Formalizing Attribute Grammars and Circularity Checking

Denis Firsov and Tarmo Uustalu

Institute of Cybernetics at TUT

October 18, 2013

Previous work and motivation

- Lexing (regular languages)
- Parsing (context-free languages)

Previous work and motivation

- Lexing (regular languages)
- Parsing (context-free languages)

Q: How to find a semantical value of an abstract syntax tree (AST)?
A: Attribute grammars

Previous work and motivation

- Lexing (regular languages)
- Parsing (context-free languages)

Q: How to find a semantical value of an abstract syntax tree (AST)?
A: Attribute grammars

Q: How to be sure that semantical value can be found for all parse trees?
A: Well-definedness property

Example: Semantics of binary numbers

$$\mathtt{N} \to \mathtt{L}$$
$$\mathtt{L}_1 \to \mathtt{L}_2\mathtt{B}$$

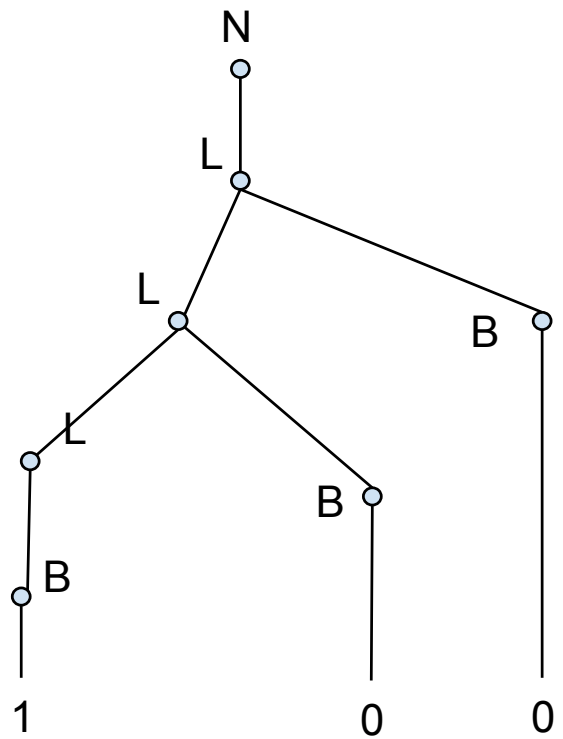
$$\mathtt{L} \to \mathtt{B}$$
$$\mathtt{B} \to \mathtt{0}$$
$$\mathtt{B} \to \mathtt{1}$$

