Foundations of Dependent Interoperability

Pierre-Évariste Dagand
UPMC - LIP6

Nicolas Tabareau
Inria

Éric Tanter
University of Chile

Dependent Interoperability

[Osera et al. 2012]

$\texttt{Vec}_{\mathbb{N}}\ \texttt{n} \longleftrightarrow \texttt{List}_{\mathbb{N}}$

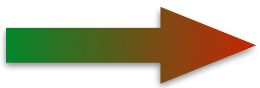
Dependent Interoperability

1. Use simply-typed library in dependently-typed context
2. Use dependently-typed library in simply-typed context
3. Dynamic verification of simply-typed components
4. Safe(r) extraction of dependently-typed programs

Examples

1. Use simply-typed library in dependently-typed context

$\texttt{rev} :$

$\texttt{List}_{\mathbb{N}} \to \texttt{List}_{\mathbb{N}}$  $\forall\, \texttt{n}.\ \texttt{Vec}_{\mathbb{N}}\ \texttt{n} \to \texttt{Vec}_{\mathbb{N}}\ \texttt{n}$

check output length
@ runtime

2. Use dependently-typed library in simply-typed context

tail:

$$\forall\ \mathtt{n}.\ \mathtt{Vec}_{\mathbb{N}}\ (\mathtt{n}+1) \to \mathtt{Vec}_{\mathbb{N}}\ \mathtt{n} \quad \Longrightarrow \quad \mathtt{List}_{\mathbb{N}} \to \mathtt{List}_{\mathbb{N}}$$

check input length @ runtime

3. Dynamic verification of simply-typed components

$$\mathtt{List}_{\mathbb{N}} \rightarrow \mathtt{List}_{\mathbb{N}}$$

$$\forall\, \mathtt{n}.\ \mathtt{Vec}_{\mathbb{N}}\ (\mathtt{n}{+}\mathtt{1}) \rightarrow \mathtt{Vec}_{\mathbb{N}}\ \mathtt{n}$$

$$\mathtt{List}_{\mathbb{N}} \rightarrow \mathtt{List}_{\mathbb{N}}$$

[Findler & Felleisen, 2002]

check “dependent contracts” @ runtime

4. Safe(r) extraction of dependently-typed programs

exec:

$\forall$ n m:$\mathbb{N}$. dinstr n m $\rightarrow$ dstack n $\rightarrow$ dstack m

type dependencies

```
int -> int -> dinstr -> dstack -> dstack
```

```
exec 0 0 (IPlus 0) [];;
Segmentation fault: 11
```

embeds necessary checks

magic inside

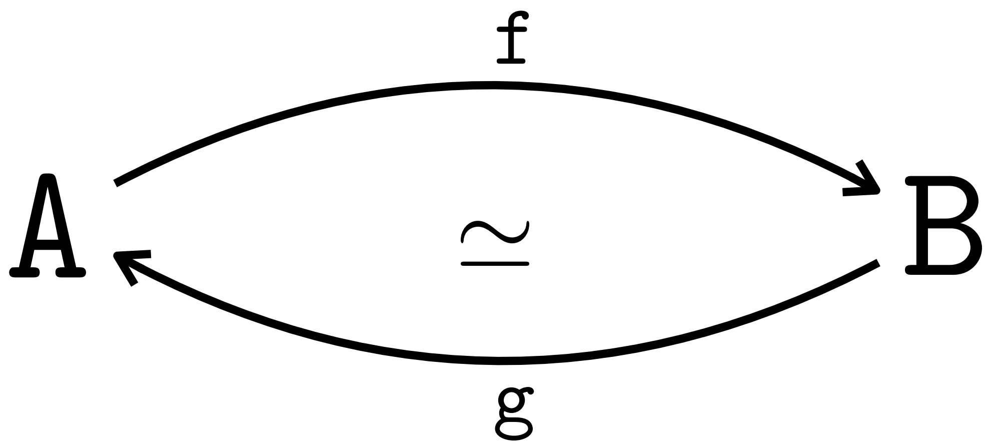
instr $\rightarrow$ List$_{\mathbb{N}}$ $\rightarrow$ List

```
instr -> int list -> int list
```

```
# exec SPlus [];;
Exception: Failure "invalid instruction"
```

