Acyclic attribute evaluation in a dependently typed setting

Denis Firsov and Tarmo Uustalu

Institute of Cybernetics at TUT

October 23, 2015

Example: Semantics of binary numbers

$$
\begin{aligned}
\mathtt{N} &\rightarrow \mathtt{L} \\
\mathtt{L}_1 &\rightarrow \mathtt{L}_2\mathtt{B}
\end{aligned}
$$

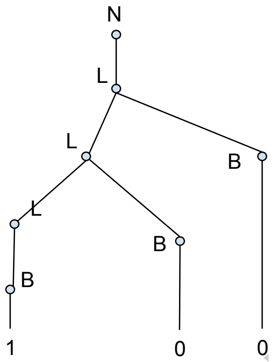
$$
\begin{aligned}
\mathtt{L} &\rightarrow \mathtt{B} \\
\mathtt{B} &\rightarrow \mathtt{0} \\
\mathtt{B} &\rightarrow \mathtt{1}
\end{aligned}
$$

Example: Semantics of binary numbers

$N \to L$	`v(N)  = v(L), s(L) = s(N), s(N) = 0`
$L_1 \to L_2B$	$v(L_1) = v(L_2) + v(B)$,
	$s(L_2) = s(L_1) + 1$, $s(B) = s(L_1)$
$L \to B$	`v(L)  = v(B), s(B) = s(L)`
$B \to 0$	`v(B)  = 0`
$B \to 1$	`v(B)  = 2^s(L)`

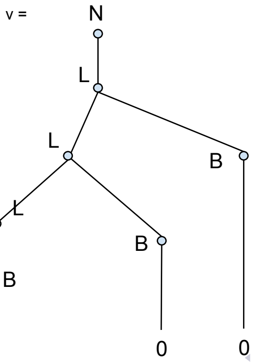
Example: Semantics of binary numbers

$N \rightarrow L$	`v(N)  = v(L), s(L) = s(N), s(N) = 0`
$L_1 \rightarrow L_2B$	$\texttt{v(L}_1\texttt{) = v(L}_2\texttt{) + v(B),}$
	$\texttt{s(L}_2\texttt{) = s(L}_1\texttt{) + 1, s(B) = s(L}_1\texttt{)}$
$L \rightarrow B$	`v(L)  = v(B), s(B) = s(L)`
$B \rightarrow 0$	`v(B)  = 0`
$B \rightarrow 1$	`v(B)  = 2^s(L)`



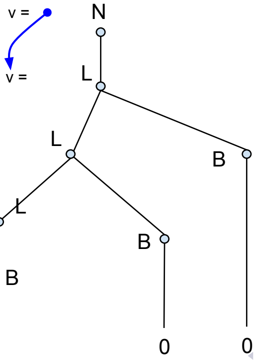
Example: Semantics of binary numbers

$N \rightarrow L$	$v(N) = v(L),\ s(L) = s(N),\ s(N) = 0$
$L_1 \rightarrow L_2B$	$v(L_1) = v(L_2) + v(B),$
	$s(L_2) = s(L_1) + 1,\ s(B) = s(L_1)$
$L \rightarrow B$	$v(L) = v(B),\ s(B) = s(L)$
$B \rightarrow 0$	$v(B) = 0$
$B \rightarrow 1$	$v(B) = 2\hat{\ }s(L)$



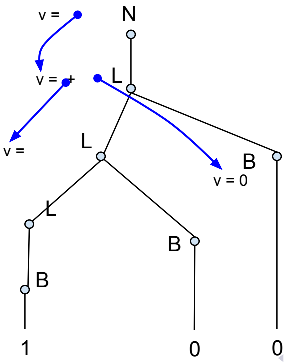
Example: Semantics of binary numbers

$N \rightarrow L$	`v(N) = v(L), s(L) = s(N), s(N) = 0`
$L_1 \rightarrow L_2B$	`v(L1) = v(L2) + v(B),`
	`s(L2) = s(L1) + 1, s(B) = s(L1)`
$L \rightarrow B$	`v(L) = v(B), s(B) = s(L)`
$B \rightarrow 0$	`v(B) = 0`
$B \rightarrow 1$	`v(B) = 2^s(L)`



