A graphical presentation of ML$^{\mathsf{F}}$ types with a linear-time incremental unification algorithm.

Didier Rémy

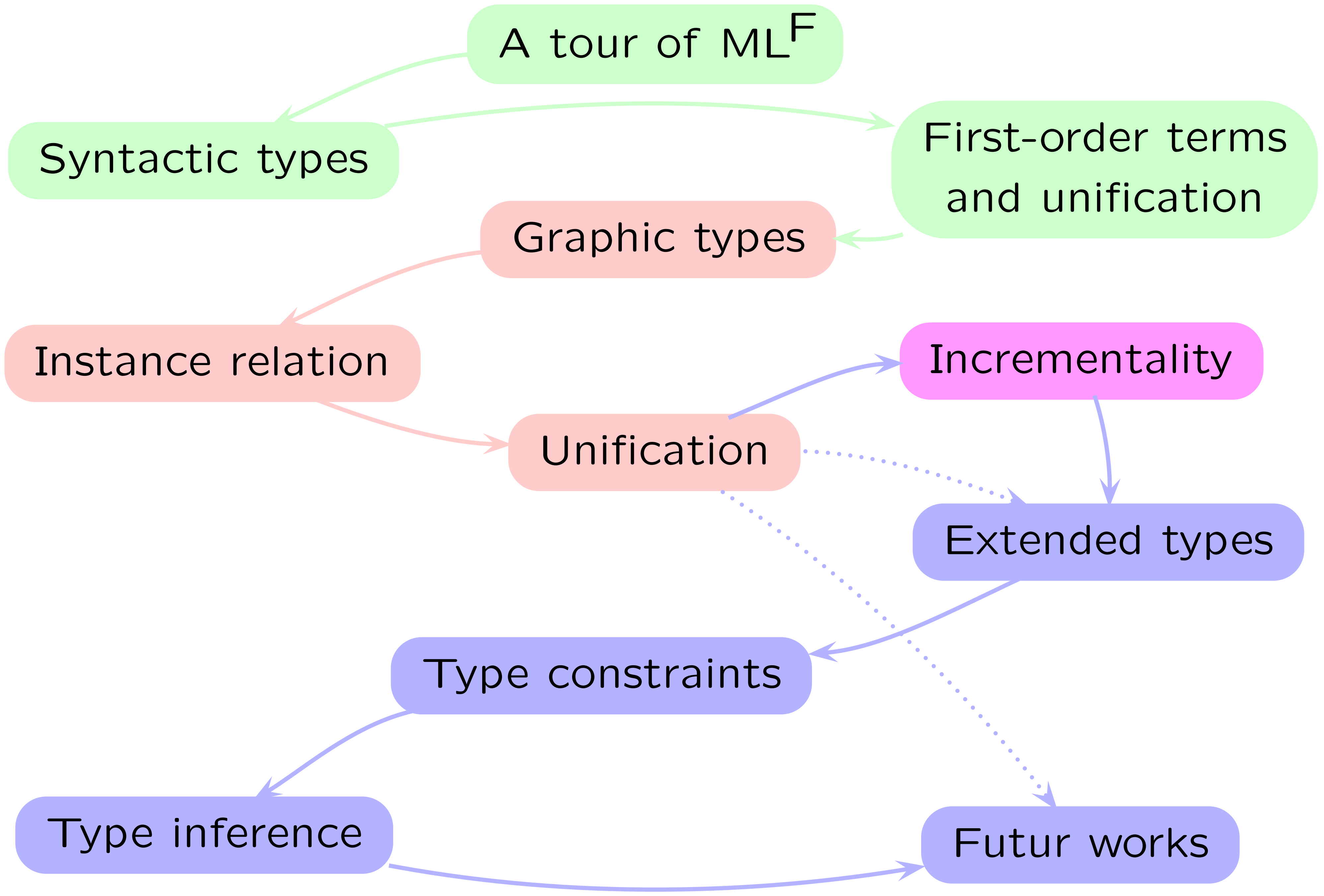
&

Boris Yakobowski

INRIA-Rocquencourt

Didier

Le Botlan


A tour of ML^F
Syntactic types
First-order terms and unification
Graphic types
Instance relation
Incrementality
Unification
Extended types
Type constraints
Type inference
Futur works

Motivations

First-class polymorphism is (sometimes) useful.

Today's solutions

- Should we give up type inference? no!
- Local type inference? no! —very fragile to program transformations
- Algorithmically specified type-inference?
- Stratifi inference? —still a backup when better solutions fail.
- Boxy types? no!

Improve System-F — regardless of type inference

- There is a gap between implicit and explicit type systems.
- Is System F the right choice? (think of F^η, $F_\leq$, F-bounded, *etc.*)

Motivations

First-class polymorphism is (sometimes) useful.

Today's solutions

- Should we give up type inference? no!
- Local type inference? no! —very fragile to program transformations
- Algorithmically specified type-inference?
- Stratified type inference? —still a backup when better solutions fail.
- Boxy types?

Improve System-F — regardless of type inference

- There is a gap between implicit and explicit type systems.
- Is System F the right choice? (think of F^η, $F_\leq$, F-bounded, *etc.*)

A key example for ML$^{\mathsf{F}}$

let choose $= \lambda(x)\ \lambda(y)$ **if** $true$ **then** x **else** $y : \forall\,\alpha \cdot \alpha \rightarrow \alpha \rightarrow \alpha$

let $id = \lambda(z)\ z : \forall\,\alpha \cdot \alpha \rightarrow \alpha$

choose $(\lambda(x)\ x)$:

A key example for ML^{F}

let choose $= \lambda(x)\ \lambda(y)$ **if** $true$ **then** x **else** $y : \forall\, \alpha \cdot \alpha \rightarrow \alpha \rightarrow \alpha$

let $id = \lambda(z)\ z : \forall\, \alpha \cdot \alpha \rightarrow \alpha$

choose $(\lambda(x)\ x)$ $: \left\{ \begin{array}{l} \forall\, \alpha \cdot (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \\ (\forall\, \alpha \cdot \alpha \rightarrow \alpha) \rightarrow (\forall\, \alpha \cdot \alpha \rightarrow \alpha) \end{array} \right.$

A key example for ML^{F}

let choose $= \lambda(x)\ \lambda(y)$ **if** true **then** x **else** $y : \forall\, \alpha \cdot \alpha \to \alpha \to \alpha$

let $id = \lambda(z)\ z : \forall\, \alpha \cdot \alpha \to \alpha$

choose $(\lambda(x)\ x)\ \ : \left\{ \begin{array}{l} \forall\, \alpha \cdot (\alpha \to \alpha) \to (\alpha \to \alpha) \\ (\forall\, \alpha \cdot \alpha \to \alpha) \to (\forall\, \alpha \cdot \alpha \to \alpha) \end{array} \right\}$ No better choice in F

A key example for ML$^{\mathsf{F}}$

let choose $= \lambda(x)\ \lambda(y)$ **if** $true$ **then** x **else** $y : \forall\,\alpha \cdot \alpha \to \alpha \to \alpha$

let $id = \lambda(z)\ z : \forall\,\alpha \cdot \alpha \to \alpha$

choose $(\lambda(x)\ x)\ : \left\{ \begin{array}{l} \forall\,\alpha \cdot (\alpha \to \alpha) \to (\alpha \to \alpha) \\ (\forall\,\alpha \cdot \alpha \to \alpha) \to (\forall\,\alpha \cdot \alpha \to \alpha) \end{array} \right\}$ No better choice in F

$: \quad \forall\,(\beta \geq \forall\,(\alpha)\ \alpha \to \alpha)\ \beta \to \beta$ in ML$^{\mathsf{F}}$

A key example for ML$^{\mathsf{F}}$

let choose $= \lambda(x)\ \lambda(y)$ **if** $true$ **then** x **else** $y : \forall\,\alpha \cdot \alpha \to \alpha \to \alpha$

let $id = \lambda(z)\ z : \forall\,\alpha \cdot \alpha \to \alpha$

choose $(\lambda(x)\ x)$ $:$ $\left\{ \begin{array}{l} \forall\,\alpha \cdot (\alpha \to \alpha) \to (\alpha \to \alpha) \\ (\forall\,\alpha \cdot \alpha \to \alpha) \to (\forall\,\alpha \cdot \alpha \to \alpha) \end{array} \right\}$ No better choice in F

$: \quad \forall\,(\beta \geq \forall\,(\alpha)\ \alpha \to \alpha)\ \beta \to \beta$ in ML$^{\mathsf{F}}$

$$\leqslant \left\{ \begin{array}{l} \forall\,(\beta = \forall\,(\alpha)\ \alpha \to \alpha)\ \beta \to \beta \\ \forall\,(\alpha)\ \forall\,(\beta = \alpha \to \alpha)\ \beta \to \beta \end{array} \right.$$

A key example for ML$^{\mathsf{F}}$

let choose $= \lambda(x)\ \lambda(y)$ **if** true **then** x **else** $y : \forall\,\alpha \cdot \alpha \to \alpha \to \alpha$

let $\mathit{id} = \lambda(z)\ z : \forall\,\alpha \cdot \alpha \to \alpha$

$$\text{choose } (\lambda(x)\ x)\ :\ \left\{ \begin{array}{l} \forall\,\alpha \cdot (\alpha \to \alpha) \to (\alpha \to \alpha) \\ (\forall\,\alpha \cdot \alpha \to \alpha) \to (\forall\,\alpha \cdot \alpha \to \alpha) \end{array} \right\}$$

No better choice in F

$$:\quad \forall\,(\beta \geq \forall\,(\alpha)\ \alpha \to \alpha)\ \beta \to \beta \quad \text{in ML}^{\mathsf{F}}$$

$$\leqslant \left\{ \begin{array}{l} \forall\,(\beta = \forall\,(\alpha)\ \alpha \to \alpha)\ \beta \to \beta \\ \forall\,(\alpha)\ \forall\,(\beta = \alpha \to \alpha)\ \beta \to \beta \end{array} \right.$$

But

$\lambda(x)\ x\ x$: ill-typed

Do not guess polymorphism!

$\lambda(x : \forall\,(\alpha)\ \alpha \to \alpha)\ x\ x$: $\forall\,(\beta = \forall\,(\alpha)\ \alpha \to \alpha)\ \beta \to \beta$

Properties

Principal types

Type inference, relies on *first-order unification in the presence of second-order types*.

Convervative over both ML and System F

ML programs need no annotations

F programs need fewer annotations: type abstractions and type applications are always inferred.

ML$^{\mathsf{F}}$ is robust (to program transformations)

For example, if $E[a_1\ a_2]$ is typable so $E[apply\ a_1\ a_2]$ where $apply$ is $\lambda(f)\ \lambda(x)\ f\ x$.