Example: Semantics of binary numbers

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

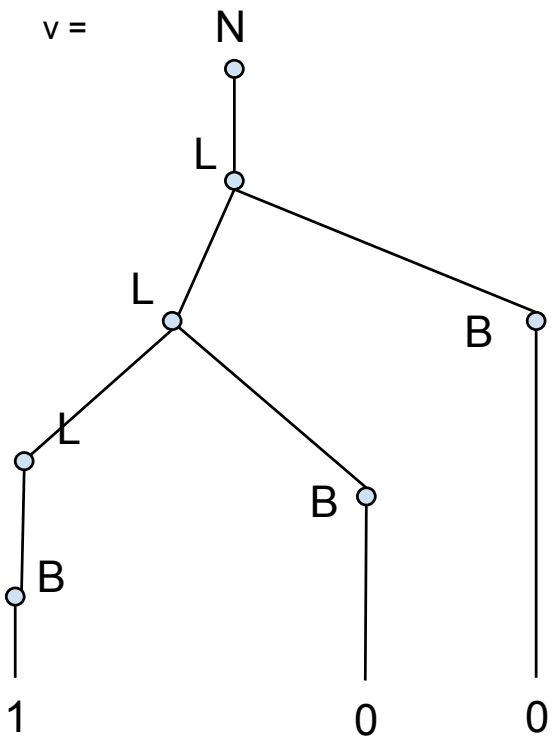
Example: Semantics of binary numbers

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



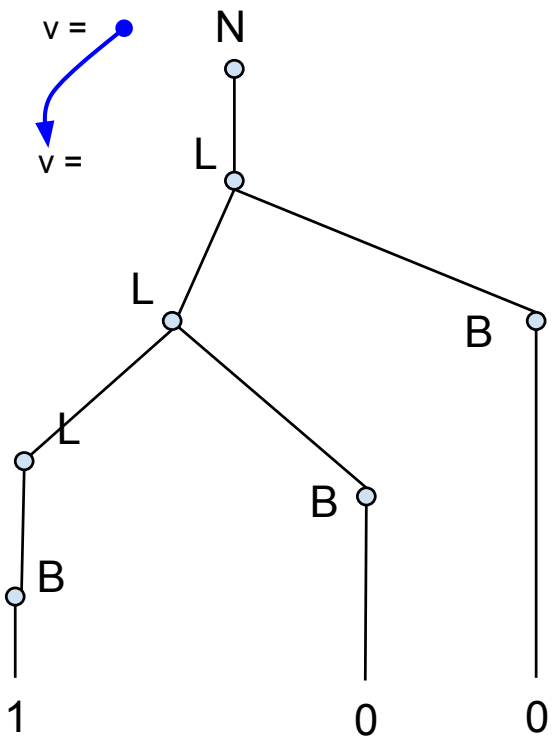
Example: Semantics of binary numbers

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



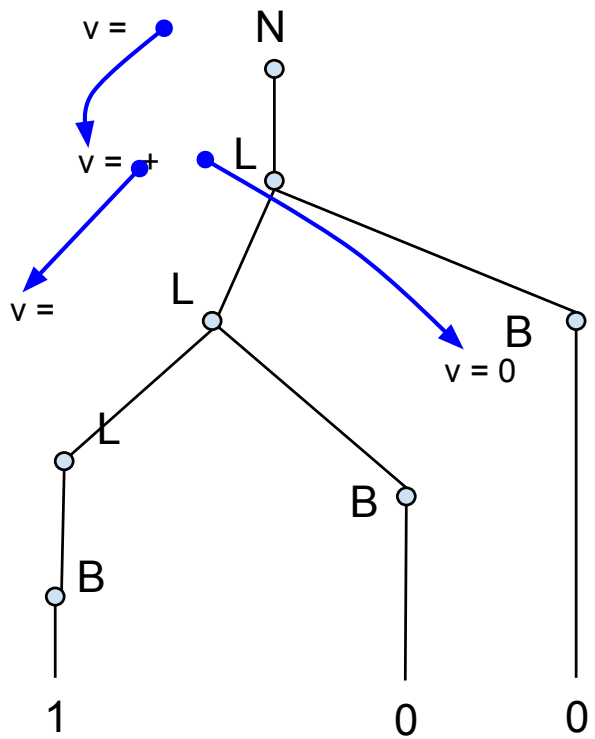
Example: Semantics of binary numbers

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



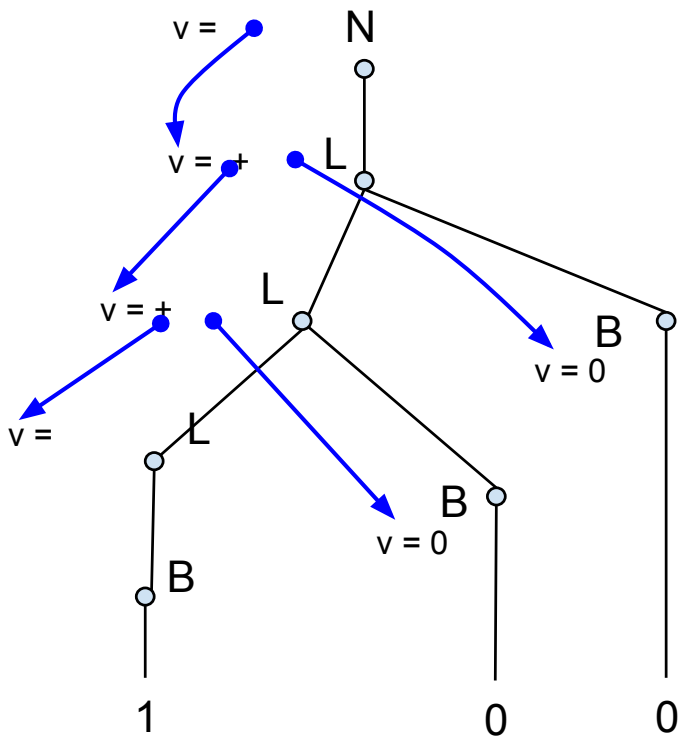
Example: Semantics of binary numbers

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



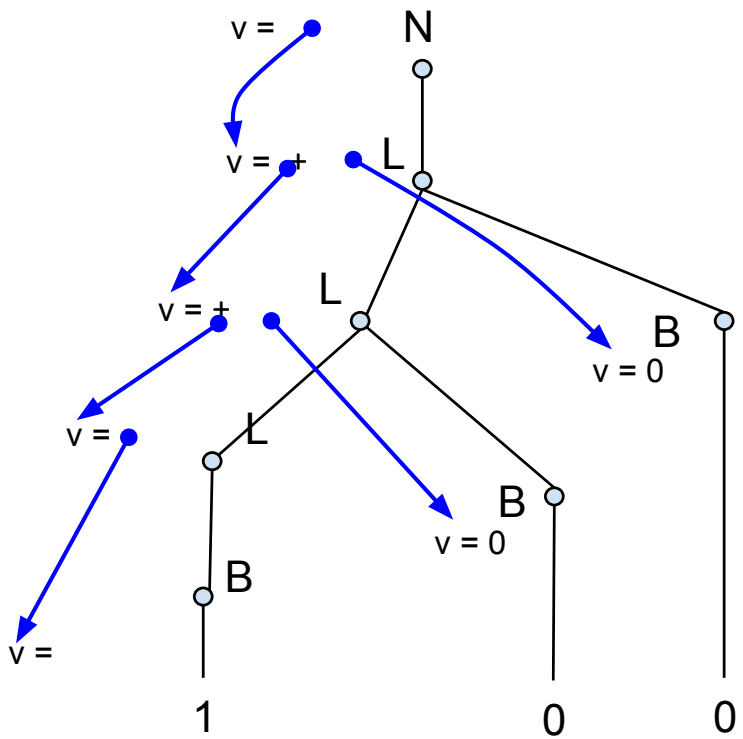
Example: Semantics of binary numbers

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



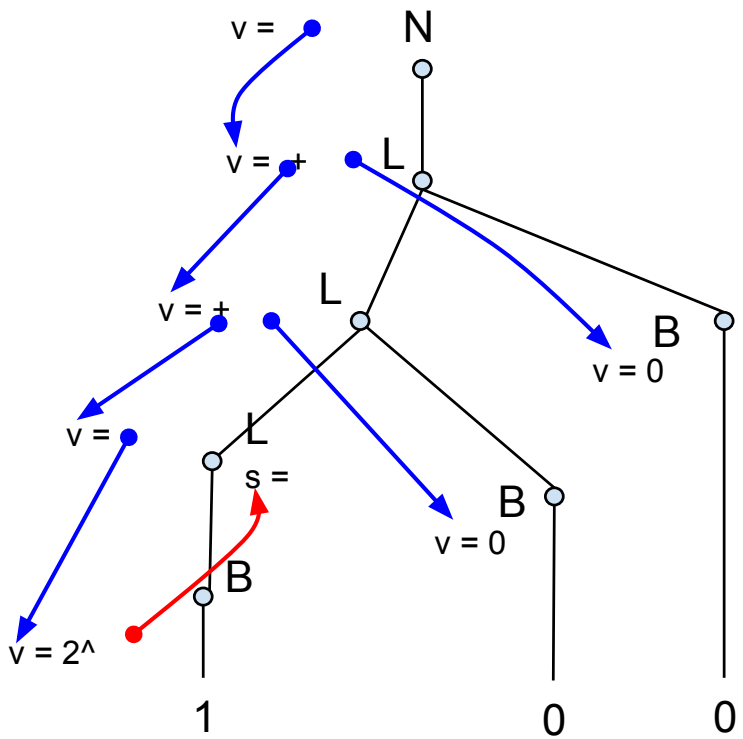
Example: Semantics of binary numbers