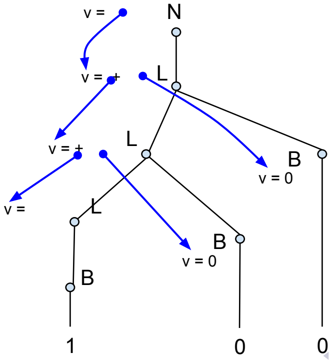
Example: Semantics of binary numbers

$N \to L$	v(N) = v(L), s(L) = s(N), s(N) = 0
$L_1 \to L_2B$	v(L_1) = v(L_2) + v(B),
	s(L_2) = s(L_1) + 1, s(B) = s(L_1)
$L \to B$	v(L) = v(B), s(B) = s(L)
$B \to 0$	v(B) = 0
$B \to 1$	v(B) = 2^s(L)



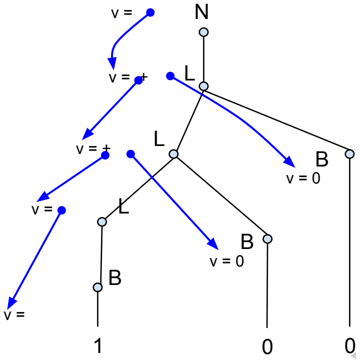
Example: Semantics of binary numbers

$N \rightarrow L$	v(N) = v(L), s(L) = s(N), s(N) = 0
$L_1 \rightarrow L_2B$	$v(L_1) = v(L_2) + v(B)$,
	$s(L_2) = s(L_1) + 1$, $s(B) = s(L_1)$
$L \rightarrow B$	v(L) = v(B), s(B) = s(L)
$B \rightarrow 0$	v(B) = 0
$B \rightarrow 1$	v(B) = 2^s(L)



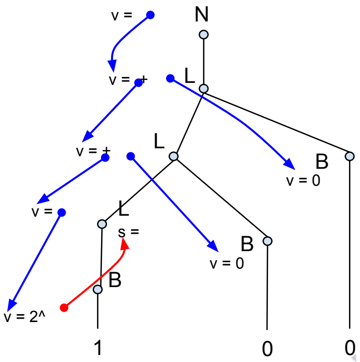
Example: Semantics of binary numbers

$N \to L$	`v(N) = v(L), s(L) = s(N), s(N) = 0`
$L_1 \to L_2 B$	`v(L1) = v(L2) + v(B),`
	`s(L2) = s(L1) + 1, s(B) = s(L1)`
$L \to B$	`v(L) = v(B), s(B) = s(L)`
$B \to 0$	`v(B) = 0`
$B \to 1$	`v(B) = 2^s(L)`



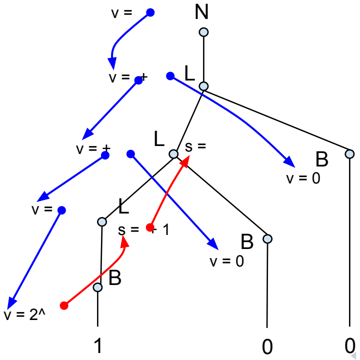
Example: Semantics of binary numbers

$N \rightarrow L$	v(N) = v(L), s(L) = s(N), s(N) = 0
$L_1 \rightarrow L_2 B$	$v(L_1) = v(L_2) + v(B)$,
	$s(L_2) = s(L_1) + 1$, $s(B) = s(L_1)$
$L \rightarrow B$	v(L) = v(B), s(B) = s(L)
$B \rightarrow 0$	v(B) = 0
$B \rightarrow 1$	v(B) = 2^s(L)



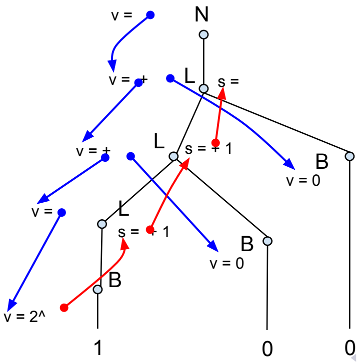
Example: Semantics of binary numbers

$N \rightarrow L$	$v(N) = v(L)$, $s(L) = s(N)$, $s(N) = 0$
$L_1 \rightarrow L_2 B$	$v(L_1) = v(L_2) + v(B)$,
	$s(L_2) = s(L_1) + 1$, $s(B) = s(L_1)$
$L \rightarrow B$	$v(L) = v(B)$, $s(B) = s(L)$
$B \rightarrow 0$	$v(B) = 0$
$B \rightarrow 1$	$v(B) = 2\hat{\ }s(L)$



