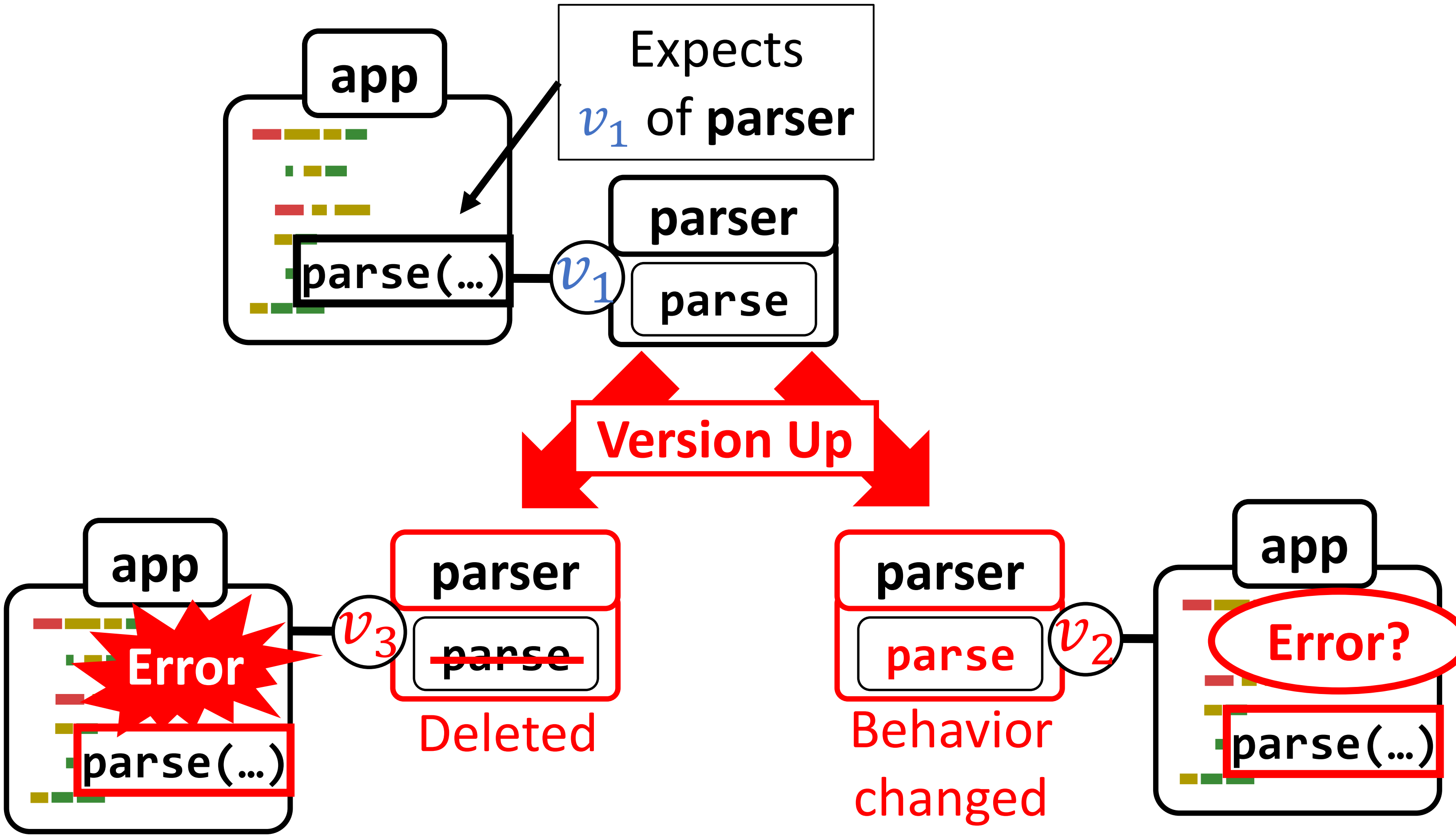
A Step toward Programming with Versions in Real-World Functional Languages

Yudai Tanabe[a)]Luthfan Anshar Lubis[a)]
Tomoyuki Aotani[b)] Hidehiko Masuhara[a)]

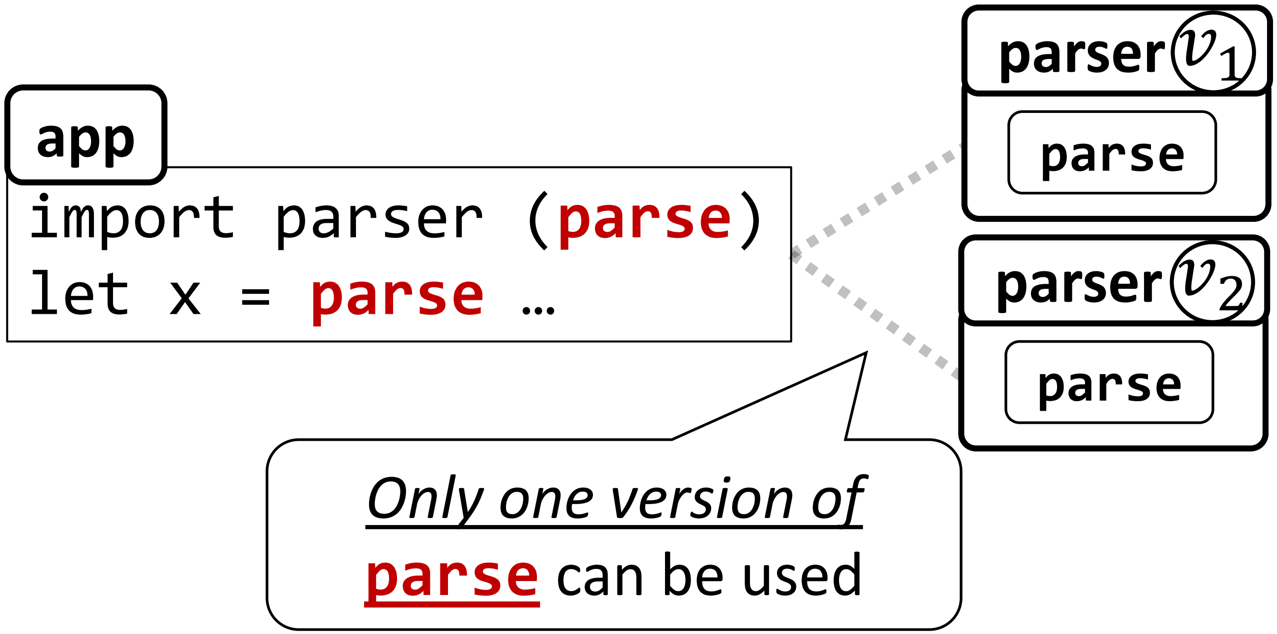
(a) Tokyo Institute of Technology (b) Mamezou