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



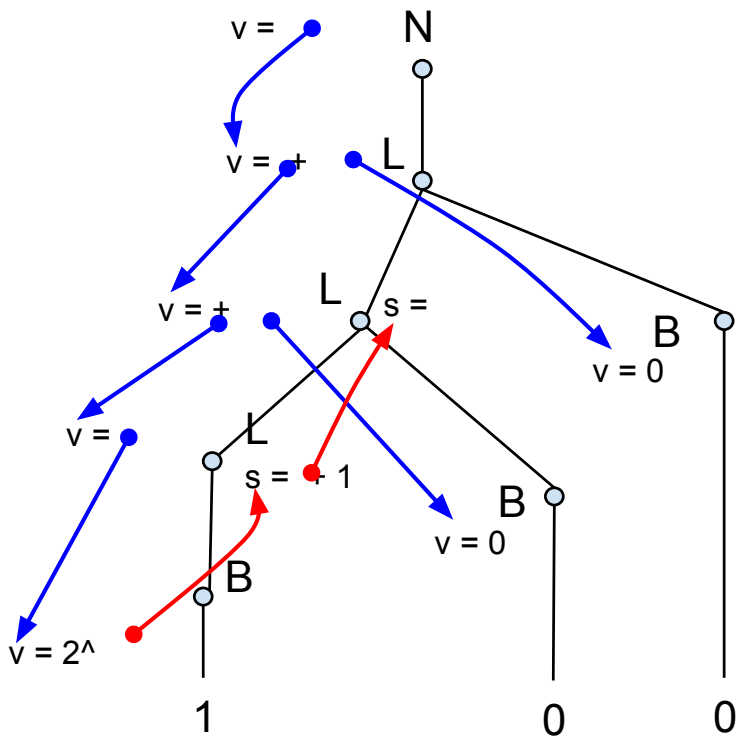
Example: Semantics of binary numbers

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



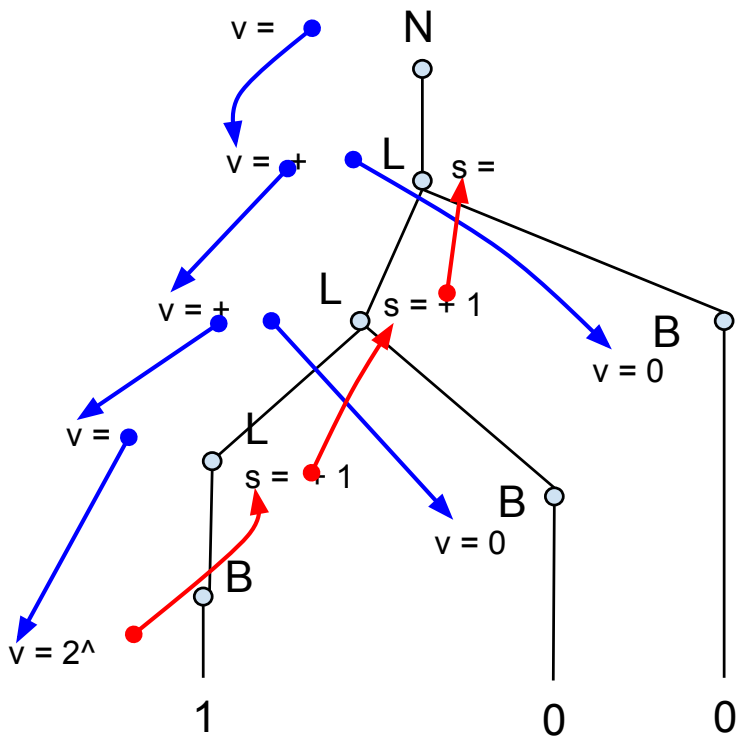
Example: Semantics of binary numbers

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



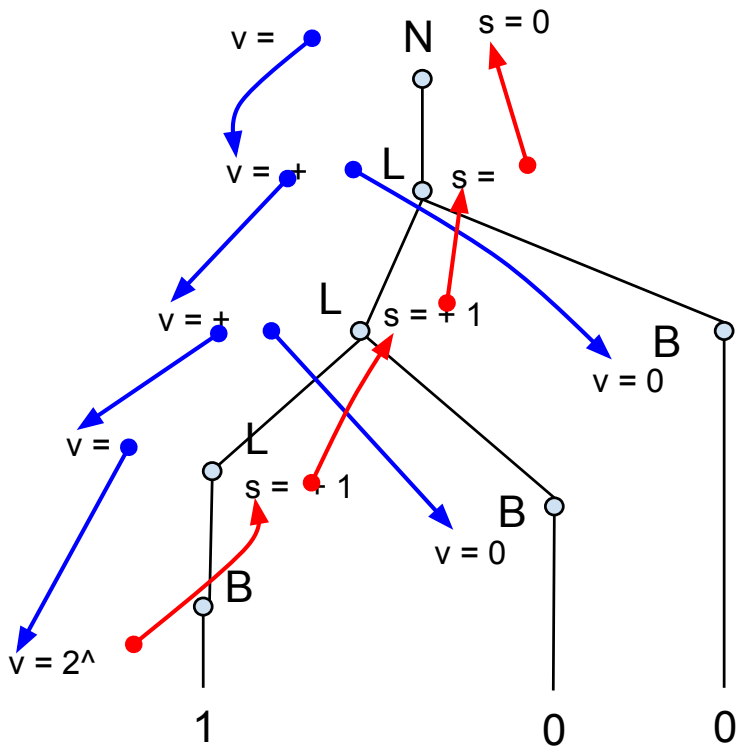
Example: Semantics of binary numbers

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



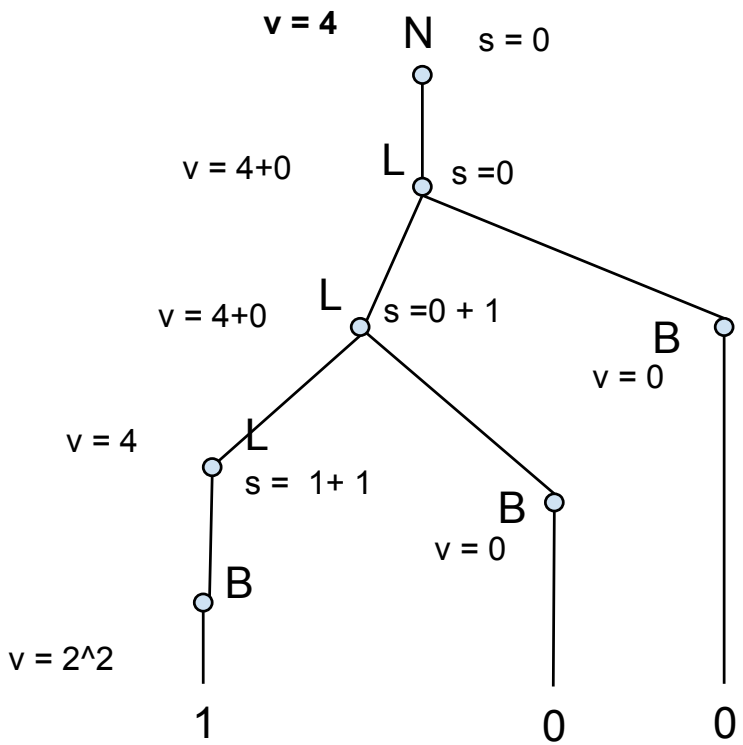
Example: Semantics of binary numbers

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



Example: Semantics of binary numbers

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



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

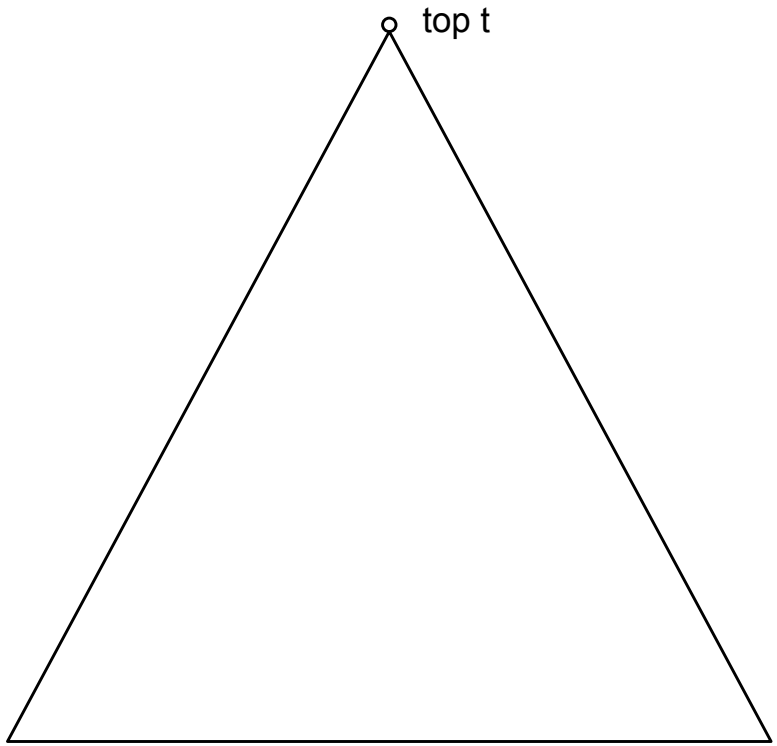
```
data PosTree  : ∀ {X : N} → Tree X → Set where
  top : ∀ {X} → (global : Tree X) → PosTree global
```