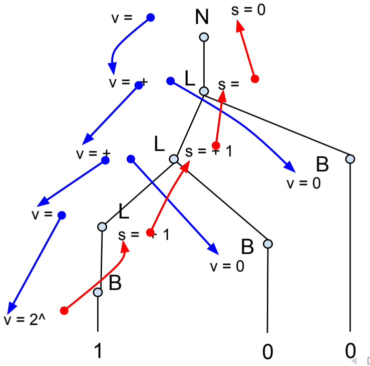
Example: Semantics of binary numbers

```
N  → L       v(N)  = v(L), s(L) = s(N), s(N) = 0
L1 → L2B     v(L1) = v(L2) + v(B),
             s(L2) = s(L1) + 1, s(B) = s(L1)
L  → B       v(L)  = v(B), s(B) = s(L)
B  → 0       v(B)  = 0
B  → 1       v(B)  = 2^s(L)
```



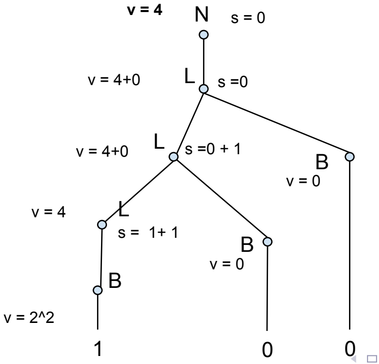
Example: Semantics of binary numbers

$N \rightarrow L$	`v(N) = v(L), s(L) = s(N), s(N) = 0`
$L_1 \rightarrow L_2 B$	`v(L1) = v(L2) + v(B),`
	`s(L2) = s(L1) + 1, s(B) = s(L1)`
$L \rightarrow B$	`v(L) = v(B), s(B) = s(L)`
$B \rightarrow 0$	`v(B) = 0`
$B \rightarrow 1$	`v(B) = 2^s(L)`



Example: Semantics of binary numbers

$N \rightarrow L$	`v(N)  = v(L), s(L) = s(N), s(N) = 0`
$L_1 \rightarrow L_2B$	$v(L_1) = v(L_2) + v(B)$,
	$s(L_2) = s(L_1) + 1$, $s(B) = s(L_1)$
$L \rightarrow B$	`v(L)  = v(B), s(B) = s(L)`
$B \rightarrow 0$	`v(B)  = 0`
$B \rightarrow 1$	`v(B)  = 2^s(L)`



Basic definitions: Parse trees

```
Attr : Set
```

Basic definitions: Parse trees

```
Attr : Set

N : Set
```

Basic definitions: Parse trees

```
Attr : Set

N : Set

Rule : N → List N → Set
```

Basic definitions: Parse trees

```
Attr : Set

N : Set

Rule : N → List N → Set

mutual
  Forest = List (∃(X : N), Tree X)

  data Tree : N → Set where
    node : ∀ {X} → (f : Forest)
         → Rule X (map proj₁ f) → Tree X
```

Basic definitions: Positions in trees

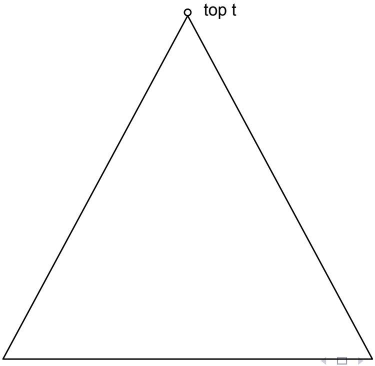
```
data PosTree  : ∀ {X : N} → Tree X → Set where
  top : ∀ {X} → (global : Tree X) → PosTree global
```

Basic definitions: Positions in trees

```
data PosTree  : ∀ {X : ℕ} → Tree X → Set where
  top : ∀ {X} → (global : Tree X) → PosTree global
  ins : ∀ {X Y}{t₁ : Tree X}{t₂ : Tree Y}
                 → PosTree t₁ → {SubTree t₂ t₁} → PosTree t₂
```