Version Update May Break Software



Background

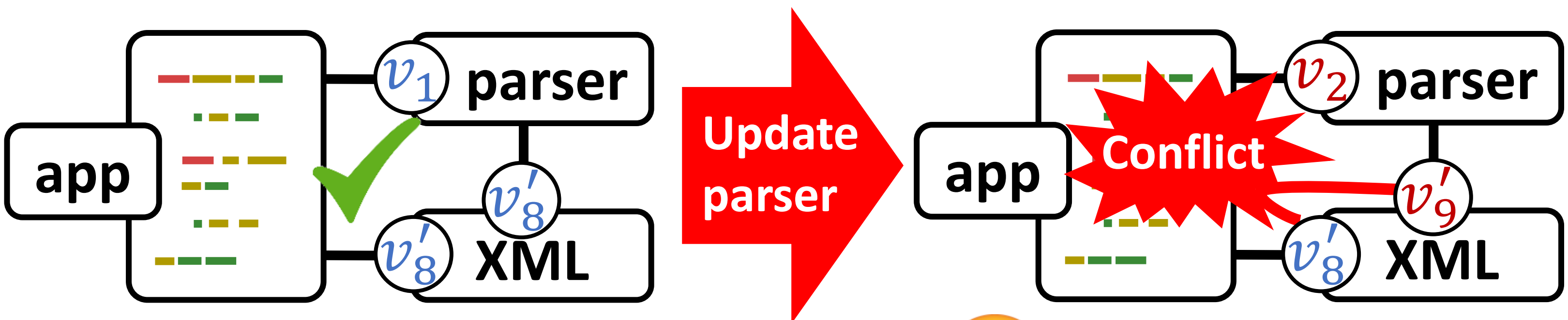
Implicit Assumption: One-version-at-a-time



Background

Dependency Hell

- Difficult to resolve indirect dependency conflicts in a software with complex dependencies

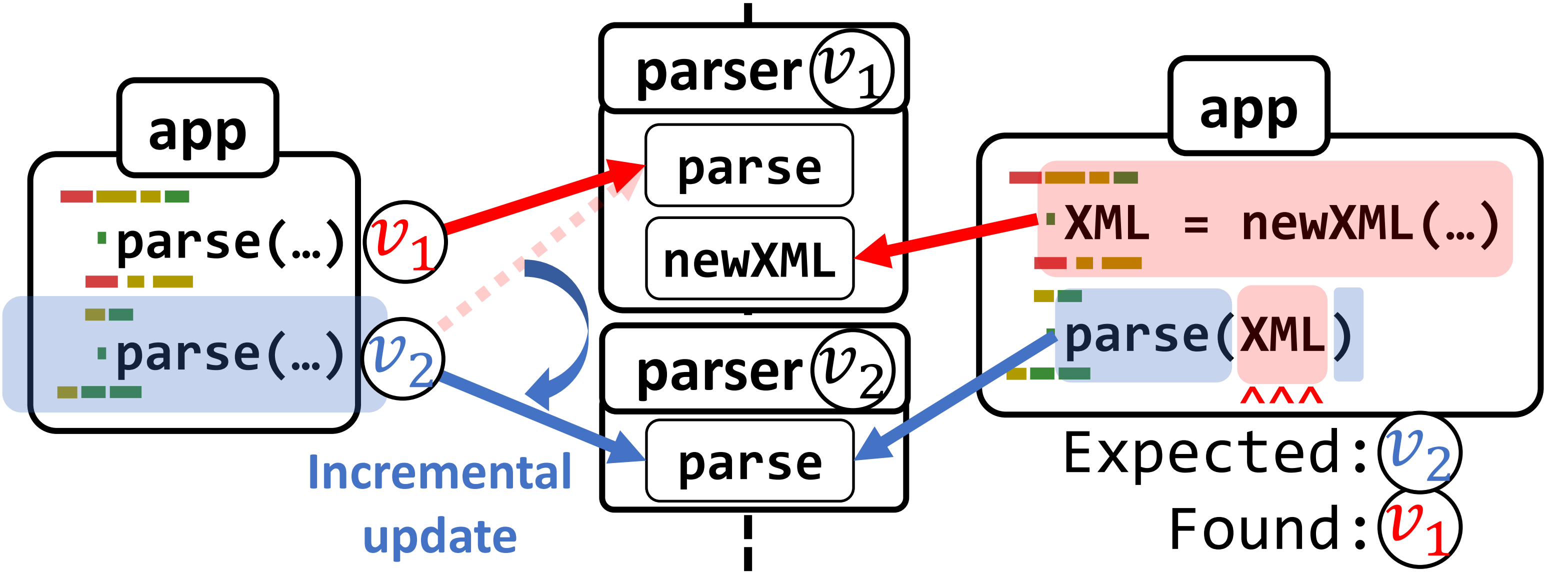


So ... how do I fix it?