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
```

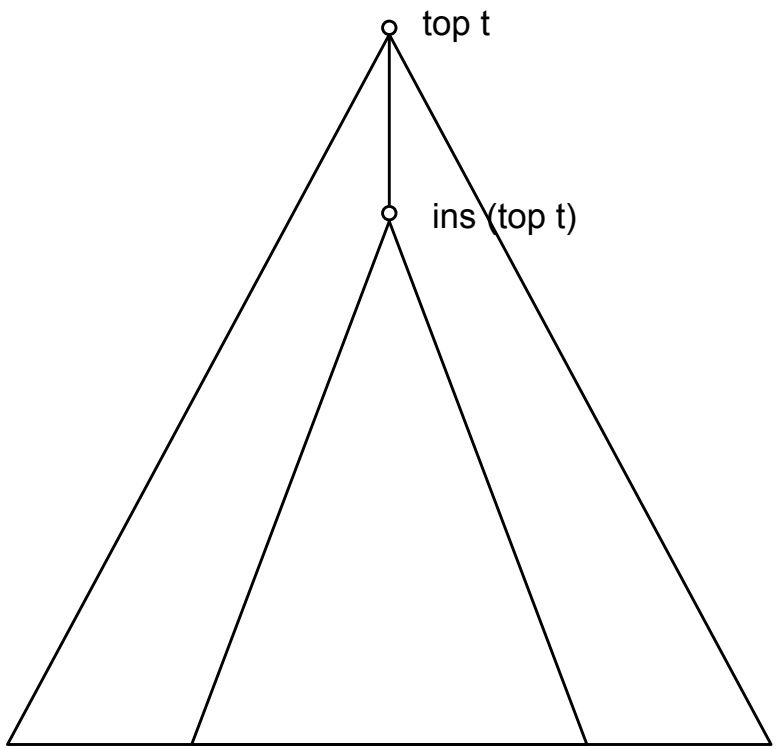
Basic definitions: Positions in trees

```
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
```



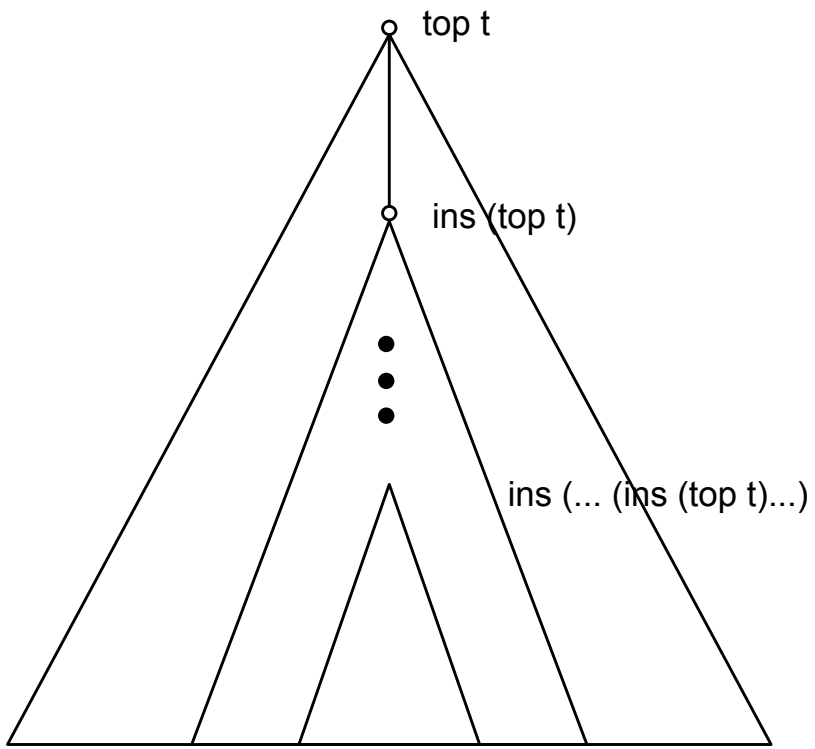
Basic definitions: Positions in trees

```
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

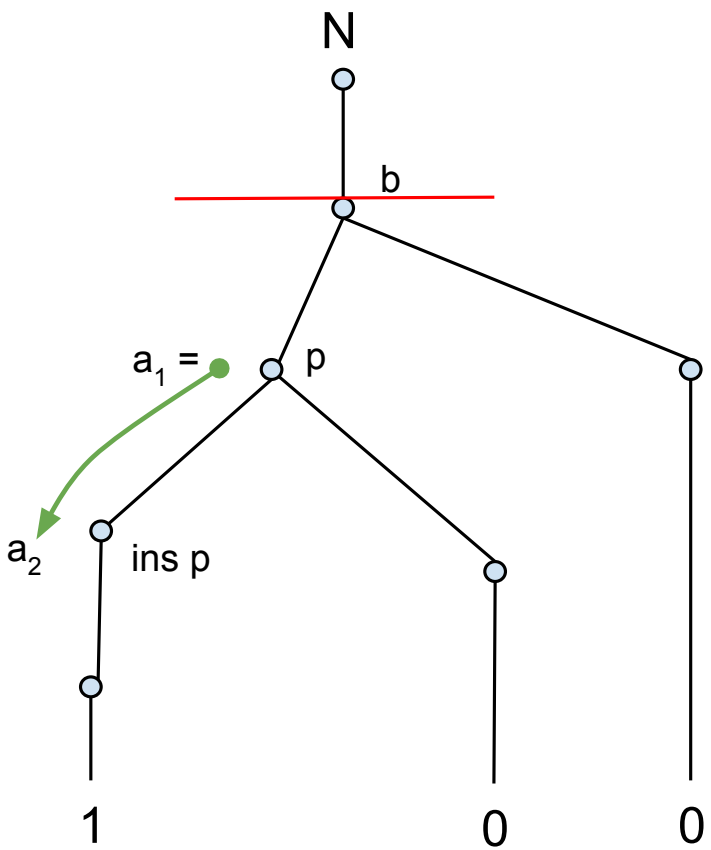
```
type Dir = ⇑ | ⇓


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 ]
```



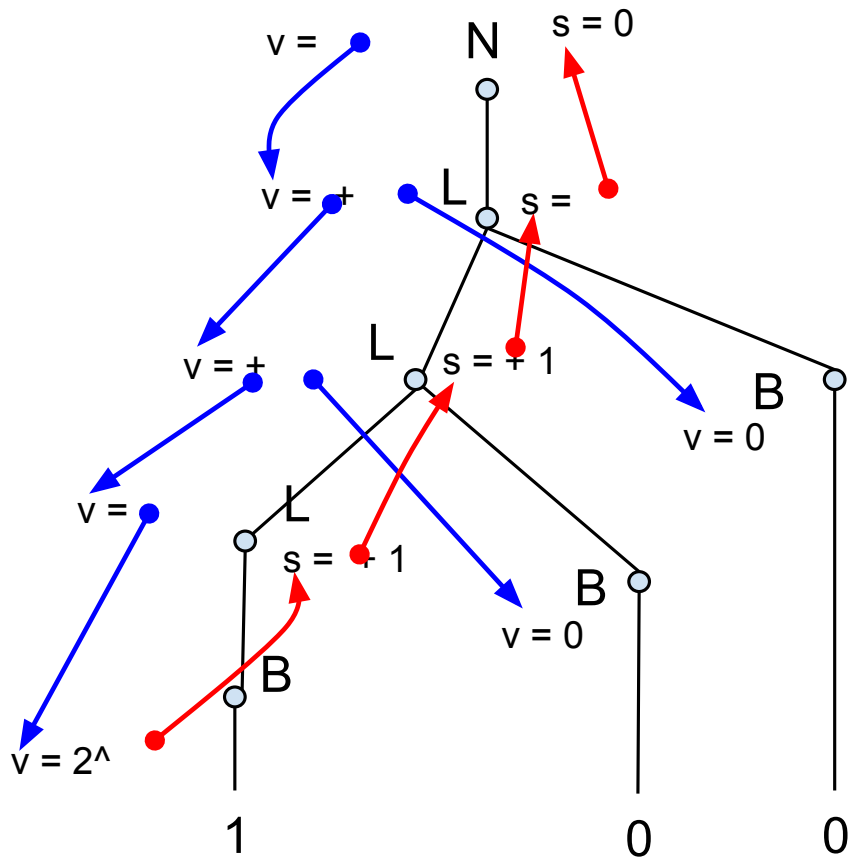
Basic definitions: Reflexive-transitive closure

