A Mechanical Soundness Proof for Subtyping over Recursive Types

Timothy Jones, David Pearce

Victoria University of Wellington

tim@ecs.vuw.ac.nz

July 19, 2016

Recursive Types

```
type IntList is { int data, IntList next } | null
```

Syntax

$$\begin{aligned} \mathtt{T} ::= {} & \mathtt{Int} \\ \mid {} & \mathtt{T} \times \mathtt{T} \\ \mid {} & \mathtt{T} \vee \mathtt{T} \end{aligned}$$

Syntax

$$
\begin{aligned}
\mathtt{T} ::= {} & \mathtt{Int} \\
\mid {} & \mathtt{T} \times \mathtt{T} \\
\mid {} & \mathtt{T} \vee \mathtt{T} \\
\mid {} & \mu \mathtt{X}.\, \mathtt{T} \\
\mid {} & \mathtt{X}
\end{aligned}
$$

Non-empty list of integers: $\mu \mathtt{X}.\, \mathtt{Int} \vee \mathtt{Int} \times \mathtt{X}$

De-Bruijn Indices

$$
\begin{aligned}
\mathtt{T} ::= {} & \mathtt{Int} \\
\mid {} & \mathtt{T} \times \mathtt{T} \\
\mid {} & \mathtt{T} \vee \mathtt{T} \\
\mid {} & \mu\, \mathtt{T} \\
\mid {} & \mathbb{N}
\end{aligned}
$$

Non-empty list of integers: $\mu\, \mathtt{Int} \vee \mathtt{Int} \times \mathtt{0}$

Syntax

```
data InductiveType (n : ℕ) : Set where
```

Syntax

```
data InductiveType (n : ℕ) : Set where
  Int : InductiveType n
```

Syntax

```
data InductiveType (n : ℕ) : Set where
  Int : InductiveType n
  _×_ : (A B : InductiveType n) → InductiveType n
  _∨_ : (A B : InductiveType n) → InductiveType n
```

Syntax

```
data InductiveType (n : ℕ) : Set where
  Int : InductiveType n
  _×_ : (A B : InductiveType n) → InductiveType n
  _∨_ : (A B : InductiveType n) → InductiveType n
  μ_ : (A : InductiveType (suc n)) → InductiveType n
```

Syntax

```
data InductiveType (n : ℕ) : Set where
  Int : InductiveType n
  _×_ : (A B : InductiveType n) → InductiveType n
  _∨_ : (A B : InductiveType n) → InductiveType n
  μ_ : (A : InductiveType (suc n)) → InductiveType n
  Var : (x : Fin n) → InductiveType n
```

Syntax

```
data InductiveType (n : ℕ) : Set where
  Int : InductiveType n
  _×_ : (A B : InductiveType n) → InductiveType n
  _∨_ : (A B : InductiveType n) → InductiveType n
  μ_ : (A : InductiveType (suc n)) → InductiveType n
  Var : (x : Fin n) → InductiveType n
```

Non-empty list of integers: μ Int $\vee$ Int $\times$ Var zero

Substitution

```
_[_] : ∀ {n} → InductiveType (suc n)
               → InductiveType n
               → InductiveType n
Int         [ A ] = Int
(B × C)     [ A ] = B [ A ] × C [ A ]
(B ∨ C)     [ A ] = B [ A ] ∨ C [ A ]
(μ B)       [ A ] = μ B [ inc A ]
Ref x       [ A ] with max? x
Ref ._      [ A ] | yes max = A
Ref x       [ A ] | no ¬p   = Ref (reduce ¬p)
```

Substitution

```
_[_] : ∀ {n} → InductiveType (suc n)
               → InductiveType n
               → InductiveType n
Int         [ A ] = Int
(B × C)     [ A ] = B [ A ] × C [ A ]
(B ∨ C)     [ A ] = B [ A ] ∨ C [ A ]
(μ B)       [ A ] = μ B [ inc A ]
Ref x       [ A ] with max? x
Ref ._      [ A ] | yes max = A
Ref x       [ A ] | no ¬p   = Ref (reduce ¬p)

unfold : Type 1 → Type 0
unfold A = A [ μ A ]
```

Nonsensical Types

Equivalent unfolding: **type** X **is** X

Contractivity: **type** Ints **is** Int | Ints

Nonsensical Types

Equivalent unfolding: $\mu \mathtt{X}.\, \mathtt{X}$

Contractivity: **type** Ints **is** Int | Ints

Nonsensical Types

Equivalent unfolding: $\mu \texttt{X}.\, \texttt{X}$

Contractivity: $\mu \texttt{X}.\, \texttt{Int} \vee \texttt{X}$

Nonsensical Types

Equivalent unfolding: $\mu\texttt{X}.\,\texttt{X}$

Contractivity: $\mu\texttt{X}.\,\texttt{Int} \vee \texttt{X}$

A type `T` is *well-formed* if every occurrence of a μ-bound variable in the body is separated from its binder by at least one $\times$.