Programming Language with Versions

Can use two versions in one client

Can detect inconsistent usages across versions



Core calculus: λ_{VL} [Tanabe'18, Tanabe'22]

Versioned Values

Version labels

$$\boxed{l_1 \mapsto \{v_1\},\ l_2 \mapsto \{v_2\}}$$

- **Versioned records** $\{\ \overline{l_i = t_i}\ \}$

  Multiple versions in one value

$$parse = \begin{Bmatrix} l_1 = parse_{v1} \\ l_2 = parse_{v2} \end{Bmatrix}$$

$parse_{v1}$

```
= \xml ->
  let elem = pop …
  in …
```

$parse_{v2}$

```
= \xml ->
  let elem = pop …
  in …
```

- **Extraction** $t.\,l_k$

  Evaluate t in the specific version

$$[parse\ XML].\,l_1$$

$\longrightarrow parse_{v1}\ obj_{v1}$

$\longrightarrow$
```
[("English","hello"),
 ("French","bonjour"),
 ("Japanese","konnichiwa")]
```

Background

λ_{VL} Type System

- ***Versions as Resources*** :
  Types are tagged with ***available*** and ***consistent*** *version labels*

$$parse : \Box_{\underline{\{l_1,l_2\}}}(\text{XML} \rightarrow A) \qquad XML : \Box_{\underline{\{l_1,l_2,l_3\}}}\text{XML}$$

$$parse = \begin{Bmatrix} \boldsymbol{l_1} = parse_{v1} \\ \boldsymbol{l_2} = parse_{v2} \end{Bmatrix} \qquad XML = \begin{Bmatrix} \boldsymbol{l_1} = obj_1 \\ \boldsymbol{l_2} = obj_2 \\ \boldsymbol{l_3} = obj_3 \end{Bmatrix}$$

$$\begin{array}{l} \text{let } [f] = parse \text{ in} \\ \text{let } [x] = XML \text{ in} \\ [f\ x] \end{array} \quad : \Box_{\underline{\{l_1,l_2\}}}\, A$$

$$\|$$

$$\{l_1, l_2\} \cap \{l_1, l_2, l_3\}$$

λ_{VL} Type System

- ***Inspect the consistency*** of version extraction

$$parse : \Box_{\{l_1,l_2\}}(\mathrm{XML} \to A)$$
$$XML : \Box_{\{l_1,l_2,l_3\}}\mathrm{XML}$$

[ERROR]
Expected l_3,
but got l_1, l_2

$\text{let } [f] = parse \text{ in}$
$\text{let } [x] = XML \text{ in}$ $: A$
$[f\ x].\,l_1$

because $l_1 \in \{l_1, l_2\} \cap \{l_1, l_2, l_3\}$

$\text{let } [f] = \underline{parse} \text{ in}$
$\text{let } [x] = XML \text{ in}$ $:$
$[f\ x].\,l_3$

because $l_3 \notin \{l_1, l_2\} \cap \{l_1, l_2, l_3\}$

Peculiar Syntax of λ_{VL}

- ***Requires high proficiency*** in the resource-aware type system

$$
\begin{aligned}
&\mathbf{let}\ [f] = parse\ \mathbf{in} \\
&\mathbf{let}\ [x] = XML\ \mathbf{in} \\
&[f\ x].l_3
\end{aligned}
$$

Substructural terms

Contextual-let binding	Promotion
$\mathbf{let}\ [x] = t\ \mathbf{in}\ t$	$[t]$

from GrMini[Orchard'19], ℓRPCF[Brunel'14]

Exposed Version in λ_{VL} Programs

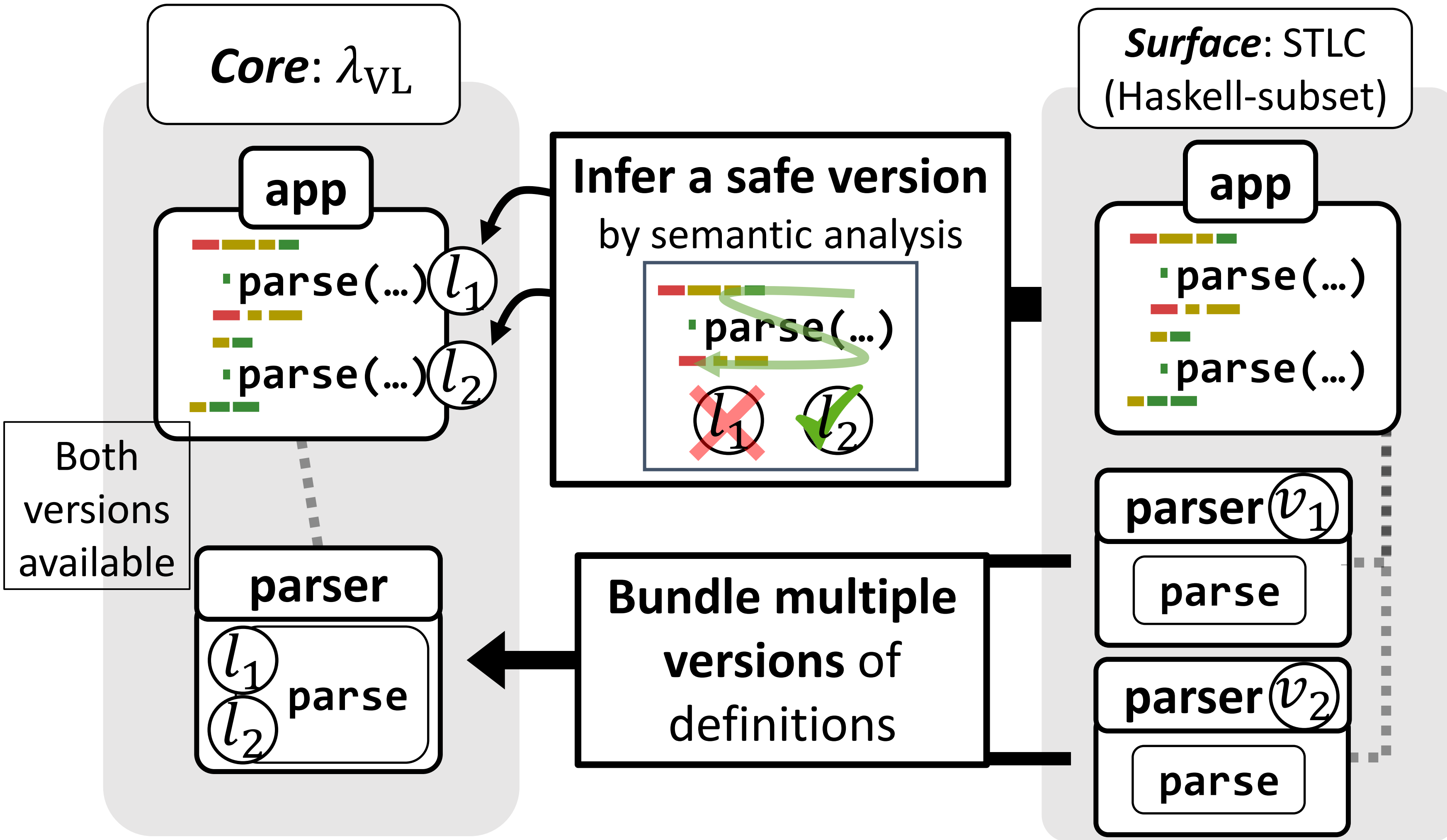
- ***Requires comprehension*** of correspondence between labels and versions