Generic Type System

Var

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

Fun

$$\frac{\Gamma, x : \tau \vdash a : \tau'}{\Gamma \vdash \lambda(x)\ a : \tau \to \tau'}$$

App

$$\frac{\Gamma \vdash a_1 : \tau_2 \longrightarrow \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1}$$

Inst

$$\frac{\Gamma \vdash a : \sigma \qquad \sigma \leqslant \sigma'}{\Gamma \vdash a : \sigma'}$$

Gen

$$\frac{\Gamma \vdash a : \sigma \qquad \mathit{dom}\,(q) \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash a : \forall\, q \cdot \sigma}$$

Let

$$\frac{\Gamma \vdash a : \sigma \qquad \Gamma, x : \sigma \vdash a' : \sigma'}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \sigma'}$$

Generic Type System

Var

$$\frac{x : \sigma \in \Gamma}{(Q)\ \Gamma \vdash x : \sigma}$$

Fun

$$\frac{(Q)\ \Gamma, x : \tau \vdash a : \tau'}{(Q)\ \Gamma \vdash \lambda(x)\ a : \tau \to \tau'}$$

App

$$\frac{(Q)\ \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad (Q)\ \Gamma \vdash a_2 : \tau_2}{(Q)\ \Gamma \vdash a_1\ a_2 : \tau_1}$$

Inst

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \leqslant \sigma'}{(Q)\ \Gamma \vdash a : \sigma'}$$

Gen

$$\frac{(Q, q)\ \Gamma \vdash a : \sigma \qquad \textit{dom}\,(q) \notin \mathsf{ftv}(\Gamma)}{(Q)\ \Gamma \vdash a : \forall\, q \cdot \sigma}$$

Let

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \Gamma, x : \sigma \vdash a' : \sigma'}{(Q)\ \Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \sigma'}$$

Generic Type System

Var

$$\frac{x : \sigma \in \Gamma}{(Q)\ \Gamma \vdash x : \sigma}$$

Fun

$$\frac{(Q)\ \Gamma, x : \tau \vdash a : \tau'}{(Q)\ \Gamma \vdash \lambda(x)\ a : \tau \rightarrow \tau'}$$

App

$$\frac{(Q)\ \Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \qquad (Q)\ \Gamma \vdash a_2 : \tau_2}{(Q)\ \Gamma \vdash a_1\ a_2 : \tau_1}$$

Inst

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \leqslant \sigma'}{(Q)\ \Gamma \vdash a : \sigma'}$$

Gen

$$\frac{(Q, q)\ \Gamma \vdash a : \sigma \qquad \textit{dom}\,(q) \notin \mathsf{ftv}(\Gamma)}{(Q)\ \Gamma \vdash a : \forall\, q \cdot \sigma}$$

Let

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \Gamma, x : \sigma \vdash a' : \sigma'}{(Q)\ \Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \sigma'}$$

(Q) binds free type variables of Γ.

(Q) could be interleaved with Γ as Γ_Q and read back by restricting the domain of Γ_Q to type variables.

Generic Type System

Var

$$\frac{x : \sigma \in \Gamma}{(Q)\ \Gamma \vdash x : \sigma}$$

Fun

$$\frac{(Q)\ \Gamma, x : \tau \vdash a : \tau'}{(Q)\ \Gamma \vdash \lambda(x)\ a : \tau \to \tau'}$$

App

$$\frac{(Q)\ \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad (Q)\ \Gamma \vdash a_2 : \tau_2}{(Q)\ \Gamma \vdash a_1\ a_2 : \tau_1}$$

Inst

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \leqslant \sigma'}{(Q)\ \Gamma \vdash a : \sigma'}$$

Gen

$$\frac{(Q, q)\ \Gamma \vdash a : \sigma \qquad \mathit{dom}\,(q) \notin \mathsf{ftv}(\Gamma)}{(Q)\ \Gamma \vdash a : \forall\, q \cdot \sigma}$$

Let

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \Gamma, x : \sigma \vdash a' : \sigma'}{(Q)\ \Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \sigma'}$$

ML

Types

$$\tau ::= \quad \alpha \mid \tau \to \tau$$

$$\sigma ::= \quad \tau \mid \forall\,(q)\ \sigma$$

$$q ::= \quad \alpha$$

Instance relation $\leqslant$

$$\forall\,(\bar{\alpha})\ \tau \leqslant \forall\,(\beta)\ \tau[\bar{\tau}'/\bar{\alpha}]$$

$$\beta \notin \mathsf{ftv}(\forall\,(\bar{\alpha})\ \tau)$$

Generic Type System

Var

$$\frac{x : \sigma \in \Gamma}{(Q)\ \Gamma \vdash x : \sigma}$$

Fun

$$\frac{(Q)\ \Gamma, x : \tau \vdash a : \tau'}{(Q)\ \Gamma \vdash \lambda(x)\ a : \tau \to \tau'}$$

App

$$\frac{(Q)\ \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad (Q)\ \Gamma \vdash a_2 : \tau_2}{(Q)\ \Gamma \vdash a_1\ a_2 : \tau_1}$$

Inst

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \leqslant \sigma'}{(Q)\ \Gamma \vdash a : \sigma'}$$

Gen

$$\frac{(Q, q)\ \Gamma \vdash a : \sigma \qquad \mathit{dom}\,(q) \notin \mathsf{ftv}(\Gamma)}{(Q)\ \Gamma \vdash a : \forall\, q \cdot \sigma}$$

Let

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \Gamma, x : \sigma \vdash a' : \sigma'}{(Q)\ \Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \sigma'}$$

System F

Types

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall\,(\alpha)\ \tau$$

$$\sigma ::= \tau$$

$$q ::= \alpha$$

Instance relation $\leqslant$

$$\forall\,(\bar\alpha)\ \tau \leqslant \forall\,(\beta)\ \tau[\bar\tau'/\bar\alpha]$$

$$\beta \notin \mathsf{ftv}(\forall\,(\bar\alpha)\ \tau)$$

Generic Type System

Var

$$\frac{x : \sigma \in \Gamma}{(Q)\ \Gamma \vdash x : \sigma}$$

Fun

$$\frac{(Q)\ \Gamma, x : \tau \vdash a : \tau'}{(Q)\ \Gamma \vdash \lambda(x)\ a : \tau \rightarrow \tau'}$$

App

$$\frac{(Q)\ \Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \qquad (Q)\ \Gamma \vdash a_2 : \tau_2}{(Q)\ \Gamma \vdash a_1\ a_2 : \tau_1}$$

Inst

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \leqslant \sigma'}{(Q)\ \Gamma \vdash a : \sigma'}$$

Gen

$$\frac{(Q, q)\ \Gamma \vdash a : \sigma \qquad \textit{dom}\,(q) \notin \mathsf{ftv}(\Gamma)}{(Q)\ \Gamma \vdash a : \forall\, q \cdot \sigma}$$

Let

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \Gamma, x : \sigma \vdash a' : \sigma'}{(Q)\ \Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \sigma'}$$

System $\mathbf{F}^{\eta}$

Types

$$\tau ::= \quad \alpha \mid \tau \rightarrow \tau \mid \forall\,(\alpha)\ \tau$$

$$\sigma ::= \quad \tau$$

$$q ::= \quad \alpha$$

Instance relation $\leqslant$

type containment :

deep, contra-variant, *etc.*

Generic Type System

Var

$$\frac{x : \sigma \in \Gamma}{(Q)\ \Gamma \vdash x : \sigma}$$

Fun

$$\frac{(Q)\ \Gamma, x : \tau \vdash a : \tau'}{(Q)\ \Gamma \vdash \lambda(x)\ a : \tau \to \tau'}$$

App

$$\frac{(Q)\ \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad (Q)\ \Gamma \vdash a_2 : \tau_2}{(Q)\ \Gamma \vdash a_1\ a_2 : \tau_1}$$

Inst

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \leqslant \sigma'}{(Q)\ \Gamma \vdash a : \sigma'}$$

Gen

$$\frac{(Q, q)\ \Gamma \vdash a : \sigma \qquad \mathit{dom}\,(q) \notin \mathsf{ftv}(\Gamma)}{(Q)\ \Gamma \vdash a : \forall\, q \cdot \sigma}$$

Let

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \Gamma, x : \sigma \vdash a' : \sigma'}{(Q)\ \Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \sigma'}$$

Explicit MLF

Types

$$\begin{aligned} \tau &::= \quad \alpha \mid \tau \to \tau \\ \sigma &::= \quad \tau \mid \forall\,(q)\ \tau \mid \bot \\ q &::= \quad (\alpha \geq \sigma) \mid (\alpha = \sigma) \end{aligned}$$

Instance relation $\leqslant$

$$\underline{\sqsubseteq}$$

Generic Type System

Var

$$\frac{x : \sigma \in \Gamma}{(Q)\ \Gamma \vdash x : \sigma}$$

Fun

$$\frac{(Q)\ \Gamma, x : \tau \vdash a : \tau'}{(Q)\ \Gamma \vdash \lambda(x)\ a : \tau \rightarrow \tau'}$$

App

$$\frac{(Q)\ \Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \qquad (Q)\ \Gamma \vdash a_2 : \tau_2}{(Q)\ \Gamma \vdash a_1\ a_2 : \tau_1}$$

Inst

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \leqslant \sigma'}{(Q)\ \Gamma \vdash a : \sigma'}$$

Gen

$$\frac{(Q, q)\ \Gamma \vdash a : \sigma \qquad \textit{dom}\,(q) \notin \text{ftv}(\Gamma)}{(Q)\ \Gamma \vdash a : \forall\, q \cdot \sigma}$$

Let

$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \Gamma, x : \sigma \vdash a' : \sigma'}{(Q)\ \Gamma \vdash \text{let}\ x = a\ \text{in}\ a' : \sigma'}$$

Implicit MLF

Types

$$\tau ::= \quad \alpha \mid \tau \rightarrow \tau \mid \forall\,(\alpha)\ \tau$$

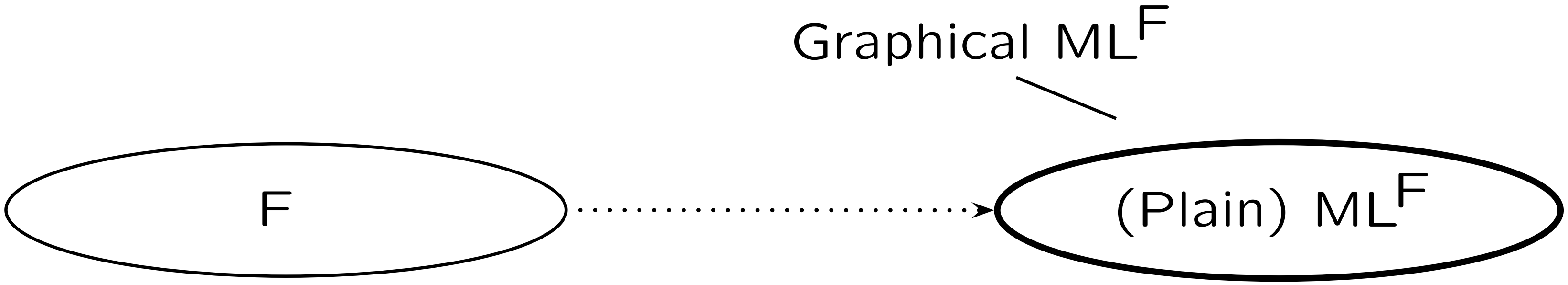
$$\sigma ::= \quad \tau \mid \forall\,(q)\ \tau \mid \bot$$

$$q ::= \quad (\alpha \geq \sigma)$$

Instance relation $\leqslant$

$\sqsubseteq$ (simpler version)

