Programming Language With Versions

🎤**Yudai Tanabe**
Tokyo Institute of Technology

https://yudaitnb.github.io/

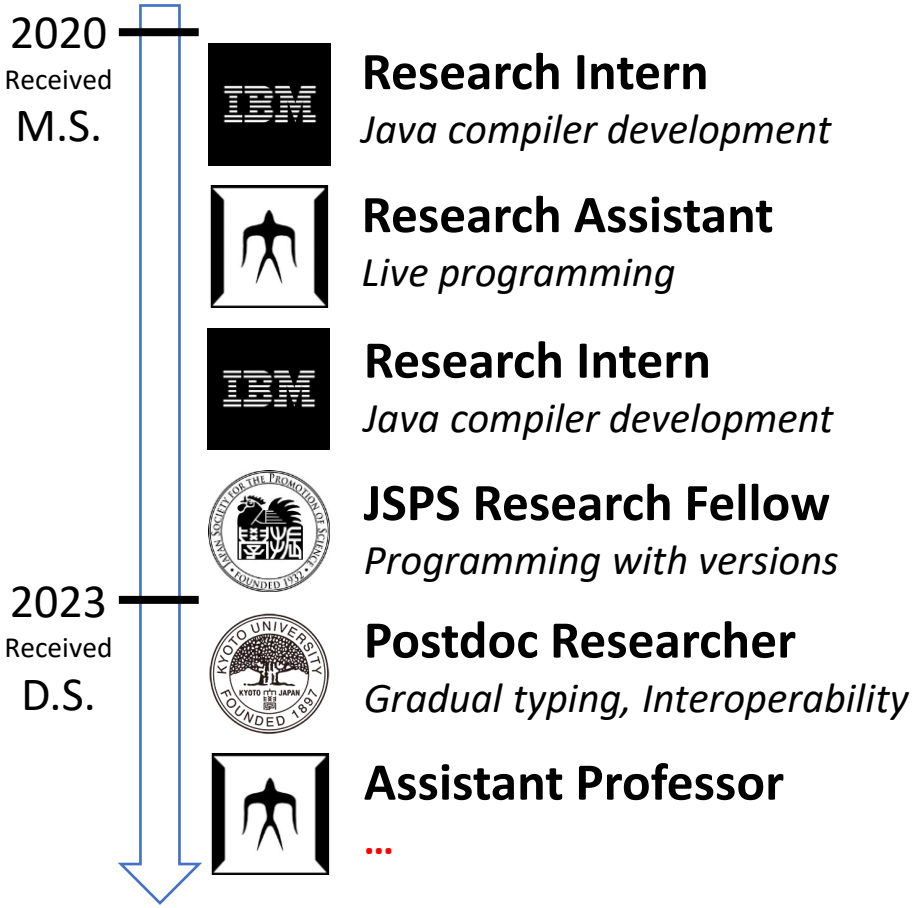
About me

Yudai Tanabe (田邉 裕大)

Assistant Professor
Tokyo Institute of Technology

Expertise:
Programming languages (PL)

- PL theory
  (effects/coeffects, gradual typing)
- Advanced modularity
  (COP, metaprogramming)
- Software maintenance in PL
  (PWV)



About me

Ongoing Projects

Today's talk

1. **Programming Language With Versions (PWV)**
   PhD research, [COP'18, ‹Programming›'22, SLE'22, COP'22, APLAS'23]
   Joint work with Sanyo-Onoda City University

2. Space-efficient Polymorphic Gradual Typing
   Postdoc research, [PLDI'24]
   Joint work with Kyoto University and NII

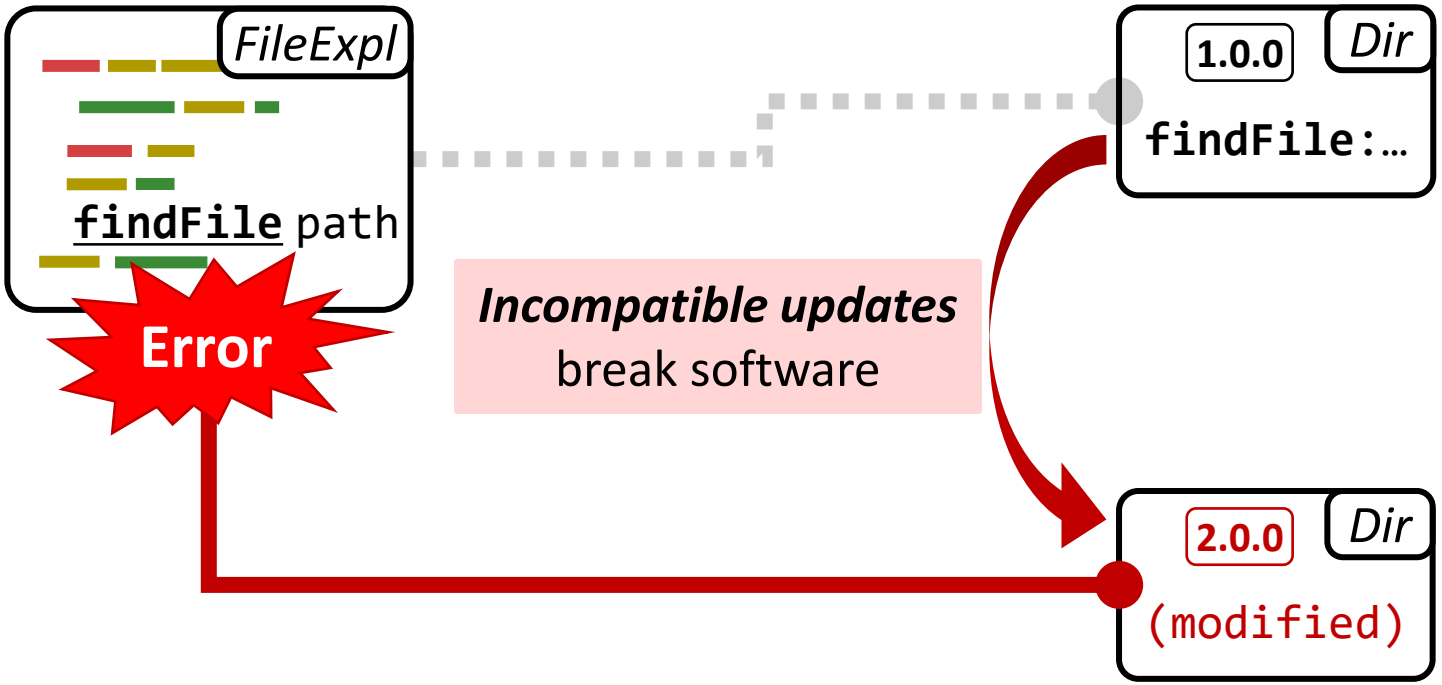
3. Safe Language Interoperability
   WIP

Working with students on their thesis projects on
PWV, type systems, GPGPU, live programming, … and more?

Update Dilemma: Enhancements ***vs.*** Adaptation Costs

[Werner'13, Bavota'15]

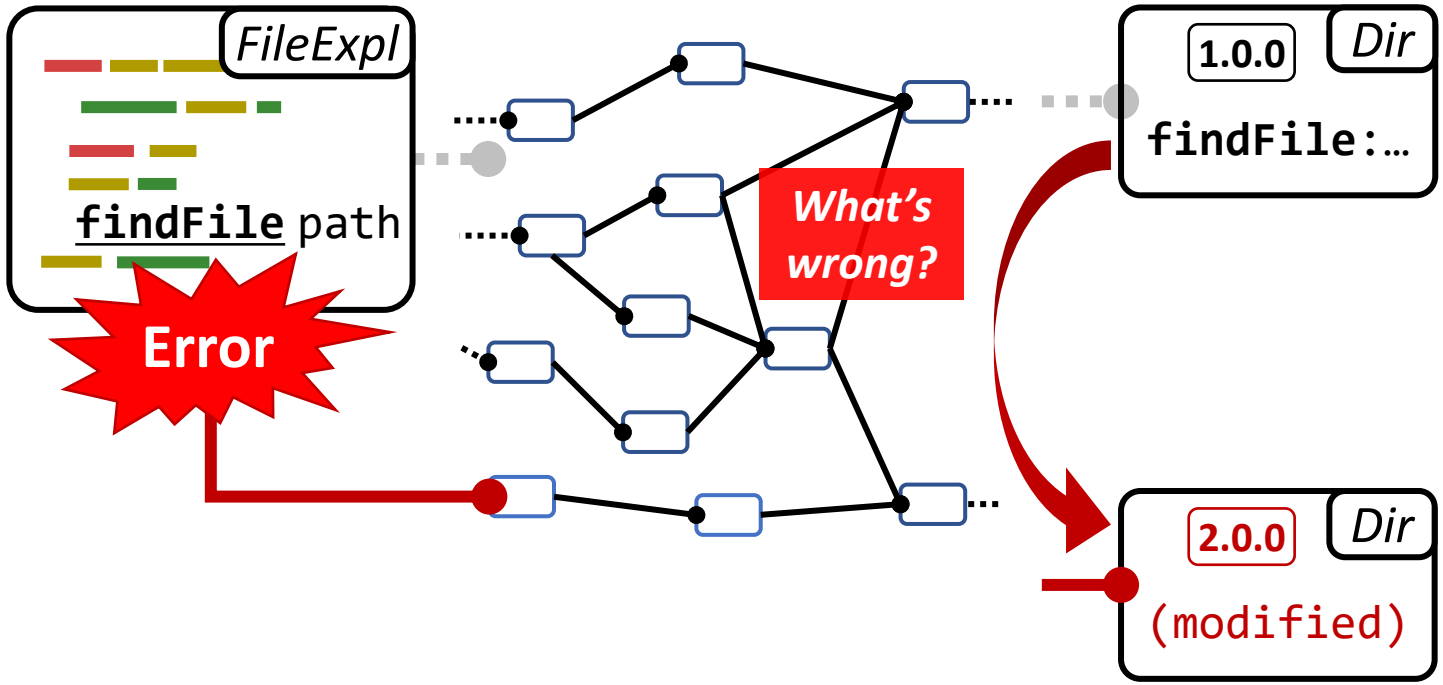
Intricate update process deterring programmers from updates



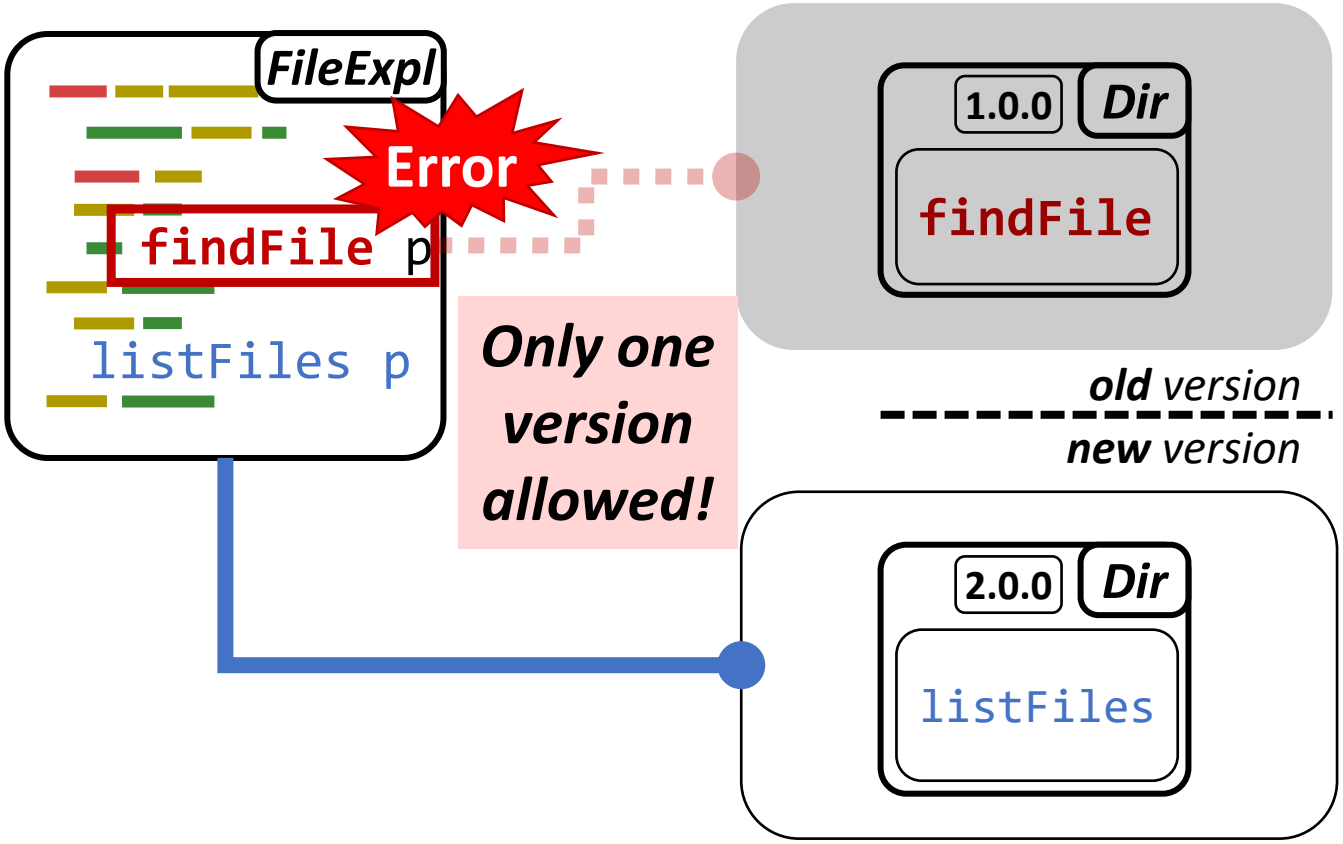
Dependency Hell

[Decan'17, Jayasuriya'19]

Indirect dependencies complicate tracing errors.



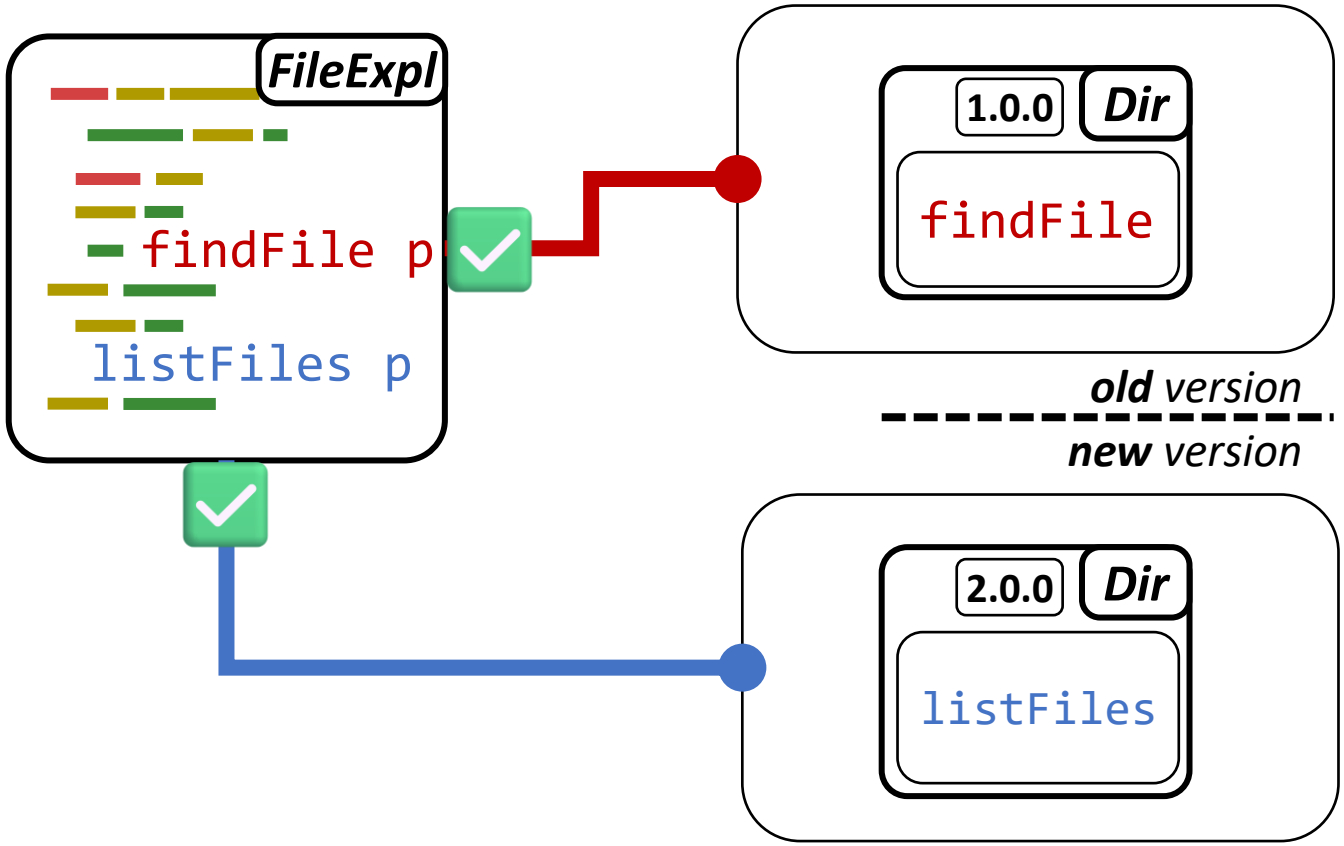
One-version-at-a-time Limitation



Proposal

Programming With Versions (PWV)

Safely handle multiple versions in one client

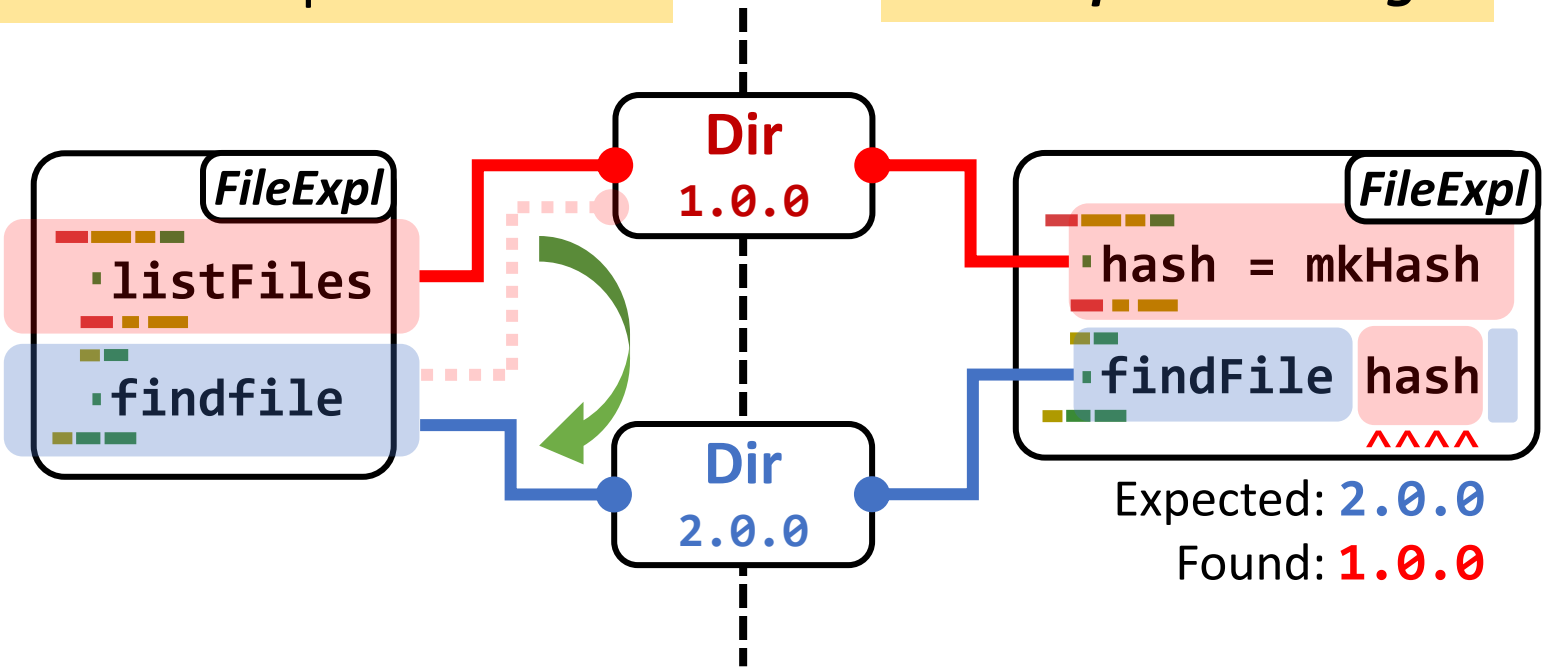


Why Language-based Approach?