Relating Types

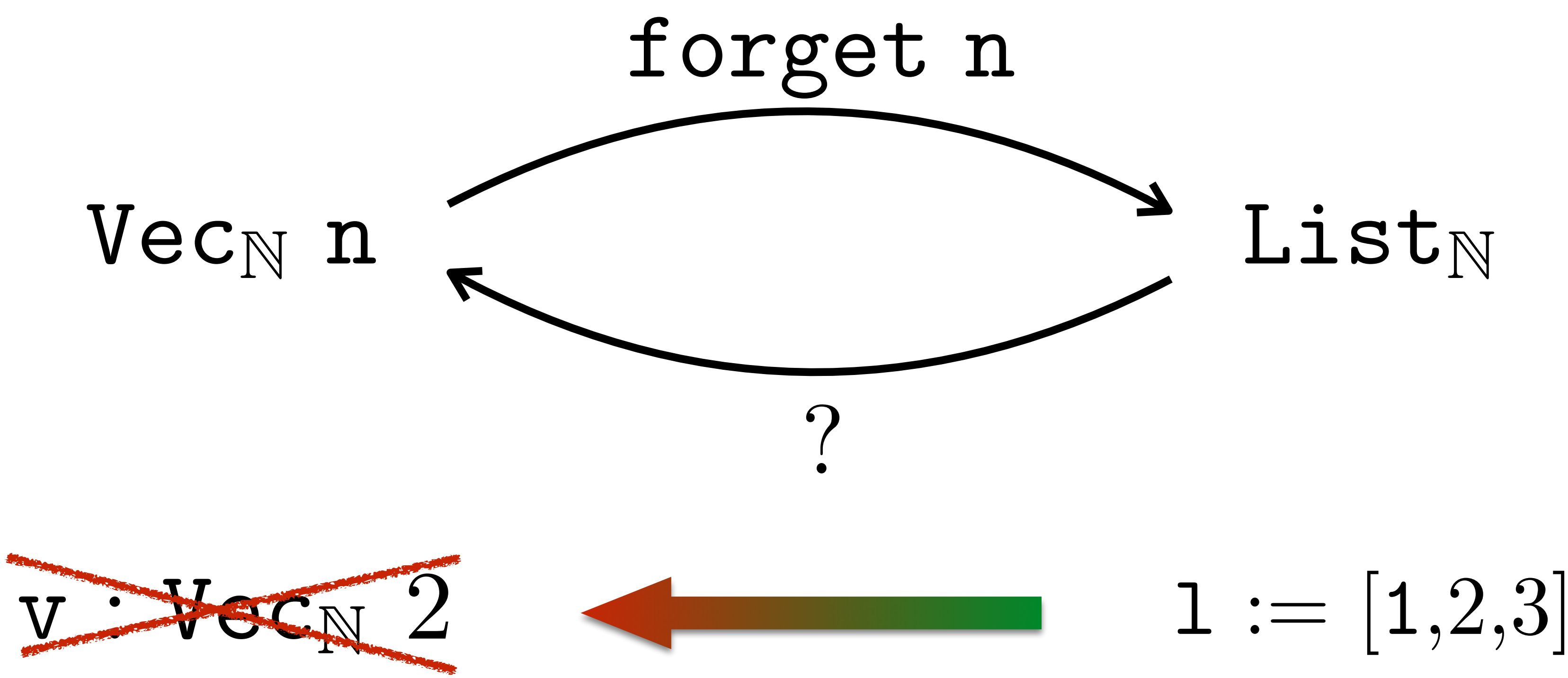
Type Equivalence



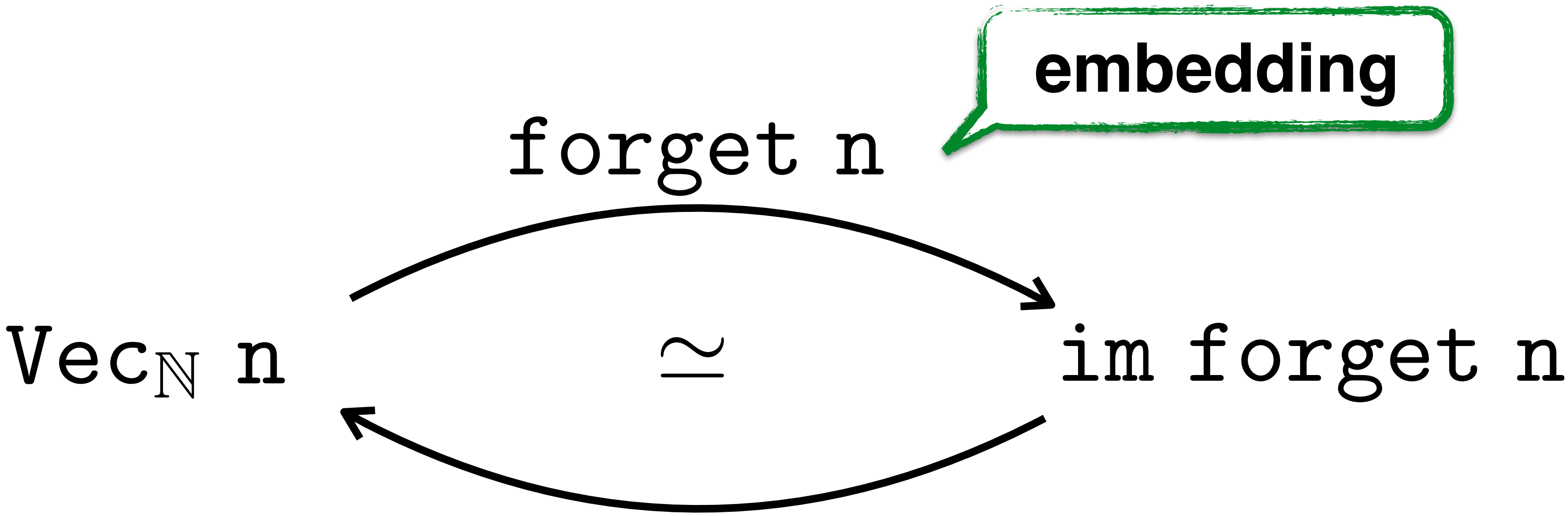

```
Class IsEquiv {A B: Type} (f: A → B) := {
  e_inv: B → A ;
  e_sect: e_inv ∘ f == id;
  e_retr: f ∘ e_inv == id;
  e_adj: ∀ x: A, e_retr (f x) = ap f
}.
```

```
Record Equiv (A B: Type) := {
   e_fun: A → B ;
   e_isequiv: IsEquiv e_fun
}.
Notation "A ≃ B" := (Equiv A B)
```

Type Equivalence ?



Type Equivalence’