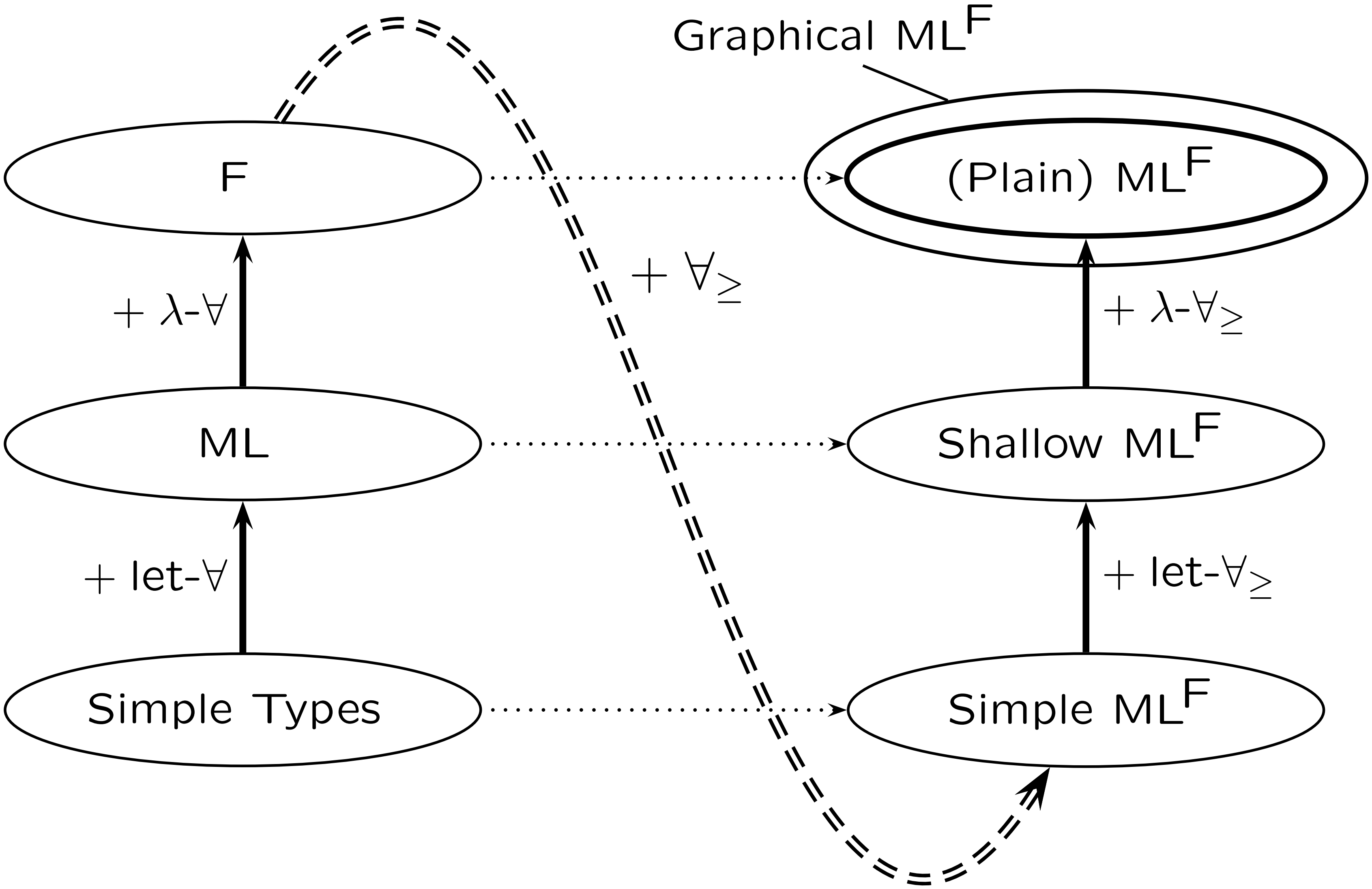
A family of languages



Standard $\forall\,\alpha\cdot$

Flexible $\forall\,(\alpha \geq \sigma)$

A family of languages


Graphical ML^{F}
F
(Plain) ML^{F}
$+\ \forall_{\geq}$
$+\ \lambda\text{-}\forall$
$+\ \lambda\text{-}\forall_{\geq}$
ML
Shallow ML^{F}
$+$ let-$\forall$
$+$ let-$\forall_{\geq}$
Simple Types
Simple ML^{F}

Standard $\forall\,\alpha\cdot$

Flexible $\forall\,(\alpha \geq \sigma)$

Syntactic type instance relation

A lot of administrative rules (See?)

- Hides the underlying principles
- Heavy proofs (i epth)
-

$$\begin{aligned}\forall\,(\alpha \geq \sigma)\;\tau \equiv\quad & \forall\,(\beta = s)\;\forall\,(\alpha \geq \forall\,(\gamma = \sigma)\;\gamma)\;\tau\\ \sqsubseteq\quad & \forall\,(\beta = s)\;\forall\,(\alpha \geq \forall\,(\gamma = \beta)\;\gamma)\;\tau\\ \equiv\quad & \forall\,(\beta = s)\;\forall\,(\alpha \geq \beta)\;\tau\\ \equiv\quad & \forall\,(\alpha = s)\;\tau\end{aligned}$$

i.e. the m

No!: An improvement was suggested by F. Pottier, but it technically collapses the syntactic instance relation via dark corners, to our surprise...

Syntactic type instance relation

A lot of administrative rules (See?)

- Hides the underlying principles
- Heavy proofs (in breadth more than in depth).
- Made extensions difficult.

Do we have the definition right?

i.e. the instance relation the best within the framework?

No!: An improvement was suggested by F. Pottier, but
it technically collapses the syntactic instance relation via dark corners,
to our surprise...

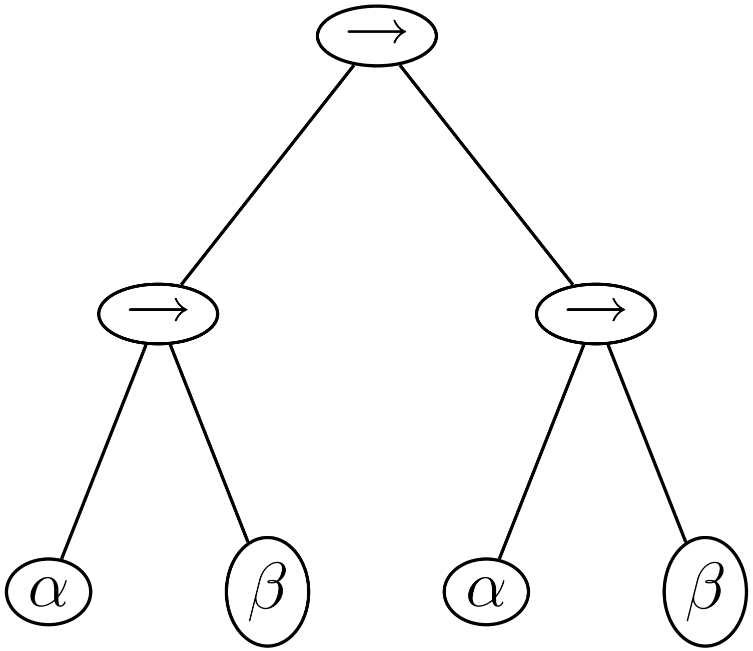
Efficiency

Expensive unification (and type inference) algorithms.

Does it scale up to large, automatically generated, programs?

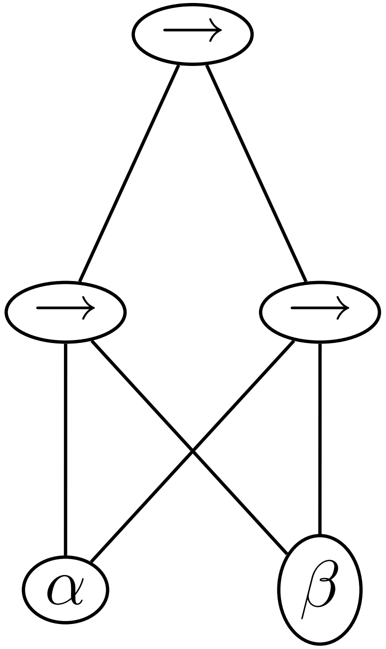
First-order terms and unification

A tree



First-order terms and unification

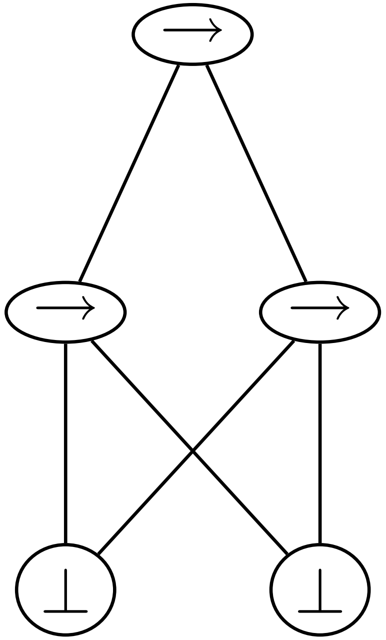
A ~~tree~~ dag



All occurrences of a variables are shared.

First-order terms and unification

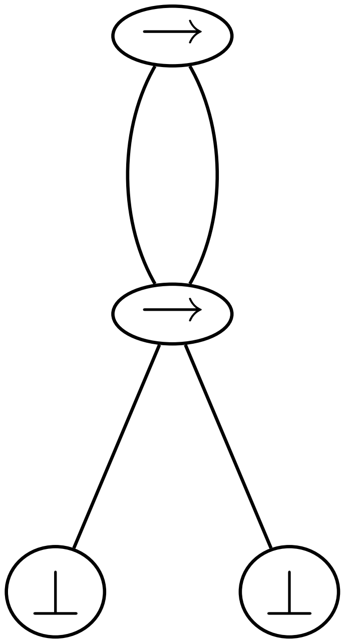
A ~~tree~~ dag



Variables need not be represented.

First-order terms and unification

A ~~tree~~ dag

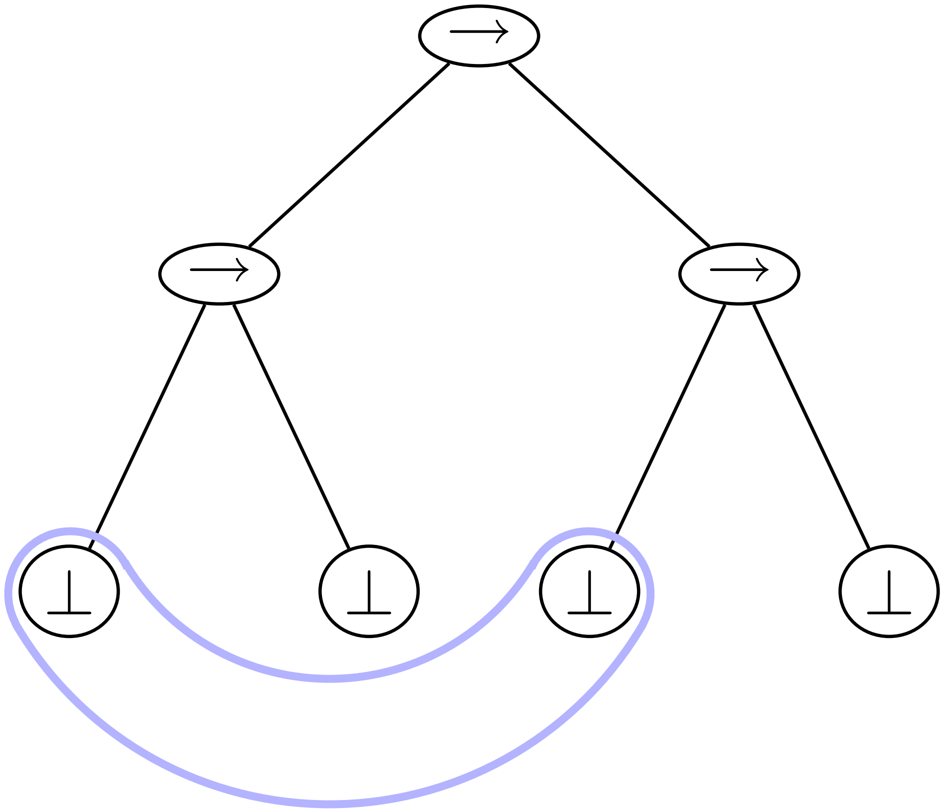


Other nodes may be also shared.