Merit: *Split Updates* Into *Smaller* Tasks

✅ ***Incremental updates*** of smaller part of codes

✅ ***Detecting incompatible usage***

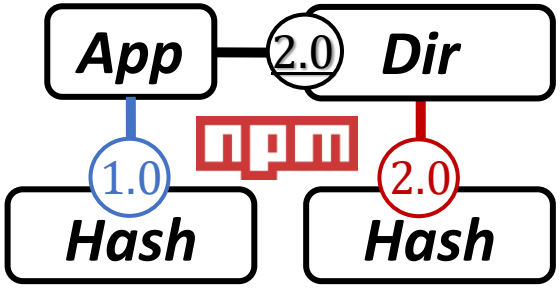


Why Language-based Approach?

Vs. Package-based Managements

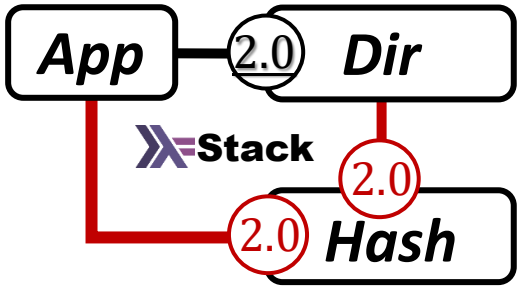
🤔 Only help externally to the language

👎 No support for incompatibility info



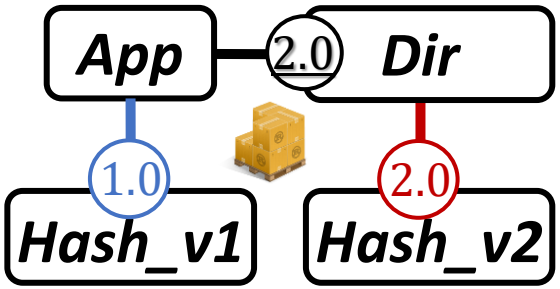
Package duplication and hoisting

👎 No flexibility in version selection

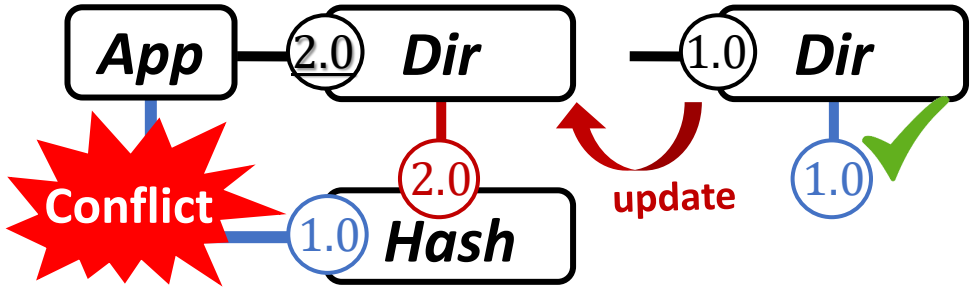


Dependency Snapshot

👎 Lost type identity between versions

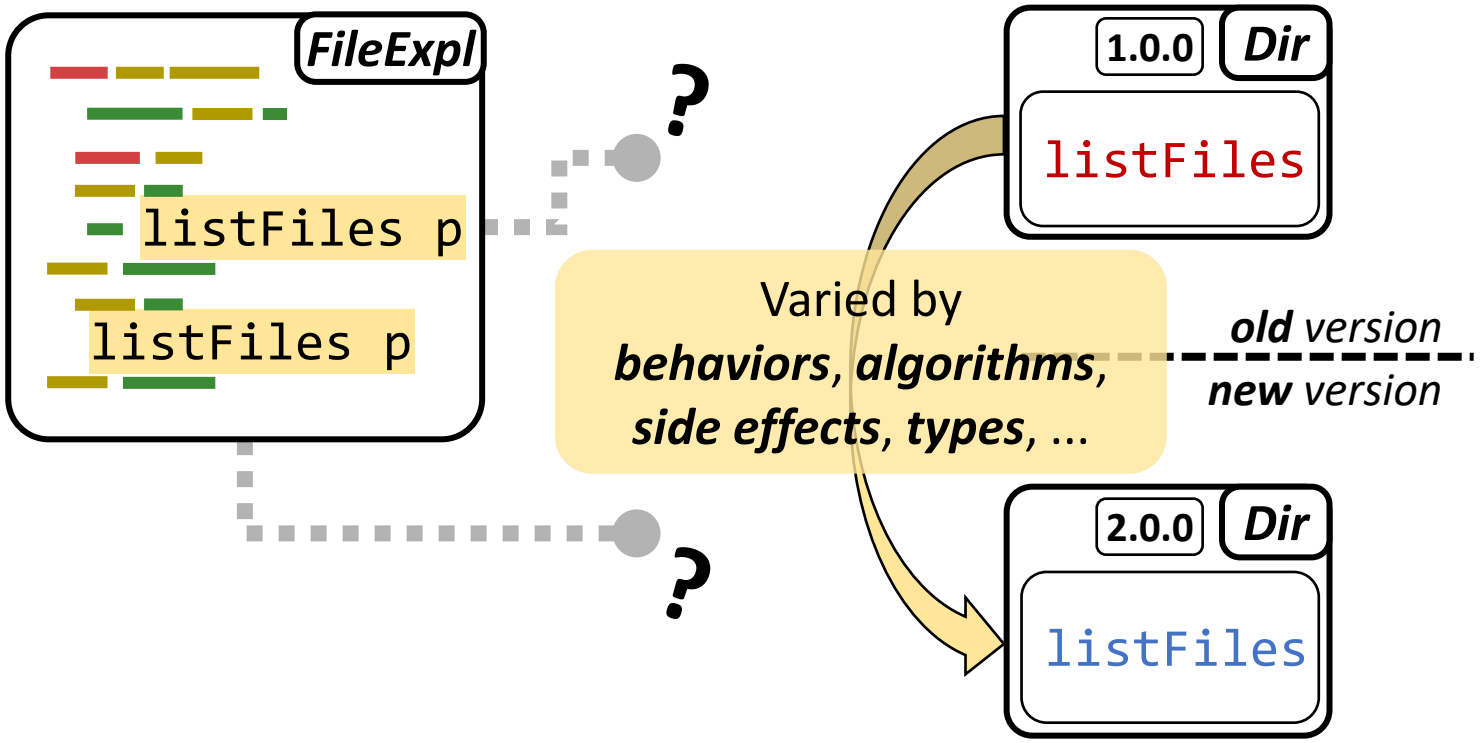


Name mangling



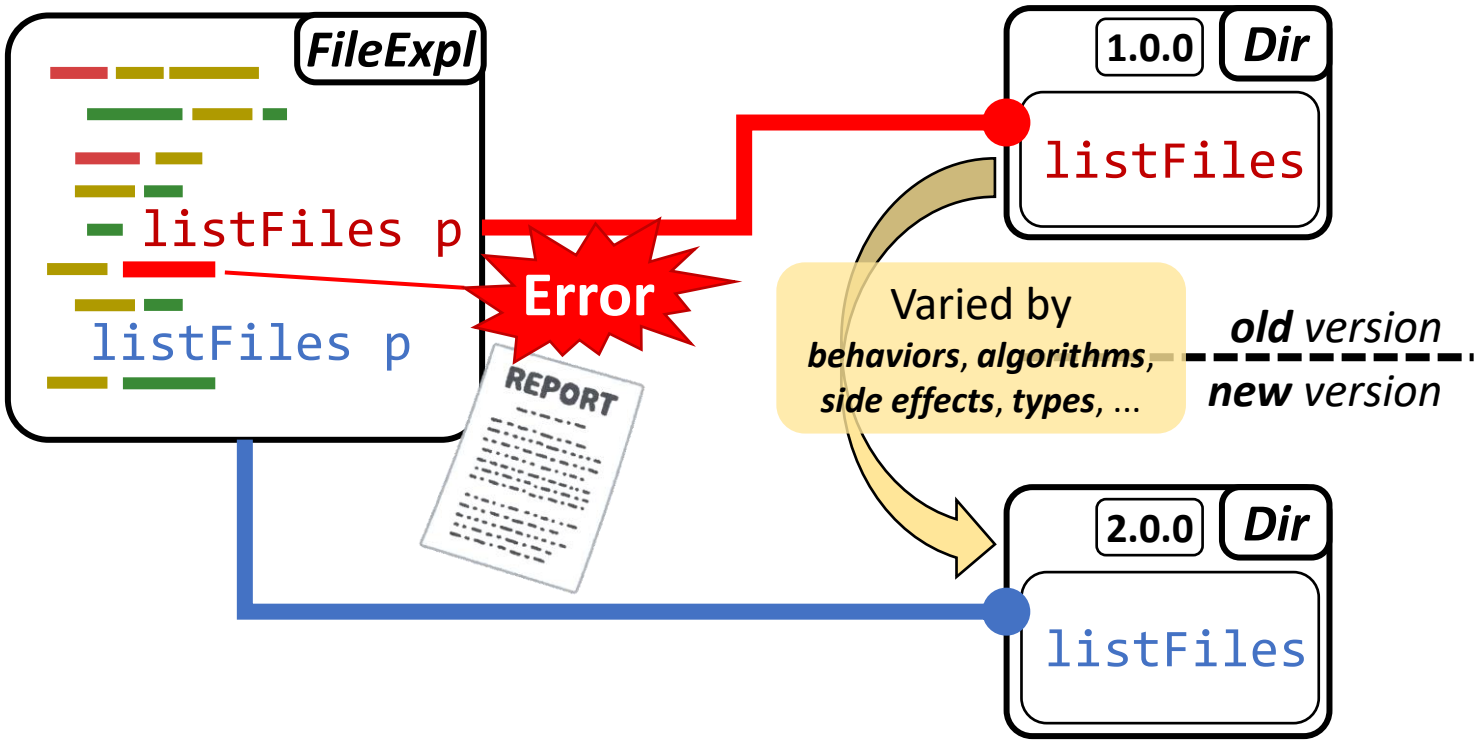
①Differentiate Programs *by Code Contexts*

… w/ as few syntactic annotations as possible



②Aid for Resolving *Semantic Incompatibility*

Addressing incompatibilities:
feedback the source and/or ***auto-resolve***?

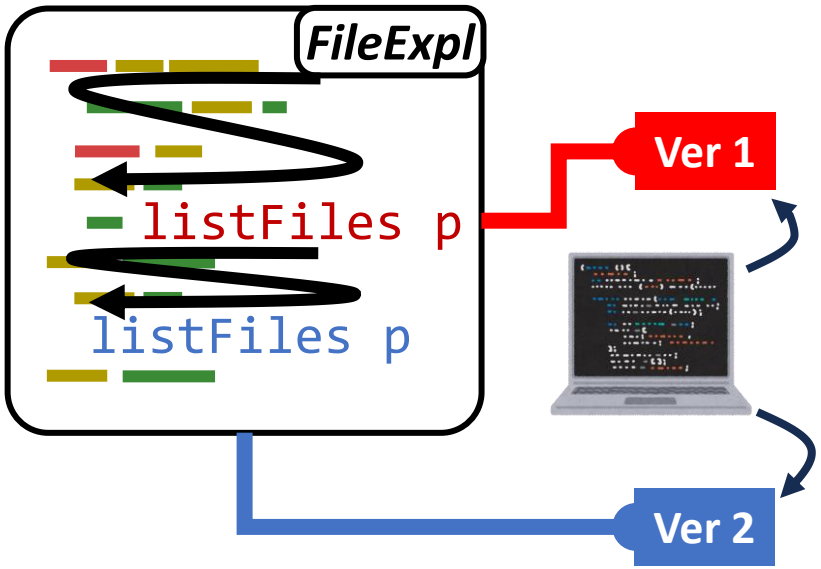


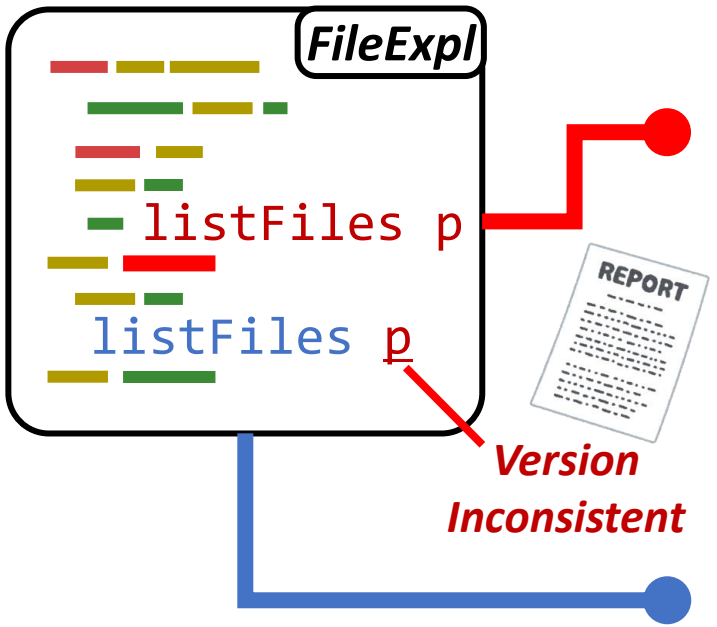
[‹Programming›'22], [APLAS'23]

Our Solution

①***Version Inference***

②Ensuring ***Version Consistency in Data Flows***





[‹Programming›'22] → [APLAS'23]

PWV ***w/o*** Version Annotations

[‹Programming›'22]
λ_{VL}

```
module FileExpl where

main () =
  let [str] = [getArg [()]] in
  let [digest] =
        [{l1=…, l2=…}[str]] in
  if [{l1=…, l2=…} [digest]].l1
  …
  [listDir [curDir]].l2
```

Cumbersome syntax

Require versions in code locations

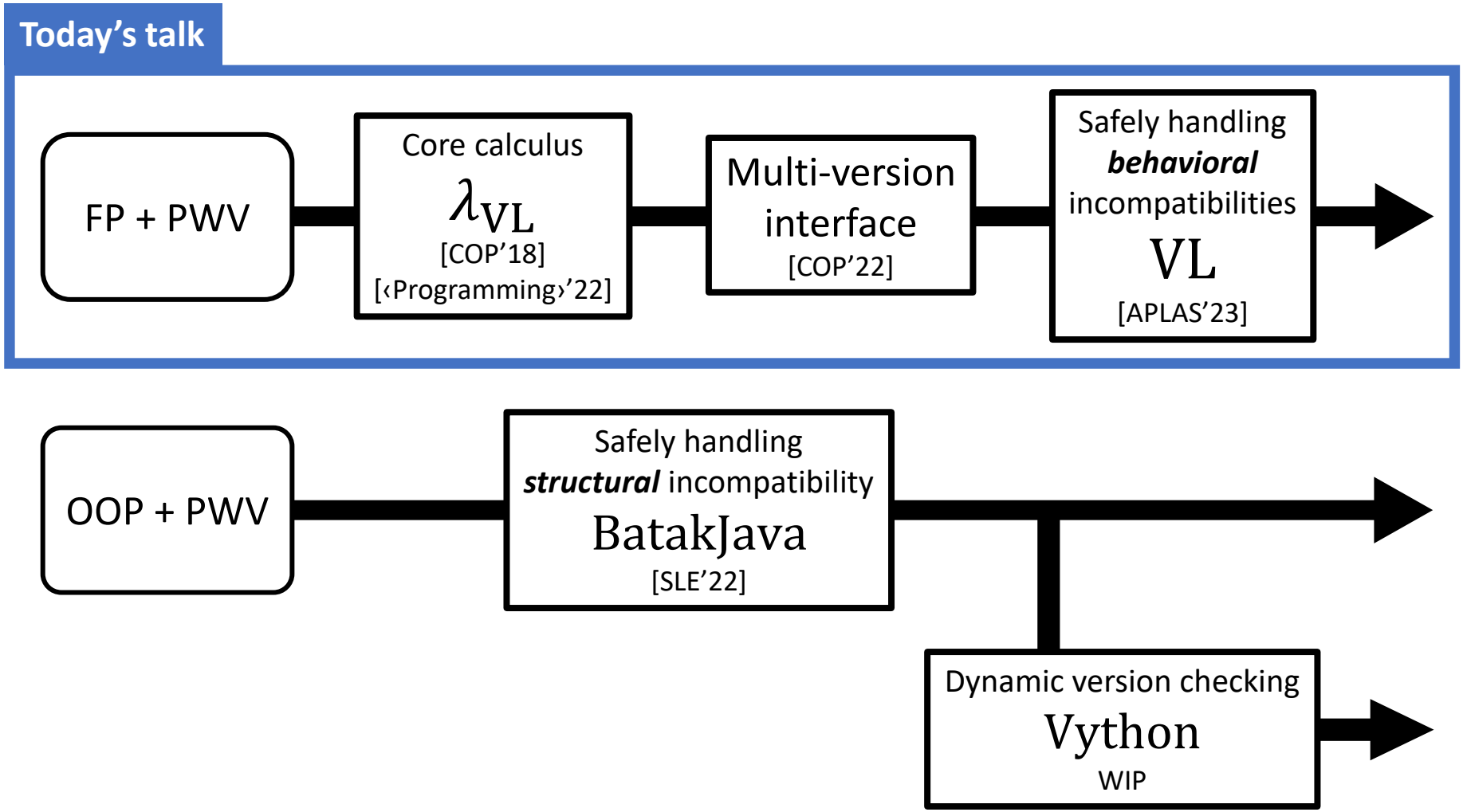
[APLAS'23]
VL

```
module FileExpl where

main () =
  let str = getArg () in
  let digest = mkHash str in
  if exist digest …
  …
  listDir curDir
```

No version annotations!

Background

PWV Languages



Rest of the Talk

Contribution

Programming with Versions ***w/o*** Version Annotations

[‹Programming›'22]
λ_{VL}
Explicit
version annotations

vs.

[APLAS'23]
VL
IR
VLMini
Version inference
incorporating implicit versions

- λ_{VL} Semantics and Type System

- **Key idea**: *Utilizing module versions for expression versions*
- Programming in VL
- Compilation

- Implementation & Evaluation
- Future work

Outline

Contribution

Programming with Versions ***w/o*** Version Annotations

[‹Programming›'22]
λ_{VL}
Explicit
version annotations

vs.

[APLAS'23]
VL
IR
VLMini
Version inference
incorporating implicit versions

- **λ_{VL} Semantics and Type System**

- Key idea:
  Utilizing module versions for expression versions
- Programming in VL
- Compilation

- Implementation & Evaluation
- Future work

[‹Programming›'22]

λ_{VL}, Versions within Semantics

Version Labels to capture version possibilities

e.g.

$$\boldsymbol{l_1} = \begin{bmatrix} Dir \mapsto 1.0.0, \\ Hash \mapsto 1.0.0 \end{bmatrix}, \boldsymbol{l_2} = \begin{bmatrix} Dir \mapsto 1.0.0, \\ Hash \mapsto 2.0.0 \end{bmatrix}$$

Multiple terms in a ***versioned value***

$$findFile = \left\{ \begin{array}{l} \boldsymbol{l_1} = \boxed{\begin{array}{l}\texttt{¥hash ->} \\ \texttt{ if exist hash …}\end{array}} \\ \boldsymbol{l_2} = \boxed{\begin{array}{l}\texttt{¥hash ->} \\ \texttt{ if exist hash …}\end{array}} \end{array} \right\}$$

Evaluate term ***in a specific version***

$$[findFile\ hash].\boldsymbol{l_1}$$

$$\rightarrow findFile_{\boldsymbol{l_1}}\ hash_{\boldsymbol{l_1}}$$

$$\rightarrow$$ `/home/yudaitnb/vl/src/file.ext`

[‹Programming›'22]

λ_{VL} Type System

*Type system to **enforce version consistency***

$$mkHash : \square_{\{l_1, l_2\}}\mathrm{Hash}$$
$$findFile : \square_{\{l_1\}}(\mathrm{Hash} \to A)$$
$$\vdash \begin{array}{l}\mathrm{let}\ [f] = findFile\ \mathrm{in} \\ \mathrm{let}\ [x] = mkHash\ \mathrm{in} \\ [f\ x].\ l_2\end{array} : \text{Well-typed?}$$

Denotes ***available versions*** of a term

Inconsistent!
because $l_2 \notin \{l_1\}$

Proved

Soundness

$$\Gamma \vdash t : A \wedge t \longrightarrow t' \Longrightarrow \Gamma \vdash t' : A \quad \text{(preservation)}$$
$$\emptyset \vdash t : A \Longrightarrow \mathrm{value}\ t \ \vee \exists t'.\, t \longrightarrow t' \quad \text{(progress)}$$

Type system is based on coeffect calculi:
$\ell\mathcal{R}$PCF[Brunel'14], GrMini[Orchard'19].

