}{\underbrace{\textbf{let}\ [x] = [\{\overline{l = t''} \mid l_k\}\ \textbf{in}\ t_2}_{t} \rightsquigarrow (\{\overline{l = t''} \mid l_k\} \rhd [x])t_2}\;(\text{E-CLET})$$

Thus, we can apply ($\rhd_{\text{ver}}$) and ($\rhd_{\text{var}}$) to obtain the following.

$$\dfrac{\dfrac{}{(\langle\overline{l = t''} \mid l_k\rangle \rhd x)t_2 = [\langle\overline{l = t''} \mid l_k\rangle/x]t_2}\;(\rhd_{\text{var}})}{(\{\overline{l = t''} \mid l_k\} \rhd [x])t_2 = [\langle\overline{l = t''} \mid l_k\rangle/x]t_2}\;(\rhd_{\text{ver}})$$

In this case, we know the typing derivation of t has the following form:

$$\underbrace{\dfrac{\dfrac{\dfrac{[\Gamma_i'] \vdash t_i'' : A}{\bigcup_i(\{l_i\} \cdot [\Gamma_i']) \;\vdash\; \{\overline{l = t''} \mid l_k\} : \Box_{\{\bar{l}\}}A}\;(\text{VER})}{\vdots}\;(\text{WEAK})\text{ or }(\text{SUB})}{\Gamma_1 \;\vdash\; \{\overline{l = t''} \mid l_k\} : \Box_{\{\bar{l}\}}A}\;(\text{WEAK})\text{ or }(\text{SUB})}_{P}$$

$$\dfrac{P \qquad \dfrac{\dfrac{\dfrac{\Gamma_2, x : A \;\vdash\; t_2 : B}{\Gamma_2, x : [A]_1 \;\vdash\; t_2 : B}\;(\text{DER})}{\Gamma_2, x : [A]_{\{\bar{l}\}} \;\vdash\; t_2 : B}\;(\text{SUB})*|\{\bar{l}\}|}{}}{\Gamma_1 + \Gamma_2 \;\vdash\; \textbf{let}\ [x] = \{\overline{l = t''} \mid l_k\}\ \textbf{in}\ t_2 : B}\;(\text{LET})$$

Then we can derive the type of $\langle\overline{l = t''} \mid l_k\rangle$ as follows:

$$\dfrac{[\Gamma_i'] \;\vdash\; t_i'' : A}{\bigcup_i(\{l_i\} \cdot [\Gamma_i']) \;\vdash\; \langle\overline{l = t''} \mid l_k\rangle : A}\;(\text{VERI})$$

By 4.7.1, we know the following:

$$\left.\begin{array}{r}\bigcup_i(\{l_i\} \cdot [\Gamma_i']) \vdash \langle\overline{l = t''} \mid l_k\rangle : A \\ \Gamma_2, x : A \vdash t_2 : B\end{array}\right\} \implies \begin{array}{l}\Gamma_2 + \bigcup_i(\{l_i\} \cdot [\Gamma_i']) \\ \qquad \vdash [\langle\overline{l = t''} \mid l_k\rangle/x]t_2 : B\end{array}$$

Finally, we can rearrange the typing context as follows:

$$\cfrac{\cfrac{\Gamma_2 + \bigcup_i(\{l_i\} \cdot [\Gamma'_i]) \vdash [\langle \overline{l = t''} \mid l_k\rangle / x]t_2 : B}{\vdots}\ (\text{WEAK}) \text{ or } (\text{SUB})}{\Gamma_2 + \Gamma_1 \vdash [\langle \overline{l = t''} \mid l_k\rangle / x]t_2 : B}\ (\text{WEAK}) \text{ or } (\text{SUB})$$

Here, there exists a derive tree to get $\Gamma_2 + \Gamma_1$ from $\Gamma_2 + \bigcup_i(\{l_i\} \cdot [\Gamma'_i])$ as for the derivation of $\{\overline{l = t''} \mid l_k\}$ which may use (WEAK) and (SUB).