Well Formedness

```
data WF {n} (m : Fin (suc n)) : InductiveType n → Set where
```

Well Formedness

```
data WF {n} (m : Fin (suc n)) : InductiveType n → Set where
  int : WF m Int
```

Well Formedness

```
data WF {n} (m : Fin (suc n)) : InductiveType n → Set where
  int : WF m Int
  pair : ∀ {A B} → WF zero A → WF zero B → WF m (A × B)
  union : ∀ {A B} → WF m A → WF m B → WF m (A ∨ B)
```

Well Formedness

```
data WF {n} (m : Fin (suc n)) : InductiveType n → Set where
  int : WF m Int
  pair : ∀ {A B} → WF zero A → WF zero B → WF m (A × B)
  union : ∀ {A B} → WF m A → WF m B → WF m (A ∨ B)
  rec : ∀ {A} → WF (suc m) A → WF m (μ A)
```

Well Formedness

```
data WF {n} (m : Fin (suc n)) : InductiveType n → Set where
  int : WF m Int
  pair : ∀ {A B} → WF zero A → WF zero B → WF m (A × B)
  union : ∀ {A B} → WF m A → WF m B → WF m (A ∨ B)
  rec : ∀ {A} → WF (suc m) A → WF m (μ A)
  ref : ∀ {x} → m ≤ inject₁ x → WF m (Var x)
```

Corresponding Proofs

```
wf_[_] : ∀ {n m A} {B : InductiveType n} → WF (inject₁ m) A
                                         → WF m B
                                         → WF m (A [ B ])
wf int        [ p ] = int
wf pair q r   [ p ] = pair (wf q [ weaken!  p ]) (wf r [ weaken!  p ])
wf union q r  [ p ] = union (wf q [ p ]) (wf r [ p ])
wf rec q      [ p ] = rec (wf q [ wf-inc p ])
wf ref {x} q  [ p ] with max? x
wf ref        q [ p ] | yes max = p
wf ref        q [ p ] | no ¬p   = wf-reduce q ¬p
```

Corresponding Proofs

```
wf_[_] : ∀ {n m A} {B : InductiveType n} → WF (inject₁ m) A
                                         → WF m B
                                         → WF m (A [ B ])
wf int          [ p ] = int
wf pair q r     [ p ] = pair (wf q [ weaken!  p ]) (wf r [ weaken!  p ])
wf union q r    [ p ] = union (wf q [ p ]) (wf r [ p ])
wf rec q        [ p ] = rec (wf q [ wf-inc p ])
wf ref {x} q    [ p ] with max? x
wf ref          q [ p ] | yes max = p
wf ref          q [ p ] | no ¬p   = wf-reduce q ¬p

wf-unfold : ∀ {n m} {A : Type (suc n)} → WF (suc m) A
                                       → WF m (A [ μ A ])
wf-unfold p = wf weaken₁ p [ rec p ]
```

Coinductive Representation

```
data CoinductiveType : Set where
  Int : CoinductiveType
  _×_ : (A B : ∞CoinductiveType) → CoinductiveType
  _∨_ : (A B : CoinductiveType) → CoinductiveType
```

Coinductive Representation

```
data CoinductiveType : Set where
  Int : CoinductiveType
  _×_ : (A B : ∞CoinductiveType) → CoinductiveType
  _∨_ : (A B : CoinductiveType) → CoinductiveType

type : ∞CoinductiveType → CoinductiveType
```

Infinite Unfolding

Linking the two representations

$$\infty\mathsf{unfold} : \{\mathsf{A} : \mathsf{InductiveType}\ 0\} \to \mathsf{WF}\ \mathsf{zero}\ \mathsf{A} \to \mathsf{CoinductiveType}$$