First-order terms and unification

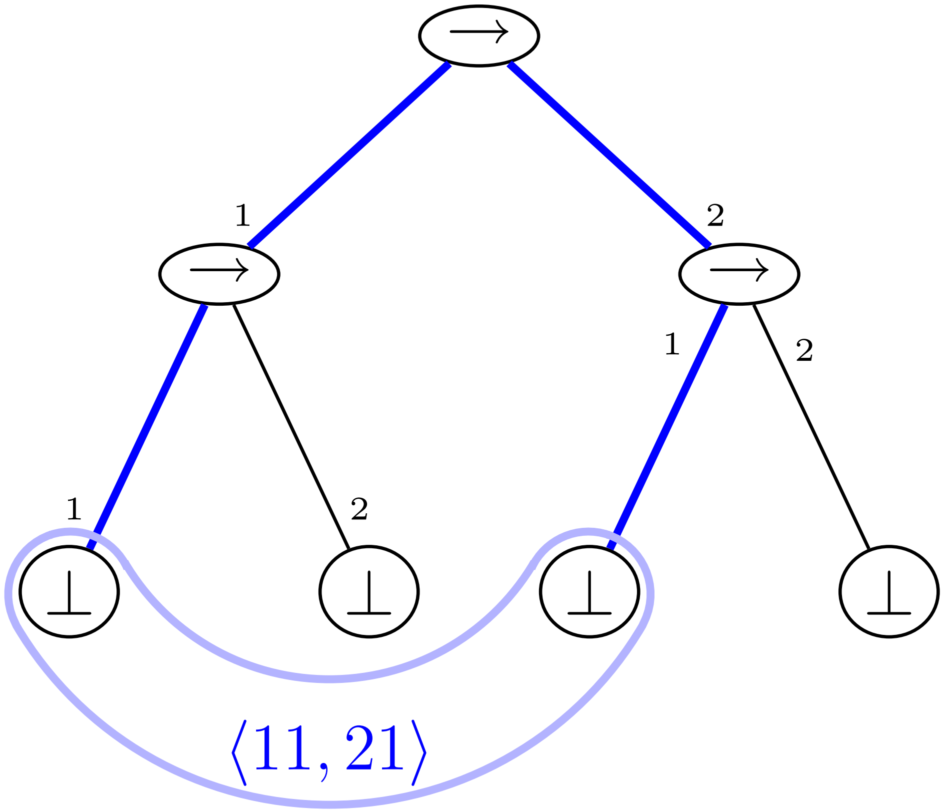
A dag τ is the superposition of

a tree $\hat{\tau}$ and an equivalence $\tilde{\tau}$ on nodes of τ



First-order terms and unification

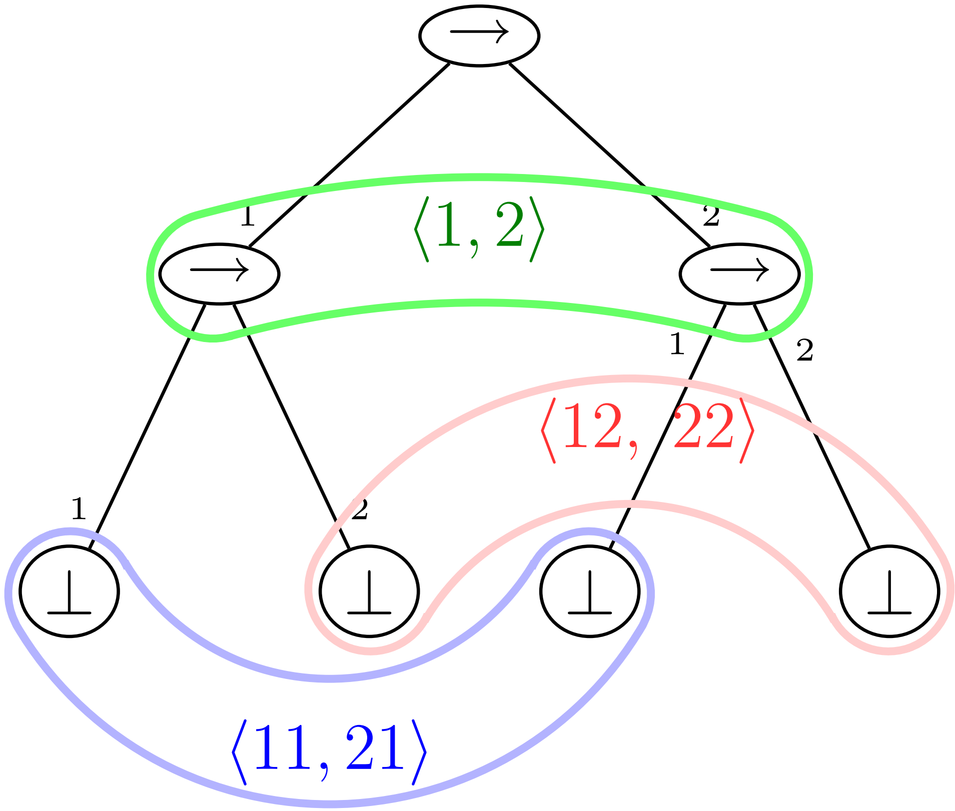
A dag τ is the superposition of
a tree $\hat{\tau}$ and an equivalence $\tilde{\tau}$ on nodes of τ



Nodes may be named after the set of paths leading to them.

A dag τ is the superposition of

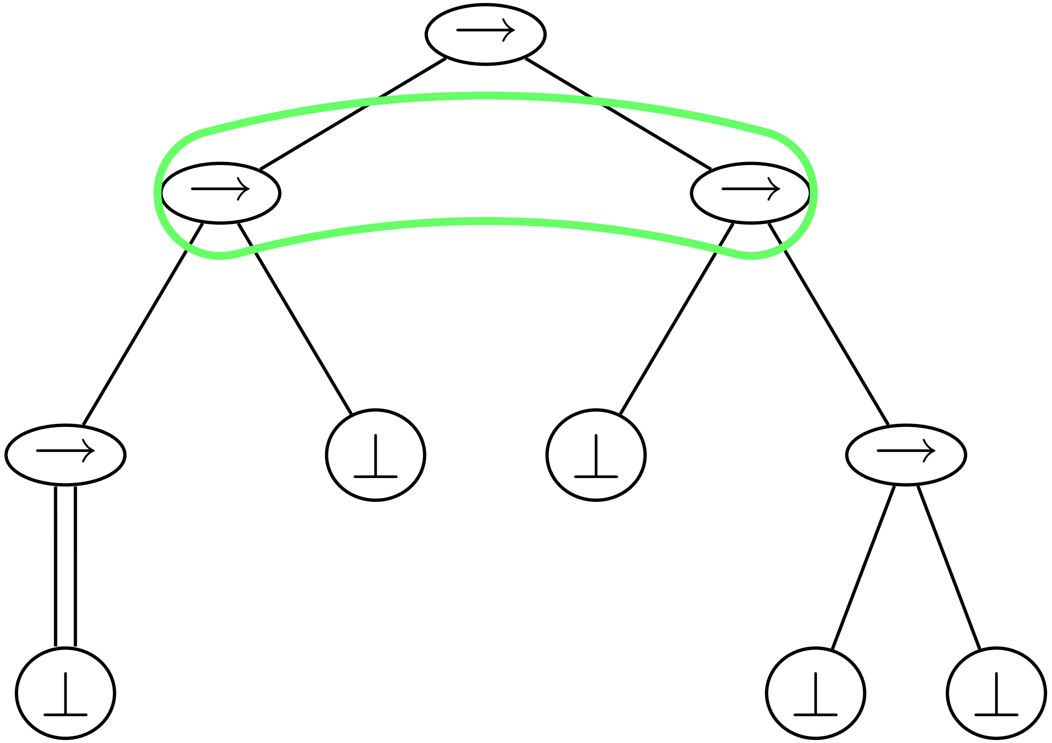
a tree $\hat{\tau}$ and an equivalence $\tilde{\tau}$ on nodes of τ



name of merged nodes = union of merged names.

First-order terms and unification

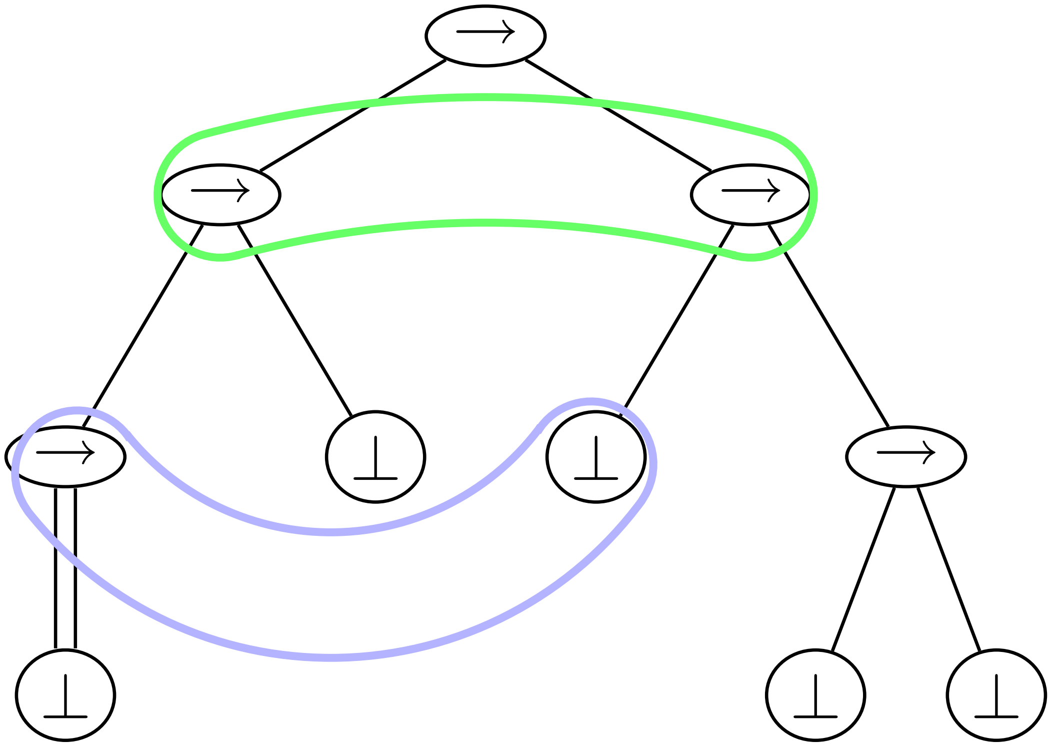
Unification computes the smallest equivalence that is congruent and consistent