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

Basic definitions: Reflexive-transitive closure

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

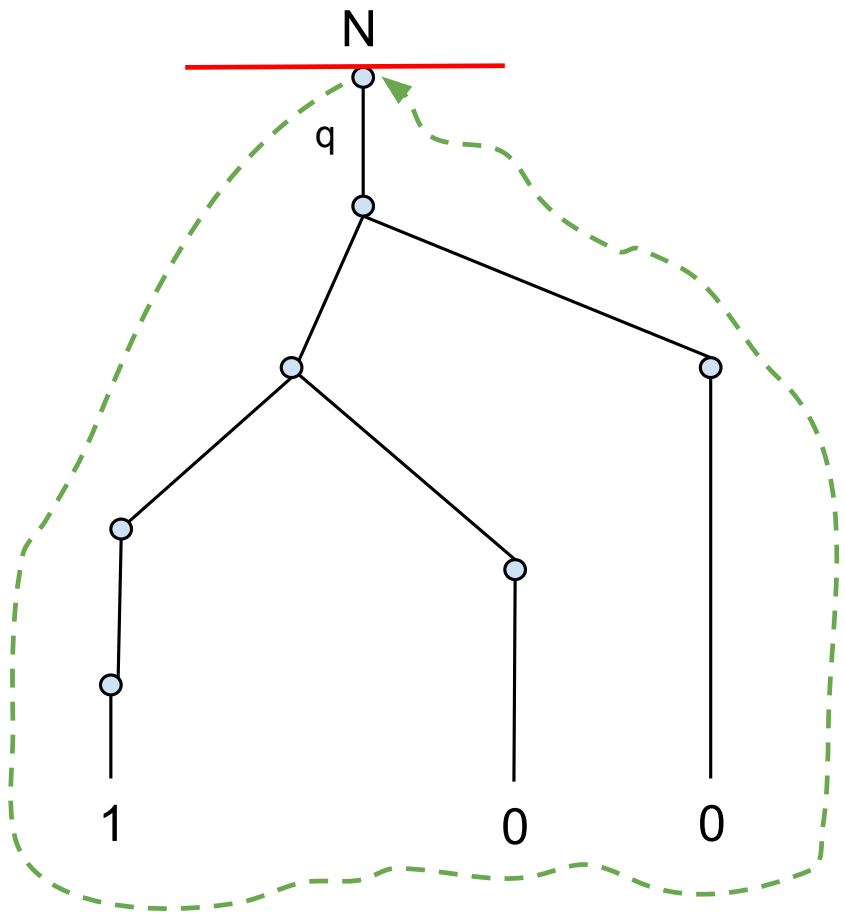
```
q |[ q , v ] ⁺↝ [ q , s ]
```



Basic definitions: Reflexive-transitive closure

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

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



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

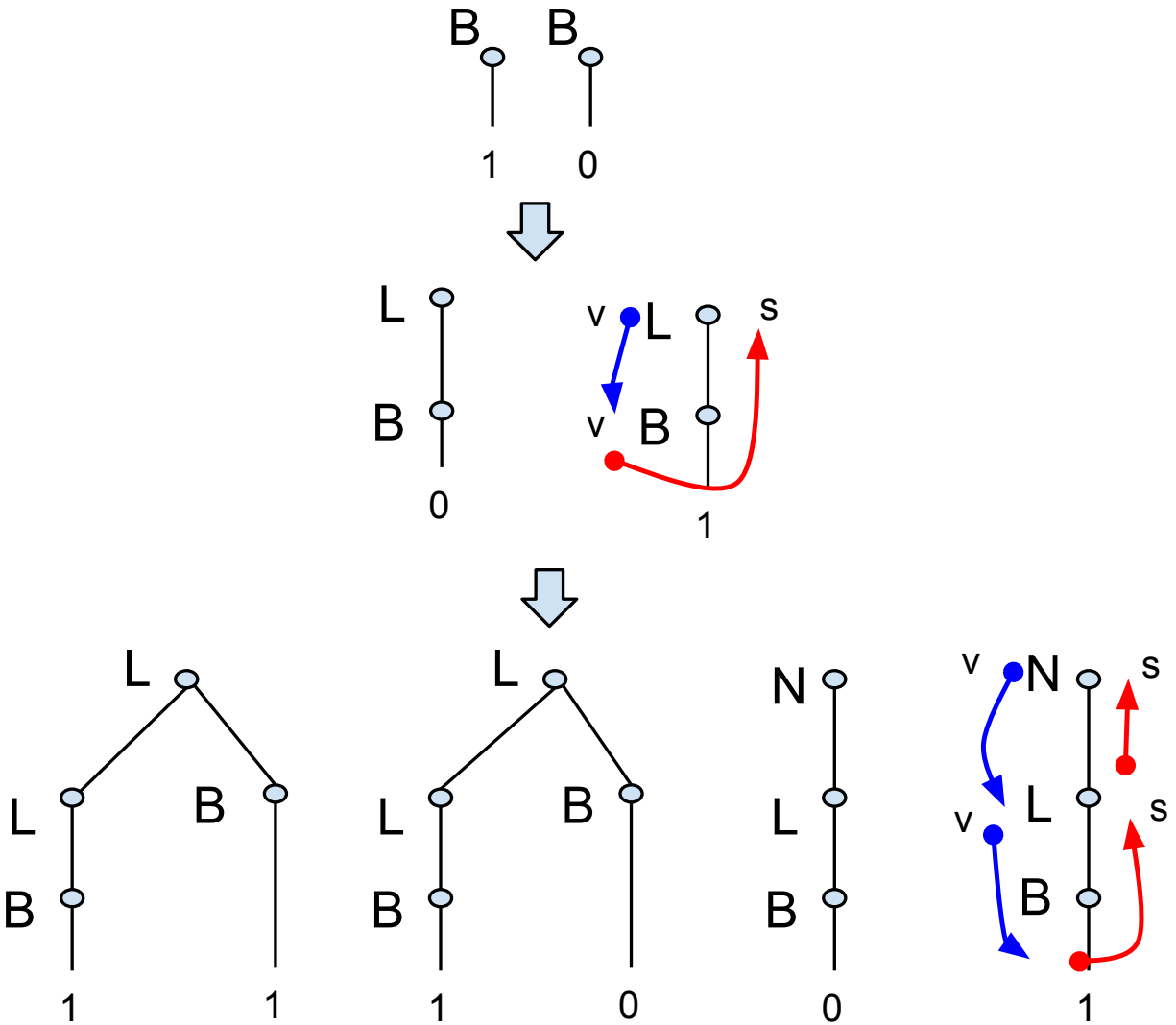
```
dependencies : {X : N} → Tree X → List (Attr × Attr)
dependencies {X} t = ...(reachability on graph)...

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'
```