Thus, by choosing $t' = [\langle \overline{l = t''} \mid l_k\rangle / x]t_2$, we obtain the conclusion of the theorem.

Case. $$\frac{\Gamma_1 \vdash t : A}{\Gamma_1 + [\Delta']_0 \vdash t : A}\ (\text{WEAK})$$

In this case, t does not change between before and after the last derivation. The induction hypothesis implies that there exists a term t'' such that:

$$t \rightsquigarrow t'' \ \wedge\ \Gamma_1 \vdash t'' : A \qquad \text{(ih)}$$

We then reapply (WEAK) to obtain the following:

$$\frac{\Gamma_1 \vdash t'' : A}{\Gamma_1 + [\Delta']_0 \vdash t'' : A}\ (\text{WEAK})$$

Thus, by choosing $t' = t''$, we obtain the conclusion of the theorem.

Case. $$\frac{\Gamma_1, x : B \vdash t : A}{\Gamma_1, x : [B]_1 \vdash t : A}\ (\text{DER})$$

In this case, t does not change between before and after the last derivation. The induction hypothesis implies that there exists a term t'' such that:

$$t \rightsquigarrow t'' \ \wedge\ \Gamma_1, x : B \vdash t'' : A \qquad \text{(ih)}$$

We then reapply (DER) to obtain the following:

$$\frac{\Gamma_1, x : B \vdash t'' : A}{\Gamma_1, x : [B]_1 \vdash t'' : A}\ (\text{DER})$$

Thus, by choosing $t' = t''$, we obtain the conclusion of the theorem.

Case. $$\frac{[\Gamma] \vdash t'' : B}{r \cdot [\Gamma] \vdash [t''] : \Box_r B}\ (\text{PR})$$

This case holds trivially because there are no reduction rules that can be applied to $[t'']$.

Case. $$\frac{[\Gamma_i] \vdash t_i : A}{\bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash \{\overline{l = t} \mid l'\} : \Box_{\{\bar{l}\}} A}\ (\text{VER})$$

This case holds trivially because there are no reduction rules that can be applied to $\{\overline{l = t} \mid l'\}$.

$$\text{Case.}\quad \frac{[\Gamma_i] \vdash t_i : A}{\bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash \langle \overline{l = t} \mid l_k \rangle : A}\ (\textsc{Veri})$$

In this case, the only reduction rule we can apply is (E-VERI).

$$\frac{}{\underbrace{\langle \overline{l = t} \mid l_k \rangle}_{t} \rightsquigarrow t_k@l_k}\ (\text{E-}\textsc{Veri})$$

By 4.7.3, we obtain the following:

$$[\Gamma_k] \vdash t_k : A \quad \Longrightarrow \quad [\Gamma_k] \vdash t_k@l_k : A$$

Finally, we can rearrange the typing context as follows:

$$\cfrac{\cfrac{[\Gamma_k] \vdash t_k@l_k : A}{\vdots}\ (\textsc{Weak}), (\textsc{Der}) \text{ or } (\textsc{Sub})}{\bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash t_k@l_k : A}\ (\textsc{Weak}), (\textsc{Der}) \text{ or } (\textsc{Sub})$$

Thus, by choosing $t' = t_k@l_k$, we obtain the conclusion of the theorem.

$$\text{Case.}\quad \frac{\Gamma \vdash t_1 : \Box_r A \quad l_k \in r}{\Gamma \vdash t_1.l_k : A}\ (\textsc{Extr})$$

In this case, there are two reduction rules that we can apply to t, dependenig on whether t_1 has the form $[t_1']$ or $\{\overline{l = t''} \mid l_k\}$.