Outline

Contribution

Programming with Versions ***w/o*** Version Annotations

[‹Programming›'22]
λ_{VL}
Explicit
version annotations

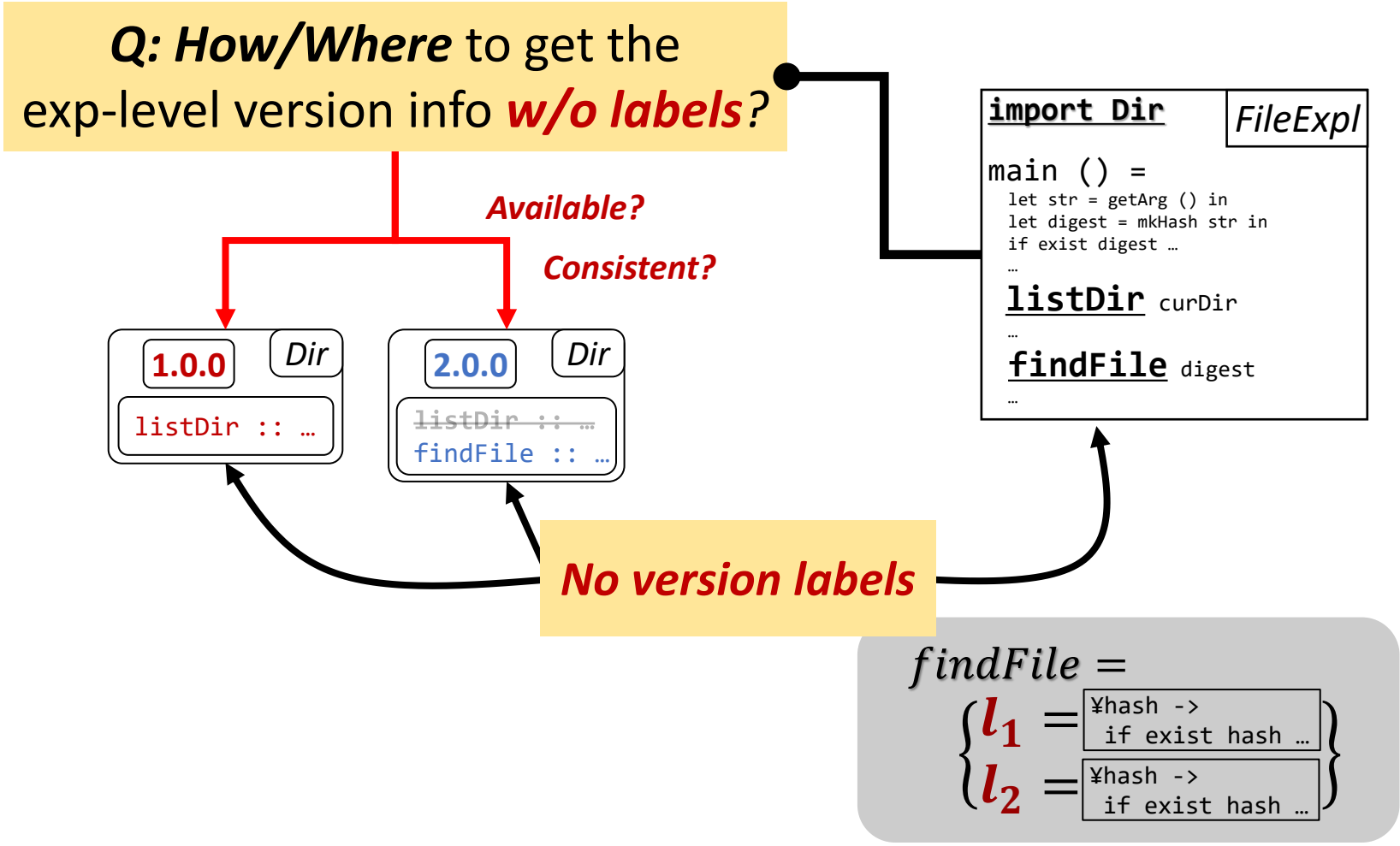
vs.

[APLAS'23]
VL
IR
VLMini
Version inference
incorporating implicit versions

- λ_{VL} Semantics and Type System
- **Key idea:** *Utilizing module versions for expression versions*
- Programming in VL
- Compilation
- Implementation & Evaluation
- Future work

Key Idea

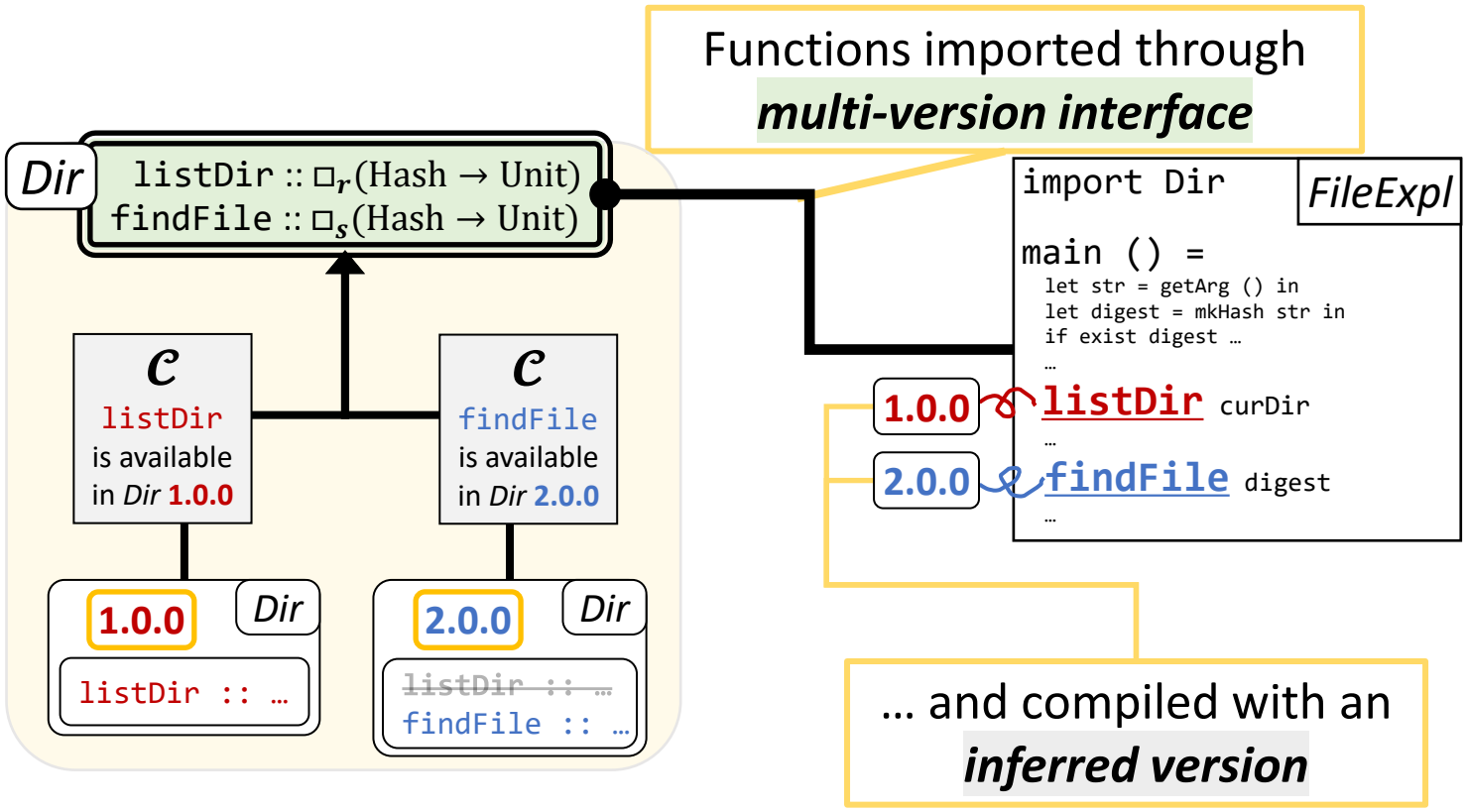
How to Omit Version Annotations?