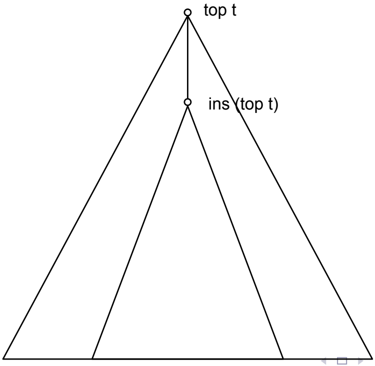
Basic definitions: Positions in trees

```
data PosTree  : ∀ {X : N} → Tree X → Set where
  top : ∀ {X} → (global : Tree X) → PosTree global
  ins : ∀ {X Y}{t₁ : Tree X}{t₂ : Tree Y}
                → PosTree t₁ → {SubTree t₂ t₁} → PosTree t₂
```



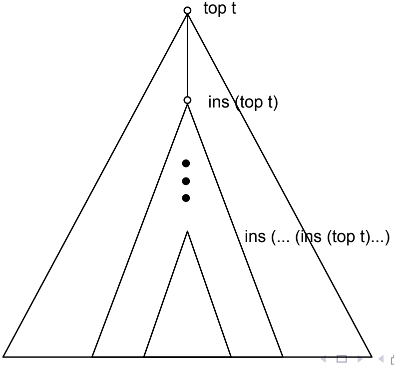
Basic definitions: Positions in trees

```
data PosTree  : ∀ {X : N} → Tree X → Set where
  top : ∀ {X} → (global : Tree X) → PosTree global
  ins : ∀ {X Y}{t₁ : Tree X}{t₂ : Tree Y}
                 → PosTree t₁ → {SubTree t₂ t₁} → PosTree t₂
```



Basic definitions: Positions in trees

```
data PosTree  : ∀ {X : N} → Tree X → Set where
  top : ∀ {X} → (global : Tree X) → PosTree global
  ins : ∀ {X Y}{t₁ : Tree X}{t₂ : Tree Y}
                → PosTree t₁ → {SubTree t₂ t₁} → PosTree t₂
```



Basic definitions: Step relation

```
type Dir = ⇑ | ⇓
```

Basic definitions: Step relation

```
type Dir = ⇑ | ⇓
```

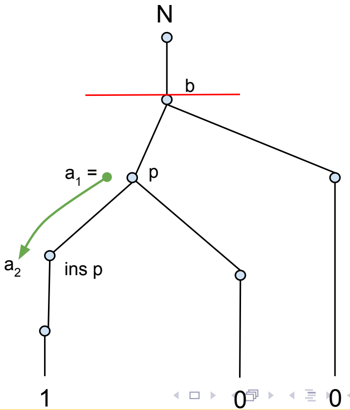
```
data _|[_]_[_] {Z : N}{t : Tree Z}(b : PosTree t) : ∀ {X Y t₁ t₂} →
         PosTree t₁ × Attr → Dir →  PosTree t₂ × Attr → Set where
```

Basic definitions: Step relation

```
type Dir = ⇑ | ⇓
```

```
data _|[_]_[_] {Z : N}{t : Tree Z}(b : PosTree t) : ∀ {X Y t₁ t₂} →
        PosTree t₁ × Attr → Dir →  PosTree t₂ × Attr → Set where
```

```
    b|[ p , a₁ ] ⇓ [ ins p , a₂ ]
```



Basic definitions: Path

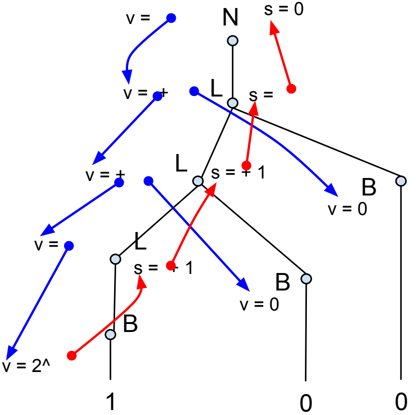
```
data _|[_]~>^{+/*}[_] {Z : N}{t : Tree Z}(b : PosTree t) : ∀{X Y t₁ t₂}
     →  PosTree t₁ × Attr → PosTree  t₂ × Attr → Set
```

$$\mathtt{data}\ \_|[\_]\overset{+/*}{\rightsquigarrow}[\_]\ \{\mathtt{Z} : \mathtt{N}\}\{\mathtt{t} : \mathtt{Tree}\ \mathtt{Z}\}(\mathtt{b} : \mathtt{PosTree}\ \mathtt{t}) : \forall\{\mathtt{X}\ \mathtt{Y}\ \mathtt{t}_1\ \mathtt{t}_2\} \to \mathtt{PosTree}\ \mathtt{t}_1 \times \mathtt{Attr} \to \mathtt{PosTree}\ \mathtt{t}_2 \times \mathtt{Attr} \to \mathtt{Set}$$