Checking for circularity: Algorithm

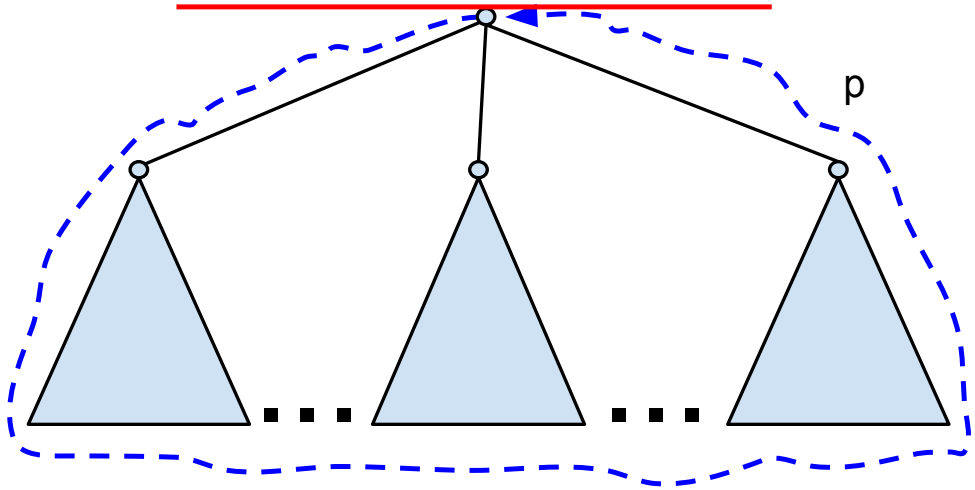
`treesToCheck : Rules` $\rightarrow$ `Forest`

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.