- Suppose $t_1 = [t_1']$.
  We know the typing derivation of t has the following form:

$$\cfrac{\cfrac{\cfrac{\cfrac{[\Gamma'] \vdash t_1' : A}{r \cdot [\Gamma'] \vdash [t_1'] : \Box_r A}\ (\textsc{Pr})}{\vdots}\ (\textsc{Weak}) \text{ or } (\textsc{Sub})}{\Gamma \vdash [t_1'] : \Box_r A}\ (\textsc{Weak}) \text{ or } (\textsc{Sub}) \qquad l_k \in r}{\Gamma \vdash [t_1'].l_k : A}\ (\textsc{Extr})$$

  By 4.7.4, we know the following:

$$r \cdot [\Gamma'] \vdash [t_1'] : \Box_r A \quad \Longrightarrow \quad \exists t'. \begin{cases} [t_1'].l_k \longrightarrow t' \\ r \cdot [\Gamma'] \vdash t' : A \end{cases}$$

  Finally, we can rearrange the typing context as follows:

$$\cfrac{\cfrac{r \cdot [\Gamma'] \vdash t' : A}{\vdots}\ (\textsc{Weak}) \text{ or } (\textsc{Sub})}{\Gamma \vdash t' : A}\ (\textsc{Weak}) \text{ or } (\textsc{Sub})$$

  Here, we follow the same manner as for the derivation of $[t_1']$ (which may use (WEAK) and (SUB)) to get Γ from $r \cdot [\Gamma']$.

  Thus, we obtain the conclusion of the theorem.

- Suppose $t_1 = \{\overline{l = t} \,|\, l'\}$.
  The last derivation is rewritten as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\Gamma_i'] \vdash t_i : A}{\bigcup_i \{l_i\} \cdot [\Gamma_i'] \vdash \{\overline{l = t} \,|\, l'\} : \Box_{\{\bar{l}\}} A}\,(\textsc{pr})}{\vdots}\,(\textsc{weak})\text{ or }(\textsc{sub})}{\Gamma \vdash \{\overline{l = t} \,|\, l'\} : \Box_{\{\bar{l}\}} A}\,(\textsc{weak})\text{ or }(\textsc{sub}) \qquad l_k \in \{\bar{l}\}}{\Gamma \vdash \{\overline{l = t} \,|\, l'\}.l_k : A}\,(\textsc{extr})$$

By 4.7.4, we know the following:

$$\bigcup_i \{l_i\} \cdot [\Gamma_i'] \vdash \{\overline{l = t} \,|\, l'\} : \Box_{\{\bar{l}\}} A \quad \Longrightarrow \quad \exists t'. \begin{cases} \{\overline{l = t} \,|\, l'\}.l_k \longrightarrow t' \\ \bigcup_i \{l_i\} \cdot [\Gamma_i'] \vdash t' : A \end{cases}$$

Finally, we can rearrange the typing context as follows:

$$\dfrac{\dfrac{\bigcup_i \{l_i\} \cdot [\Gamma_i'] \vdash t' : A}{\vdots}\,(\textsc{weak})\text{ or }(\textsc{sub})}{\Gamma \vdash t' : A}\,(\textsc{weak})\text{ or }(\textsc{sub})$$

Here, we follow the same manner as for the derivation of $\{\overline{l = t} \,|\, l'\}$ (which may use (WEAK) and (SUB)) to get Γ from $\bigcup_i \{l_i\} \cdot [\Gamma_i']$.

Thus, we obtain the conclusion of the theorem.

$$\text{Case.} \quad \dfrac{\Gamma_1, x : [B]_r, \Gamma_2 \vdash t : A \quad r \sqsubseteq s}{\Gamma_1, x : [B]_s, \Gamma_2 \vdash t : A}\,(\textsc{sub})$$

In this case, t does not change between before and after the last derivation. The induction hypothesis implies that there exists a term t'' such that:

$$t \rightsquigarrow t'' \;\wedge\; \Gamma_1, x : [B]_r, \Gamma_2 \vdash t'' : A \qquad \text{(ih)}$$

We then reapply (SUB) to obtain the following:

$$\dfrac{\Gamma_1, x : [B]_r, \Gamma_2 \vdash t'' : A \qquad r \sqsubseteq s}{\Gamma_1, x : [B]_s, \Gamma_2 \vdash t'' : A}\,(\textsc{sub})$$

Thus, by choosing $t' = t''$, we obtain the conclusion of the theorem. □

Theorem 4.7.6. [Type preservation for evaluations]

$$\left.\begin{array}{l}\Gamma \vdash t : A \\ t \longrightarrow t'\end{array}\right\} \implies \Gamma \vdash t' : A$$

Proof. The proof is given by induction on the typing derivation of t. Consider the cases for the last rule used in the typing derivation of the first assumption.