Infinite Unfolding

$$\infty\text{unfold} : \{\text{A} : \text{InductiveType } 0\} \to \text{WF zero A} \to \text{CoinductiveType}$$

$$\begin{aligned}
\infty\text{unfold int} &= \text{Int} \\
\infty\text{unfold (pair p q)} &= \infty\text{unfold p} \times \infty\text{unfold q} \\
\infty\text{unfold (union p q)} &= \infty\text{unfold p} \vee \infty\text{unfold q} \\
\infty\text{unfold (rec p)} &= \infty\text{unfold (wf-unfold p)} \\
\infty\text{unfold (ref p)} &= \bot\text{-elim } (<\text{-bound p})
\end{aligned}$$

Infinite Unfolding

$$\infty\text{unfold} : \{A : \text{InductiveType } 0\} \to \text{WF zero } A \to \text{CoinductiveType}$$

$$\begin{aligned}
\infty\text{unfold int} &= \text{Int} \\
\infty\text{unfold (pair p q)} &= \infty\text{unfold p} \times \infty\text{unfold q} \\
\infty\text{unfold (union p q)} &= \infty\text{unfold p} \vee \infty\text{unfold q} \\
\infty\text{unfold (rec p)} &= \infty\text{unfold (wf-unfold p)} \\
\infty\text{unfold (ref p)} &= \bot\text{-elim } (<\text{-bound p})
\end{aligned}$$

Infinite Unfolding

$$\infty\text{unfold} : \{A : \text{InductiveType } 0\} \to \text{WF zero } A \to \text{CoinductiveType}$$

$$\text{delay-unfold} : \{A : \text{InductiveType } 0\} \to \text{WF zero } A \to \infty\text{CoinductiveType}$$

$$\begin{aligned}
\infty\text{unfold int} &= \text{Int} \\
\infty\text{unfold (pair p q)} &= \infty\text{unfold p} \times \infty\text{unfold q} \\
\infty\text{unfold (union p q)} &= \infty\text{unfold p} \vee \infty\text{unfold q} \\
\infty\text{unfold (rec p)} &= \infty\text{unfold (wf-unfold p)} \\
\infty\text{unfold (ref p)} &= \bot\text{-elim } (<\text{-bound p})
\end{aligned}$$

Infinite Unfolding

$$\infty\text{unfold} : \{A : \text{InductiveType } 0\} \to \text{WF zero } A \to \text{CoinductiveType}$$

$$\text{delay-unfold} : \{A : \text{InductiveType } 0\} \to \text{WF zero } A \to \infty\text{CoinductiveType}$$

$$\begin{aligned}
\infty\text{unfold int} &= \text{Int} \\
\infty\text{unfold (pair p q)} &= \infty\text{unfold p} \times \infty\text{unfold q} \\
\infty\text{unfold (union p q)} &= \infty\text{unfold p} \vee \infty\text{unfold q} \\
\infty\text{unfold (rec p)} &= \infty\text{unfold (wf-unfold p)} \\
\infty\text{unfold (ref p)} &= \bot\text{-elim } (<\text{-bound p})
\end{aligned}$$

$$\text{type (delay-unfold p)} = \infty\text{unfold p}$$

Infinite Unfolding

$$\infty\text{unfold} : \{A : \text{InductiveType } 0\} \to \text{WF zero } A \to \text{CoinductiveType}$$

$$\text{delay-unfold} : \{A : \text{InductiveType } 0\} \to \text{WF zero } A \to \infty\text{CoinductiveType}$$

$$\begin{aligned}
\infty\text{unfold int} &= \text{Int} \\
\infty\text{unfold (pair p q)} &= \text{delay-unfold p} \times \text{delay-unfold q} \\
\infty\text{unfold (union p q)} &= \infty\text{unfold p} \vee \infty\text{unfold q} \\
\infty\text{unfold (rec p)} &= \infty\text{unfold (wf-unfold p)} \\
\infty\text{unfold (ref p)} &= \bot\text{-elim } (<\text{-bound p})
\end{aligned}$$

$$\text{type (delay-unfold p)} = \infty\text{unfold p}$$

Infinite Unfolding

$$\infty\text{unfold} : \{A : \text{InductiveType } 0\} \to \text{WF zero } A \to \text{CoinductiveType}$$

$$\text{delay-unfold} : \{A : \text{InductiveType } 0\} \to \text{WF zero } A \to \infty\text{CoinductiveType}$$