$$\begin{array}{l}\mathbf{let}\ [f] = parse\ \mathbf{in}\\ \mathbf{let}\ [x] = XML\ \mathbf{in}\\ [f\ x].\boldsymbol{l_3}\end{array}$$

Label-dependent terms

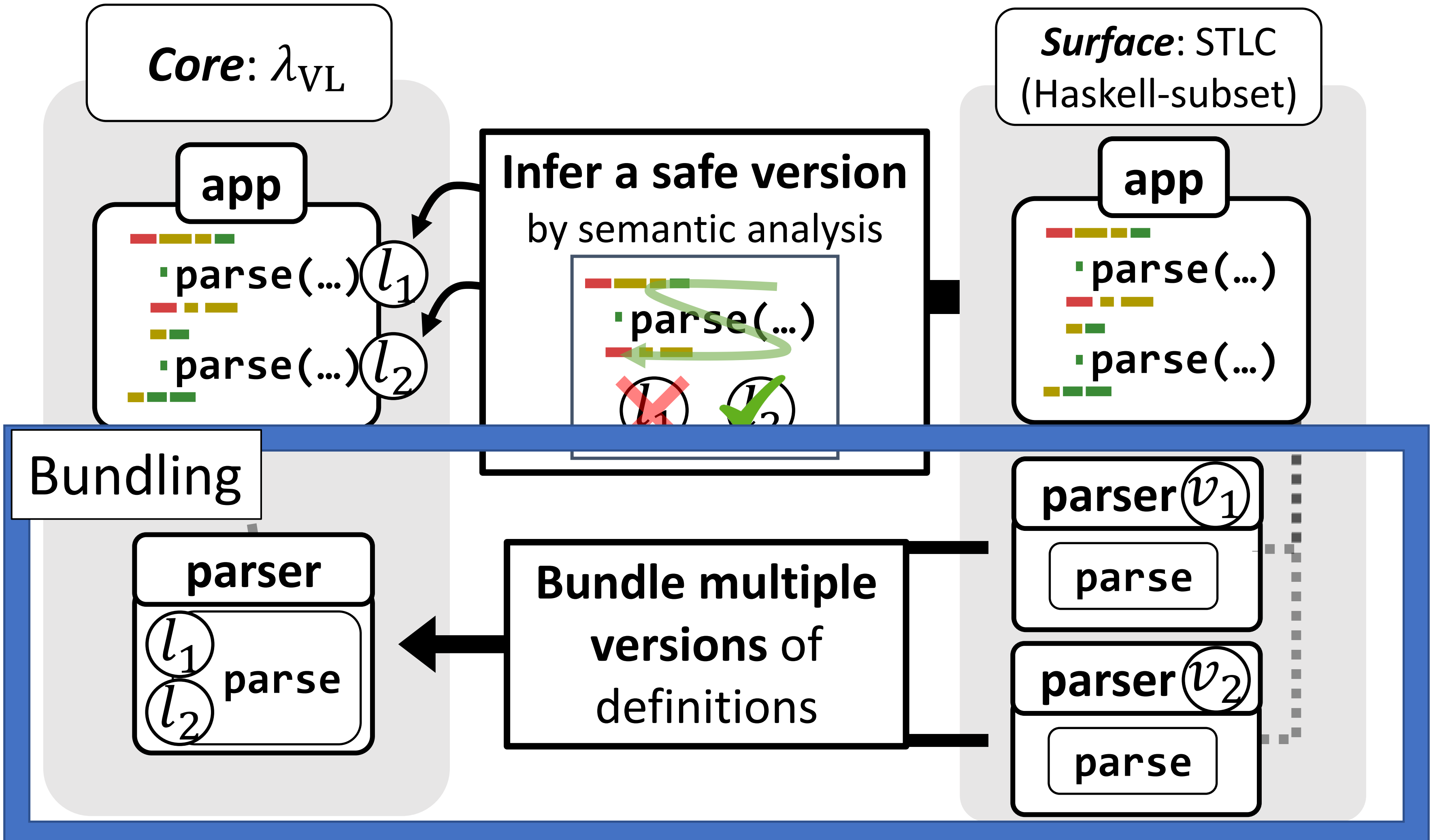
Versioned records	Extraction
$\{\ \overline{\boldsymbol{l_i} = t_i}\ \}$	$t.\boldsymbol{l_k}$