Case. $$\frac{}{\emptyset \vdash n : \mathsf{Int}}\ (\textsc{Int})$$

This case holds trivially because there are no evaluation rules that can be applied to n.

Case. $$\frac{}{x : A \vdash x : A}\ (\textsc{Var})$$

This case holds trivially because there are no evaluation rules that can be applied to x.

Case. $$\frac{\Gamma, x : A_1 \ \vdash\ t_1 : A_2}{\Gamma\ \vdash\ \lambda x.t_1 : A_1 \to A_2}\ (\textsc{Abs})$$

This case holds trivially because there are no evaluation rules that can be applied to $\lambda x.t_1$.

Case. $$\frac{\begin{array}{c}\Gamma_1 \vdash t_1 : B \to A \\ \Gamma_2 \vdash t_2 : B\end{array}}{\Gamma_1 + \Gamma_2 \vdash t_1\, t_2 : A}\ (\textsc{App})$$

In this case, there are two evaluation rules that can be applied to t.

- Suppose the evaluation rule matches to $[\cdot]$.
  We know the evaluation of the assumption has the following form:

$$\frac{\dfrac{}{(\lambda x.t_1')\, t_2 \rightsquigarrow (t_2 \rhd x)\, t_1'}\ \textsc{E-Abs}}{\underbrace{(\lambda x.t_1')\, t_2}_{t} \longrightarrow (t_2 \rhd x)\, t_1'}$$

  By 4.7.5, we know the following:

$$\left.\begin{array}{l}\Gamma_1 + \Gamma_2 \vdash (\lambda x.t_1')\, t_2 : A \\ (\lambda x.t_1')\, t_2 \rightsquigarrow (t_2 \rhd x)\, t_1'\end{array}\right\} \implies \Gamma_1 + \Gamma_2 \vdash (t_2 \rhd x)\, t_1' : A$$

  Thus, we obtain the conclusion of the theorem.

- Suppose the evaluation rule matches to $E\, t$.
  We know the evaluation of the assumption has the following form:

$$\dfrac{\dfrac{}{t_1' \rightsquigarrow t_1''}}{\underbrace{E[t_1']\, t_2}_{t} \longrightarrow E[t_1'']\, t_2}$$

where $t_1 = E[t_1']$.

By induction hypothesis, we know the following:

$$\left.\begin{array}{l} \Gamma_1 \vdash E[t_1'] : B \to A \\ E[t_1'] \longrightarrow E[t_1''] \end{array}\right\} \implies \Gamma_1 \vdash E[t_1''] : B \to A \qquad \text{(ih)}$$

We then reapply (APP) to obtain the following:

$$\dfrac{\Gamma_1 \vdash E[t_1''] : B \to A \quad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash E[t_1'']\, t_2 : A}\,(\textsc{app})$$

Thus, we obtain the conclusion of the theorem.

$$\text{Case.}\ \dfrac{\Gamma_1 \vdash t_1 : \Box_r A \quad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ [x] = t_1\ \textbf{in}\ t_2 : B}\,(\textsc{let})$$

In this case, there are two evaluation rules that we can apply to t.

- Suppose the evaluation rule matches to $[\cdot]$.
  We know the evaluation of the assumption has the following form:

$$\dfrac{\dfrac{}{\textbf{let}\ [x] = [t_1']\ \textbf{in}\ t_2 \rightsquigarrow ([t_1'] \rhd [x])t_2}\,(\textsc{E-clet})}{\underbrace{\textbf{let}\ [x] = [t_1']\ \textbf{in}\ t_2}_{t} \longrightarrow ([t_1'] \rhd [x])t_2}$$

  By 4.7.5, we know the following:

$$\left.\begin{array}{l} \Gamma_1 + \Gamma_2 \vdash \textbf{let}\ [x] = [t_1']\ \textbf{in}\ t_2 : B \\ \textbf{let}\ [x] = [t_1']\ \textbf{in}\ t_2 \rightsquigarrow ([t_1'] \rhd [x])t_2 \end{array}\right\} \implies \Gamma_1 + \Gamma_2 \vdash ([t_1'] \rhd [x])t_2 : B$$

  Thus, we obtain the conclusion of the theorem.