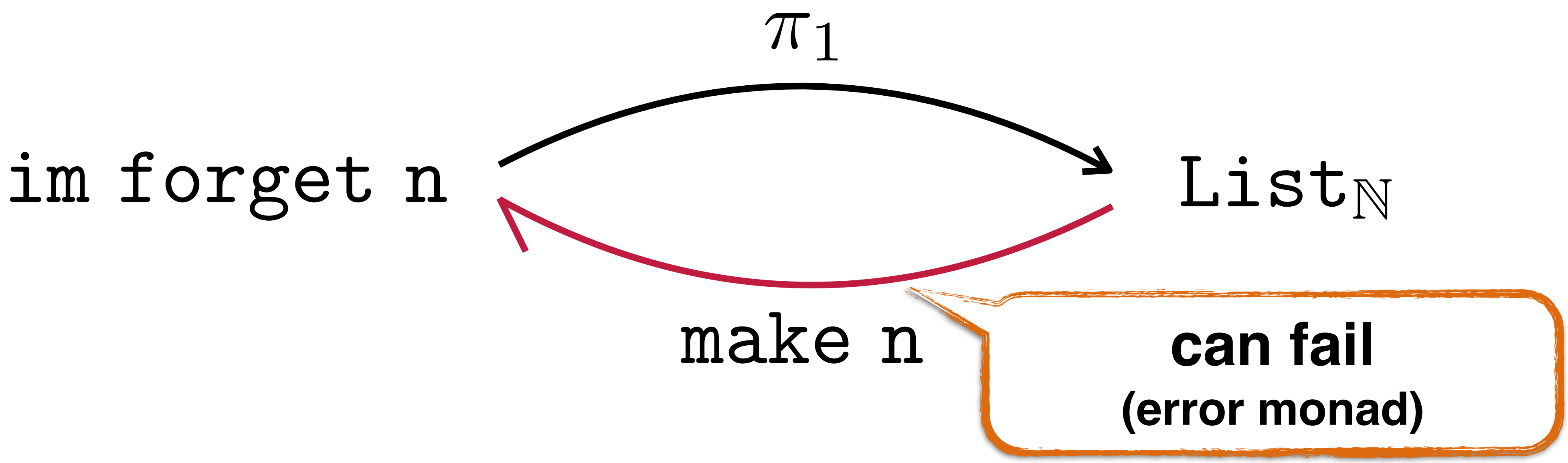


$\texttt{im forget n} \equiv$

meaning of the index

$\{\ \texttt{l: List}_{\mathbb{N}}\ \&\ \exists\ \texttt{v: Vec}_{\mathbb{N}}\ \texttt{n, forget n v} = \texttt{l}\ \}$

$\Leftrightarrow\ \texttt{length l} = \texttt{n}$



but $\forall$ l: im forget n, make n (π_1 l) = Some l

(lift π_1) $\circ_K$ (make n) $\preceq$ creturn

creturn $\preceq$ (make n) $\circ_K$ (lift π_1)