Ordinary Functional Language as a Surface Language for λ_{VL}



Research Proposal

Ordinary Functional Language as a Surface Language for λ_{VL}



An Example of Bundling

λ_{VL}

Main

$$main : \Box_{\{l_1,l_2\}}\mathrm{Int}$$
$$main = [id\ [n]]$$

ID

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

Bundling

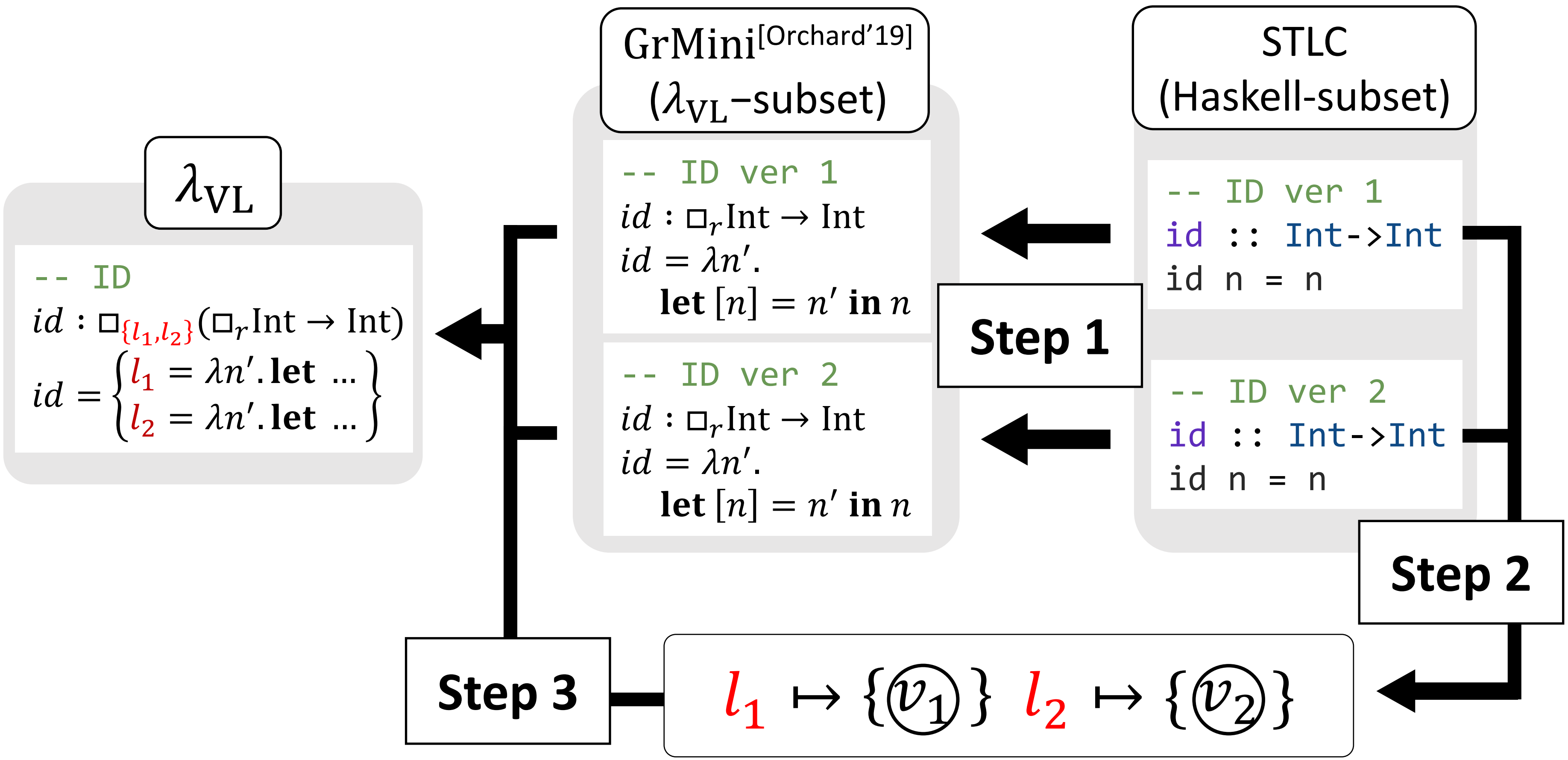
STLC (Haskell-subset)

```haskell
-- Main
main :: Int
main = id n
```

```haskell
-- ID version 1
id :: Int -> Int
id n = n
n :: Int
n = 1
```

```haskell
-- ID version 2
id :: Int -> Int
id n = n
n :: Int
n = 2
```

Bundling Overview



Bundling Overview

λ_{VL}

```
-- ID
```

$id : \square_{\{l_1, l_2\}}(\square_r \mathrm{Int} \to \mathrm{Int})$

$id = \left\{\begin{array}{l} l_1 = \lambda n'.\,\mathbf{let}\ \ldots \\ l_2 = \lambda n'.\,\mathbf{let}\ \ldots \end{array}\right\}$

GrMini[Orchard'19]
(λ_{VL}–subset)

```
-- ID ver 1
```

$id : \square_r \mathrm{Int} \to \mathrm{Int}$
$id = \lambda n'.$
$\quad \mathbf{let}\ [n] = n'\ \mathbf{in}\ n$

```
-- ID ver 2
```

$id : \square_r \mathrm{Int} \to \mathrm{Int}$
$id = \lambda n'.$
$\quad \mathbf{let}\ [n] = n'\ \mathbf{in}\ n$

STLC
(Haskell-subset)