- Suppose the evaluation rule matches to $\textbf{let}\ [x] = E\ \textbf{in}\ t$.
  We know the evaluation of the assumption has the following form:

$$\dfrac{\dfrac{}{t_1' \rightsquigarrow t_1''}}{\underbrace{\textbf{let}\ [x] = E[t_1']\ \textbf{in}\ t_2}_{t} \longrightarrow \textbf{let}\ [x] = E[t_1'']\ \textbf{in}\ t_2}$$

  where $t_1 = E[t_1']$.

  By induction hypothesis, we know the following:

$$\left.\begin{array}{l} \Gamma_1 \vdash E[t_1'] : \Box_r A \\ E[t_1'] \longrightarrow E[t_1''] \end{array}\right\} \implies \Gamma_1 \vdash E[t_1''] : \Box_r A \qquad \text{(ih)}$$

  We then reapply (LET) to obtain the following:

$$\frac{\Gamma_1 \vdash E[t_1''] : \Box_r A \quad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let } [x] = E[t_1''] \textbf{ in } t_2 : B} \text{(LET)}$$

Thus, we obtain the conclusion of the theorem.

$$\text{Case.} \quad \frac{\Gamma_1 \vdash t : A}{\Gamma_1 + [\Delta']_0 \vdash t : A} \text{(WEAK)}$$

In this case, t does not change between before and after the last derivation. The induction hypothesis implies that there exists a term t' such that:

$$t \longrightarrow t' \ \wedge \ \Gamma_1 \vdash t' : A \qquad \text{(ih)}$$

We then reapply (WEAK) to obtain the following:

$$\frac{\Gamma_1 \vdash t' : A}{\Gamma_1 + [\Delta']_0 \vdash t' : A} \text{(WEAK)}$$

Thus, we obtain the conclusion of the theorem.

$$\text{Case.} \quad \frac{\Gamma_1, x : B \vdash t : A}{\Gamma_1, x : [B]_1 \vdash t : A} \text{(DER)}$$

In this case, t does not change between before and after the last derivation. The induction hypothesis implies that there exists a term t' such that:

$$t \longrightarrow t' \ \wedge \ \Gamma_1, x : B \vdash t' : A \qquad \text{(ih)}$$

We then reapply (DER) to obtain the following:

$$\frac{\Gamma_1, x : B \vdash t' : A}{\Gamma_1, x : [B]_1 \vdash t' : A} \text{(DER)}$$

Thus, we obtain the conclusion of the theorem.

$$\text{Case.} \quad \frac{[\Gamma] \vdash t'' : B}{r \cdot [\Gamma] \vdash [t''] : \Box_r B} \text{(PR)}$$

This case holds trivially because there are no evaluation rules that can be applied to $[t'']$.

$$\text{Case.} \quad \frac{[\Gamma_i] \vdash t_i : A}{\bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash \{\overline{l = t} \,|\, l'\} : \Box_{\{\bar{l}\}} A} \text{(VER)}$$

This case holds trivially because there are no evaluation rules that can be applied to $\{\overline{l = t} \,|\, l'\}$.

$$\text{Case.} \quad \frac{[\Gamma_i] \vdash t_i : A}{\bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash \langle \overline{l = t} \,|\, l_k \rangle : A} \text{(VERI)}$$

In this case, the only evaluation rule we can apply is evaluation for $[\cdot]$. We know the evaluation of the assumption has the following form:

$$\dfrac{\dfrac{}{\langle \overline{l = t} \,|\, l_k \rangle \rightsquigarrow t_k@l_k}\ \text{E-VERI}}{\underbrace{\langle \overline{l = t} \,|\, l_k \rangle}_{t} \longrightarrow t_k@l_k}$$