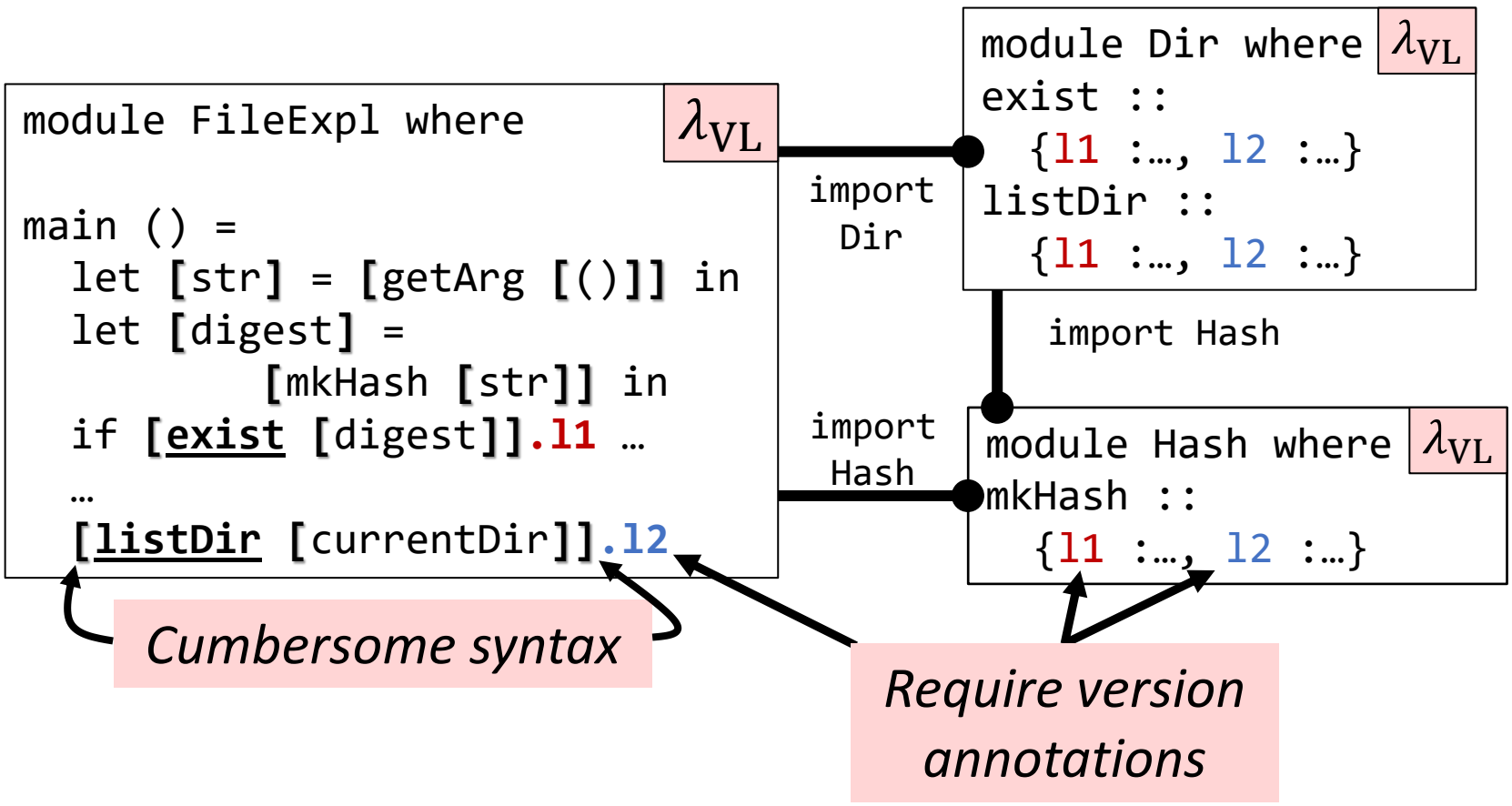
Key Idea

Compilation with *Implicit* Versions

***A.** Utilizing module vers. to denote expression vers.*

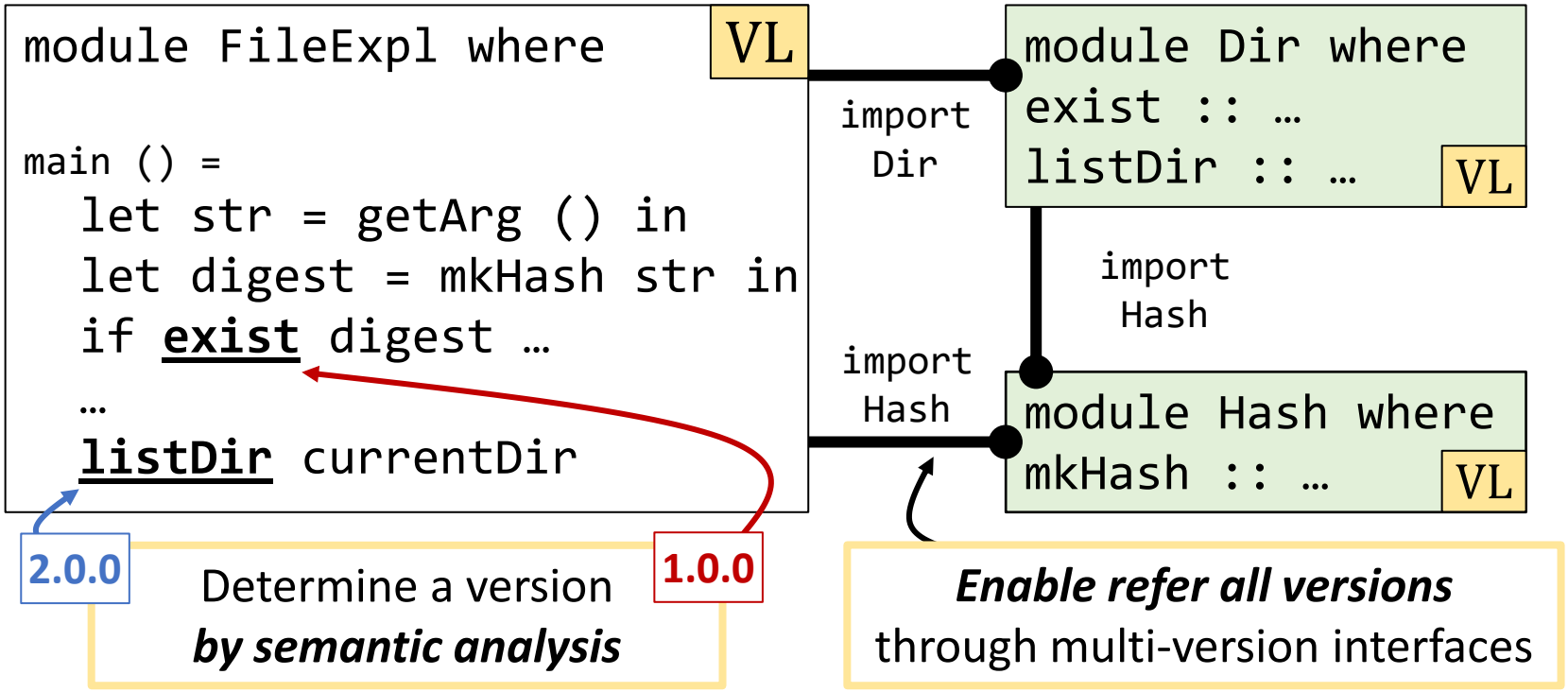


VL *vs.* λ_{VL}: *w/* Version Labels

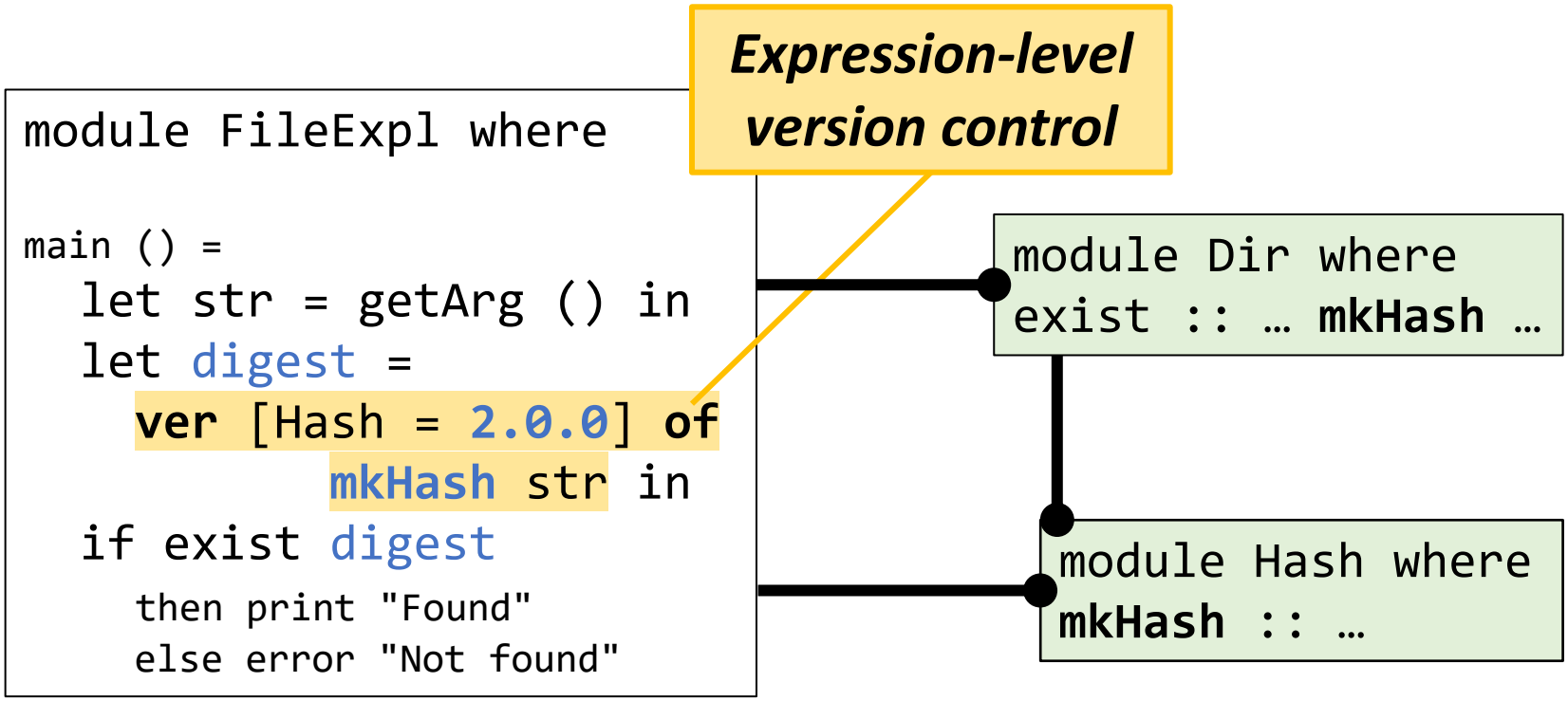


VL *vs.* λ_{VL} : *w/o* Version Labels

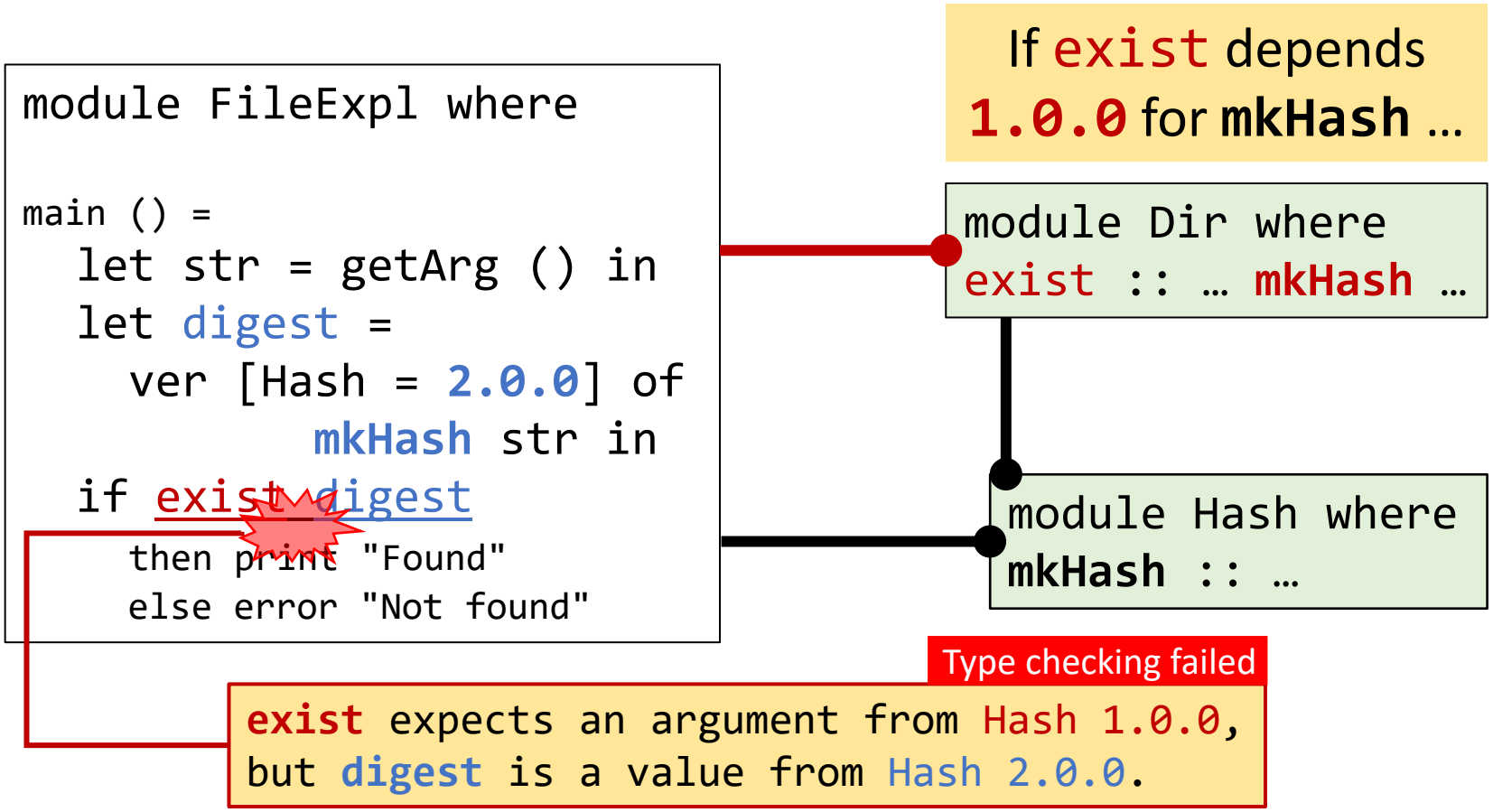
No version annotations!



Handling Multiple Versions in One Client



Detecting *Incompatible* Version Usage



Collaboration with *Compatible* Version

```
module FileExpl where

main () =
  let str = getArg () in
  let digest =
    ver [Dir = 2.0.0] of
          mkHash str in
  let exist' = unver exist
  if exist' digest
    then print "Found"
    else error "Not found"
```

If programmers know **1.0.0** and **2.0.0** of Hash are ***compatible*** …

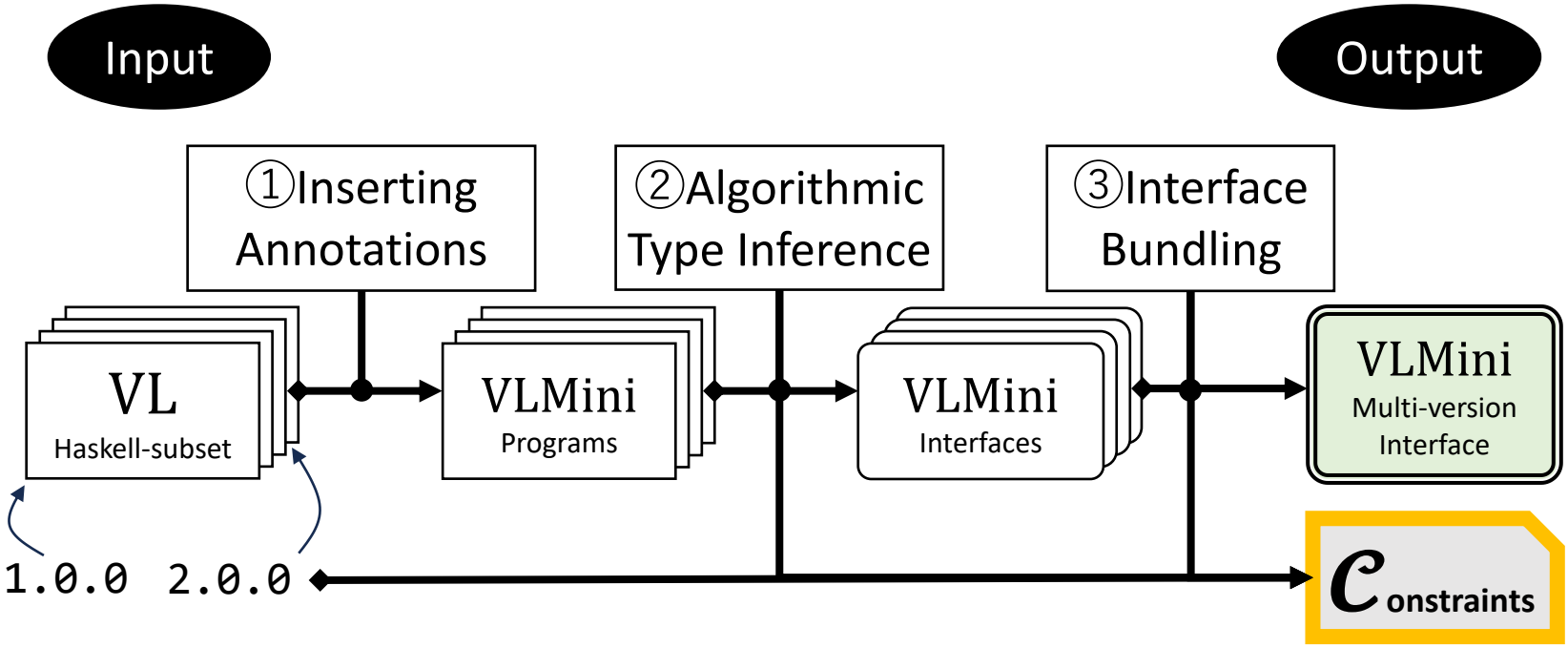
```
module Dir where
exist :: … mkHash …
```

```
module Hash where
mkHash :: …
```