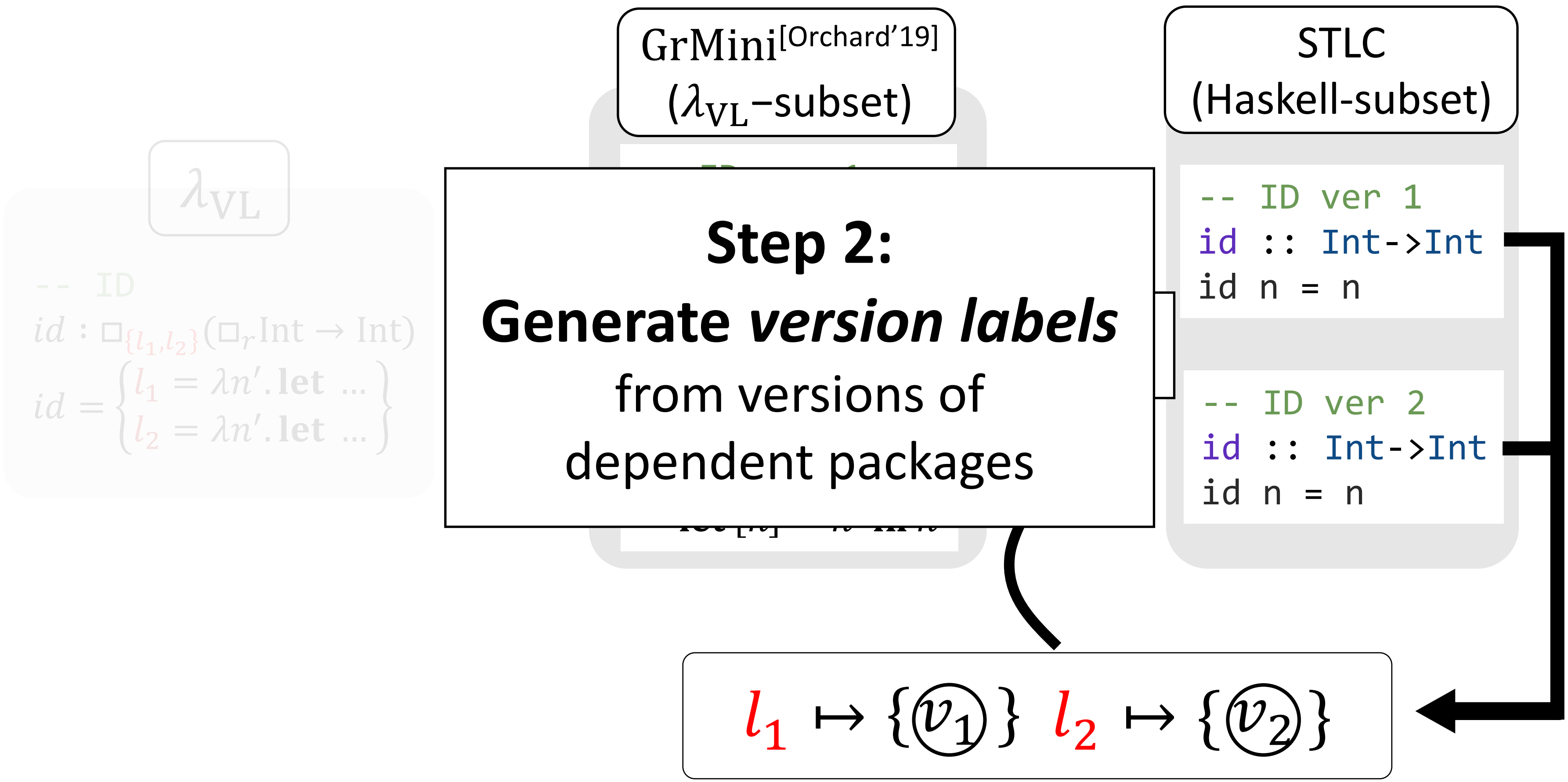
```haskell
-- ID ver 1
id :: Int->Int
id n = n
```

```haskell
-- ID ver 2
id :: Int->Int
id n = n
```

Step 1: Girard's translation

Any term and type derivation in STLC can be translated into GrMini (λ_{VL}-subset)[Orchard'19]

Bundling Overview



Methods

Bundling Overview

λ_{VL}

```
-- ID
```

$id : \Box_{\{l_1,l_2\}}(\Box_r \mathrm{Int} \to \mathrm{Int})$

$id = \begin{Bmatrix} l_1 = \lambda n'.\mathbf{let}\ \dots \\ l_2 = \lambda n'.\mathbf{let}\ \dots \end{Bmatrix}$

GrMini[Orchard'19] (λ_{VL}–subset)

```
-- ID ver 1
```

$id : \Box_r \mathrm{Int} \to \mathrm{Int}$
$id = \lambda n'.$
$\quad \mathbf{let}\ [n] = n'\ \mathbf{in}\ n$

```
-- ID ver 2
```

$id : \Box_r \mathrm{Int} \to \mathrm{Int}$
$id = \lambda n'.$
$\quad \mathbf{let}\ [n] = n'\ \mathbf{in}\ n$