Basic definitions: Path

$$\texttt{data \_|[\_]}\overset{+/*}{\rightsquigarrow}\texttt{[\_] \{Z : N\}\{t : Tree Z\}(b : PosTree t) :} \forall\texttt{\{X Y } t_1\ t_2\texttt{\}}$$
$$\rightarrow \texttt{PosTree } t_1 \times \texttt{Attr} \rightarrow \texttt{PosTree } t_2 \times \texttt{Attr} \rightarrow \texttt{Set}$$

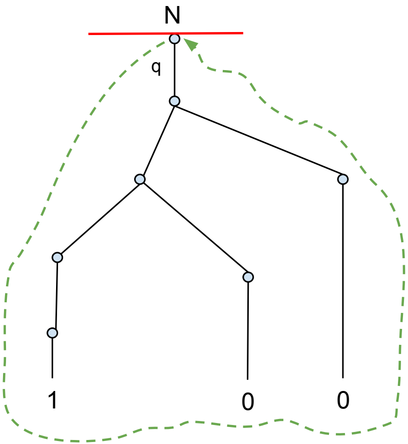
$$\texttt{q |[q , v]} \overset{+}{\rightsquigarrow} \texttt{[q , s]}$$



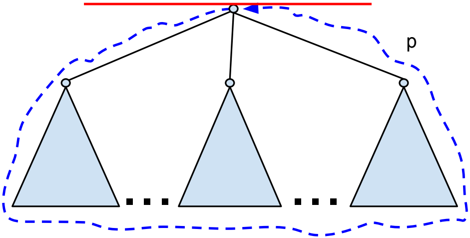
Basic definitions: Path

$$\texttt{data \_|[\_]} \overset{+/*}{\leadsto} \texttt{[\_] \{Z : N\}\{t : Tree Z\}(b : PosTree t) :} \forall \texttt{\{X Y } t_1 \ t_2\texttt{\}}$$
$$\rightarrow \texttt{PosTree } t_1 \times \texttt{Attr} \rightarrow \texttt{PosTree } t_2 \times \texttt{Attr} \rightarrow \texttt{Set}$$

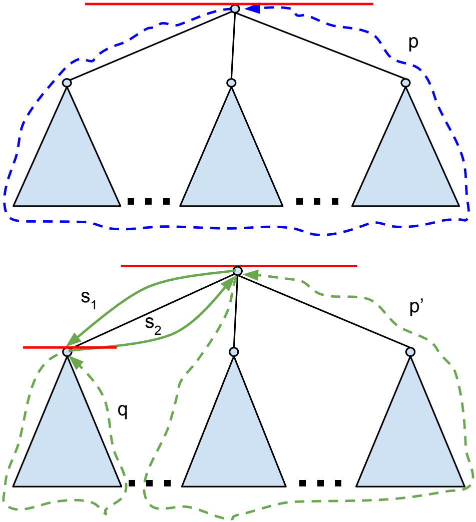
$$\texttt{q |[q , v]} \overset{+}{\leadsto} \texttt{[q , s]}$$



Cycle decomposition



Cycle decomposition



p ≡ s_1 ++ q ++ s_2 ++ p'

Induction principle

Theorem (Induction principle)

Let `P` *be a property of paths of type* `p |[ p ] f [ p ]`, *where* `p` *is any position in any tree. Then to conclude that* `P` *holds for all cycles on all trees, we need to establish the following:*

- `P` *holds for empty cycles on all trees.*
- *Any proofs that* `P` *holds for a cycle* `q` *of type* `(ins p c)|[ ins p c ] f [ ins p c ]` *and* `P` *holds for a cycle* `p'` *of type* `p |[ p ] f [ p ]` *can be converted into a proof that* `P` *holds for the cycle* $\mathtt{s}_1$ `++ q ++` $\mathtt{s}_2$ `++ p'` *where* $\mathtt{s}_1$ *and* $\mathtt{s}_2$ *are steps down from and up to* `p`.