Incorporate compatibility into type checking

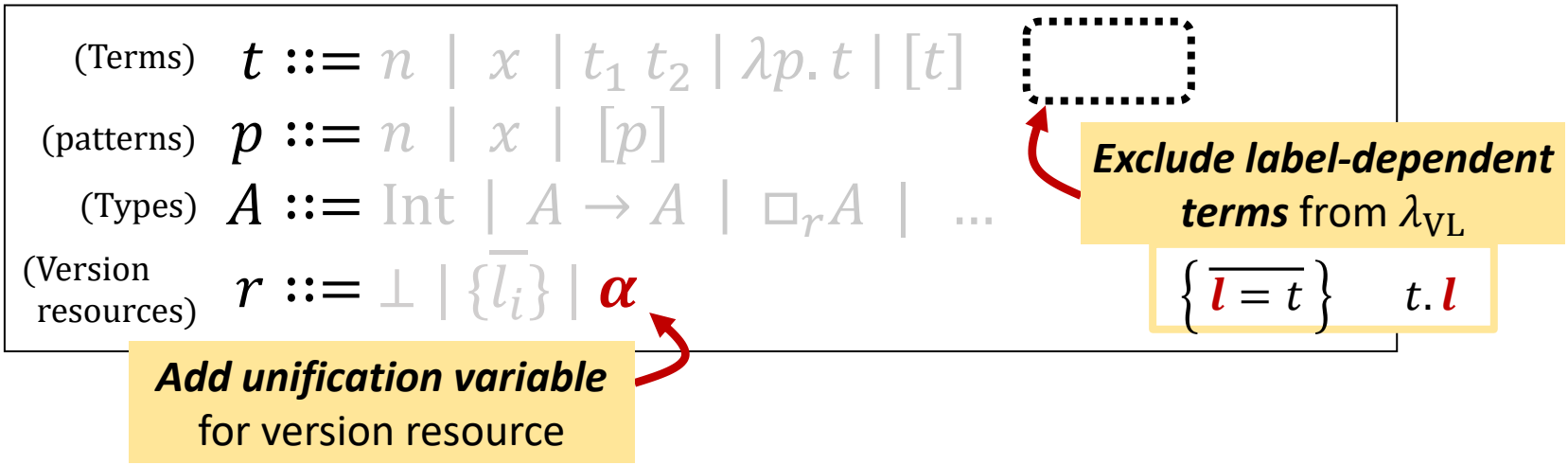
Compilation Overview

I/O of Compilation



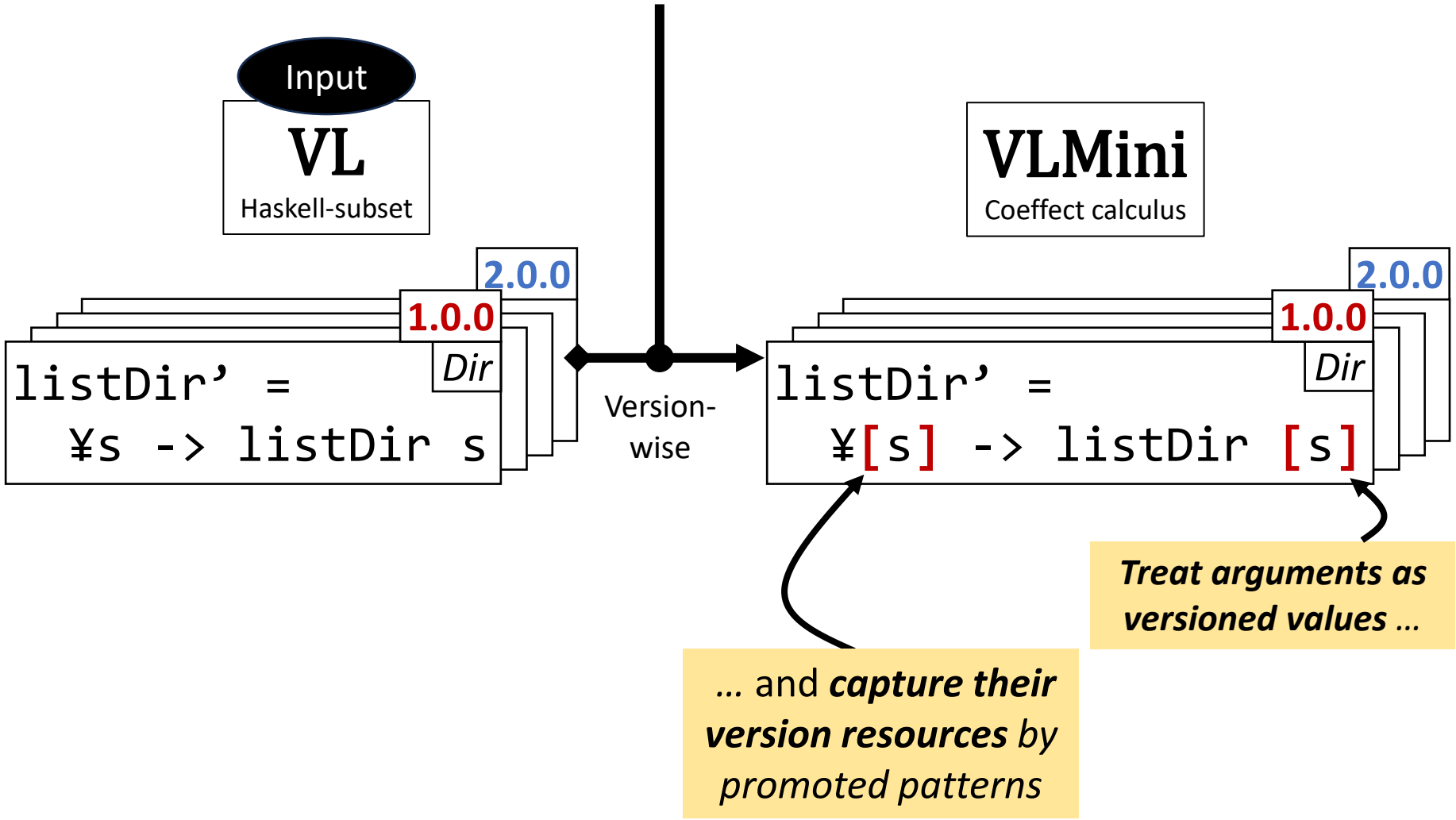
Compilation Overview

VLMini, A *label-free* variant of λ_{VL}



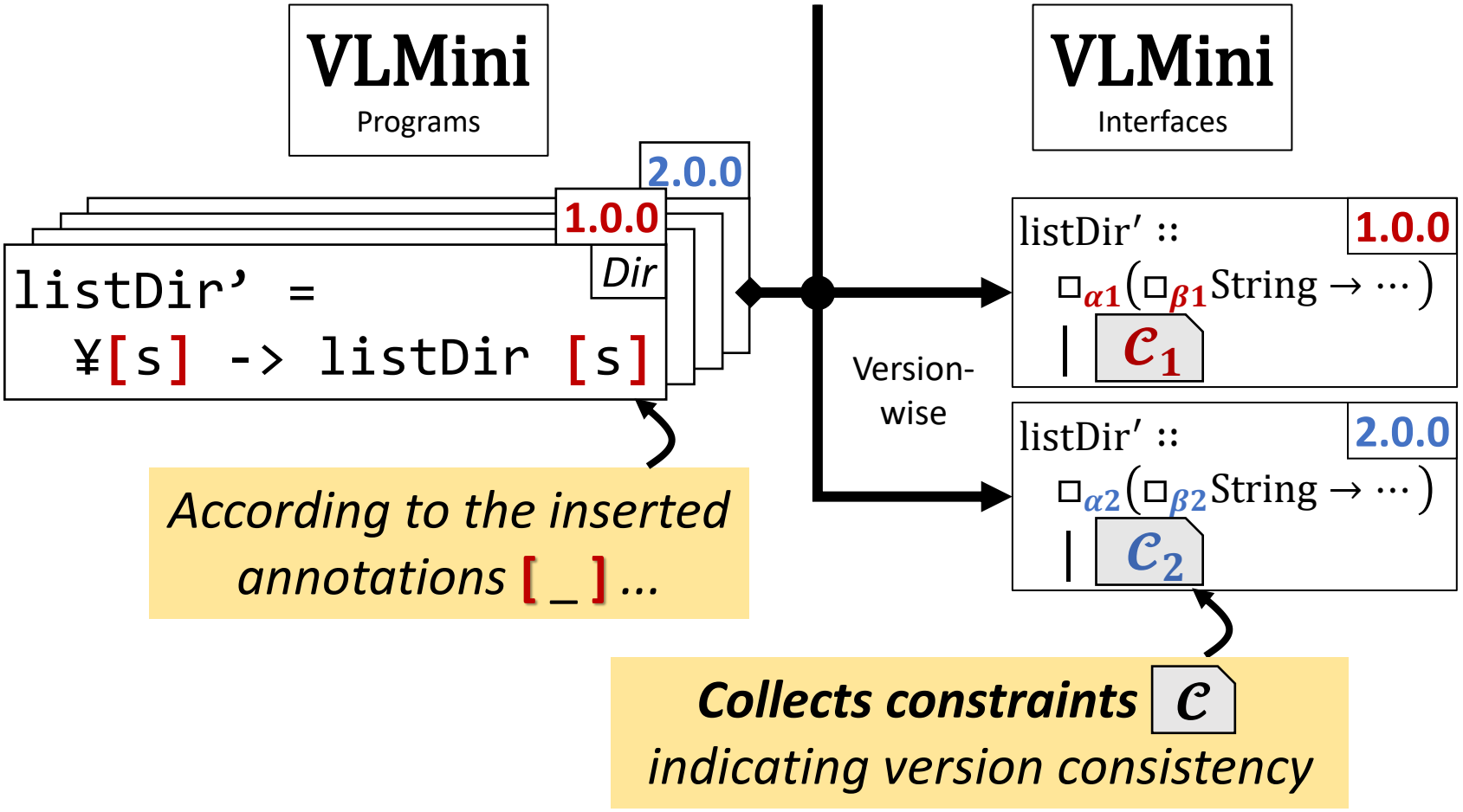
Compilation Overview

① Inserting Annotations



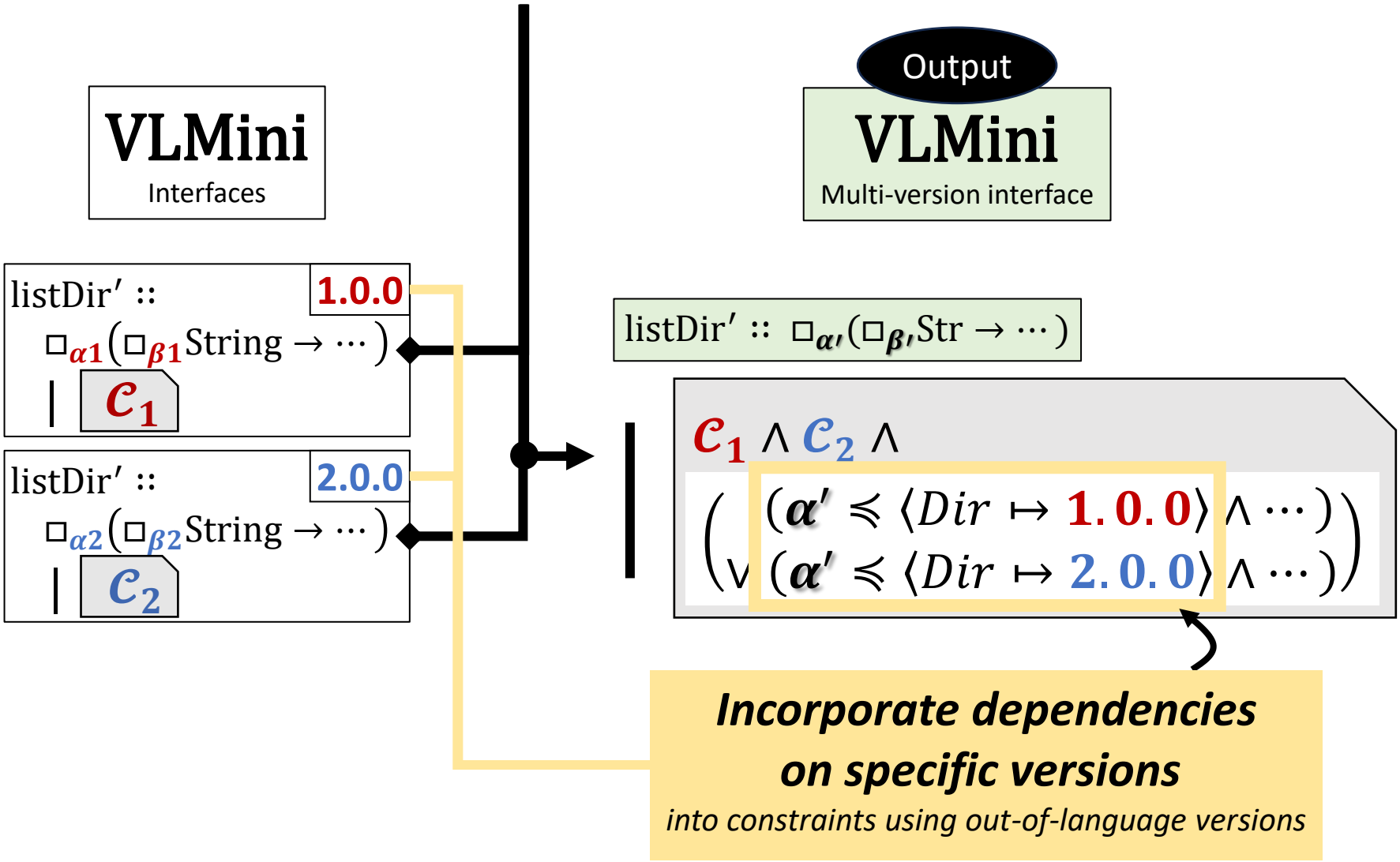
Compilation Overview

② Version Inference



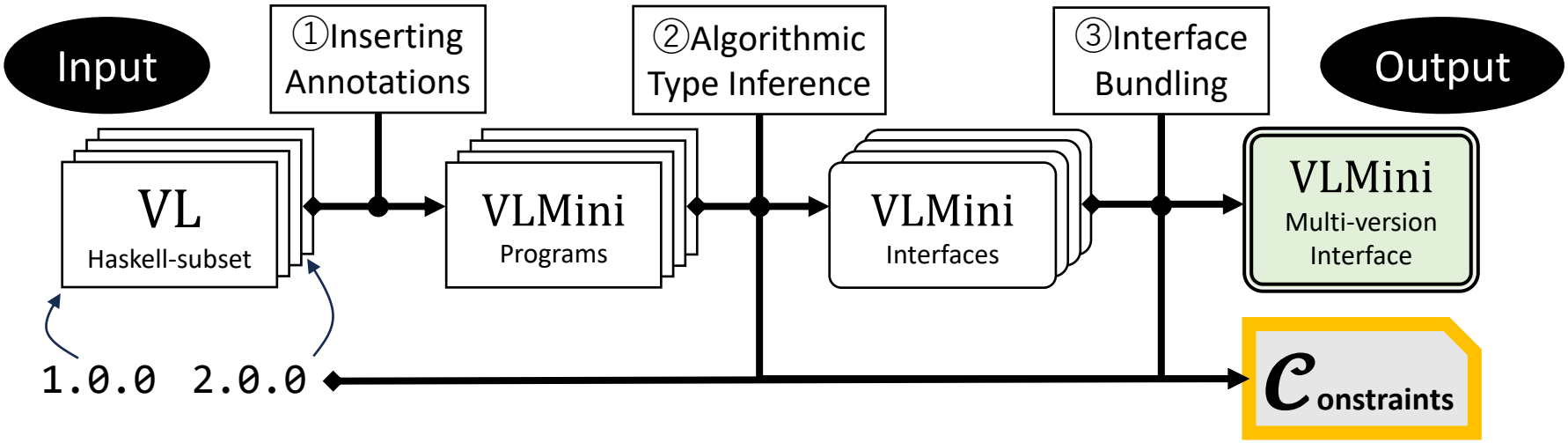
Compilation Overview

③ Interface Bundling



Compilation Overview

I/O of Compilation



Next slide 👉

How the VL compiler uses generated constraints

Constraints

"$\preccurlyeq$" represents dependencies

(Constraints) $\mathcal{C} ::= \top \mid \mathcal{C}_1 \wedge \mathcal{C}_2 \mid \mathcal{C}_1 \vee \mathcal{C}_2 \mid \alpha \preccurlyeq \alpha' \mid \alpha \preccurlyeq \mathcal{D}$

(Dependencies) $\mathcal{D} ::= \langle \overline{M_i \mapsto V_i} \rangle$

Module name

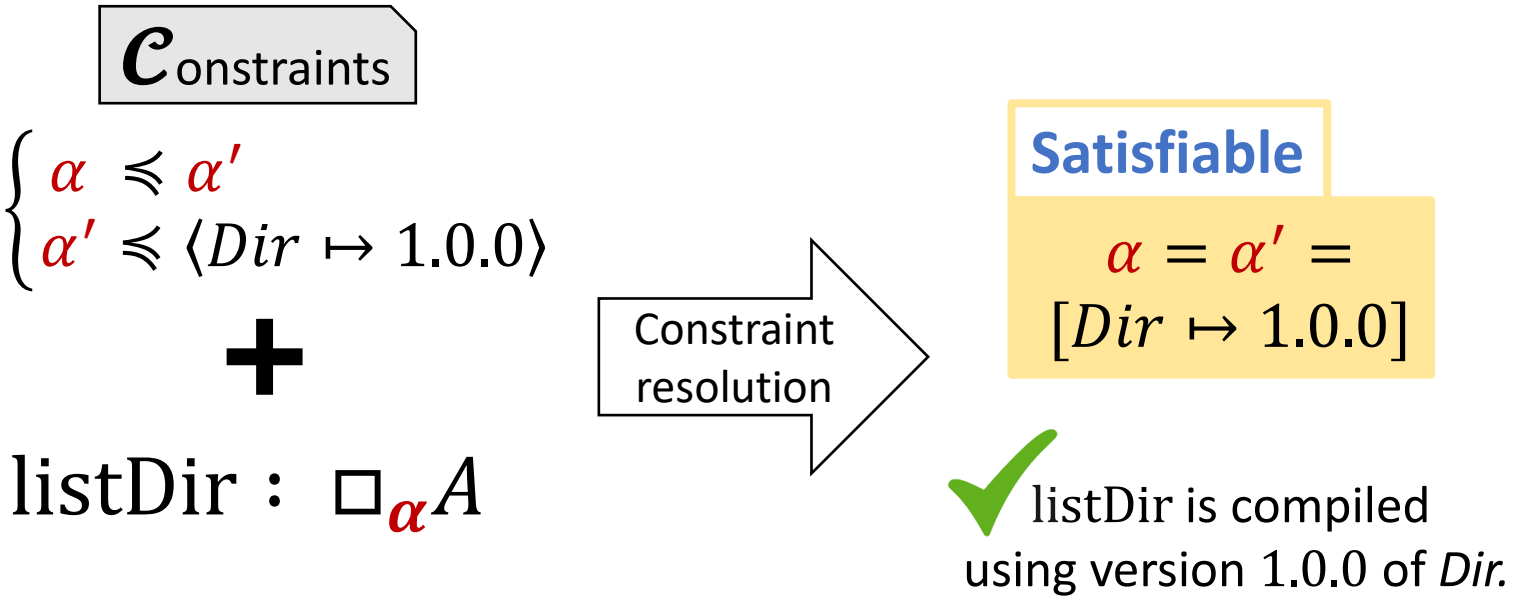
Version number

*"If a version label for **RHS** expects a specific version, …*

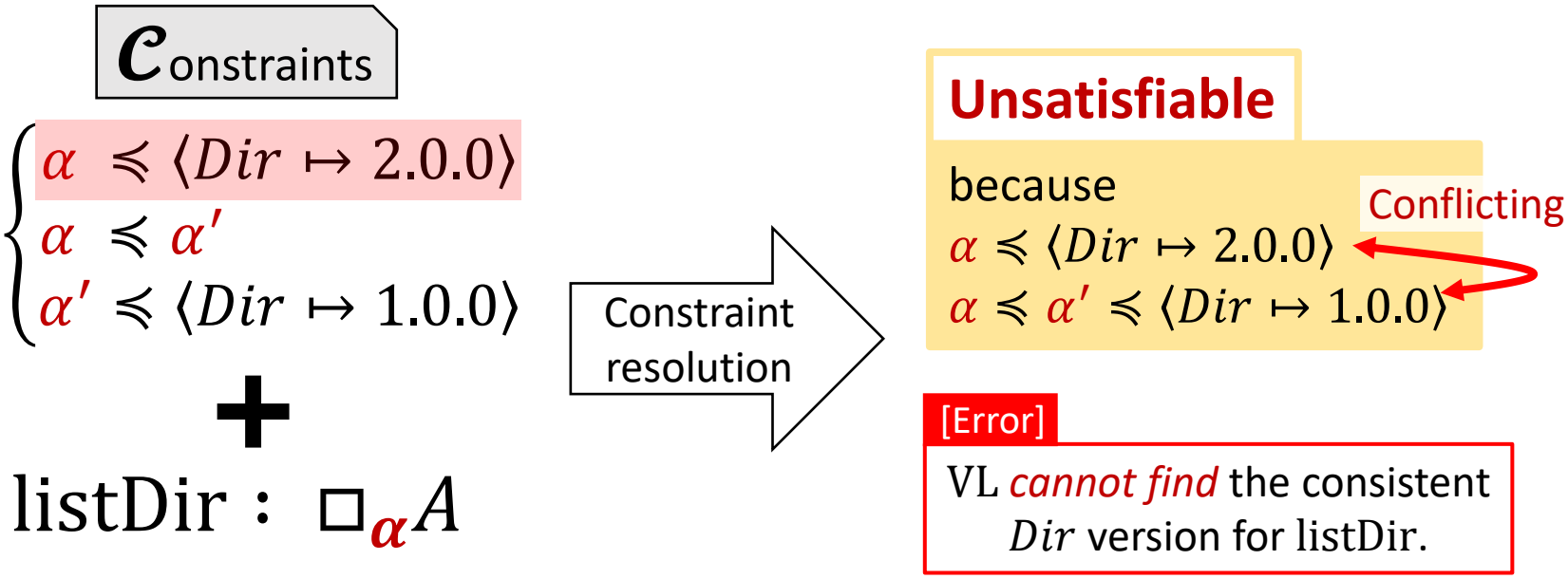
$\alpha \preccurlyeq \alpha' \qquad \alpha \preccurlyeq \langle Dir \mapsto 1.0.0 \rangle$

*… then **α** (LHS) also expects the same version."*

Satisfiable Constraints



Unsatisfiable Constraints



Outline

Contribution

Programming with Versions ***w/o*** Version Annotations

[‹Programming›'22]
λ_{VL}
Explicit
version annotations

vs.

[APLAS'23]
VL
IR
VLMini
Version inference
incorporating implicit versions

- λ_{VL} Semantics and Type System

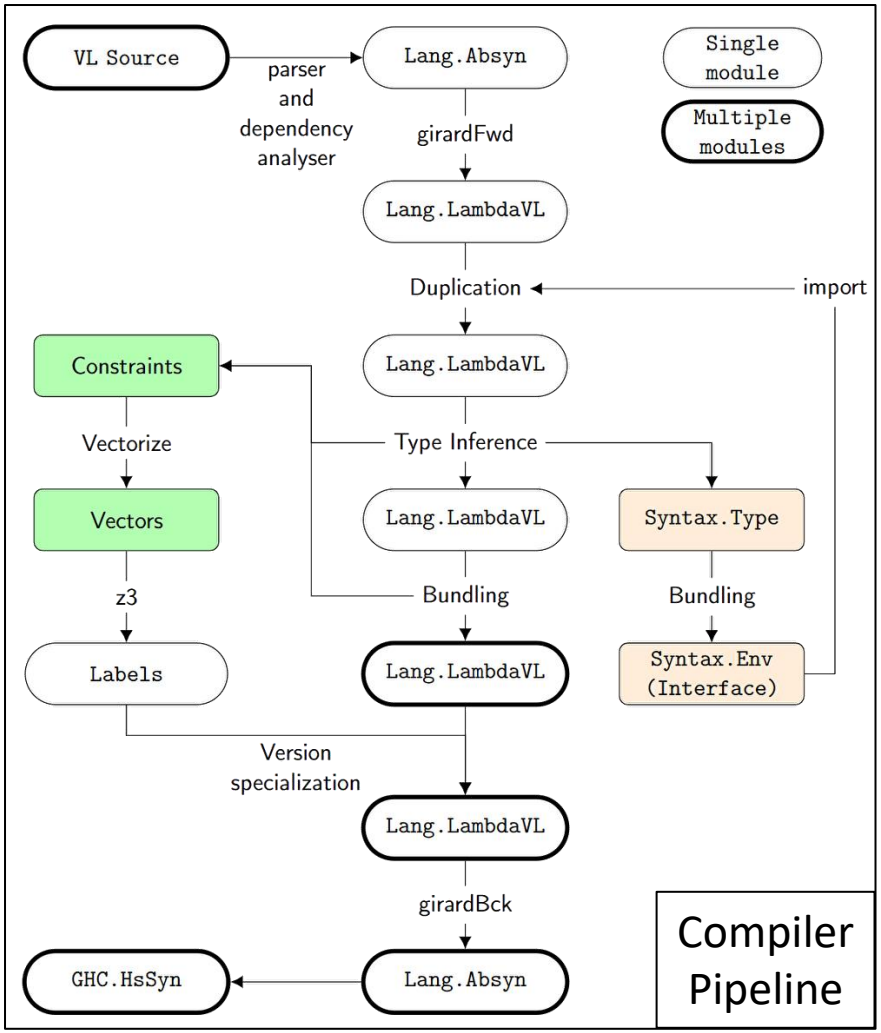
- **Key idea**: Utilizing module versions for expression versions
- Programming in VL
- Compilation

- Implementation & Evaluation
- Future work

Implementation

The VL Compiler

- Implemented on
  GHC 9.2.4
  https://github.com/yudaitnb/vl

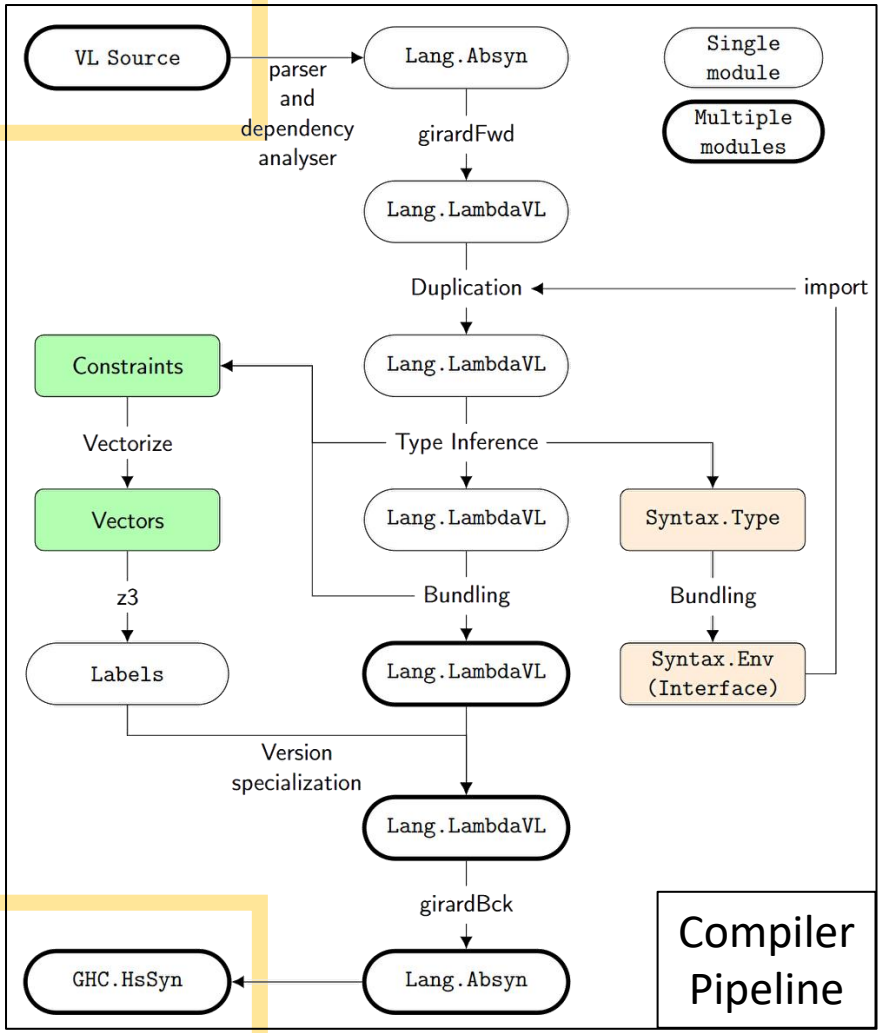


Implementation

The VL Compiler

- Implemented on GHC 9.2.4
  https://github.com/yudaitnb/vl

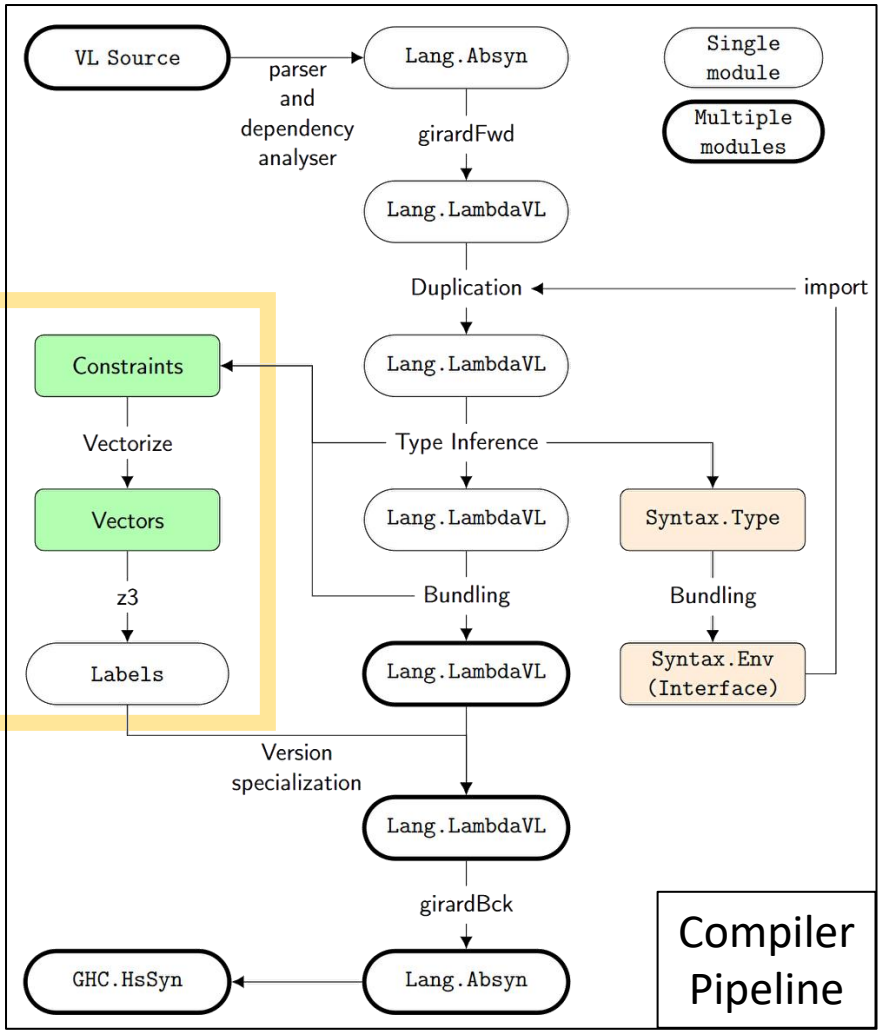
- Input and output are Haskell ASTs



Implementation

The VL Compiler

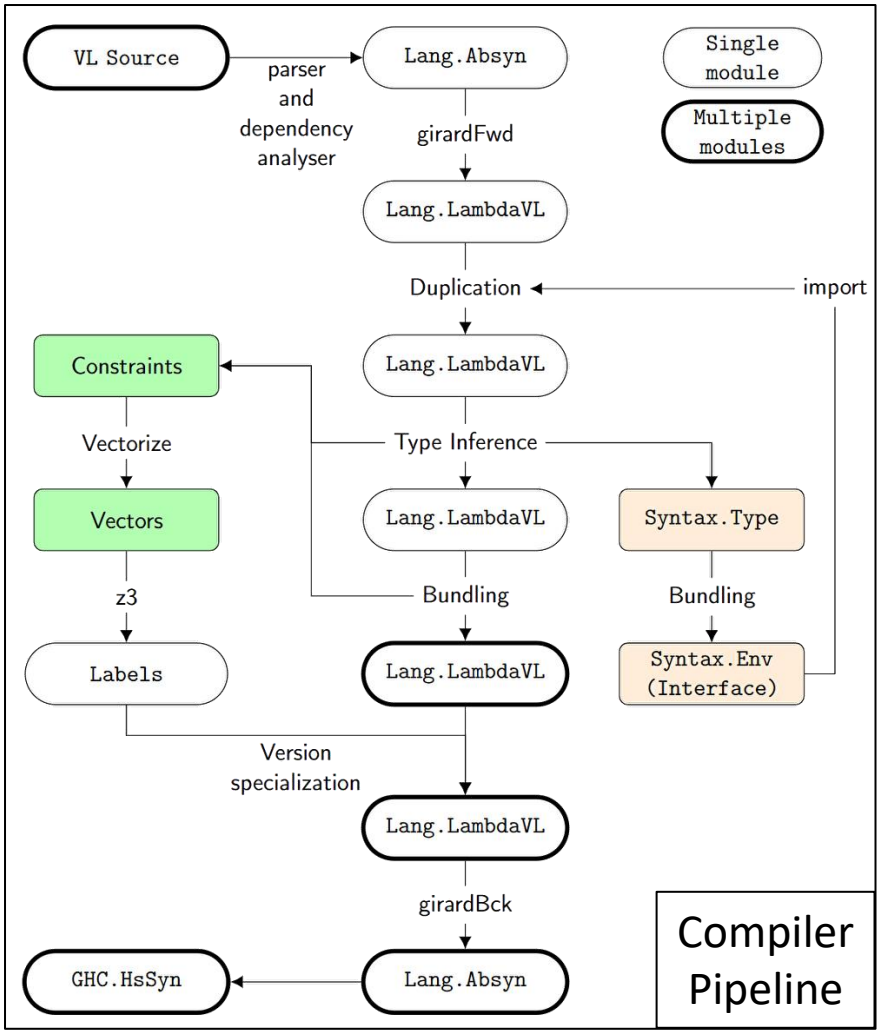
- Implemented on GHC 9.2.4
  https://github.com/yudaitnb/vl
- Input and output are Haskell ASTs
- Resolve constraints using Z3 [De Moura'08]