$$\begin{aligned}
\infty\text{unfold int} &= \text{Int} \\
\infty\text{unfold (pair p q)} &= \text{delay-unfold p} \times \text{delay-unfold q} \\
\infty\text{unfold (union p q)} &= \infty\text{unfold p} \vee \infty\text{unfold q} \\
\infty\text{unfold (rec p)} &= \infty\text{unfold (wf-unfold p)} \\
\infty\text{unfold (ref p)} &= \bot\text{-elim } (<\text{-bound p})
\end{aligned}$$

$$\text{type (delay-unfold p)} = \infty\text{unfold p}$$

Substitution Delay

Substitutions must be held until they can be delayed

Substitution Delay

Substitutions must be held until they can be delayed

```
data Substs : (n : ℕ) → Fin (suc n) → Set where
  []  : ∀ {n} → Substs n zero
  _::_ : ∀ {n m} {A : InductiveType n} → WF m A
                                       → Substs n m
                                       → Substs (suc n) (suc m)
```

Infinite Unfolding

$$\infty\mathsf{unfold}' : \forall\ \{\mathsf{n}\ \mathsf{B}\} \to \mathsf{WF}\ (\mathsf{from}\mathbb{N}\ \mathsf{n})\ \mathsf{B} \to \mathsf{Substs}\ \mathsf{n}\ (\mathsf{from}\mathbb{N}\ \mathsf{n}) \to \mathsf{CoinductiveType}$$

Infinite Unfolding

$$\infty\text{unfold}' : \forall\ \{\mathsf{n}\ \mathsf{B}\} \to \mathsf{WF}\ (\text{from}\mathbb{N}\ \mathsf{n})\ \mathsf{B} \to \mathsf{Substs}\ \mathsf{n}\ (\text{from}\mathbb{N}\ \mathsf{n}) \to \mathsf{CoinductiveType}$$

$$\text{apply-substs} : \forall\ \{\mathsf{n}\}\ \{\mathsf{B} : \mathsf{InductiveType}\ \mathsf{n}\} \to \mathsf{WF}\ \mathsf{zero}\ \mathsf{B} \to \mathsf{Substs}\ \mathsf{n}\ (\text{from}\mathbb{N}\ \mathsf{n}) \to \infty\ \mathsf{CoinductiveType}$$

Infinite Unfolding

```
∞unfold′ : ∀ {n B} → WF (fromℕ n) B
                   → Substs n (fromℕ n)
                   → CoinductiveType

apply-substs : ∀ {n} {B : InductiveType n} → WF zero B
                                           → Substs n (fromℕ n)
                                           → ∞ CoinductiveType

∞unfold′ int          v = Int
∞unfold′ (pair p q)   v = apply-substs p v × apply-substs q v
∞unfold′ (union p q) v = ∞unfold′ p v ∨ ∞unfold′ q v
∞unfold′ (rec p)      v = ∞unfold′ p (rec p :: v)
∞unfold′ (ref p)      v = ⊥-elim (<-bound p)
```

Infinite Unfolding

```
∞unfold′ : ∀ {n B} → WF (fromℕ n) B
                   → Substs n (fromℕ n)
                   → CoinductiveType

apply-substs : ∀ {n} {B : InductiveType n} → WF zero B
                                           → Substs n (fromℕ n)
                                           → ∞ CoinductiveType

∞unfold′ int          v = Int
∞unfold′ (pair p q)   v = apply-substs p v × apply-substs q v
∞unfold′ (union p q) v = ∞unfold′ p v ∨ ∞unfold′ q v
∞unfold′ (rec p)      v = ∞unfold′ p (rec p :: v)
∞unfold′ (ref p)      v = ⊥-elim (<-bound p)

type (apply-substs {0} p []) = ∞unfold′ p []
apply-substs p (q :: v)      = apply-substs (wf p [ weaken! q ]) v
```

Infinite Unfolding

```
∞unfold : {A : InductiveType 0} → WF zero A
                                → CoinductiveType
∞unfold p = ∞unfold′ p []
```

Subtyping

```
data _≤_ : CoinductiveType → CoinductiveType → Set where
```

Subtyping

```
data _≤_ : CoinductiveType → CoinductiveType → Set where
  int : Int ≤ Int
```

Subtyping

```
data _≤_ : CoinductiveType → CoinductiveType → Set where
  int : Int ≤ Int
  left : ∀ {A B C} → A ≤ B → A ≤ B ∨ C
  right : ∀ {A B C} → A ≤ C → A ≤ B ∨ C
```