congruent: successors of merged nodes are merged

First-order terms and unification

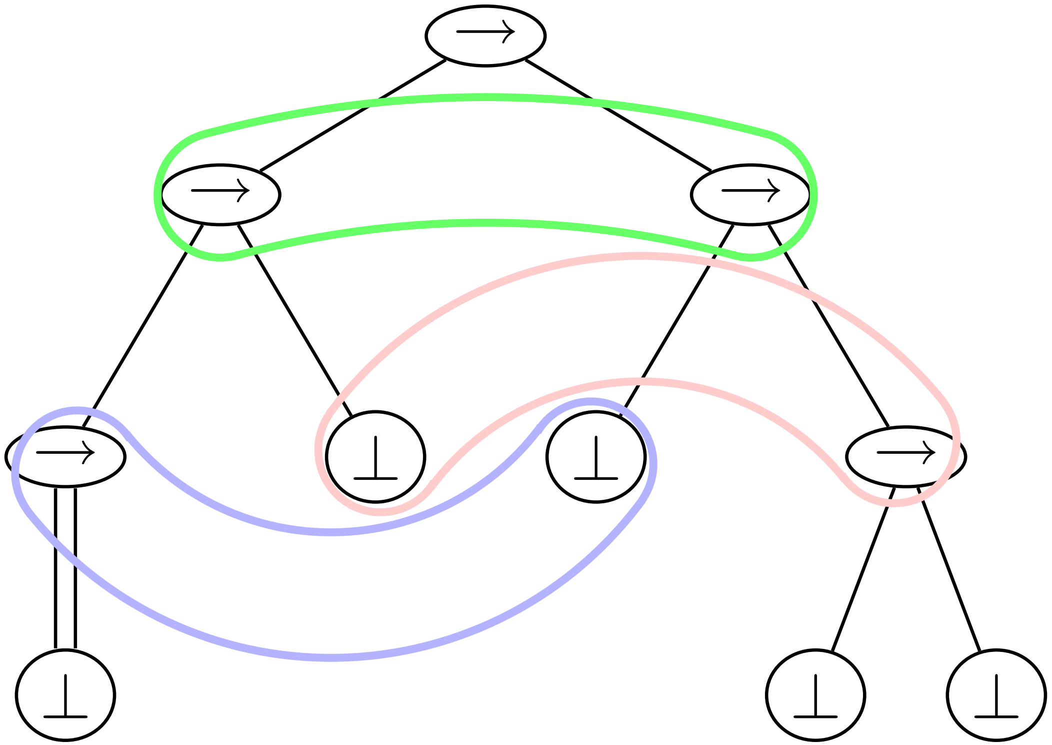
Unification computes the smallest equivalence that is congruent and consistent



consistent : no symbol class, but ⊥ is a pseudo-symbol that never clashes

First-order terms and unification

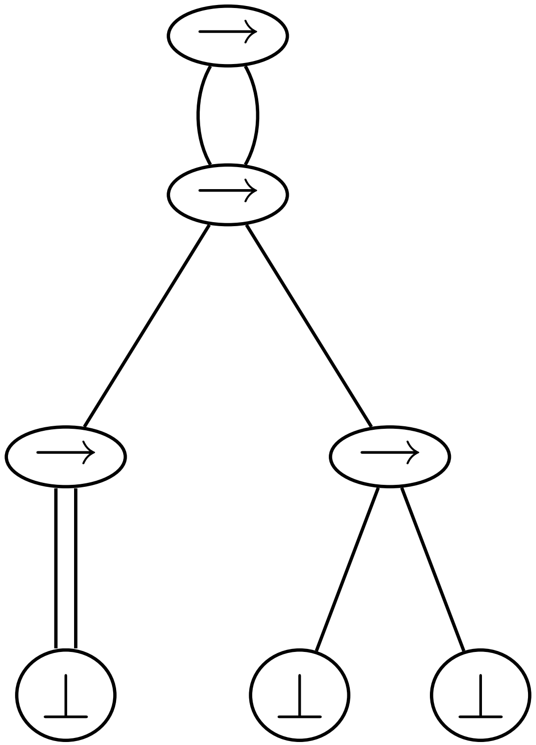
Unification computes the smallest equivalence that is congruent and consistent



consistent : no symbol class, but ⊥ is a pseudo-symbol that never clashes

First-order terms and unification

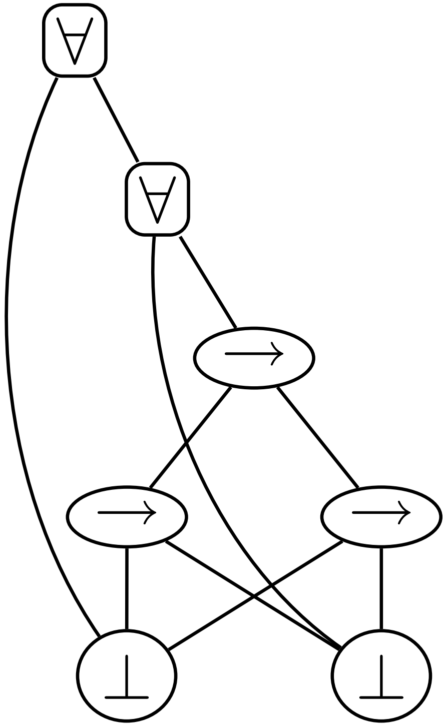
Unification computes the smallest equivalence that is congruent and consistent



Drawn as a graph.

Representing binders

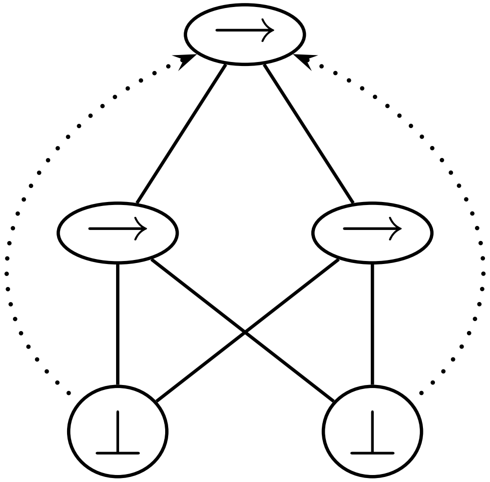
Explicitly with forward pointers (as usual)



Problem: binders do not commute and cannot be removed implicitly.

Representing binders

Implicitly with backward pointers (bindings edges)

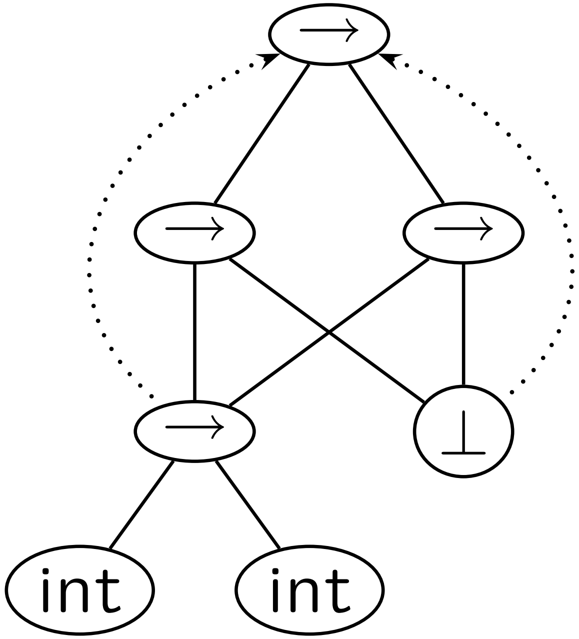


$$\forall\ (\beta \geq \bot, \gamma \geq \bot)\ \boxed{\beta \to \gamma} \to \boxed{\beta \to \gamma}$$

Binding edges point to the node where they (as variables) would have been introduced.

Commutation of binders come for free!

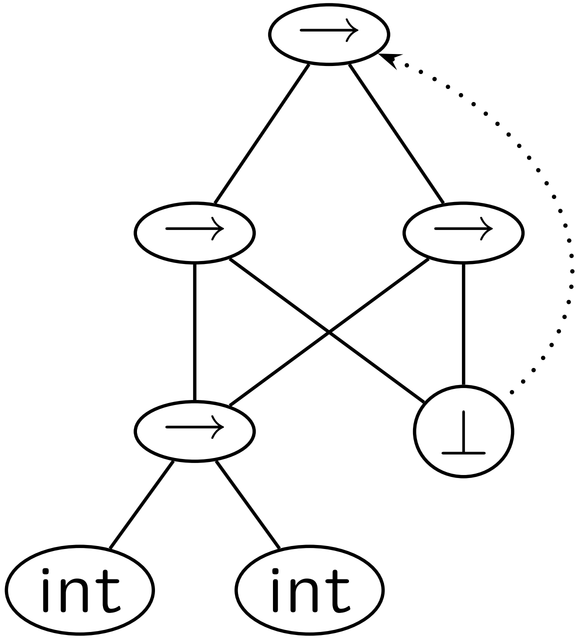
Representing binders



$$\forall\ (\beta = \mathsf{int} \to \mathsf{int}, \gamma \geq \bot)\ \boxed{\beta \to \gamma} \to \boxed{\beta \to \gamma}$$

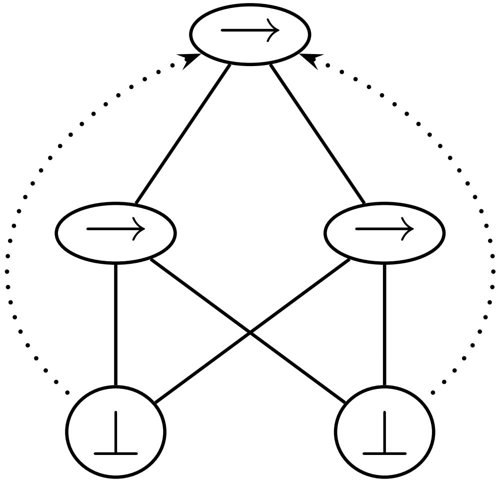
Useless binders may be removed (GC).

Representing binders



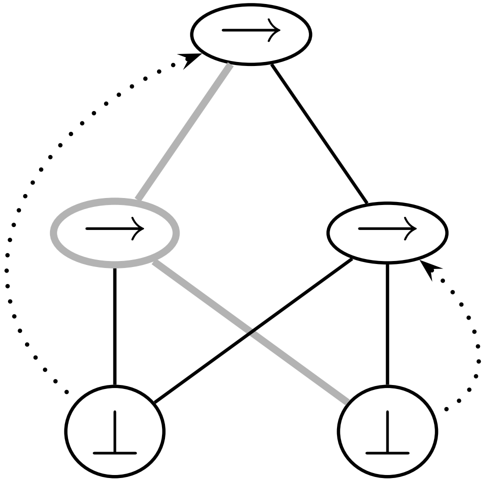
$$\forall\ (\beta = \mathsf{int} \to \mathsf{int}, \gamma \geq \bot)\ \boxed{\beta \to \gamma} \to \boxed{\beta \to \gamma}$$

Representing binders



$$\forall\ (\beta \geq \bot, \gamma \geq \bot)\ \boxed{\beta \rightarrow \gamma} \rightarrow \boxed{\beta \rightarrow \gamma}$$

Well-formed conditions (1)

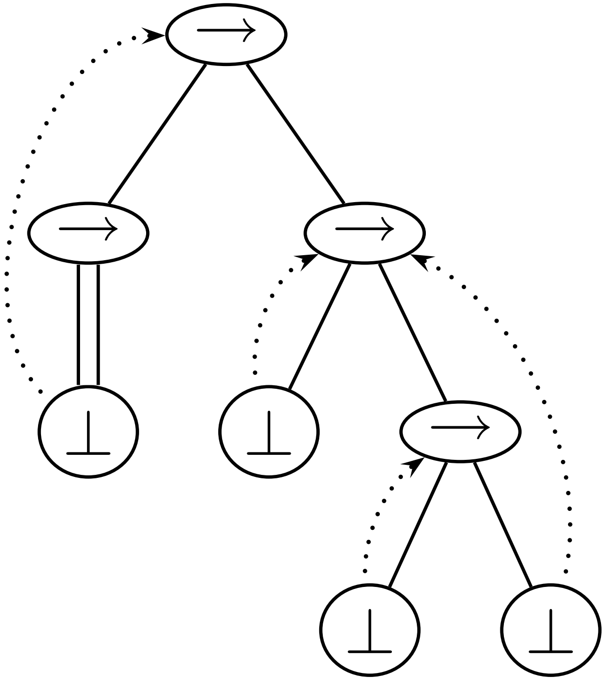


$$\forall\ (\beta \geq \bot)\ \boxed{\beta \rightarrow \gamma} \rightarrow \boxed{\forall\ (\gamma \geq \bot)\ \beta \rightarrow \gamma}$$

(1) The binding of a node must be one of its dominators.