Implementation

The VL Compiler

- Implemented on GHC 9.2.4
  https://github.com/yudaitnb/vl
- Both in-/out-put are Haskell ASTs
  (subset)
- Resolve constraints using Z3 [De Moura'08]



Evaluations (next slides)

1. ***Case study*** to confirm VL achieving PWV benefits
2. ***Compiler performance***

Evaluation

1. Case Study

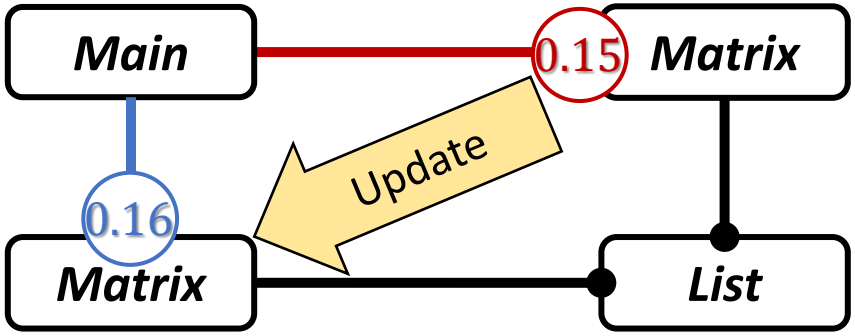
VL *achieves our goals:*

✓ *Handling two versions* in one client

✓ *Detecting inconsistent version*

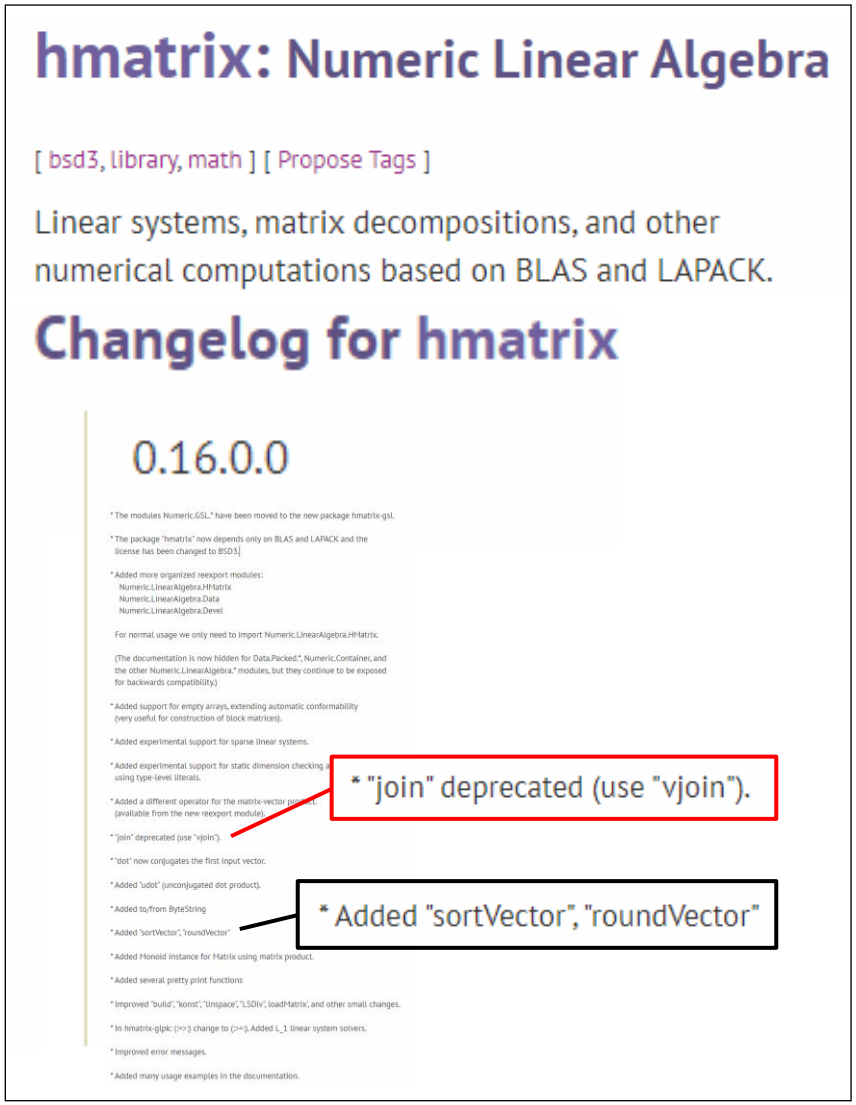
Setting

- Port **hmatrix** to ***Matrix***
- Simulating breaking updates in VL



≈ Data.List

base (Haskell standard library)



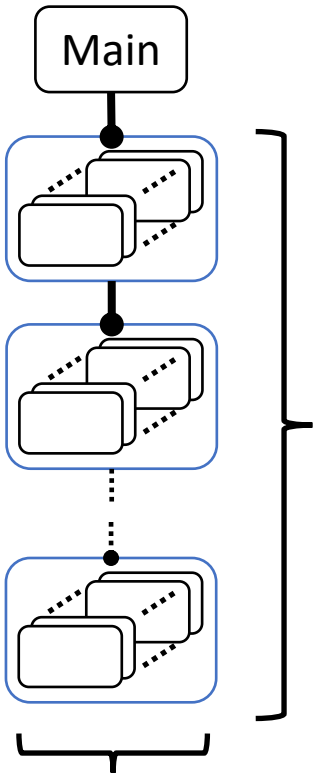
https://hackage.haskell.org/package/hmatrix-0.20.2
https://hackage.haskell.org/package/hmatrix-0.20.2/changelog

Evaluation

2. Compiler Performance

Benchmark setting

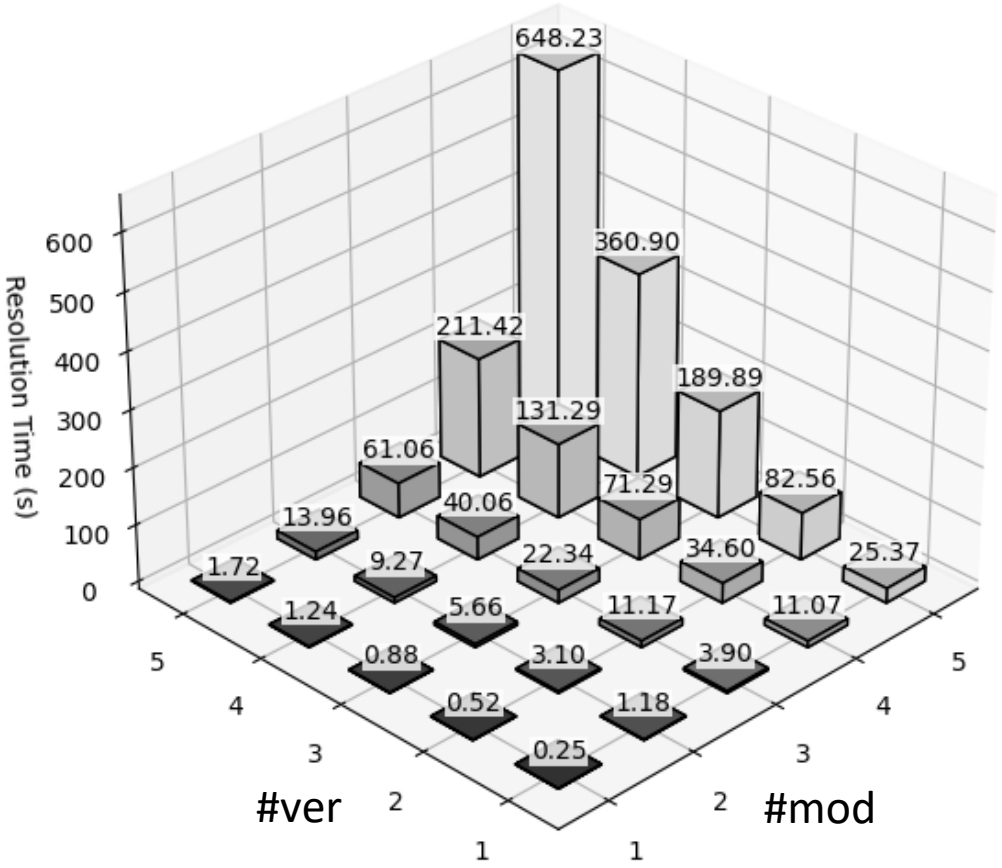
Ubuntu 22.04
Ryzen 7950X
Z3 version 4.12.2



Importing **#mod**-times nested dependencies

Each module has **#ver** versions

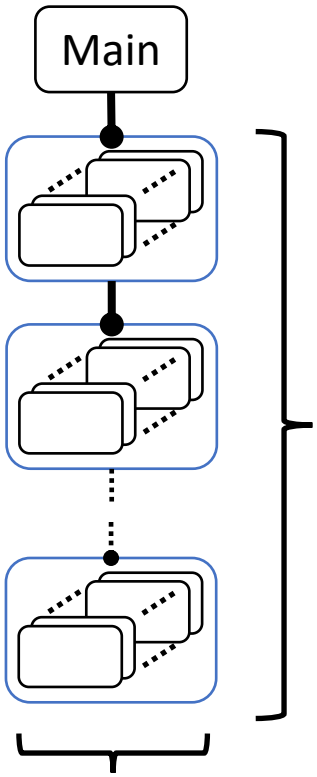
~500LOC / mod · ver



Evaluation

2. Compiler Performance

Benchmark setting

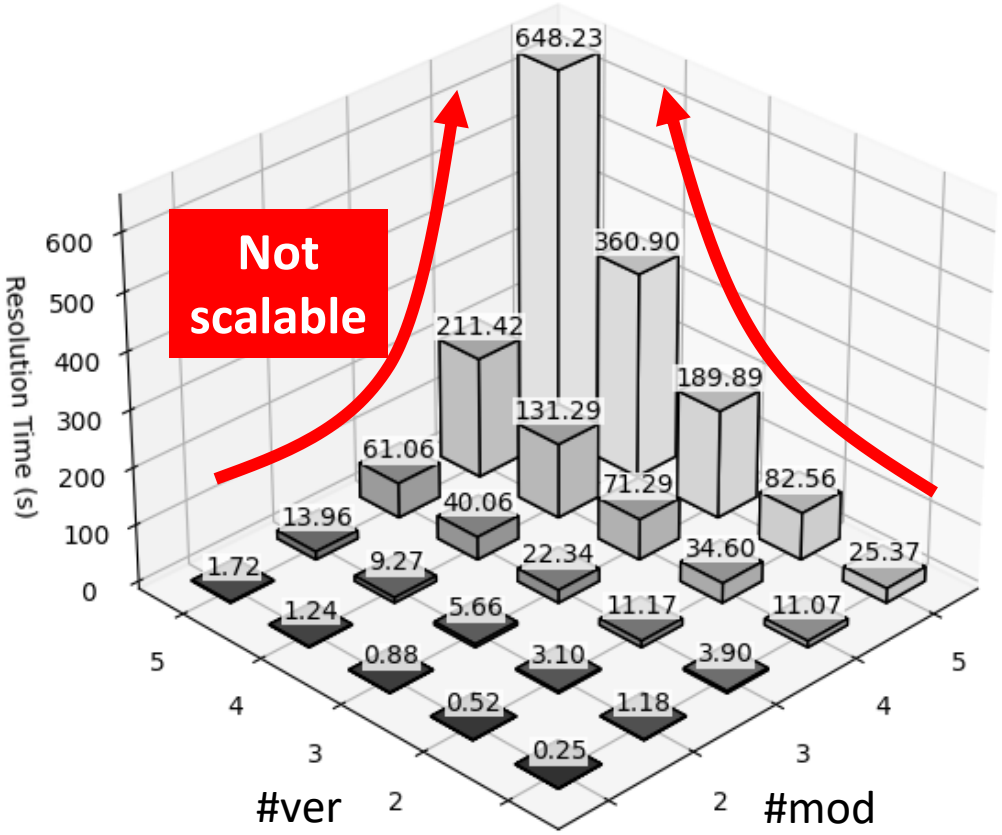


Ubuntu 22.04
Ryzen 7950X
Z3 version 4.12.2

Importing **#mod**-times nested dependencies

Each module has **#ver** versions

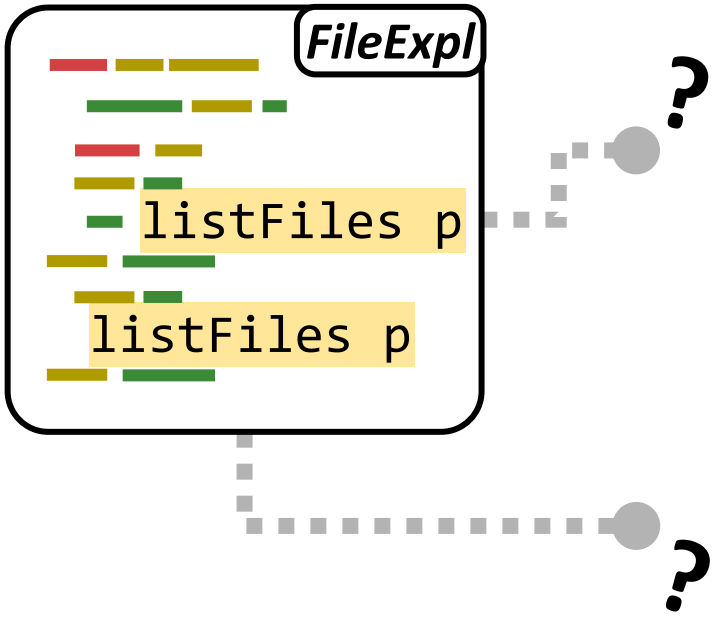
~500LOC / mod · ver



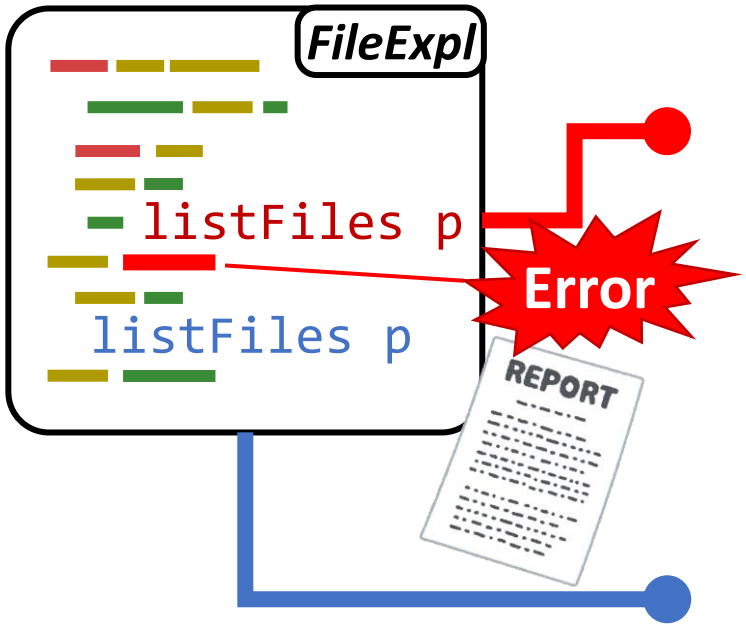
… but existing techniques can optimize the constraint resolution
(out-of-scope, short discussion in the paper)

Revisit Technical Challenges in PWV

①*Differentiate Programs by Contexts*



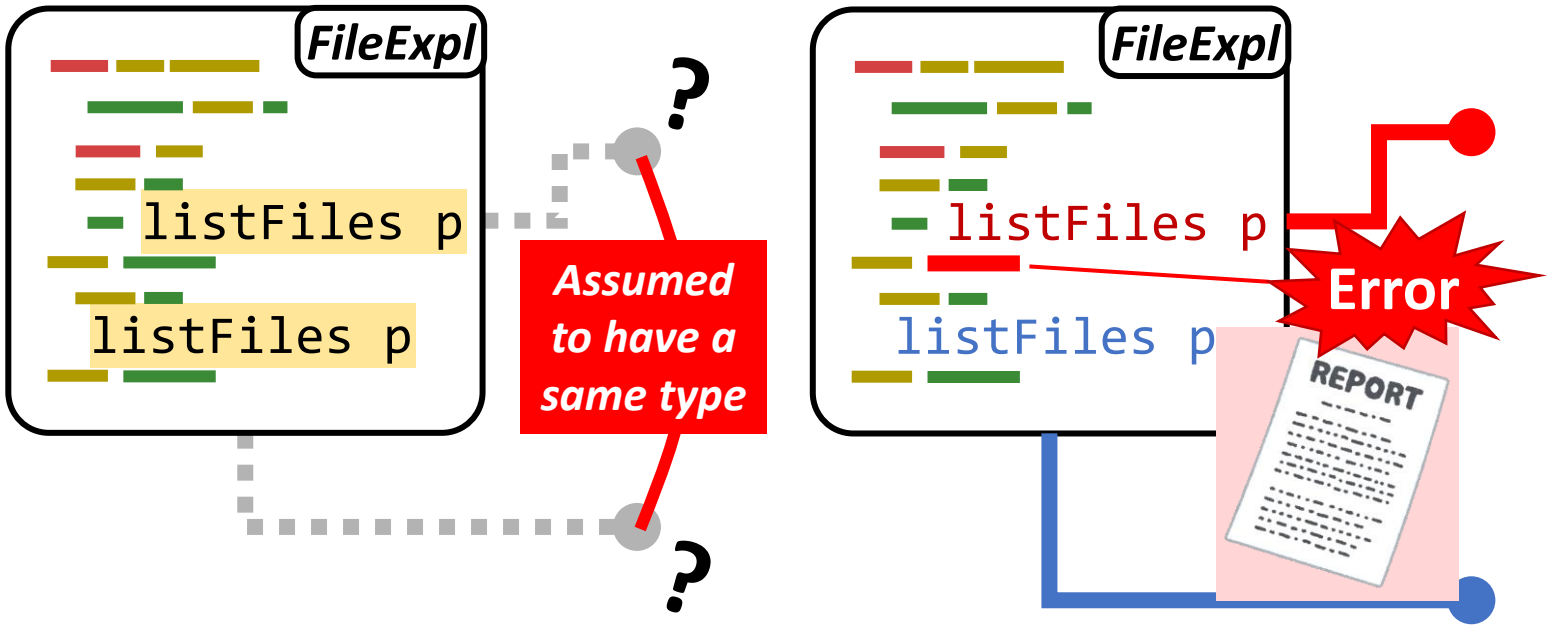
②*Aid for Resolving Semantic Incompatibility*



Current Limitation in VL

①Limited Support for Compatibility

②Mere Code Location Feedback



Future Work

①Further Compatibility Support

✅ ***Supported***

- Availability of functions
  (add/delete definitions)
- Behavioral incompatibility
  (w/o interface-level changes)
- Add/delete imports
  (w/o cyclic dependencies)

❓ ***Unsupported***

- **Type changes**

Batak-Java [SLE'22]

- Add/delete methods
- Class inheritance changes

①The Path Forward for VL

Can version inference work with ***varied function-*** and ***data types***?

```
main = do
  res <- fetchData url
  putStr $ "Data:" ++ res
```

Ver 1?

```
auth :: AuthMethod -> …
auth x =
  case x with
    Password msg -> …
  | OTP msg      -> …
```

Ver 2?

```
module SecureDataFetcher …

data AuthMethod =
    Password String
  | TwoFactor String
  | OTP String

fetchData ::     Response
    String -> IO String
```

Future Work

②Data Interoperability Across Versions

Can we address ***semantic incompatibilities*** automatically?

```
main = do
  let u1 = User
         { name = "Alice"
         , age  = 30 }
  let u2 = updateBirthYear u1
  let u3 = updateAge u2
  putStr $ show u3
```

```
data User = User
  { name :: String
  , age  :: Int
  , birthYear :: Int }

updateAge = …
updateBirthYear = …
```

Boundary of expected versions

Inserting adaptors?