Subtyping

```
data _≤_ : CoinductiveType → CoinductiveType → Set where
  int : Int ≤ Int
  left : ∀ {A B C} → A ≤ B → A ≤ B ∨ C
  right : ∀ {A B C} → A ≤ C → A ≤ B ∨ C
  union : ∀ {A B C} → A ≤ C → B ≤ C → A ∨ B ≤ C
```

Subtyping

```
data _≤_ : CoinductiveType → CoinductiveType → Set where
  int : Int ≤ Int
  left : ∀ {A B C} → A ≤ B → A ≤ B ∨ C
  right : ∀ {A B C} → A ≤ C → A ≤ B ∨ C
  union : ∀ {A B C} → A ≤ C → B ≤ C → A ∨ B ≤ C
  pair : ∀ {A B C D} → A ∞≤ C → B ∞≤ D → A × B ≤ C × D
```

Subtyping

```
data _≤_ : CoinductiveType → CoinductiveType → Set where
  int : Int ≤ Int
  left : ∀ {A B C} → A ≤ B → A ≤ B ∨ C
  right : ∀ {A B C} → A ≤ C → A ≤ B ∨ C
  union : ∀ {A B C} → A ≤ C → B ≤ C → A ∨ B ≤ C
  pair : ∀ {A B C D} → A ∞≤ C → B ∞≤ D → A × B ≤ C × D


_∞≤_ : ∞CoinductiveType → ∞CoinductiveType → Set
```

Subtyping

```
data _≤_ : CoinductiveType → CoinductiveType → Set where
  int : Int ≤ Int
  left : ∀ {A B C} → A ≤ B → A ≤ B ∨ C
  right : ∀ {A B C} → A ≤ C → A ≤ B ∨ C
  union : ∀ {A B C} → A ≤ C → B ≤ C → A ∨ B ≤ C
  pair : ∀ {A B C D} → A ∞≤ C → B ∞≤ D → A × B ≤ C × D

_∞≤_ : ∞CoinductiveType → ∞CoinductiveType → Set

sub : ∀ {A B} → A ∞≤ B → type A ≤ type B
```

Connecting Back

```
_<:_ : ∀ {A B} → WF zero A → WF zero B → Set
_<:_ p q = ∞unfold p ≤ ∞unfold q
```

Reflexivity

```
reflexive : Reflexive _≤_
reflexive {Int}     = int
reflexive {A × B} = pair ∞reflexive ∞reflexive
reflexive {A ∨ B} = union (left reflexive) (right reflexive)

∞reflexive : Reflexive _∞≤_
sub ∞reflexive = reflexive
```

Transitivity

```
transitive : Transitive _≤_
transitive p           int          = p
transitive p           (left q)     = left (transitive p q)
transitive p           (right q)    = right (transitive p q)
transitive (left p)    (union q r)  = transitive p q
transitive (right p)   (union q r)  = transitive p r
transitive (pair p q)  (pair r s)   = pair (∞transitive p r)
                                           (∞transitive q s)
transitive (union p q) r            = union (transitive p r)
                                            (transitive q r)

∞transitive : Transitive _∞≤_
sub (∞transitive p q) = transitive (sub p) (sub q)
```

Semantics

Standard to prove correspondence with semantic subtyping

How do we give meaning to our types in Agda?

Values

```
data Value : Set where
  int : ℤ → Value
  _,_ : Value → Value → Value
```

Values

```
data Value : Set where
  int : ℤ → Value
  _,_ : Value → Value → Value

⟦_⟧ : CoinductiveType → Set
```

Values

```
data Value : Set where
  int : ℤ → Value
  _,_ : Value → Value → Value

⟦_⟧ : CoinductiveType → Set

embed : ∀ {A} → ⟦ A ⟧ → Value
```

Meanings

$$
\begin{aligned}
&[\![\_]\!] : \mathsf{CoinductiveType} \to \mathsf{Set} \\
&[\![\ \mathsf{Int}\ ]\!] &&= \mathbb{Z} \\
&[\![\ \mathsf{A} \times \mathsf{B}\ ]\!] &&= [\![\ \mathsf{type\ A}\ ]\!] \times [\![\ \mathsf{type\ B}\ ]\!] \\
&[\![\ \mathsf{A} \vee \mathsf{B}\ ]\!] &&= [\![\ \mathsf{A}\ ]\!] \uplus [\![\ \mathsf{B}\ ]\!]
\end{aligned}
$$