Representing binders

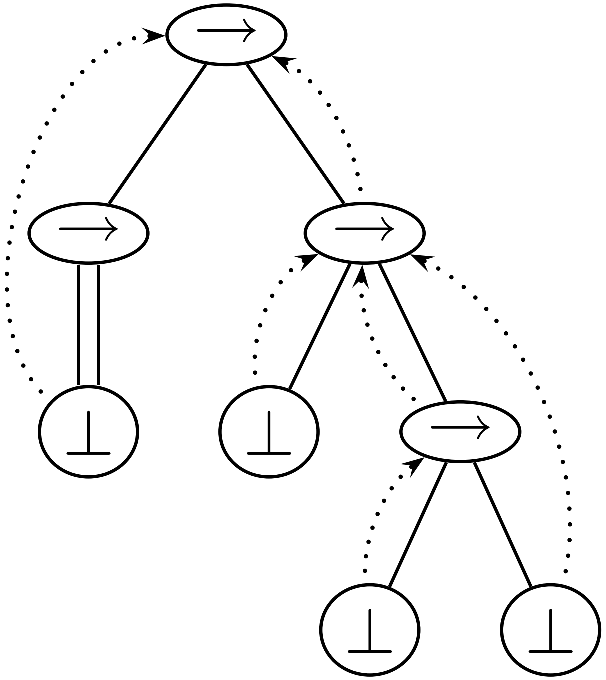
Well-formed conditions (2)



$$\forall\ (\beta_1 \geq \bot)\ \boxed{\beta_1 \to \beta_1} \to \boxed{\forall\ (\beta_2 \geq \bot, \beta_3 \geq \bot)\ \boxed{\forall\ (\beta_4)\ \ \beta_4 \to \beta_3} \to \beta_2}$$

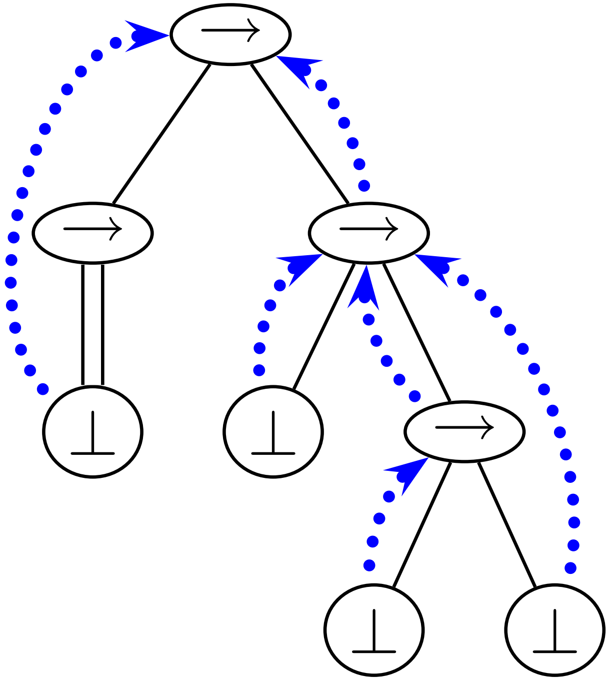
(2) Binding paths are upward closed.

Well-formed conditions (2)



$$\forall \left(\beta_1 \geq \bot, \alpha_1 = \boxed{\forall \left(\beta_2 \geq \bot, \beta_3 \geq \bot, \alpha_2 = \boxed{\forall\ (\beta_4)\ \ \beta_4 \to \beta_3}\right)\ \ \beta_2 \to \alpha_2}\right)\ \ \boxed{\beta_1 \to \beta_1} \to \alpha_1$$

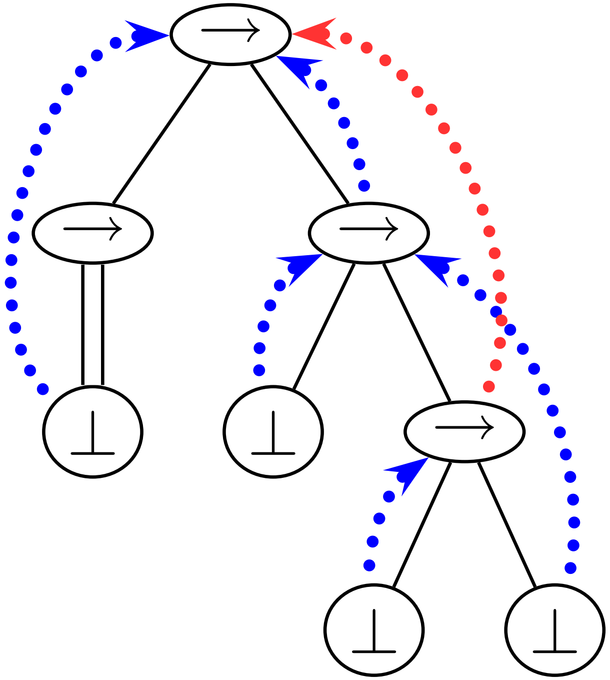
Well-formed conditions (2)



$$\forall\left(\beta_1 \geq \bot, \alpha_1 = \boxed{\forall\left(\beta_2 \geq \bot, \beta_3 \geq \bot, \alpha_2 = \boxed{\forall\ (\beta_4)\ \ \beta_4 \to \beta_3}\right)\ \ \beta_2 \to \alpha_2}\right)\ \boxed{\beta_1 \to \beta_1} \to \alpha_1$$

(2) Inverse binding edges form a tree (with the same root)

Well-formed conditions (3)

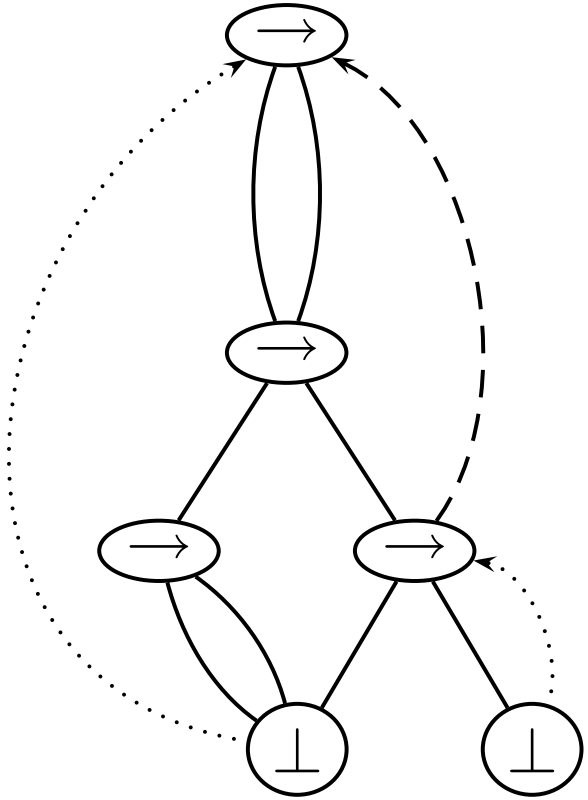


$$\forall\left(\beta_1 \geq \bot, \alpha_2 = \boxed{\forall\ (\beta_4)\ \ \beta_4 \to \beta_3}, \alpha_1 = \boxed{\forall\ (\beta_2 \geq \bot, \beta_3 \geq \bot)\ \ \beta_2 \to \alpha_2}\right)\ \boxed{\beta_1 \to \beta_1} \to \alpha_1$$

(3) Binding edges cannot cross (to be made precise)

Graphic types

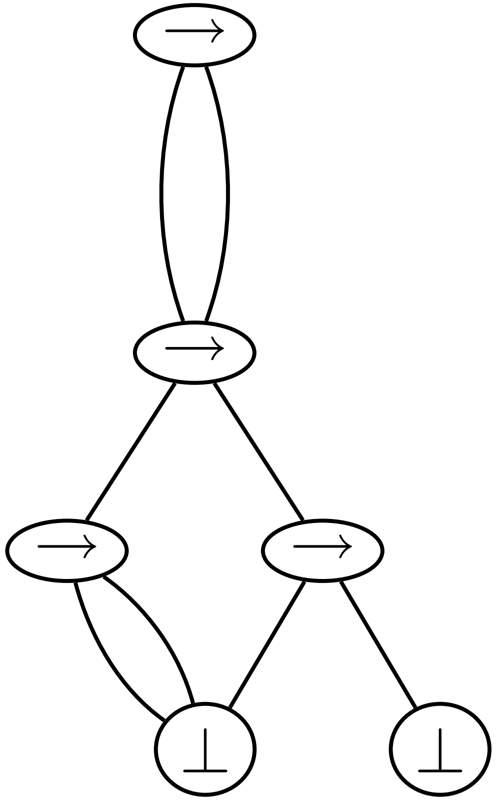
A graphic type. . .



$$\forall\,(\beta \geq \bot, \alpha = \forall\,(\gamma \geq \bot)\;\beta \to \gamma, \alpha' = (\beta \to \beta) \to \alpha)\;\alpha' \to \alpha'$$

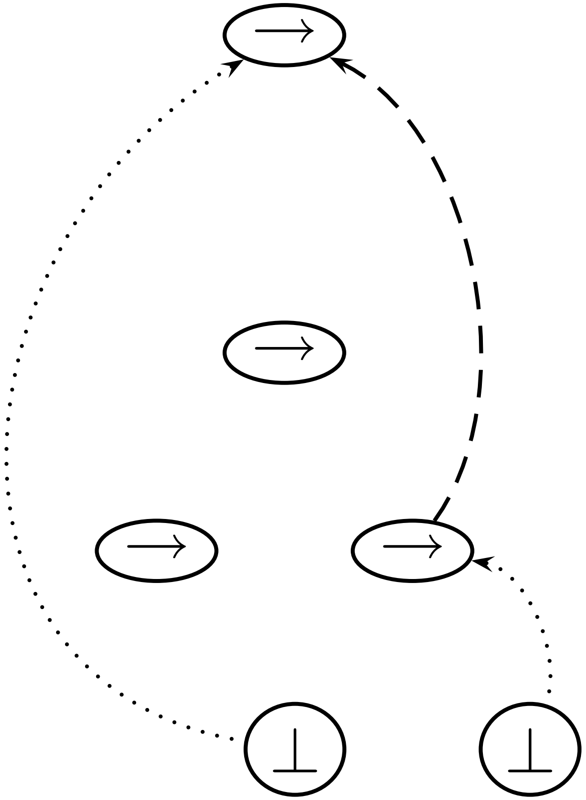
Graphic types

is a first-order term graph...