`yearToAge ageToYear`

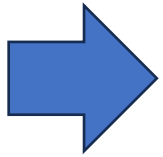
Generating common interface?

Summary

Contribution

Programming with versions
w/o version annotations

[‹Programming›'22]
λ_{VL}
Explicit
version annotations



[APLAS'23]
VL

IR
VLMini

Version inference
incorporating implicit versions

Implementation

on GHC with Z3
https://github.com/yudaitnb/vl

Full version

arXiv Formalization
Proof of soundness

Version Resource Semiring $\mathcal{R}$

Coeffect calculus: $\ell\mathcal{R}$PCF[Brunel'14], GrMini[Orchard'19]

$t ::= \ldots \mid x \mid t_1 t_2 \mid \lambda x.t \mid [t] \mid \mathrm{let}\,[x] = t_1 \,\mathrm{in}\, t_2$

$A ::= \ldots \mid A \to A \mid \Box_r A$

$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A]_r$

$r \in (\mathcal{R}, \oplus, 0, \otimes, 1)$

$t ::= \ldots \mid \{\overline{l = t}\} \mid t.l$

versioned values con-/de-structors

$\mathcal{R} = \mathbb{L}$ (version labels)

$r ::= \bot \mid \emptyset \mid \{l_i\} \mid r_1 \oplus r_2 \mid r_1 \otimes r_2$

λ_{VL} … and some corresponding typing rules

$\mathcal{R} = \{\mathrm{Irrelevant}, \mathrm{Private}, \mathrm{Public}\}$ (security level[Orchard'19])
e.g. $\Box_{\mathrm{Private}} A, \Box_{\mathrm{Public}} A$

$\mathcal{R} = \mathbb{N}$ (exact usage[Petricek'14])
e.g. $\Box_0 A, \Box_2 A$

Version Awareness

Additive part: *resource splitting*

$$\frac{\Gamma_1 \vdash t_1 : A \to B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \text{app}$$

Splitting resources for sub judgments

$$(\Gamma, x : [A]_r) + (\Gamma', x : [A]_s) = (\Gamma + \Gamma'), x : [A]_{r \oplus s}$$

Multiplication part: *resource demanding*

$$\frac{[\Gamma] \vdash t : A}{r * [\Gamma] \vdash [t] : \square_r A}\ \text{pr}$$

Requiring resources from a context

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

"$[t]$ available in r requires
all assumptions to be available in r."

Intuition to 0 and 1 in Semiring

Both 0 and 1 indicate unavailable resources.

Treated differently only in multiplication $\otimes$.

$$r_1 \otimes r_2 = \begin{cases} \bot & (r_1 = \bot \vee r_2 = \bot) \\ r_1 \cup r_2 & (otherwise) \end{cases}$$

$$\frac{\Gamma \vdash t : A}{\Gamma, [\Delta]_{0} \vdash t : A}\ \text{weak} \qquad (=\bot)$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_{1} \vdash t : B}\ \text{der} \qquad (=\emptyset)$$

$$\frac{\Gamma, x : [A]_{r}, \Gamma' \vdash t : B \quad r \sqsubseteq s}{\Gamma, x : [A]_{s}, \Gamma' \vdash t : B}\ \text{sub}$$

$\bot \sqsubseteq \emptyset \sqsubseteq \{l_i\} \sqsubseteq \cdots$

In other coeffect calculi, the semantic difference between 0 and 1 may be meaningful.

i.e.) Exact usage $(\mathbb{N}, +, 0, \cdot, 1, \equiv)$[Patriceik'14,Orchard'19]

λ_{VL} Typing Rules

$$\frac{}{\emptyset \vdash n : \mathrm{Int}}\ \text{int} \qquad \frac{}{x : A \vdash x : A}\ \text{var}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B}\ \text{abs}$$

$$\frac{\Gamma_1 \vdash t_1 : A \to B \quad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1\ t_2 : B}\ \text{app} \qquad \frac{\Gamma_1 \vdash t_1 : \square_r A \quad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ [x] = t_1\ \mathbf{in}\ t_2 : B}\ \text{let}$$

$$\frac{\Gamma \vdash t : A}{\Gamma, [\Delta]_0 \vdash t : A}\ \text{weak} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ \text{der} \qquad \frac{[\Gamma] \vdash t : A}{r * [\Gamma] \vdash [t] : \square_r A}\ \text{pr}$$

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

$$\frac{[\Gamma_i] \vdash t_i : A}{\bigcup(\{l_i\} * [\Gamma_i]) \vdash \left\langle \overline{l = t} \mid l_i \right\rangle : A}\ \text{veri} \qquad \frac{[\Gamma_i] \vdash t_i : A}{\bigcup(\{l_i\} * [\Gamma_i]) \vdash \left\{ \overline{l = t} \mid l_i \right\} : \square_{\bar{\{l\}}} A}\ \text{ver}$$

Properties

Well-typed versioned substitutions

(Well-typed linear substitutions hold as well)

Proved

$$\begin{cases} [\Delta] \vdash t' : A \\ \Gamma, x : [A]_r, \Gamma' \vdash t : B \end{cases} \Longrightarrow \Gamma + r \cdot \Delta + \Gamma' \vdash [t' \mapsto x]t : B$$

Type-safe extractions

Proved

$$[\Gamma] \vdash v : \square_{\boldsymbol{r}} A \implies \forall \boldsymbol{l_k} \in \boldsymbol{r}. \exists t'. \begin{cases} v.\boldsymbol{l_k} \longrightarrow t' \\ [\Gamma] \vdash t' : A \end{cases}$$

①Translate **VL** to **VLMini**

Girard's Translation

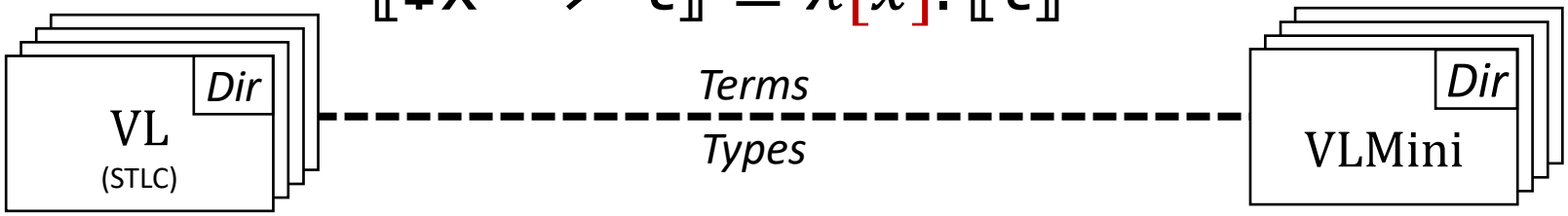
A generalization of the original for linear λ-calculus[Girard'87] to GrMini[Orchard'19]

Inserting ***syntactic annotation*** [_] where a value should be treated as a versioned value

Treat arguments as versioned values …

… and capture their version resources by promoted patterns

$$[\![\texttt{x}]\!] \equiv x$$
$$[\![\texttt{t s}]\!] \equiv [\![\texttt{t}]\!]\ [[\![\texttt{s}]\!]]$$
$$[\![\texttt{¥x -> t}]\!] \equiv \lambda[x].[\![\texttt{t}]\!]$$



$$[\![\texttt{A}]\!] \equiv A$$
$$[\![\texttt{A -> A}]\!] \equiv \square_r [\![A]\!] \to [\![B]\!]$$

②Version Inference

Constraint Generation

$$f : [\dots]_{\alpha_f}, x : [\dots]_{\alpha_x} \vdash [f\ x] \Rightarrow \square_{\alpha} A;\ \alpha \preccurlyeq \alpha_f \wedge \alpha \preccurlyeq \alpha_x \quad (\Rightarrow_{\mathrm{PR}})$$

②Algorithmic Type Inference

Variable dependencies

generated by inserted promotion

1.0.0 $\mathrm{listDir}' :: \square_{\alpha 1}(\square_{\beta 1}\mathrm{Str} \to \cdots) \mid C_1$

2.0.0 $\mathrm{listDir}' :: \square_{\alpha 2}(\square_{\beta 2}\mathrm{Str} \to \cdots) \mid C_2$

②Version Inference & ③Interface Bundling

Constraint Generation

Please see the paper for more details!

$$f : [\dots]_{\alpha_f}, x : [\dots]_{\alpha_x} \vdash [f\ x] \Rightarrow \square_{\alpha} A;\ \alpha \preccurlyeq \alpha_f \wedge \alpha \preccurlyeq \alpha_x \quad (\Rightarrow_{\mathrm{PR}})$$

②Algorithmic Type Inference

Variable dependencies

generated by inserted promotion

1.0.0 $\mathrm{listDir}' :: \square_{\alpha 1}(\square_{\beta 1}\mathrm{Str} \to \cdots) \mid \mathcal{C}_1$

2.0.0 $\mathrm{listDir}' :: \square_{\alpha 2}(\square_{\beta 2}\mathrm{Str} \to \cdots) \mid \mathcal{C}_2$

$\mathrm{listDir}' :: \square_{\alpha'}(\square_{\beta'}\mathrm{Str} \to \cdots)$

$$\mid \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \left(\begin{array}{l} (\alpha' \preccurlyeq \langle Dir \mapsto \mathbf{1.0.0} \rangle \wedge \cdots) \\ \vee\ (\alpha' \preccurlyeq \langle Dir \mapsto \mathbf{2.0.0} \rangle \wedge \cdots) \end{array} \right)$$

③Interface Bundling

Label dependencies

generated by availability checking

② Version Inference

Algorithmic Type Inference

Allocate resource variables and *collect constraints*

$$\underbrace{\Sigma; \Gamma \vdash t}_{\textbf{Input}} \Rightarrow \underbrace{A; \Sigma'; \theta; \mathcal{C}}_{\textbf{Output}}$$

Input

- t : Term
- Γ : Typing context
- Σ : Type variable kinds

Output

- A : Type
- $\mathcal{C}$: Constraints
- Σ' : Type variable kinds
- θ : Substitution

② Version Inference

Pattern Type Synthesis

$$(\lambda[x].\, t)\ [y]$$

Resource contexts

$$\underbrace{\Sigma;\, R \vdash p : A}_{\text{Input}} \rhd \underbrace{\Gamma; \Sigma'; \theta}_{\text{Output}}$$

Aggregate resources by $[p]$

$$\frac{\Sigma'; \alpha \vdash p : \beta \rhd \Delta; \Sigma''; \theta \quad \Sigma' \vdash A \sim \square_\alpha \beta \rhd \theta'}{\Sigma; - \vdash [p] : A \rhd \Delta; \Sigma''; \theta \uplus \theta'}\ (\text{p}\square)$$

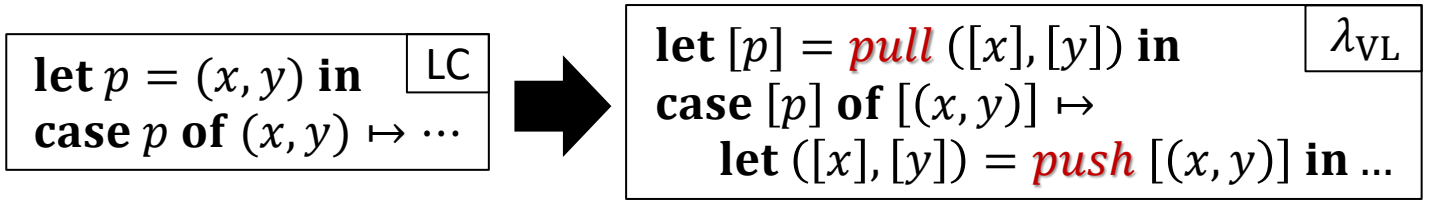
Convert the resource into assumption

$$\frac{\Sigma \vdash A : \text{Type} \quad \Sigma \vdash r : \text{Labels}}{\Sigma; r \vdash x : A \rhd x : [A]_r; \Sigma; \emptyset}\ [\text{pVar}]$$

② Version Inference

Data Structure Support

- Inserting *distributive combinators*[Huges'21]



Granule
[Orchard'19, Huges'21]

```
push : (a, b)[r] -> (a[r], b[r])
push [(x, y)]  = ([x], [y])
pull : (a[n], b[m]) -> (a, b)[n ⊓ m]
pull ([x], [y]) = [(x, y)]
```

- Motivation:
  How to propagate resources in-/out-side a data structure?

A versioned value of a tuple	*A tuple of versioned values*

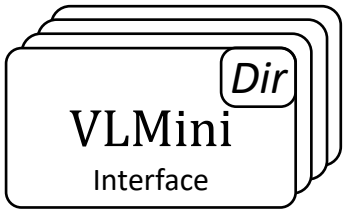
$fst\ p$ is ill-typed

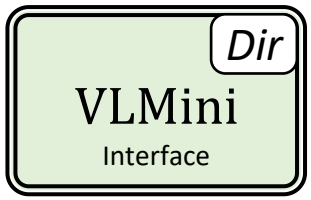
$$fst :: \square_{r'}(\mathrm{Int}, \mathrm{Int}) \to \mathrm{Int}$$
$$fst = \lambda[x].\ \mathbf{case}\ [x]\ \mathbf{of}\ [(x, y)] \mapsto x$$

$$p :: (\square_r \mathrm{Int}, \square_s \mathrm{Int})$$
$$p = ([x], [y])$$

③ Interface Bundling

Generate Multi-version Interface





listDir′ :: $\square_{\alpha 1}(\square_{\beta 1}\mathrm{String} \to \cdots)$ | $\mathcal{C}_1$ — 1.0.0

listDir′ :: $\square_{\alpha 2}(\square_{\beta 2}\mathrm{String} \to \cdots)$ | $\mathcal{C}_2$ — 2.0.0

listDir′ :: $\square_{\alpha'}(\square_{\beta'}\mathrm{String} \to \cdots)$

Generated constraints

$$\mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \left(\left(\begin{array}{c}\alpha' \preccurlyeq \langle Dir \mapsto \mathbf{1.0.0}\rangle \\ \wedge\ \alpha' \preccurlyeq \alpha_1 \\ \wedge\ \beta' \preccurlyeq \beta_1\end{array}\right)\right.$$ for 1.0.0

$$\left.\vee\left(\begin{array}{c}\alpha' \preccurlyeq \langle Dir \mapsto \mathbf{2.0.0}\rangle \\ \wedge\ \alpha' \preccurlyeq \alpha_2 \\ \wedge\ \beta' \preccurlyeq \beta_2\end{array}\right)\right)$$ for 2.0.0

Restriction
All versions must have a same type

$$\approx \square_{\left\{\begin{array}{l}[Dir \mapsto \mathbf{1.0.0}], \\ [Dir \mapsto \mathbf{2.0.0}]\end{array}\right\}}(\square_r \mathrm{Int} \to \mathrm{Int})$$

λ_{VL}

Implementation – How to use SMT solver

Vectorizing Constraints

Translate constraints to *symbolic lists*[SBV]

Label / Constraints		Symbolic lists
$\begin{bmatrix} A \mapsto 1.0.0 \\ B \mapsto 2.0.0 \end{bmatrix}$	$\approx$	$[1_A, 2_B]$
$\alpha_2 \preccurlyeq \langle B \mapsto 2.0.0 \rangle$	$\approx$	$v_{\alpha 2}.2 = 2_B$
$\alpha_1 \preccurlyeq \alpha_2$	$\approx$	$\forall i.\ v_{\alpha 1}.i = v_{\alpha 2}.i$

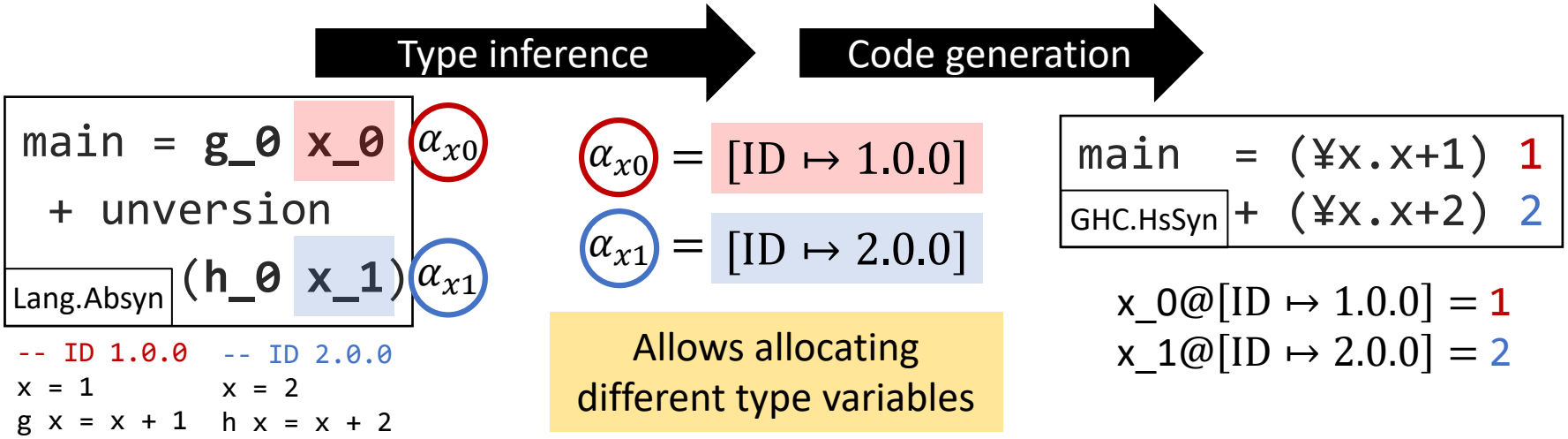
M_i	A	B
id_{mod}	1	2

id_{ver}	A	B
1.0.0	1_A	1_B
2.0.0	2_A	2_B

A label $[M_i \mapsto V_i]$ indicates that
the $id_{mod}(M_i)$-th element of a symbolic list is $id_{ver}(M_i, V_i)$.