STLC (Haskell-subset)

```
-- ID ver 1
id :: Int->Int
id n = n
```

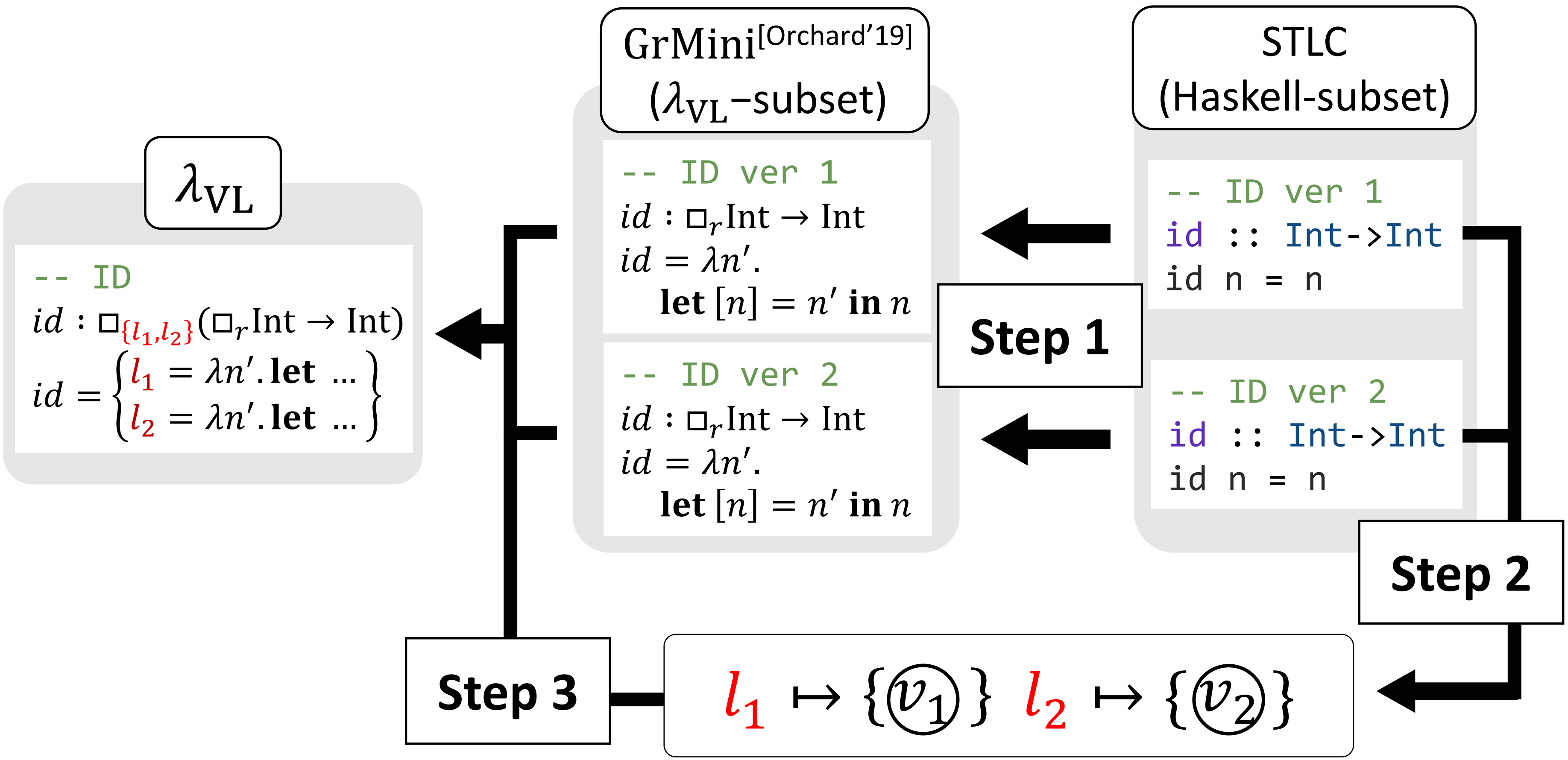
```
-- ID ver 2
id :: Int->Int
id n = n
```

$l_1 \mapsto \{ v_1 \}\quad l_2 \mapsto \{ v_2 \}$

Step 3:
Bundle top-level values
tagged with version labels
as a versioned value

Methods

Bundling Overview



Step 1: Girard's Translation

Version resource variable

(instantiated in the type checking)

Types:

$$[\![\mathsf{A}]\!] \equiv A$$

$$[\![\mathsf{A} \ \texttt{->} \ \mathsf{A}]\!] \equiv \square_{r} [\![A]\!] \rightarrow [\![B]\!]$$

Terms:

$$[\![\mathsf{x}]\!] \equiv x$$

$$[\![\backslash\mathsf{x} \ \texttt{->} \ \mathsf{t}]\!] \equiv \lambda x'.\, \mathbf{let}\, [x] = x' \,\mathbf{in}\, [\![\mathsf{t}]\!]$$

$$[\![\mathsf{t} \ \ \mathsf{s}]\!] \equiv [\![\mathsf{t}]\!] \, [[\![\mathsf{s}]\!]]$$

STLC
(Haskell-subset)

```haskell
-- Main
main :: Int
main = id n
```

```haskell
-- ID version 1
id :: Int -> Int
id n = n
n :: Int
n = 1
```

```haskell
-- ID version 2
id :: Int -> Int
id n = n
n :: Int
n = 2
```

Step 1: Girard's Translation

GrMini (λ_{VL}–subset)

$$id : \square_r \text{Int} \to \text{Int}$$
$$id = \lambda n'.\,\mathbf{let}\ [n] = n'\ \mathbf{in}\ n$$
$$n : \text{Int}$$
$$n = 1$$

STLC (Haskell-subset)

```haskell
id :: Int -> Int
id n = n
n :: Int
n = 1
```

$$\llbracket \texttt{\textbackslash n -> n} \rrbracket \equiv \lambda n'.\,\mathbf{let}\ [n] = n'\ \mathbf{in}\ \llbracket n \rrbracket$$
$$\equiv \lambda n'.\,\mathbf{let}\ [n] = n'\ \mathbf{in}\ n$$

$$\llbracket \texttt{Int -> Int} \rrbracket \equiv \square_r \llbracket \texttt{Int} \rrbracket \to \llbracket \texttt{Int} \rrbracket$$
$$= \square_r \text{Int} \to \text{Int}$$

Methods

Step 1: Girard's Translation

GrMini
(λ_{VL}–subset)

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

STLC
(Haskell-subset)

```
-- Main
main :: Int
main = id n
```

$$\llbracket \texttt{id n} \rrbracket \equiv \llbracket id \rrbracket\ [\llbracket n \rrbracket] \equiv id\ [n]$$

$$\llbracket \texttt{Int} \rrbracket \equiv \mathrm{Int}$$

Step 2: Generate Version Labels

- Maps *version labels* to versions of external module

$$l_1 \mapsto \{\texttt{ID version 1}\}$$
$$l_2 \mapsto \{\texttt{ID version 2}\}$$

- Consider cartesian product If multiple external modules exist

$$l_1 \mapsto \{\texttt{ID ver 1, N ver 1}\}$$
$$l_2 \mapsto \{\texttt{ID ver 1, N ver 2}\}$$
$$l_3 \mapsto \{\texttt{ID ver 2, N ver 1}\}$$
$$l_4 \mapsto \{\texttt{ID ver 2, N ver 2}\}$$