By 4.7.5, we know the following:

$$\left.\begin{array}{l} \bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash \langle \overline{l = t} \,|\, l_k \rangle : A \\ \langle \overline{l = t} \,|\, l_k \rangle \rightsquigarrow t_k@l_k \end{array}\right\} \implies \bigcup_i(\{l_i\} \cdot [\Gamma_i]) \vdash t_k@l_k : A$$

Thus, we obtain the conclusion of the theorem.

$$\text{Case.} \quad \dfrac{\Gamma \vdash t_1 : \Box_r A \quad l_k \in r}{\Gamma \vdash t_1.l_k : A}\ (\text{EXTR})$$

In this case, there are two evaluation rules that we can apply to t.

- Suppose the evaluation rule matches to $[\cdot]$.
  We know the evaluation of the assumption has the following form:

$$\dfrac{\dfrac{}{t_1.l_k \rightsquigarrow t_1'}\ \text{E-EX1 or E-EX2}}{\underbrace{t_1.l_k}_{t} \longrightarrow t_1'}$$

  By 4.7.5, we know the following:

$$\left.\begin{array}{l} \Gamma \vdash t_1.l_k : A \\ t_1.l_k \rightsquigarrow t_1' \end{array}\right\} \implies \Gamma \vdash t_1' : A$$

  Thus, we obtain the conclusion of the theorem.

- Suppose the evaluation rule matches to $E.l$.
  We know the evaluation of the assumption has the following form:

$$\dfrac{\dfrac{}{t_1' \rightsquigarrow t_1''}}{\underbrace{E[t_1'].l_k}_{t} \longrightarrow E[t_1''].l_k}$$

  where $t_1 = E[t_1']$.
  By induction hypothesis, we know the following:

$$\left.\begin{array}{l} \Gamma \vdash E[t_1'] : \Box_r A \\ E[t_1'] \longrightarrow E[t_1''] \end{array}\right\} \implies \Gamma \vdash E[t_1''] : \Box_r A \qquad \text{(ih)}$$

  We the reapply (EXTR) to obtain the following:

$$\dfrac{\Gamma \vdash E[t_1''] : \Box_r A \quad l_k \in r}{\Gamma \vdash E[t_1''].l_k : A}\ (\text{EXTR})$$

  Thus, we obtain the conclusion of the theorem.

Case. $$\frac{\Gamma_1, x : [B]_r, \Gamma_2 \vdash t : A \quad r \sqsubseteq s}{\Gamma_1, x : [B]_s, \Gamma_2 \vdash t : A} \text{(SUB)}$$

In this case, t does not change between before and after the last derivation. The induction hypothesis implies that there exists a term t' such that:

$$t \longrightarrow t' \ \wedge \ \Gamma_1, x : [B]_r, \Gamma_2 \vdash t' : A \tag{ih}$$

We then reapply (SUB) to obtain the following:

$$\frac{\Gamma_1, x : [B]_r, \Gamma_2 \vdash t' : A \qquad r \sqsubseteq s}{\Gamma_1, x : [B]_s, \Gamma_2 \vdash t' : A} \text{(SUB)}$$

Thus, we obtain the conclusion of the theorem. □

Theorem 4.7.7. [λ_{VL} progress]

$$\varnothing \vdash t : A \Longrightarrow (\mathsf{value}\ t) \vee (\exists t'.t \longrightarrow t')$$

Proof. The proof is given by induction on the typing derivation of t. Consider the cases for the last rule used in the typing derivation of the assumption.

$$\text{Case.} \quad \frac{}{\varnothing \vdash n : \mathsf{Int}}\ (\textsc{Int})$$

This case holds trivially because value n.