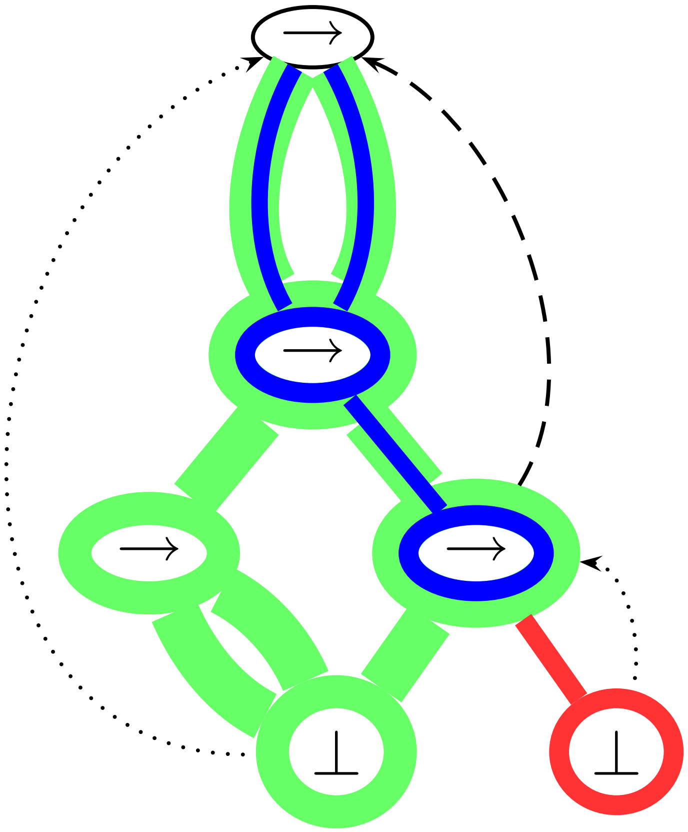
Graphic types

...plus a binding tree...



Graphic types

with relations between them.

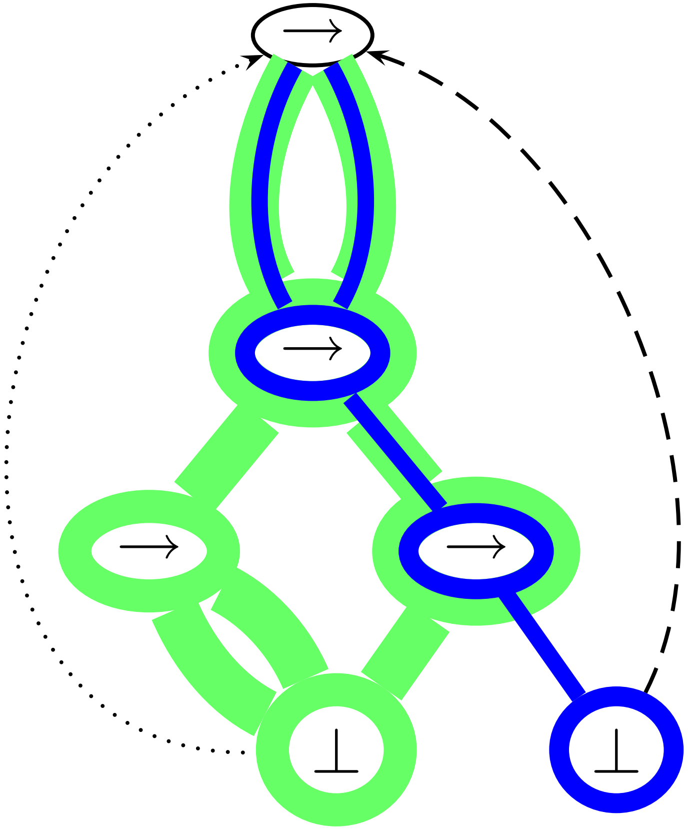


$\mathcal{B}(n) = \{m \mid n \circ\!\!\longrightarrow m \circ\!\!\longrightarrow \breve{n}\}$ where $n \succ\!\!\!\longrightarrow \breve{n}$.

If $m \in \mathcal{B}(n)$, then $\mathcal{B}(m) \subseteq \mathcal{B}(n)$

Graphic types

with relations between them.

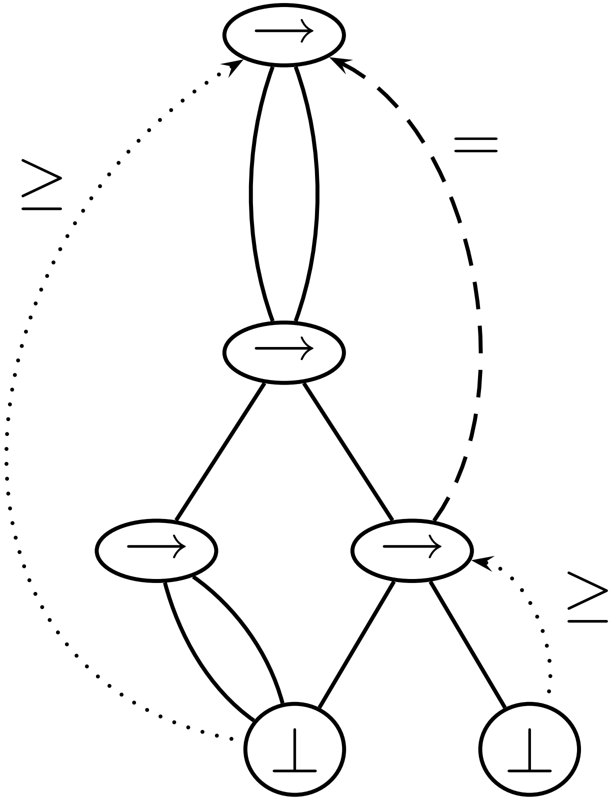


$\mathcal{B}(n) = \{m \mid n \mathrel{\circ\!\!\longrightarrow} m \mathrel{\circ\!\!\longrightarrow} \breve{n}\}$ where $n \mathrel{\succ\!\!\!\longrightarrow} \breve{n}$.

If $m \in \mathcal{B}(n)$, then $\mathcal{B}(m) \subseteq \mathcal{B}(n)$

Graphic types

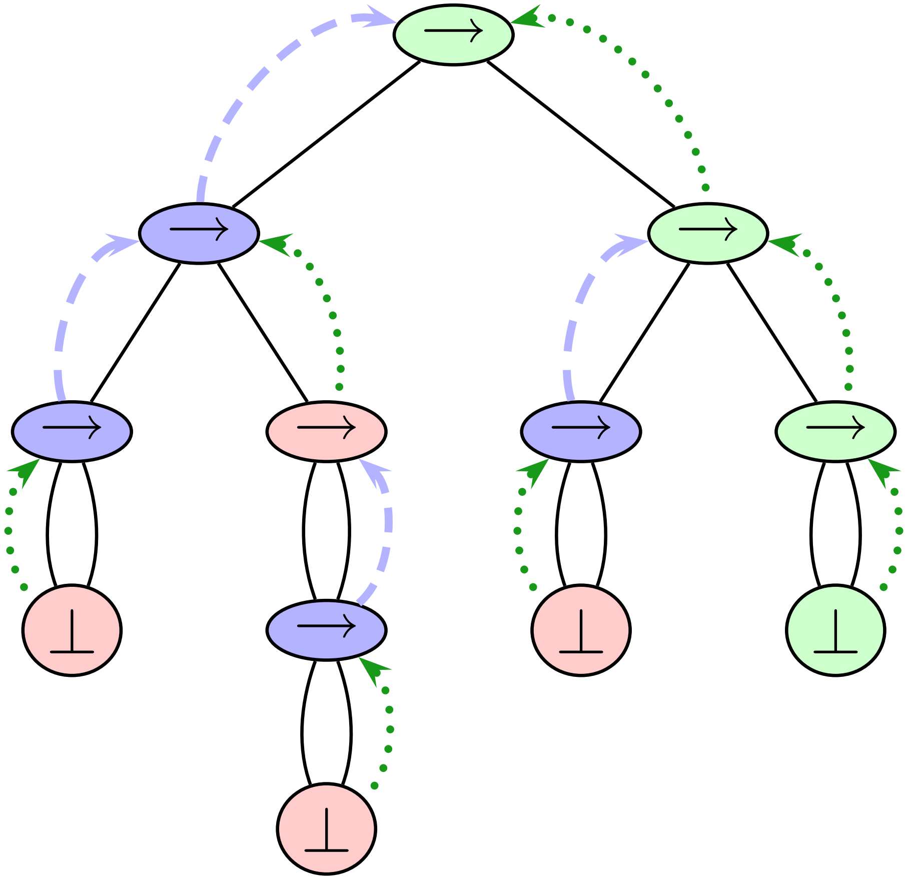
Two kinds of binding arrows



— Flexible binding (≥ flag, dotted arrows): mean instances may be taken.

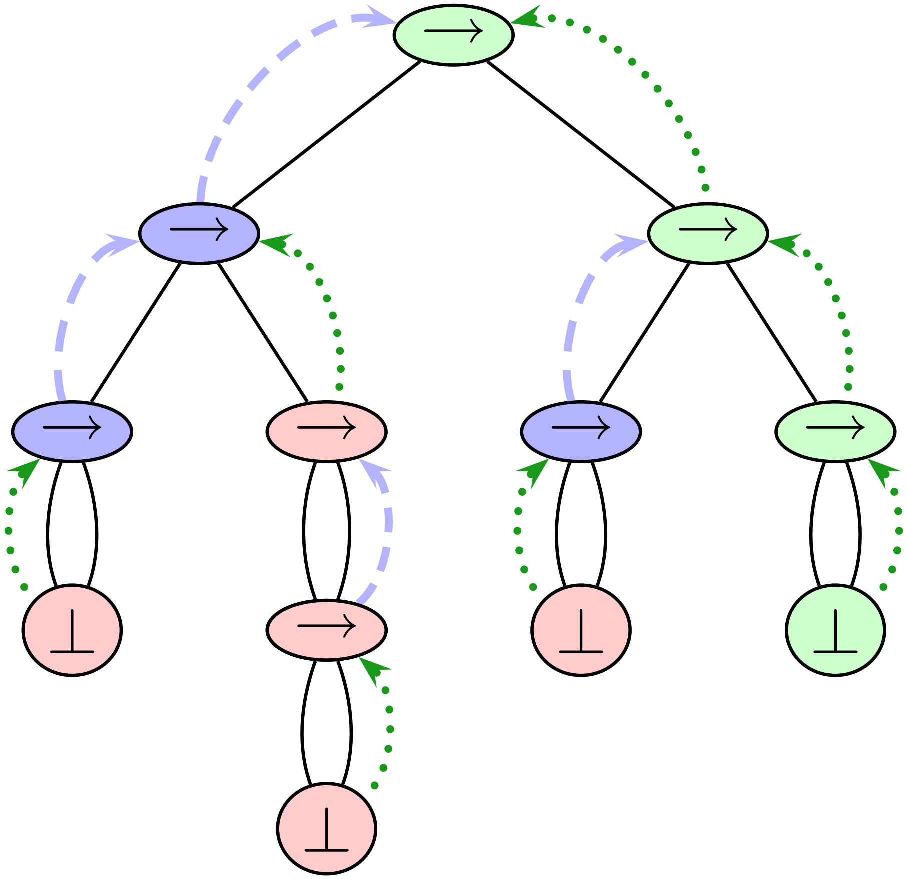
— Rigid (= flag, dashed arrows): mean no instance may not be taken.