Meanings

$$
\begin{aligned}
&[\![\_]\!] : \mathsf{CoinductiveType} \to \mathsf{Set} \\
&[\![\ \mathsf{Int}\ ]\!] &&= \mathbb{Z} \\
&[\![\ \mathsf{A} \times \mathsf{B}\ ]\!] &&= [\![\ \mathsf{type\ A}\ ]\!] \times [\![\ \mathsf{type\ B}\ ]\!] \\
&[\![\ \mathsf{A} \vee \mathsf{B}\ ]\!] &&= [\![\ \mathsf{A}\ ]\!] \uplus [\![\ \mathsf{B}\ ]\!]
\end{aligned}
$$

Meanings

```
⟦_⟧ : CoinductiveType → Set
⟦ Int ⟧       = ℤ
⟦ A × B ⟧     = ⟦ type A ⟧ × ⟦ type B ⟧
⟦ A ∨ B ⟧     = ⟦ A ⟧ ⊎ ⟦ B ⟧
```

```
data _×_ (A B : CoinductiveType) : Set where
  _,_ : ⟦ A ⟧ → ⟦ B ⟧ → A × B
```

Meanings

$$
\begin{aligned}
&[\![\_]\!] : \mathsf{CoinductiveType} \to \mathsf{Set} \\
&[\![\ \mathsf{Int}\ ]\!] \quad\ = \mathbb{Z} \\
&[\![\ \mathsf{A} \times \mathsf{B}\ ]\!] = \mathsf{type\ A} \times \mathsf{type\ B} \\
&[\![\ \mathsf{A} \vee \mathsf{B}\ ]\!] = [\![\ \mathsf{A}\ ]\!] \uplus [\![\ \mathsf{B}\ ]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{data}\ \_\times\_\ (\mathsf{A\ B} : \mathsf{CoinductiveType}) : \mathsf{Set}\ \mathsf{where} \\
&\quad \_,\_ : [\![\ \mathsf{A}\ ]\!] \to [\![\ \mathsf{B}\ ]\!] \to \mathsf{A} \times \mathsf{B}
\end{aligned}
$$

Embedding

$$
\begin{aligned}
&\mathsf{embed} : \forall\ \{\mathsf{A}\} \to [\![\ \mathsf{A}\ ]\!] \to \mathsf{Value} \\
&\mathsf{embed}\ \{\mathsf{Int}\}\quad\ \mathsf{x}\quad\ = \mathsf{int}\ \mathsf{x} \\
&\mathsf{embed}\ \{\mathsf{A} \times \mathsf{B}\}\ (\mathsf{x}\ ,\ \mathsf{y}) = \mathsf{embed}\ \mathsf{x}\ ,\ \mathsf{embed}\ \mathsf{y} \\
&\mathsf{embed}\ \{\mathsf{A} \vee \mathsf{B}\}\ (\mathsf{inj}_1\ \mathsf{x}) = \mathsf{embed}\ \mathsf{x} \\
&\mathsf{embed}\ \{\mathsf{A} \vee \mathsf{B}\}\ (\mathsf{inj}_2\ \mathsf{y}) = \mathsf{embed}\ \mathsf{y}
\end{aligned}
$$

Semantic Subtyping

$$\llbracket \_ \rrbracket \subseteq \llbracket \_ \rrbracket : \mathsf{CoinductiveType} \to \mathsf{CoinductiveType} \to \mathsf{Set}$$
$$\llbracket\ \mathsf{A}\ \rrbracket \subseteq \llbracket\ \mathsf{B}\ \rrbracket = (\mathsf{x} : \llbracket\ \mathsf{A}\ \rrbracket) \to \Sigma[\ \mathsf{y} \in \llbracket\ \mathsf{B}\ \rrbracket\ ]\ \mathsf{embed}\ \mathsf{x} \equiv \mathsf{embed}\ \mathsf{y}$$

Soundness

```
sound : ∀ {A B} → A ≤ B → ⟦ A ⟧⊆⟦ B ⟧
sound int          x       = x , refl
sound (left p)     x       with sound p x
sound (left p)     x       | y , q = inj₁ y , q
sound (right p)    x       with sound p x
sound (right p)    x       | y , q = inj₂ y , q
sound (pair p q)   (w , x) with sound (sub p) w | sound (sub q) x
sound (pair p q)   (w , x) | y , r | z , s = (y , z) , cong₂ _,_ r s
sound (union p q) (inj₁ x) = sound p x
sound (union p q) (inj₂ y) = sound q y
```