Case. (VAR)

This case holds trivially because $x : A$ cannot be $\varnothing$.

$$\text{Case.} \quad \frac{x : A_1 \vdash t : A_2}{\varnothing \vdash \lambda x.t : A_1 \rightarrow A_2}\ (\textsc{Abs})$$

This case holds trivially because value $\lambda x.t$.

$$\text{Case.} \quad \frac{\begin{array}{c}\varnothing \vdash t_1 : B \rightarrow A \\ \varnothing \vdash t_2 : B\end{array}}{\varnothing \vdash t_1\, t_2 : A}\ (\textsc{App})$$

There are two cases whether t_1 is a value or not.

- Suppose t_1 is a value.
  By the inversion lemma (A.4.1), we know that there exists a term t_1' and $t_1 = \lambda x.t_1'$. Thus, we can apply (E-ABS) to t.

$$\frac{\frac{}{(\lambda x.t_1')\, t_2 \rightsquigarrow (t_2 \rhd x)\, t_1'}\ (\textsc{E-Abs})}{\underbrace{(\lambda x.t_1')\, t_2}_{t} \longrightarrow (t_2 \rhd x)\, t_1'}$$

  Furthermore, we know the following:

$$\frac{}{(t_2 \rhd x)\, t_1' = [t_2/x]t_1'}\ (\rhd_{\text{var}})$$

  By choosing $t = [t_2/x]t_1'$, we obtain the conclusion of the theorem.

- Suppose t_1 is not a value.
  There exists a term t_1' such that:

$$\frac{t_1 \rightsquigarrow t_1'}{t_1 \longrightarrow t_1'}$$

  Also, we can apply evaluation for application to t.

$$\frac{t_1 \rightsquigarrow t_1'}{\underbrace{t_1\, t_2}_{t} \longrightarrow t_1'\, t_2}$$

Thus, by choosing $t' = t_1'\, t_2$, we obtain the conclusion of the theorem.

Case. $$\frac{\emptyset \vdash t_1 : \Box_r A \quad x : [A]_r \vdash t_2 : B}{\emptyset \vdash \textbf{let}\ [x] = t_1\ \textbf{in}\ t_2 : B}\ (\textsc{Let})$$

There are two cases whether t_1 is a value or not.

- Suppose t_1 is a value.
  By the inversion lemma (A.4.1), we know that t_1 has either a form of $[t_1']$ or and $\{\overline{l = t''} \mid l_k\}$.
  - Case $t_1 = [t_1']$.
    In this case, we can apply (E-CLET) to obtain the following.

    $$\frac{}{\textbf{let}\ [x] = [t_1']\ \textbf{in}\ t_2 \rightsquigarrow ([t_1'] \rhd [x])t_2}\ (\text{E-}\textsc{clet})$$

    Thus, we can apply ($\rhd_\Box$) and ($\rhd_{\text{var}}$) to obtain the following.

    $$\frac{\dfrac{}{(t_1' \rhd x)t_2 = [t_1'/x]t_2}\ (\rhd_{\text{var}})}{([t_1'] \rhd [x])t_2 = [t_1'/x]t_2}\ (\rhd_\Box)$$

    Thus, by choosing $t' = [t_1'/x]t_2$, we obtain the conclusion of the theorem.
  - Case $t_1 = \{\overline{l = t''} \mid l_k\}$.
    In this case, we can apply (E-CLET) to obtain the following:

    $$\frac{\dfrac{}{\textbf{let}\ [x] = \{\overline{l = t''} \mid l_k\}\ \textbf{in}\ t_2 \rightsquigarrow (\langle\overline{l = t''} \mid l_k\rangle \rhd [x])t_2}\ (\text{E-}\textsc{clet})}{\underbrace{\textbf{let}\ [x] = \{\overline{l = t''} \mid l_k\}\ \textbf{in}\ t_2}_{t} \longrightarrow (\langle\overline{l = t''} \mid l_k\rangle \rhd [x])t_2}$$

    Also, we can apply ($\rhd_{\text{ver}}$) and ($\rhd_{\text{var}}$) to obtain the following.

    $$\frac{\dfrac{}{(\langle\overline{l = t''} \mid l_k\rangle \rhd x)t_2 = [\langle\overline{l = t''} \mid l_k\rangle/x]t_2}\ (\rhd_{\text{var}})}{(\{\overline{l = t''} \mid l_k\} \rhd [x])t_2 = [\langle\overline{l = t''} \mid l_k\rangle/x]t_2}\ (\rhd_{\text{ver}})$$

    Thus, by choosing $t' = [\langle\overline{l = t''} \mid l_k\rangle/x]t_2$, we obtain the conclusion of the theorem.