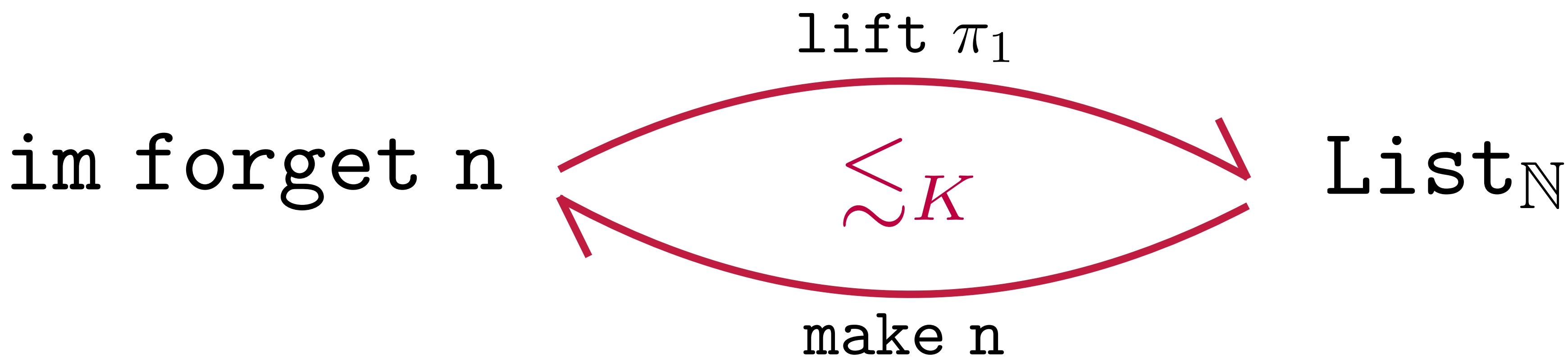
cannot fail

monadic composition

Some x $\preceq$ Some y $\iff$ x = y

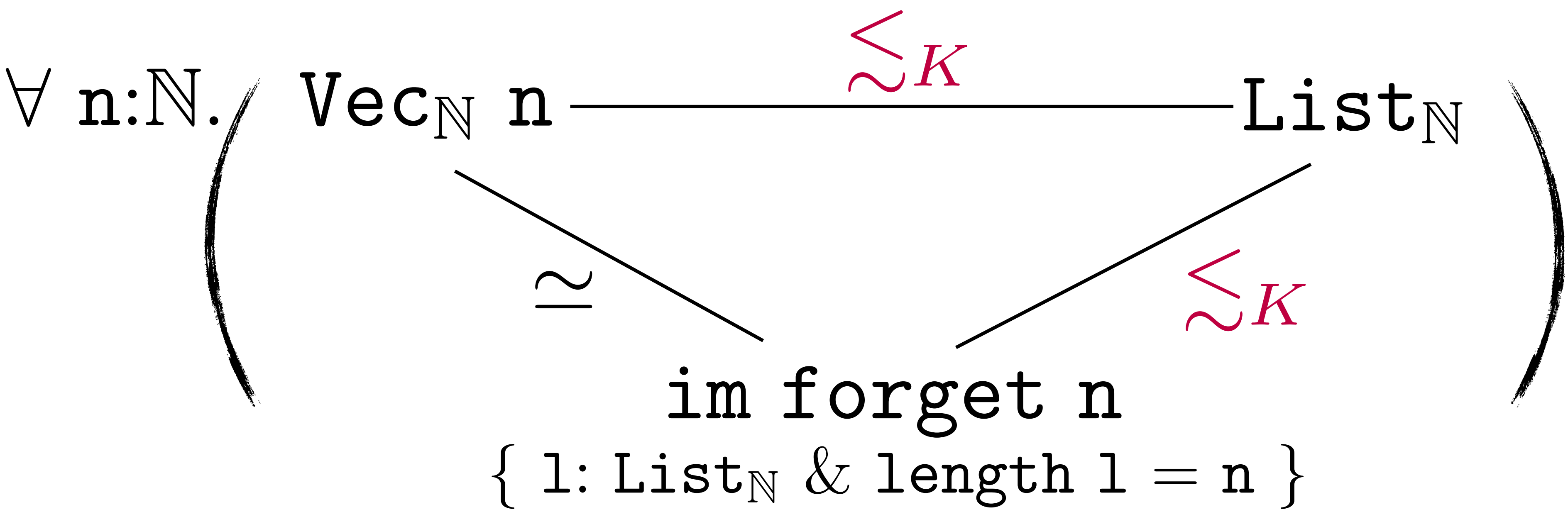
None $\preceq$ Some x

Partial Galois Connection




```
Class IsConnectionK {A B: HSet} (f: A ⇀ B) := {
  pc_inv: B ⇀ A;
  pc_sect: creturn ≼ pc_inv ∘K f;
  pc_retr: f ∘K pc_inv ≼ creturn ;
}.
```

Dependent Connection



$$\texttt{Vec}_{\mathbb{N}} \lesssim_K^{\square} \texttt{List}_{\mathbb{N}}$$

Higher-Order Connections

$$\forall\, n.\ \mathtt{Vec}_{\mathbb{N}}\ (n+1) \rightharpoonup \mathtt{Vec}_{\mathbb{N}}\ n \quad \lessapprox \quad \mathtt{List}_{\mathbb{N}} \rightharpoonup \mathtt{List}_{\mathbb{N}}$$

standard Galois connection

```
Class IsConnection {A B}
    (f: A ⟶ B) := {
  c_inv: B ⟶ A;
  c_sect: id ⪯ c_inv ∘ f ;
  c_retr: f ∘ c_inv ⪯ id;
}.
```

can fail more

Dependent Interoperability

Constructive Foundations of Dependent Interoperability

DICoq

prove

first-order connections

verified (for free)

higher-order connections

$\mathtt{Vec}_{\mathbb{N}} \lesssim^{\square}_{K} \mathtt{List}_{\mathbb{N}}$

$\forall\ \mathtt{n}.\ \mathtt{Vec}_{\mathbb{N}}\ (\mathtt{n}+1) \rightharpoonup \mathtt{Vec}_{\mathbb{N}}\ \mathtt{n}$