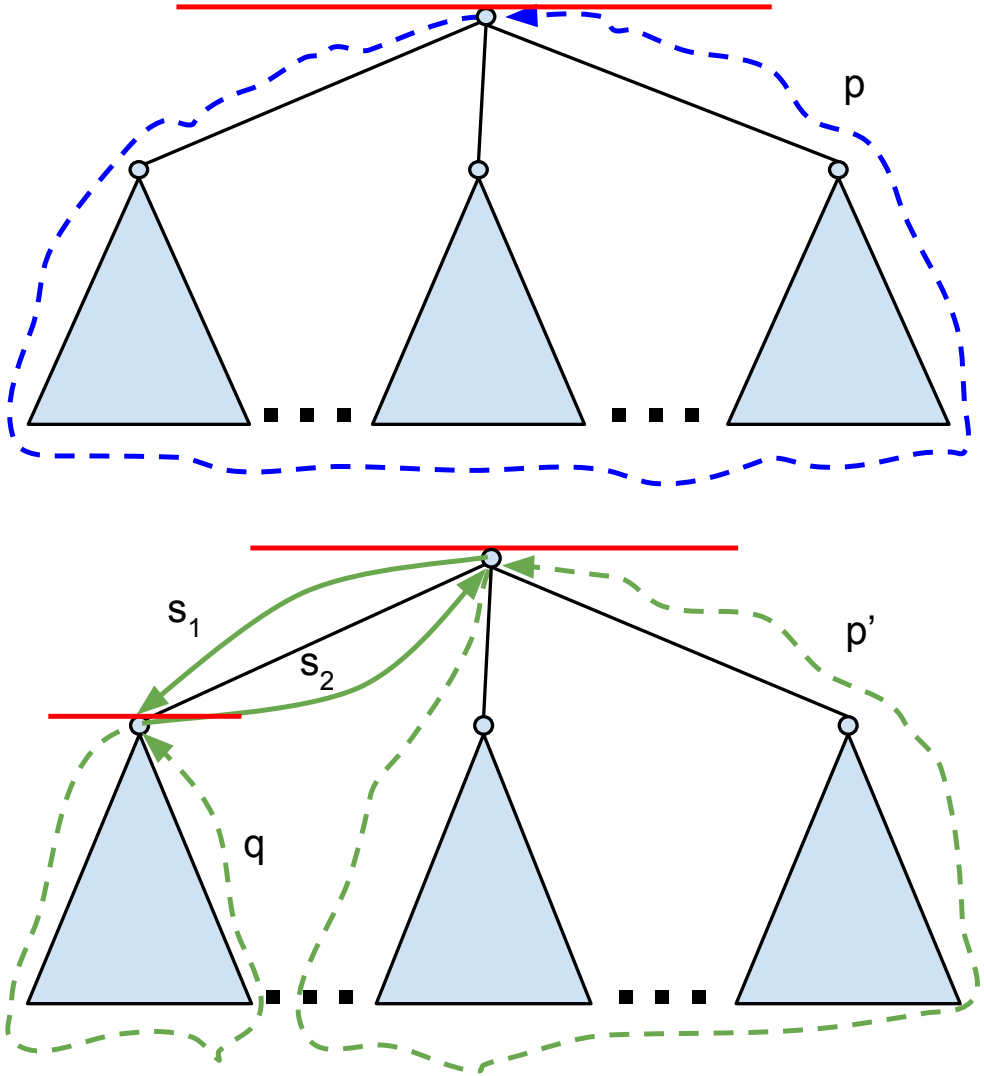
Checking for circularity: Example



Cycle decomposition



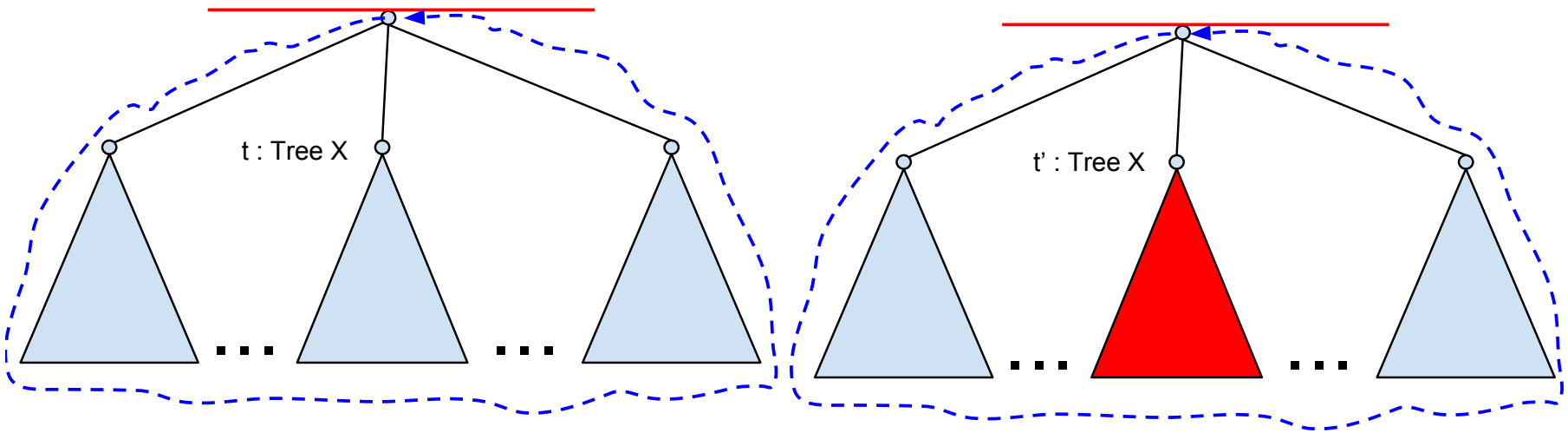
Cycle decomposition



$$p \equiv s_1 \mathbin{+\!\!+} q \mathbin{+\!\!+} s_2 \mathbin{+\!\!+} p'$$

Substitution lemma

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

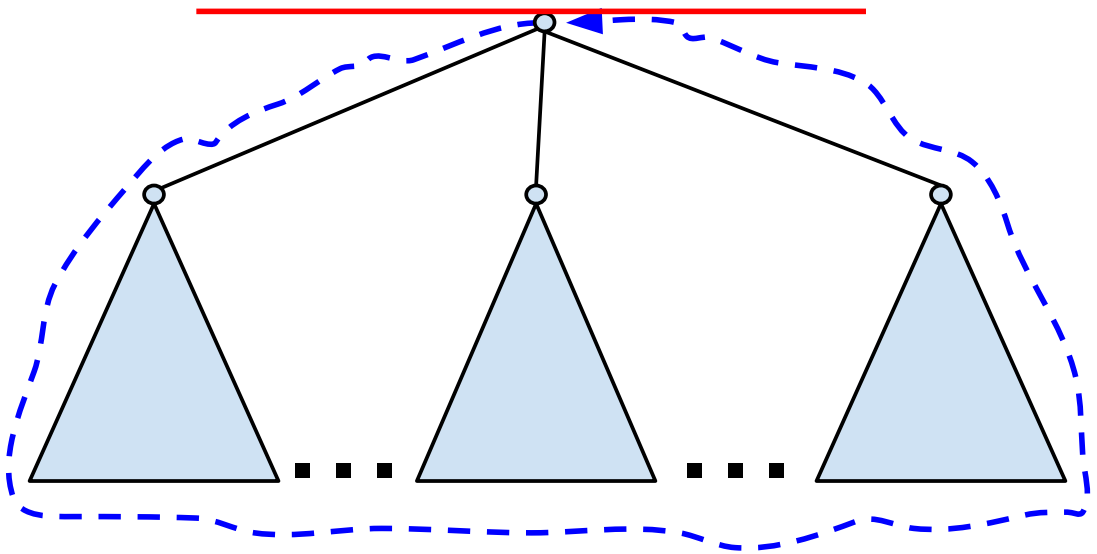


Completeness

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

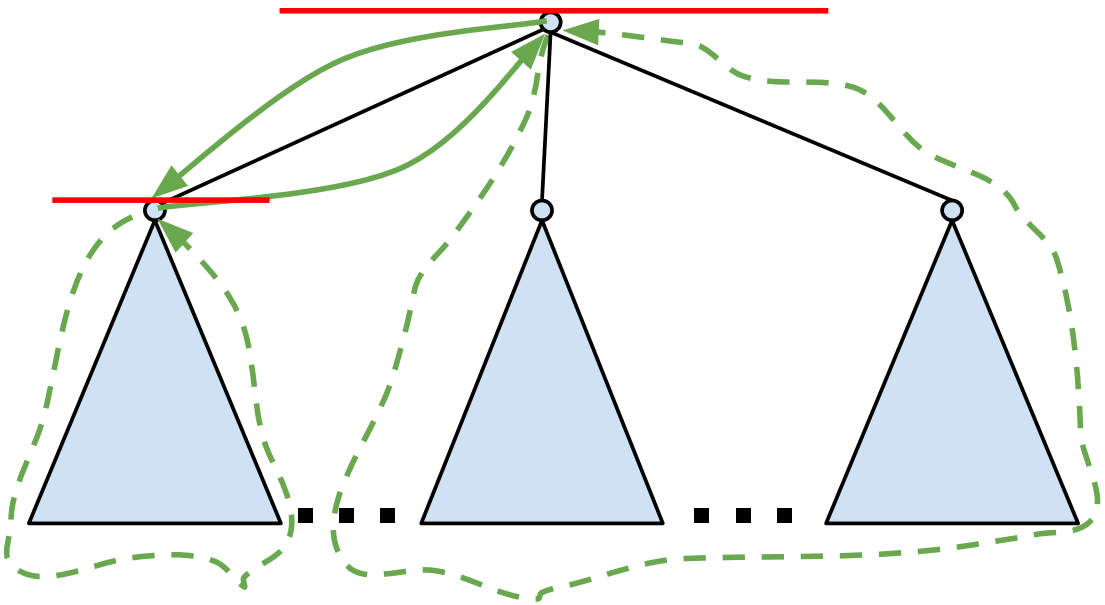
Completeness