Ad-hoc Polymorphism *via* Duplication

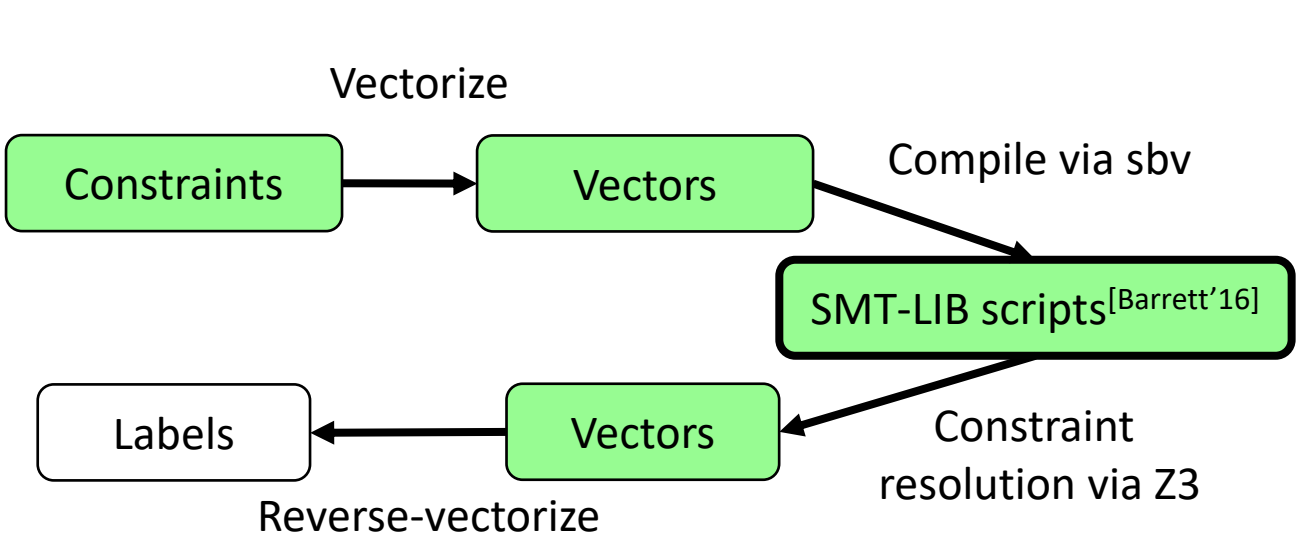
- ***Rename*** all occurrences of ***external symbols***
  - Replicate those in constraints and contexts as well

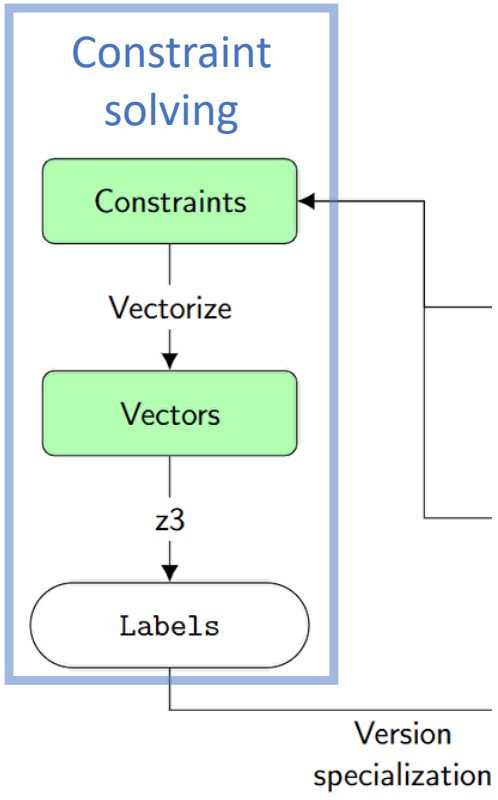


※ Full-resource polymorphism[Orchard'19] requires a revised compilation scheme and an extension to core calculus.

How to Estimate Complexity

Exponential for ***number of variables***, how to estimate*?*






```
(declare-const a Bool)
(declare-const b Bool)
(define-fun demorgan () Bool
    (= (and a b) (not (or (not a) (not b)))))
(assert (not demorgan))
(check-sat)
```

SMT-Lib Scripts

```
(declare-fun s0 () (_ BitVec 8))
(declare-fun s1 () (_ BitVec 8))
…
(define-fun s1552 () (_ BitVec 8) #x00)
                -- Special int value indicating undefined version
…
(define-fun s1553 () Bool (distinct s1 s1552))
(define-fun s1554 () Bool (= s0 s1))
(define-fun s1555 () Bool (and s1553 s1554))
(define-fun s1556 () Bool (= s1 s1552))
(define-fun s1557 () Bool (xor s1555 s1556))
…
(assert s3842)   -- Represents all constraints
(minimize s4362) -- Maximize the number of
                 -- undefined version elements
(check-sat)
(get-objectives)
…
```

Declare symbolic variables

Constraints

New symbolic variables per variable/label dependency

Assertion

Inspecting solution models

Size of Constraints

Number of definitions

Vector size

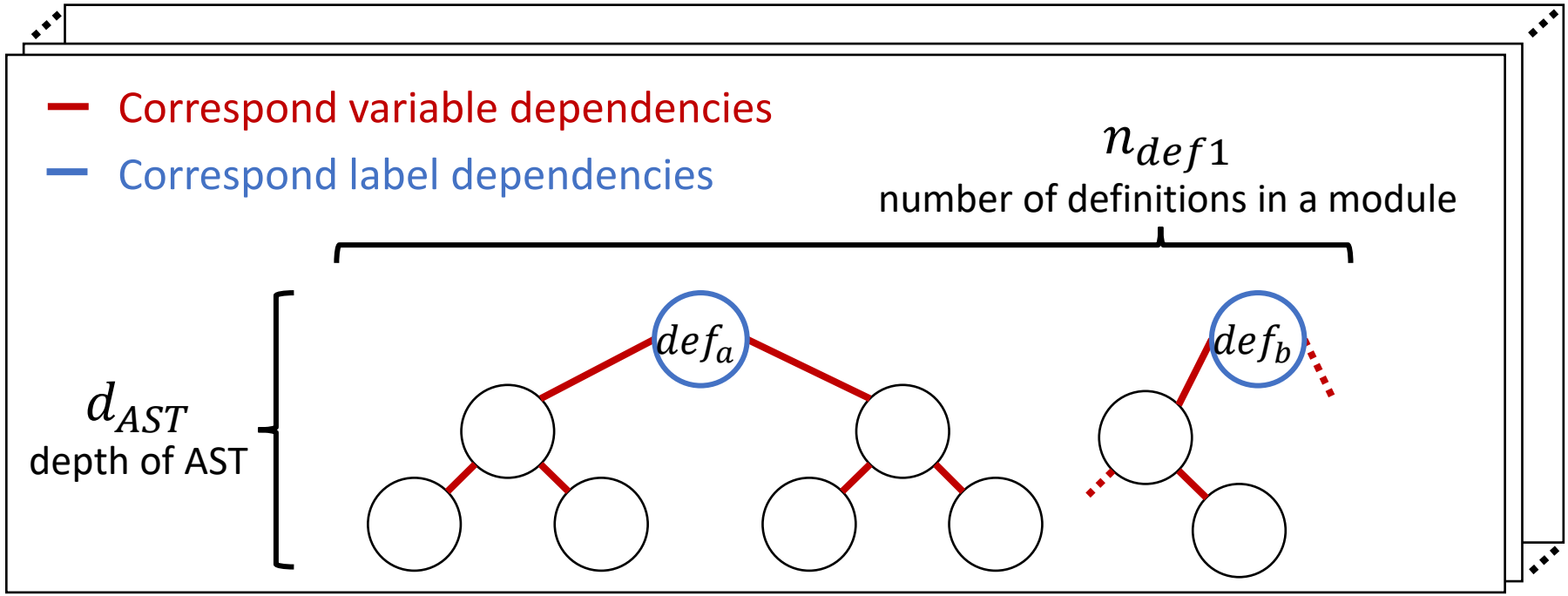
Number of AST edges

Size of *variable dep.* : $O\left(n_{mod} n_{def} \mathbf{2^{d_{AST}}}\right)$

Size of *label dep.* : $O(n_{mod} n_{def})$

$O(n_{def}) = O(n_{def1} n_{ver} n_{mod})$

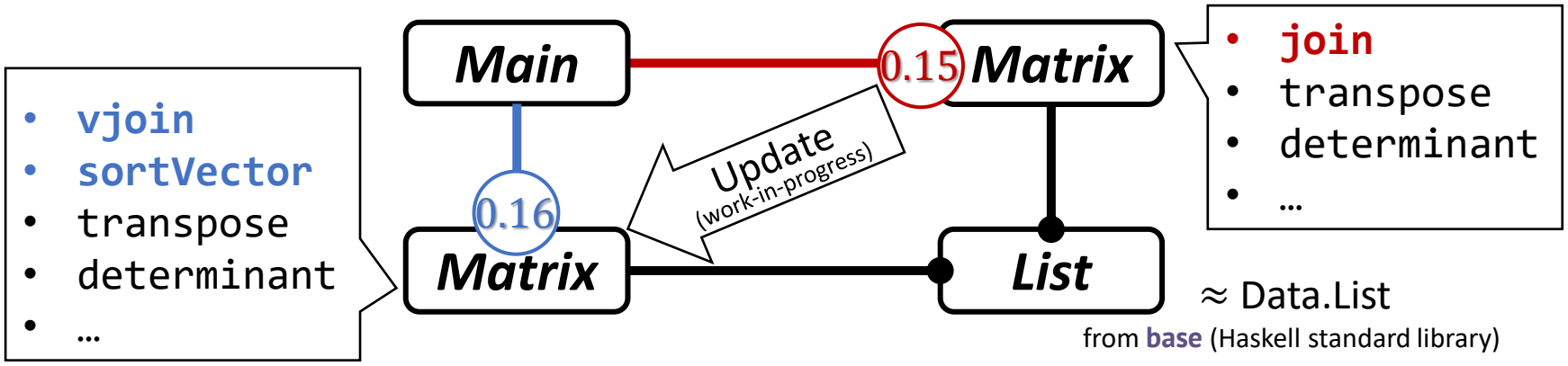
$\times n_{ver} \times n_{mod}$



Case Study – **hmatrix**

Module Structure

Reproducing incompatibility with ***Matrix*** and ***List***



version	join	sortVector
< 0.15.0	***available***	unavailable
≥ 0.16.0	Deleted (replaced vjoin)	***available***

hmatrix

```
Vector a  =>  List Int
Matrix a  =>  List (List Int)
```

Matrix

Case Study – **hmatrix**

Handling Multiple Versions in a Code

Compile

Dispatching to a consistent version of programs, with **unversion** as boundaries

```
main =
  let vec    = [2, 1]
      sorted = unversion
                 (sortVector vec)
      m22 = join -- [[1,2],[2,1]]
              (singleton sorted)
              (singleton vec)
  in determinant m22
```

Haskell AST (prettyprint)

```
module Main where
main = ...
  -- join
  (let join = ¥xs -> ¥ys ->
      case xs of
        [] -> ys
        x : xs -> (:) x (join xs ys)
  in join)
...
  -- sortVector
  (let sortVector = ¥xs ->
      case xs of
        []  -> []
        [x] -> [x]
        xs  -> (¥r -> (let vjoin = ...  in vjoin)
                        (sortVector
                          ((let init = ... in init) r))
                        [(let last = ... in last) r])
               ((let bubble = ... in bubble) xs)
  in sortVector)
...
```

Consistent in 0.15

Consistent in 0.16

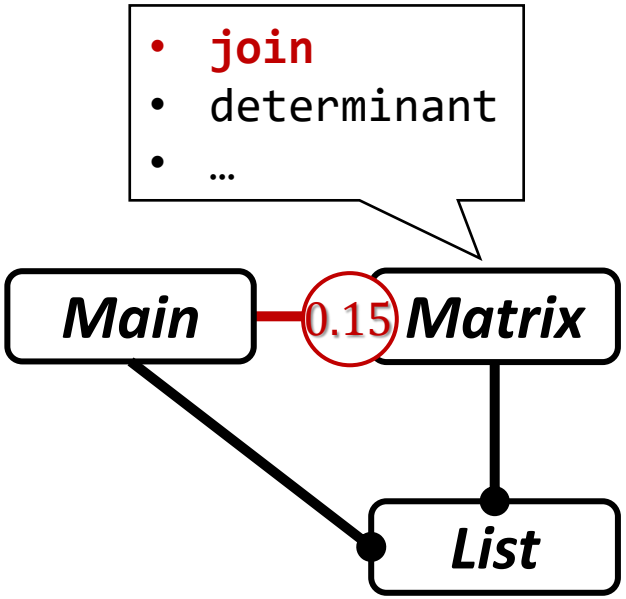
```
> ghc –o main Main.hs
> ./main
-3
```

Note: inserting IO function “print” manually

`Main` Module (***before*** update)

Haskell

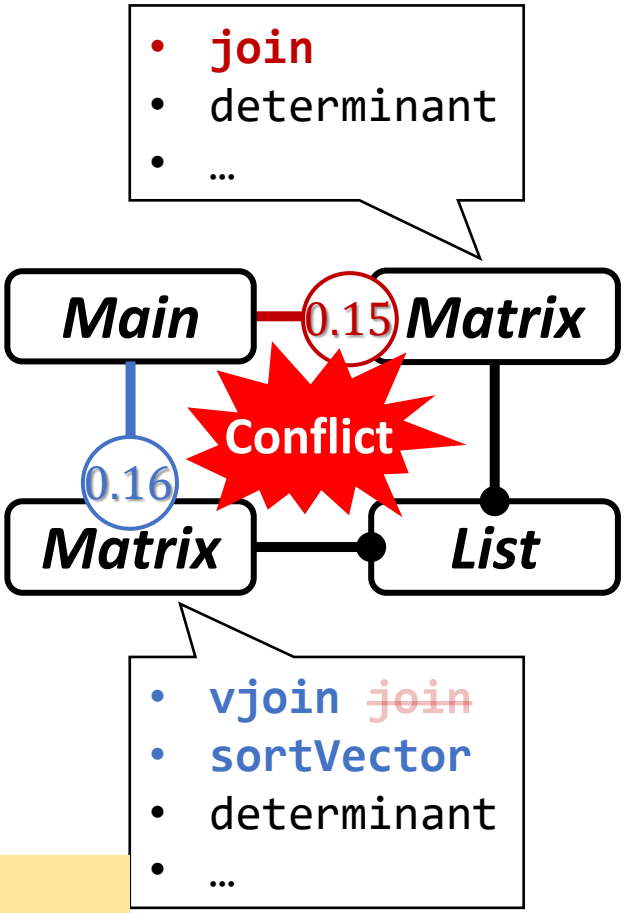
```
module Main where

import Matrix
import List

main =
  let vec  = [2, 1]
      vec' = [1, 2]
      m22 = join -- [[1,2],[2,1]]
             (singleton vec')
             (singleton vec)
  in determinant m22
```



Main Module (*after* update)

```haskell
module Main where

import Matrix
import List

main =
  let vec  = [2, 1]
      sorted = sortVector vec
      m22 = join -- [[1,2],[2,1]]
              (singleton sorted)
              (singleton vec)
  in determinant m22
```

Haskell




```
main.hs:1:38: error:
    Variable not in scope: sortVector
```

Case Study – **hmatrix**

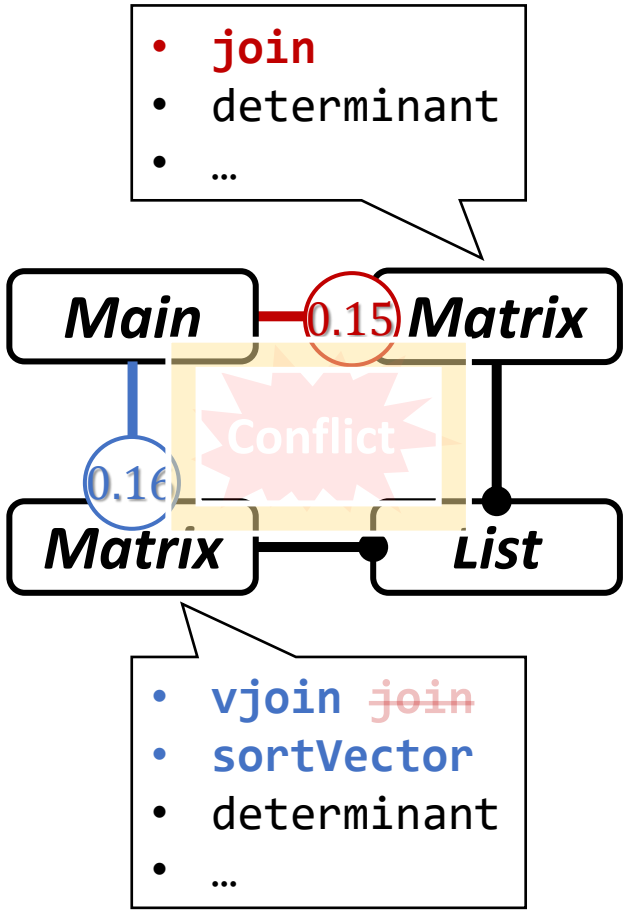
Detecting Inconsistent Version

```
module Main where

import Matrix
import List

main =
  let vec  = [2, 1]
      sorted = sortVector vec
      m22 = join -- [[1,2],[2,1]]
            (singleton sorted)
            (singleton vec)
  in determinant m22
```

VL

Version conflicts resolved,
but ***no consistent versions***

Inconsistent

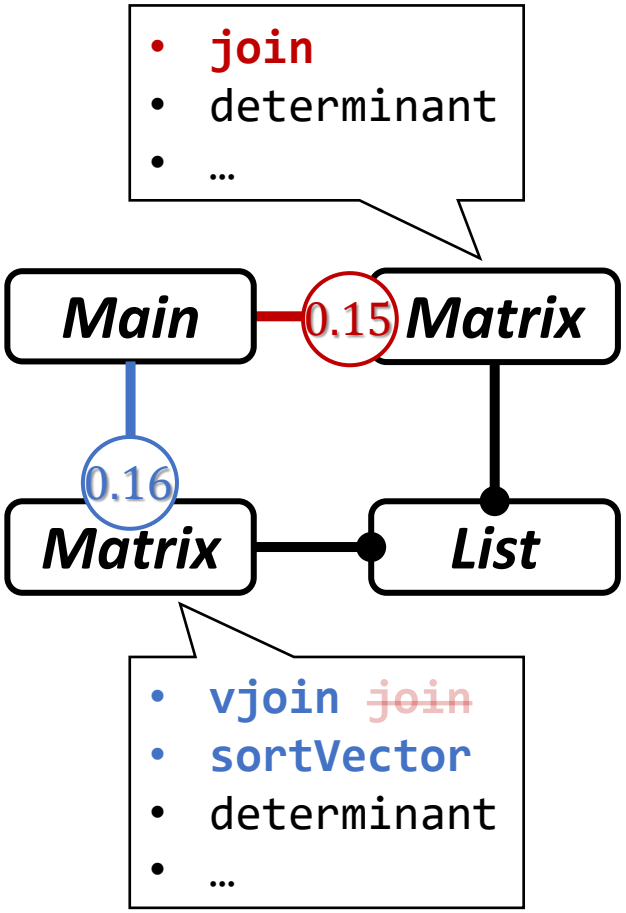


Handling Two Versions in One Client

```
module Main where

import Matrix
import List

main =
  let vec    = [2, 1]
      sorted = unversion
                 (sortVector) vec
      m22 = join -- [[1,2],[2,1]]
                 (singleton sorted)
                 (singleton vec)
  in determinant m22
```

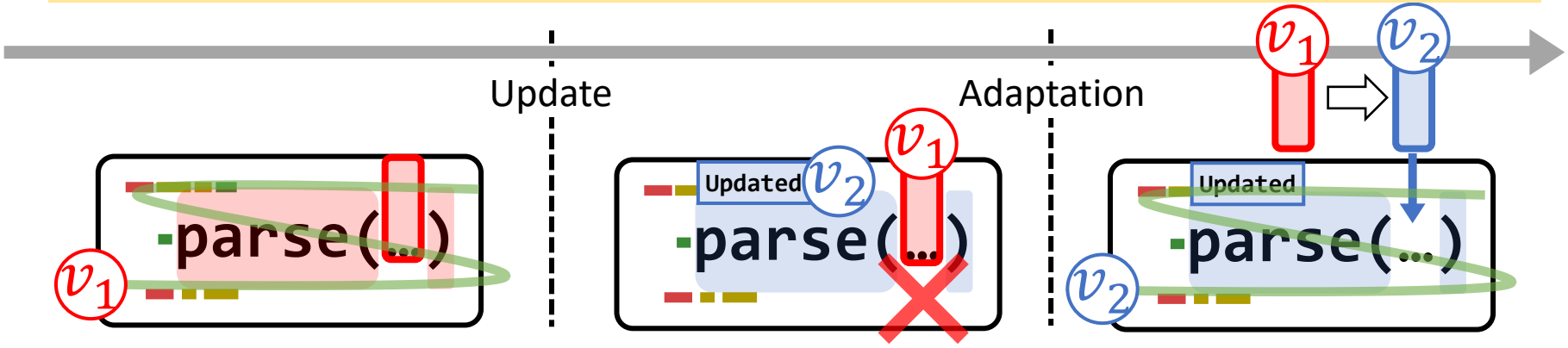
VL

No longer depends on 0.16



Automatic Adaptations

Detecting incompatibilities and ***inserting adapters*** automatically



Concept:

- **Code repository**, a *persistent definition/package store*
- Working environment(s) that are ***views*** into the code repository

Nix [Dolstra'04]: Hash-tagged packages + Nix package manager
Unison: Hash-tagged definitions + Unison code base manager

+ ***Consistency checking within expressions***