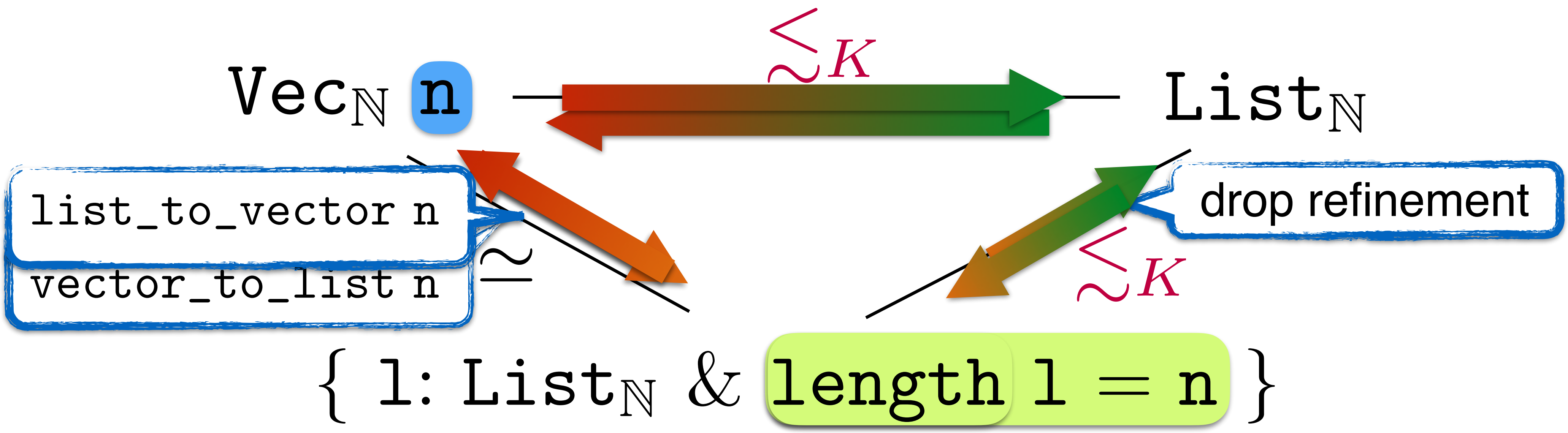
verified lifting operators

$\lesssim$

$\mathtt{List}_{\mathbb{N}} \rightharpoonup \mathtt{List}_{\mathbb{N}}$

Automatic Liftings

1. simple to dependent
2. dependent to simple



n free ⇒ compute index

must be decidable

n bound ⇒ check index predicate

```
Class Decidable (A: HProp) := dec: A + (not A).
```

Connections & Anticonnections

Galois connections (asymmetric)

```
Class IsConnectionK … := {
  …
  pc_sect: creturn ≼ pc_inv ∘K f
  …
}
```

$\lesssim_K$

- ✓ section **cannot fail**
- ✓ **unique** inverse
- ✓ subset type ≃ image restriction
- ✓ predicate is **sound & complete**

Anticonnections (symmetric)

```
Class IsAnticonnectionK … := {
  …
  apc_sect: pc_inv ∘K f ≼ creturn
  …
}
```

$\approx_K$

- ✓ section **can fail**
- ✓ **many** possible inverses
- ✓ predicate can be **incomplete**

Anticonnections

automatic liftings

```
Class Checkable (A: HProp) := {
  check: HProp;
  check_dec: Decidable check ;
  convert: check → A
}.
```

decidable and sound approximation

trade completeness for

✓ efficiency

eg. bounded check (length, membership)

✓ undecidability

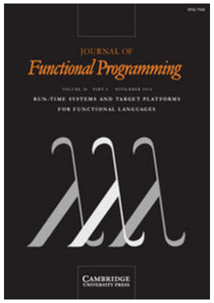
eg. potentially-infinite lists

et voilà…

Perspectives

- Compensate limitations of the target type system
  - induce verified safeguards
  - expressible as runtime checks
- Extension to effectful setting
- Type connections & framework beyond DI
- Relation to gradual typing (esp. precision)

Foundations of Dependent Interoperability



to appear at JFP
preprint & code online