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



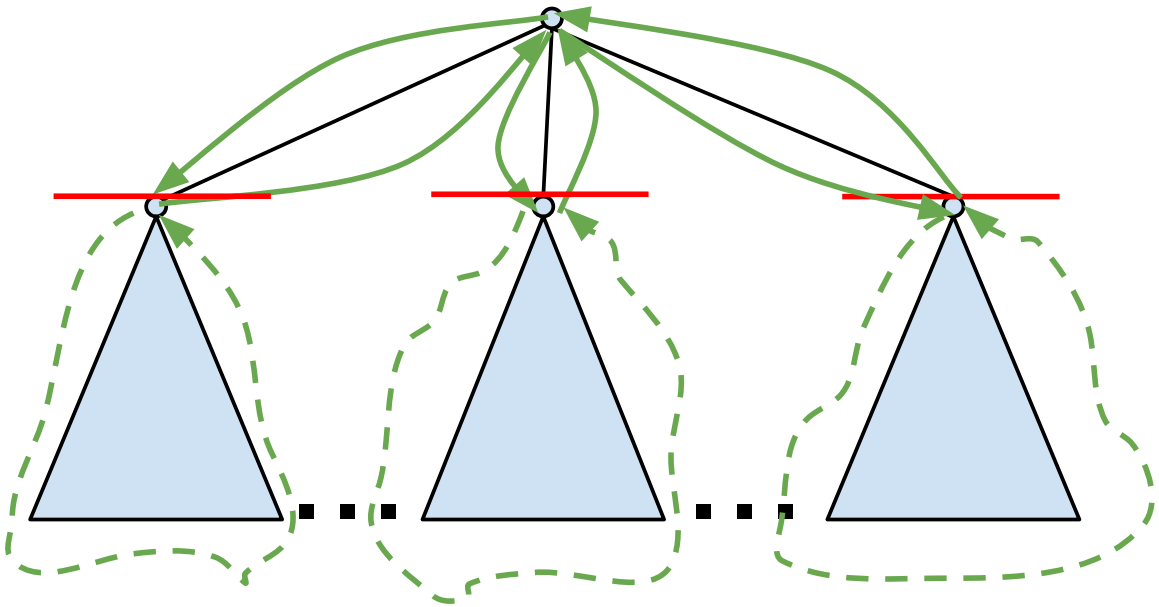
Completeness

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



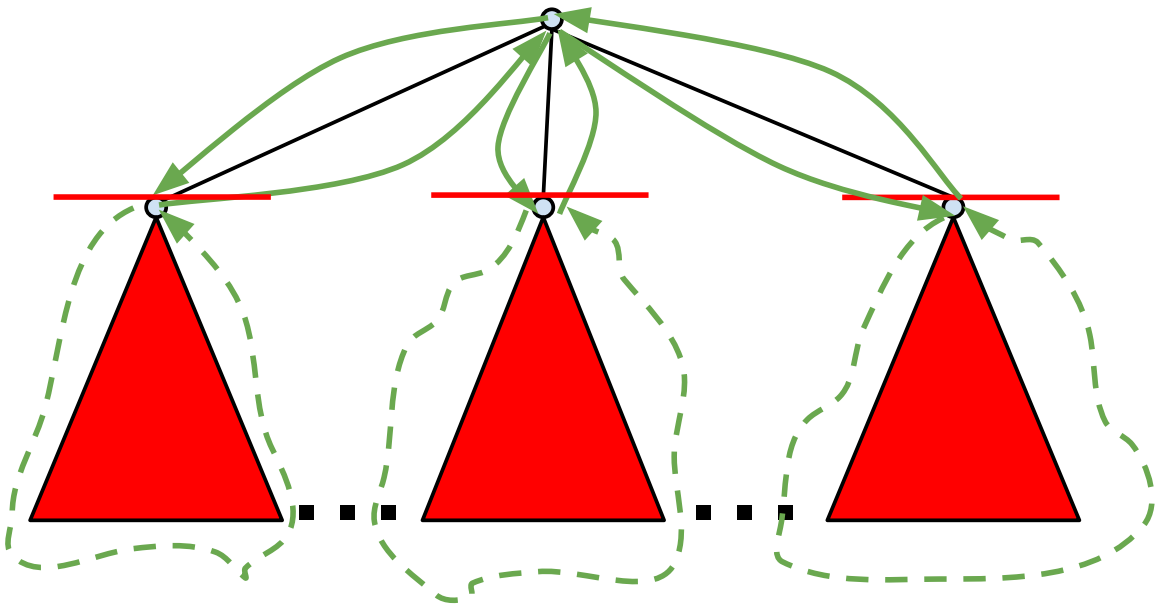
Completeness

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



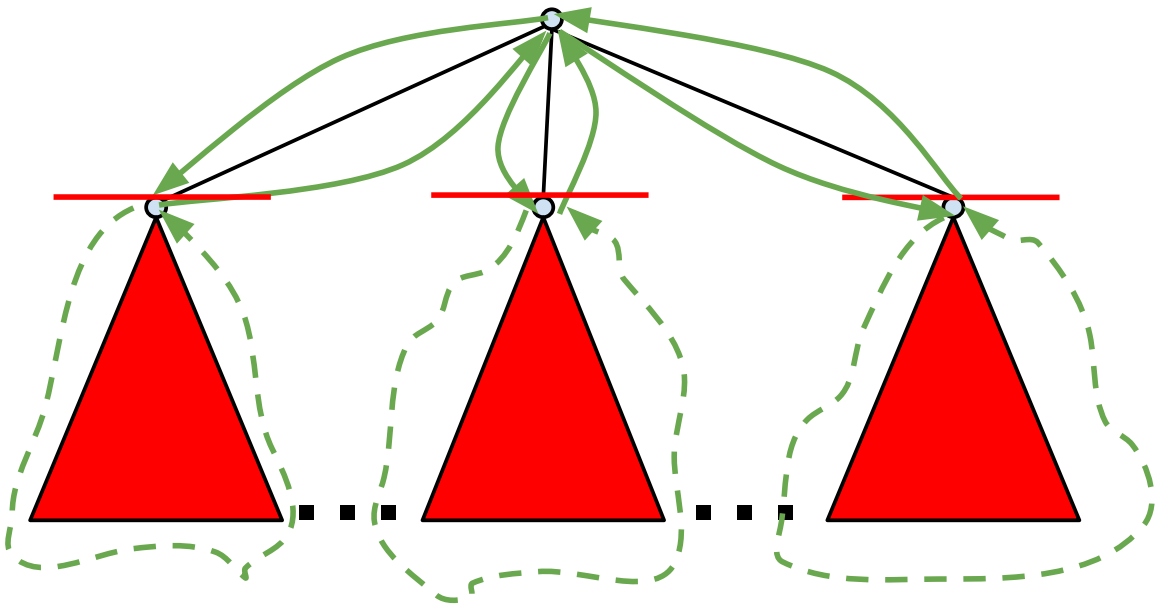
Completeness

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



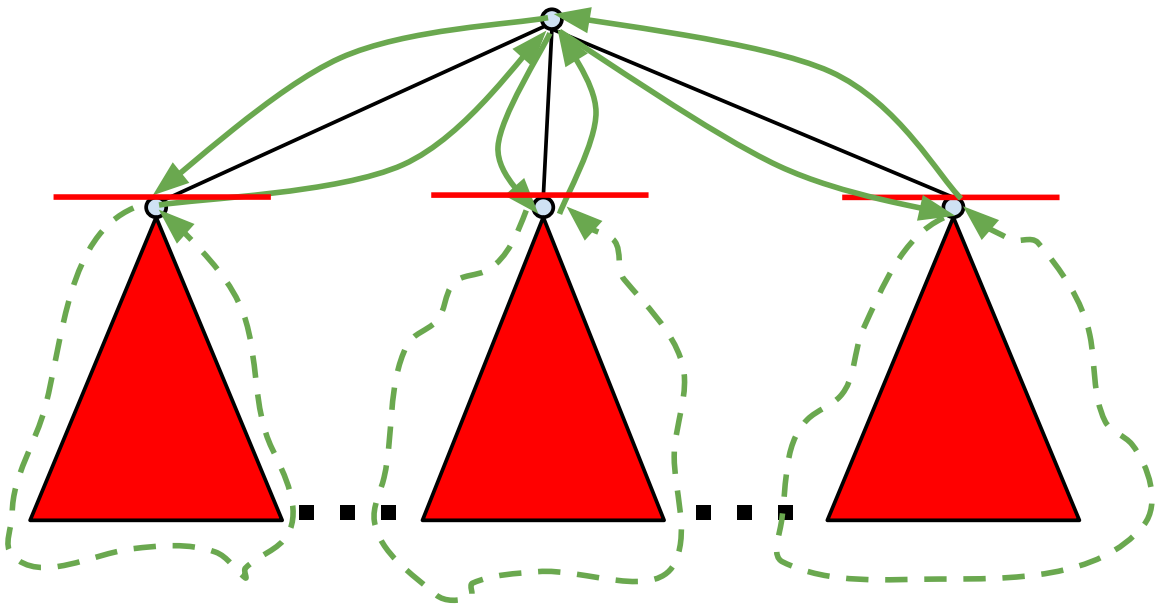
Completeness

```
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?