Basic definitions: Tree root dependencies

```
dependencies : {X : N} → Tree X → List (Attr × Attr)
dependencies {X} t = ...  -- reachability on graph
```

Basic definitions: Tree root dependencies

```
dependencies : {X : N} → Tree X → List (Attr × Attr)
dependencies {X} t = ...  -- reachability on graph

dep-sound : {X : N}(t : Tree X)(a b : Attr)
 → (a, b) ∈ dependencies t → top t | [ top t, a ] ⇝⁺ [ top t, b ]
```

Basic definitions: Tree root dependencies

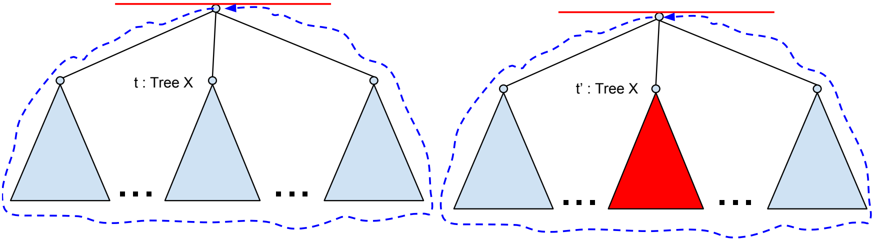
```
dependencies : {X : N} → Tree X → List (Attr × Attr)
dependencies {X} t = ...  -- reachability on graph

dep-sound : {X : N}(t : Tree X)(a b : Attr)
 → (a, b) ∈ dependencies t → top t | [ top t, a ] ⇝⁺ [ top t, b ]

dep-complete : {X : N}(t : Tree X)(a b : Attr)
  → top t | [ top t, a ] ⇝⁺ [ top t, b ] → (a, b) ∈ dependencies t
```

Basic definitions: Tree root dependencies

```
dependencies : {X : N} → Tree X → List (Attr × Attr)
dependencies {X} t = ...  -- reachability on graph

dep-sound : {X : N}(t : Tree X)(a b : Attr)
 → (a, b) ∈ dependencies t → top t | [ top t, a ] ⇝⁺ [ top t, b ]

dep-complete : {X : N}(t : Tree X)(a b : Attr)
  → top t | [ top t, a ] ⇝⁺ [ top t, b ] → (a, b) ∈ dependencies t


_≈_ : {X : N} → Tree X → Tree X → Set
t ≈ t' = dependencies t ≡ dependencies t'
```

Substitution lemma

If t $\approx$ t' then `node (f₁ ++ [ t ] ++ f₂)` $\approx$ `node (f₁ ++ [ t' ] ++ f₂))`

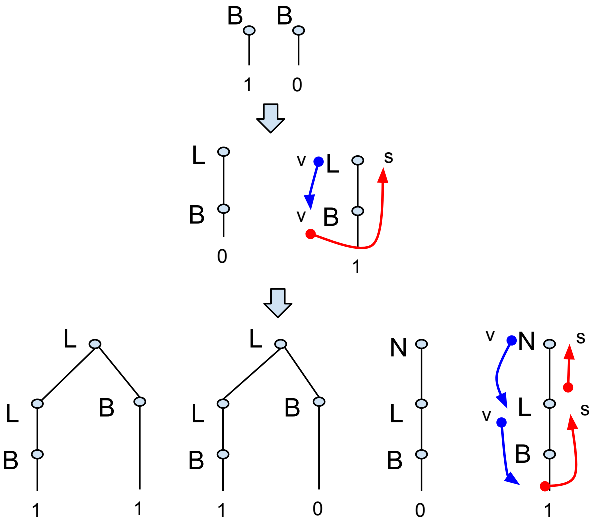


Checking for circularity: Algorithm

`treesToCheck : Rules → Forest`

1. Initialize accumulator with terminal rules
2. Build new trees from existing ones in accumulator
3. For each new tree `t`:
    1. Compute the `dependencies t`
    2. If there is no tree `t'` in accumulator such that `t` $\approx$ `t'` then add `t` to accumulator
4. If at least one tree was added then go to step 2, otherwise return accumulator.