STLC
(Haskell-subset)

```haskell
-- Main
main :: Int
main = id n
```

```haskell
-- ID version 1
id :: Int -> Int
id n = n
n :: Int
n = 1
```

```haskell
-- ID version 2
id :: Int -> Int
id n = n
n :: Int
n = 2
```

Step 3: Bundling Top-level Values

λ_{VL}

$$id : \square_{\{l_1, l_2\}}(\square_r \mathrm{Int} \to \mathrm{Int})$$

$$id = \left\{ \begin{array}{l} l_1 = \lambda n'.\,\mathbf{let}\,[n] = n'\,\mathbf{in}\,n \\ l_2 = \lambda n'.\,\mathbf{let}\,[n] = n'\,\mathbf{in}\,n \end{array} \right\}$$

$$l_1 \mapsto \{\texttt{ID version 1}\}$$
$$l_2 \mapsto \{\texttt{ID version 2}\}$$

GrMini (λ_{VL}–subset)

$$id : \square_r \mathrm{Int} \to \mathrm{Int}$$
$$id = \lambda n'.\,\mathbf{let}\,[n] = n'\,\mathbf{in}\,n$$
$$n : \mathrm{Int}$$
$$n = 1$$

`ID version 1`

$$id : \square_r \mathrm{Int} \to \mathrm{Int}$$
$$id = \lambda n'.\,\mathbf{let}\,[n] = n'\,\mathbf{in}\,n$$
$$n : \mathrm{Int}$$
$$n = 2$$

`ID version 2`

Step 3: Bundling Top-level Values

λ_{VL}

$main : \Box_{s}\mathrm{Int}$
$main = [id\ [n]]$

$\longleftarrow$

GrMini (λ_{VL}–subset)

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

Step 3: Bundling Top-level Values

λ_{VL}

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

GrMini (λ_{VL}–subset)

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

$$\begin{aligned} main &= \mathbf{let}\ [id'] = id\ \mathbf{in}\ \mathbf{let}\ [n'] = n\ \mathbf{in}\ [id'\ [n']] \\ &\equiv \mathbf{let}\ [id'] = \{l_1 = \cdots, l_2 = \cdots\}\ \mathbf{in} \\ &\quad\ \mathbf{let}\ [n'] = \{l_1 = \cdots, l_2 = \cdots\}\ \mathbf{in}\ [id'\ [n']] \\ &\quad\ : \Box_{\{l_1, l_2\}} \mathrm{Int} \end{aligned}$$

$$main.l_1 \to^* 1 \qquad main.l_2 \to^* 2$$

Future Work

Formalization

- To prove desirable properties for bundling:
  - Ex.) Preserve its meaning/type of the original program

- ***Challenge***: Inverse translation of bundling
  - In a independent manner of the λ_{VL} dynamic semantics

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

Similar to the macro dispatch mechanism?

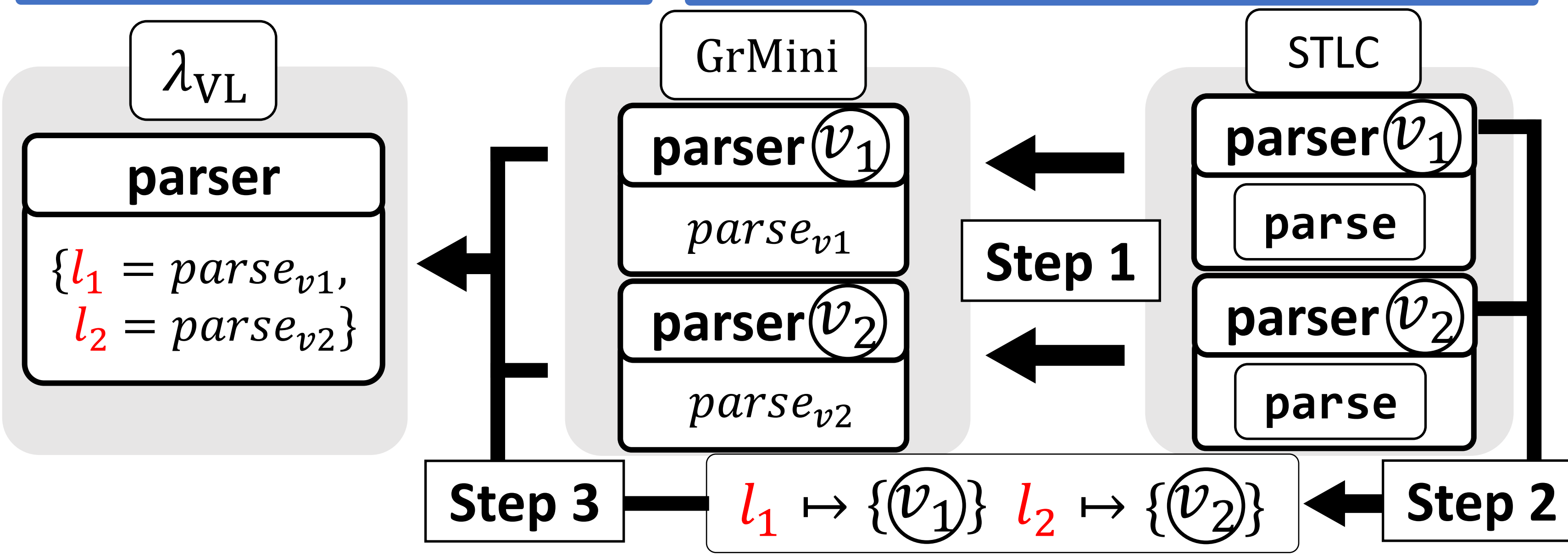
Summary

Problem

Infeasibility of λ_{VL} Programming

Proposal

Ordinary Functional Language as a Surface Language for λ_{VL}



Methods: Bundling

Compiling STLC to λ_{VL}

Future work

- ❑ Formalization