- Suppose t_1 is not a value.
  There exists terms t_1' such that:

  $$\frac{t_1 \rightsquigarrow t_1'}{t_1 \longrightarrow t_1'}$$

  Also, we can apply evaluation for contextual let bindings to t.

  $$\frac{t_1 \rightsquigarrow t_1'}{\underbrace{\textbf{let}\ [x] = t_1\ \textbf{in}\ t_2}_{t} \longrightarrow \textbf{let}\ [x] = t_1'\ \textbf{in}\ t_2}$$

Thus, by choosing $t' = (\textbf{let}\ [x] = t_1'\ \textbf{in}\ t_2)$, we obtain the conclusion of the theorem.

Case. $$\frac{\varnothing \vdash t : A}{\varnothing \vdash t : A}\ (\textsc{weak})$$

In this case, t does not change between before and after the last derivation. Thus, we can obtain the conclusion of the theorem by induction hypothesis.

Case. (DER)

This case hold trivially because $\Gamma_1, x : [B]_1$ cannot be $\varnothing$.

Case. $$\frac{\varnothing \vdash t : B}{\varnothing \vdash [t] : \Box_r B}\ (\textsc{pr})$$

This case holds trivially because $[t]$ is a value.

Case. $$\frac{\varnothing \vdash t_i : A}{\varnothing \vdash \{\overline{l = t} \mid l'\} : \Box_{\{\bar{l}\}} A}\ (\textsc{ver})$$

This case holds trivially because $\{\overline{l = t} \mid l'\}$ is a value.

Case. $$\frac{\varnothing \vdash t_i : A}{\varnothing \vdash \langle \overline{l = t} \mid l_k \rangle : A}\ (\textsc{veri})$$

In this case, we can apply (E-VERI).

$$\frac{\overline{\langle \overline{l = t} \mid l_k \rangle \rightsquigarrow t_k @ l_k}\ (\text{E-}\textsc{veri})}{\langle \overline{l = t} \mid l_k \rangle \longrightarrow t_k @ l_k}$$

Thus, by choosing $t' = t_k @ l_k$, we obtain the conclusion of the theorem.

Case. $$\frac{\varnothing \vdash t_1 : \Box_r A \quad l_k \in r}{\varnothing \vdash t_1.l_k : A}\ (\textsc{extr})$$

In this case, we have two cases whether t_1 is a value or not.

- Suppose t_1 is a value. $(t_1 = v_1)$
  By 4.7.4, we know the following:

  $$\varnothing \vdash v_1 : \Box_r A \quad \Longrightarrow \quad \exists t'. \begin{cases} v_1.l_k \longrightarrow t' \\ \varnothing \vdash t' : A \end{cases}$$

  Thus, we obtain the conclusion of the theorem.

- Suppose t_1 is not a value.
  There exists a term t_1 such that:

  $$\frac{t_1 \rightsquigarrow t_1'}{t_1 \longrightarrow t_1'}$$

  Also, we can apply an exaluation rule for extraction to t.

  $$\frac{t_1 \rightsquigarrow t_1'}{\underbrace{t_1.l_k}_{t} \longrightarrow t_1'.l_k}$$

  Thus, by choosing $t' = t_1'.l_k$, we obtain the conclusion of the theorem.

Case. $$\frac{\Gamma_1, x : [B]_r, \Gamma_2 \vdash t : A \quad r \sqsubseteq s}{\Gamma_1, x : [B]_s, \Gamma_2 \vdash t : A} \text{(SUB)}$$

In this case, t does not change between before and after the last derivation. Thus, by induction hypothesis, we obtain the conclusion of the theorem. □