Checking for circularity: Example

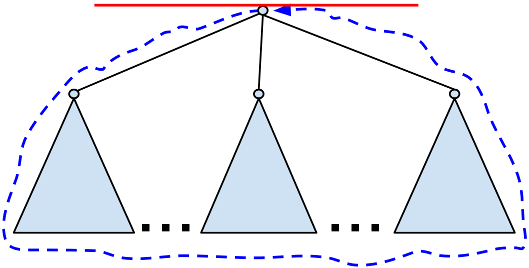


Completeness

```
completeness: (t : Tree X)
     → ∃ (t' : Tree X) , t ≈ t' × (X, t') ∈ treesToCheck Rs
```

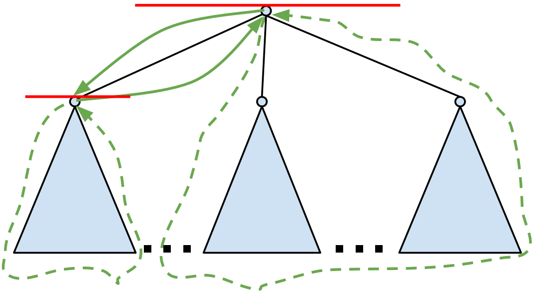
Completeness

```
completeness: (t : Tree X)
     → ∃ (t' : Tree X) , t ≈ t' × (X, t') ∈ treesToCheck Rs
```



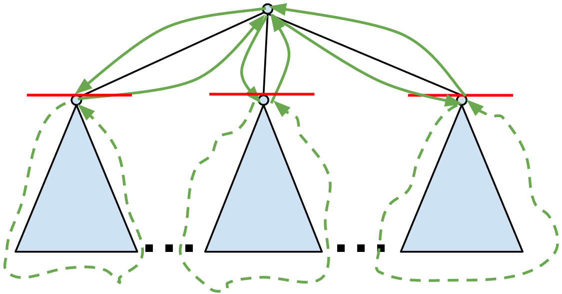
Completeness

```
completeness: (t : Tree X)
     → ∃ (t' : Tree X) , t ≈ t' × (X, t') ∈ treesToCheck Rs
```



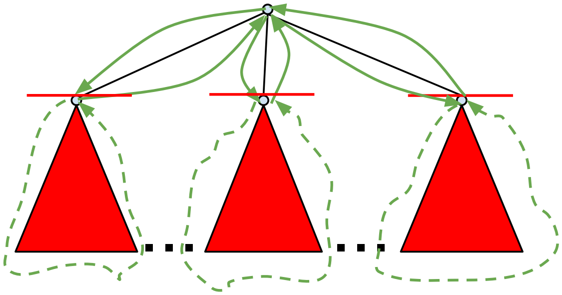
Completeness

```
completeness: (t : Tree X)
     → ∃ (t' : Tree X) , t ≈ t' × (X, t') ∈ treesToCheck Rs
```



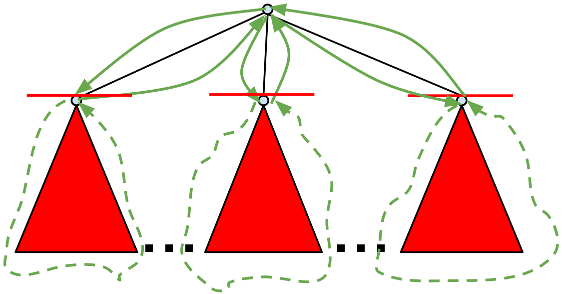
Completeness

```
completeness: (t : Tree X)
     → ∃ (t' : Tree X) , t ≈ t' × (X, t') ∈ treesToCheck Rs
```



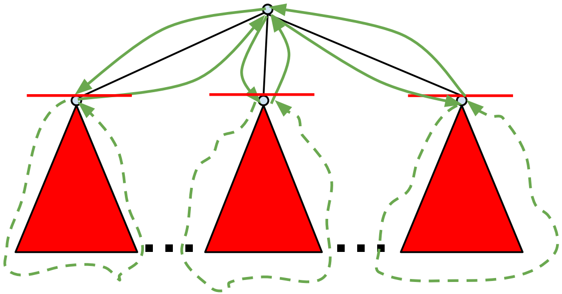
Completeness

```
completeness: (t : Tree X)
     → ∃ (t' : Tree X) , t ≈ t' × (X, t') ∈ treesToCheck Rs
```



Completeness

```
completeness: (t : Tree X)
     → ∃ (t' : Tree X) , t ≈ t' × (X, t') ∈ treesToCheck Rs
```



```
circular? : Rules → Bool
circular? = ⋁ { a ≟ b | t ← treesToCheck Rs, (a, b) ← dependencies t }
```

Thank you for your attention!
Questions?