Graphic types: flags and permissions



Binding path	Permissions
$\geq^*$	$\{\geq, =\}$
$=(\geq\|=)^*$	$\{=\}$
Others	$\{\}$

Graphic types: flags and permissions

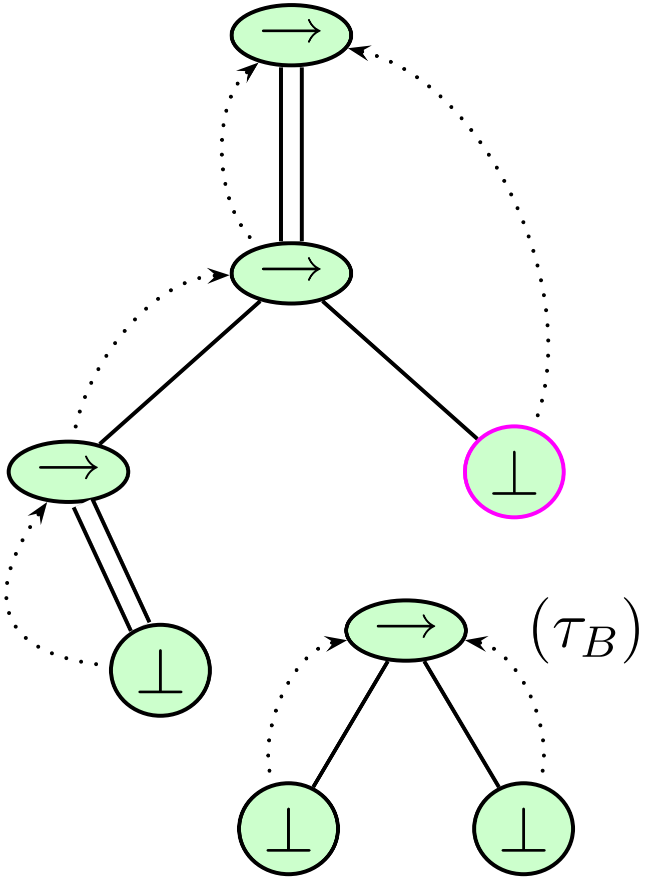


Binding path	Permissions
$\geq^*$	$\{\geq, =\}$
$=(\geq\|=)^*$	$\{=\}$
Others	$\{\}$

$=^+\geq^*$

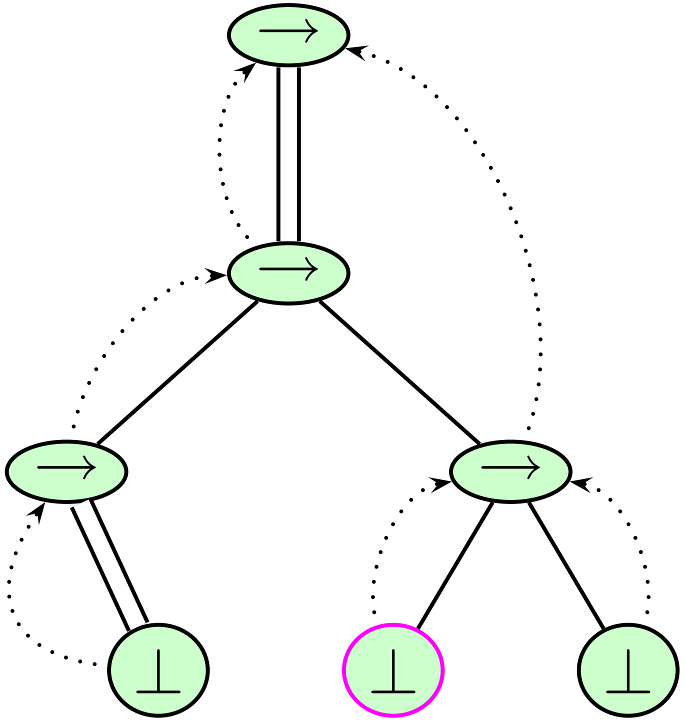
Instance relation

Grafting



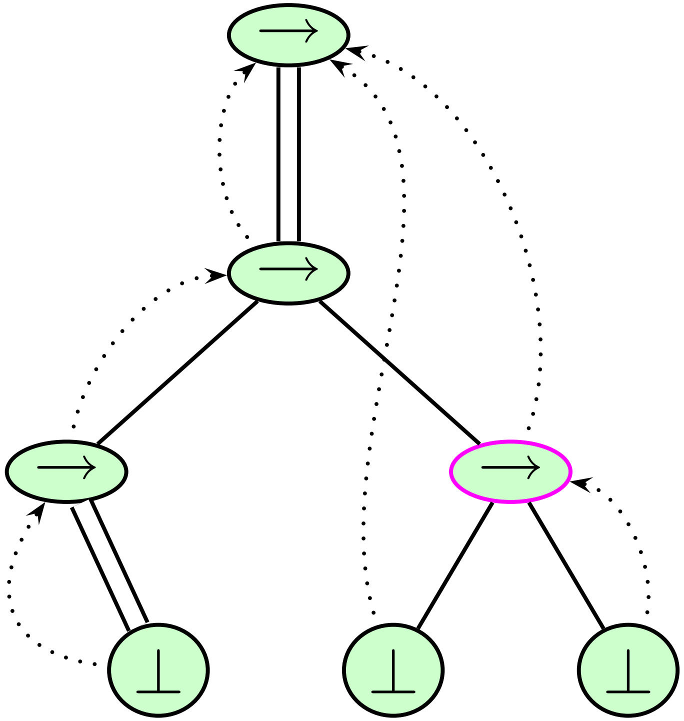
Instance relation

Raising



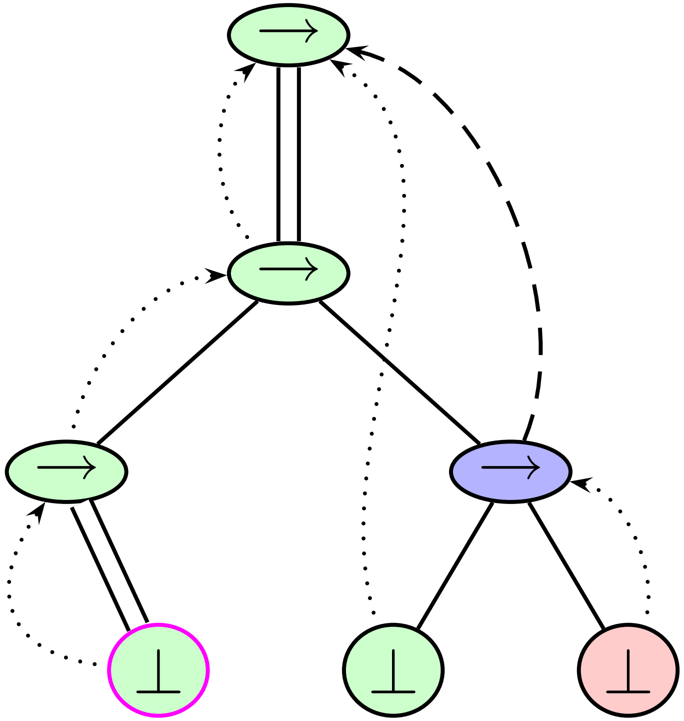
Instance relation

Weakening



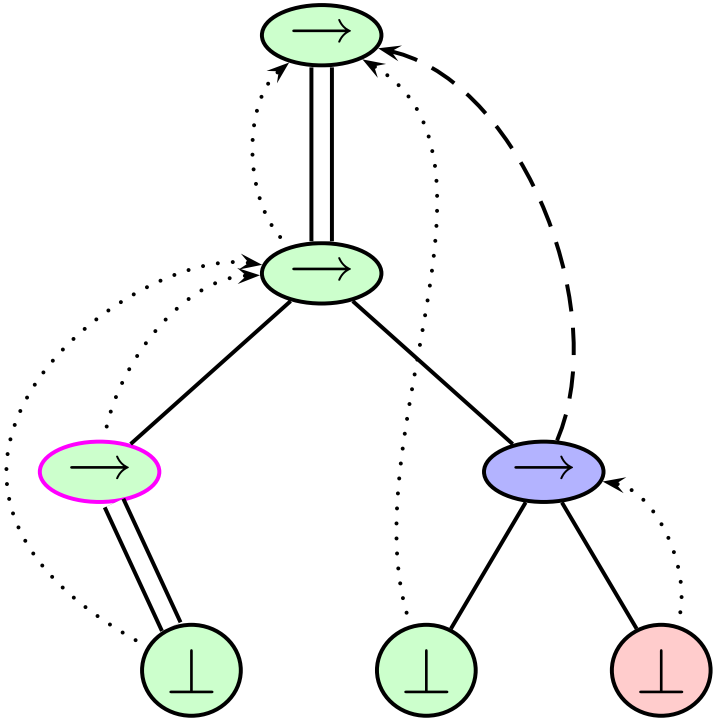
Instance relation

Raising



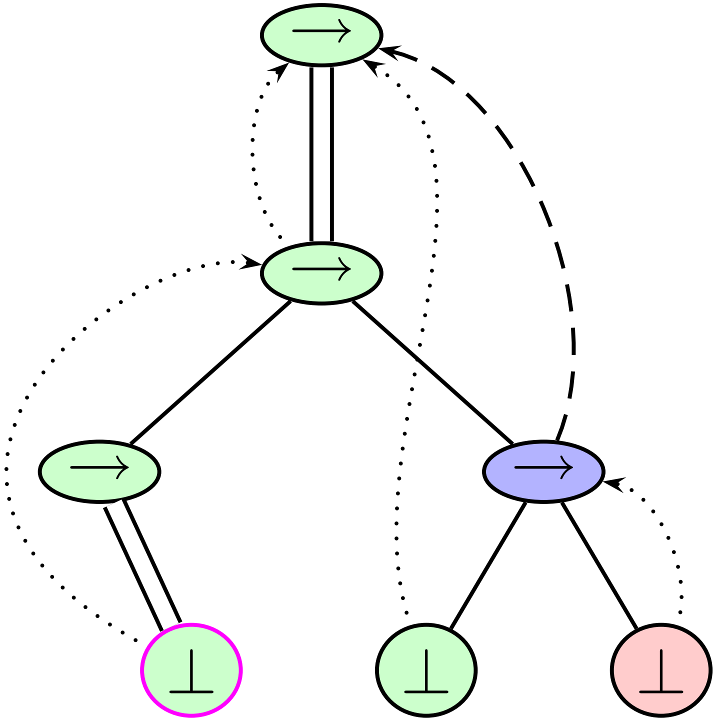
Instance relation

Deletion (implicit)

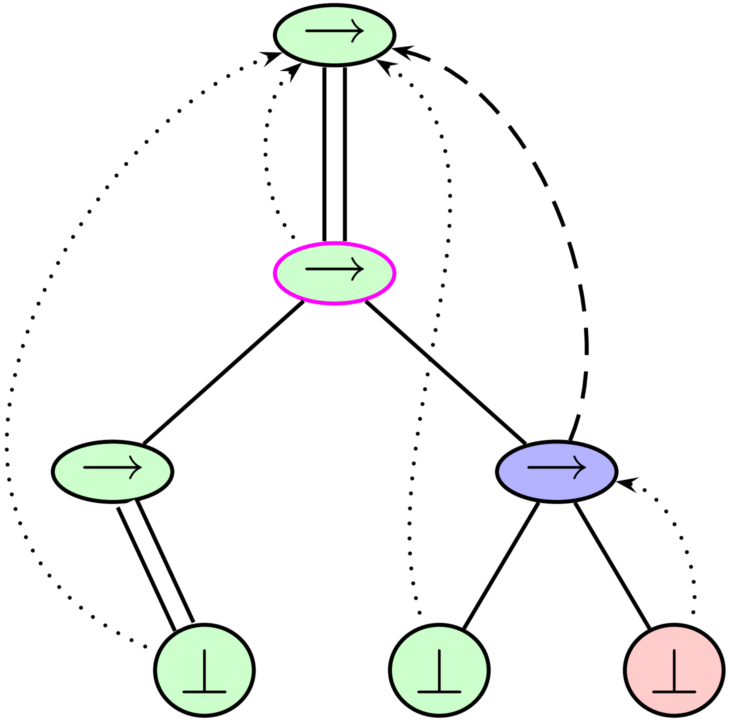


Instance relation

Raising

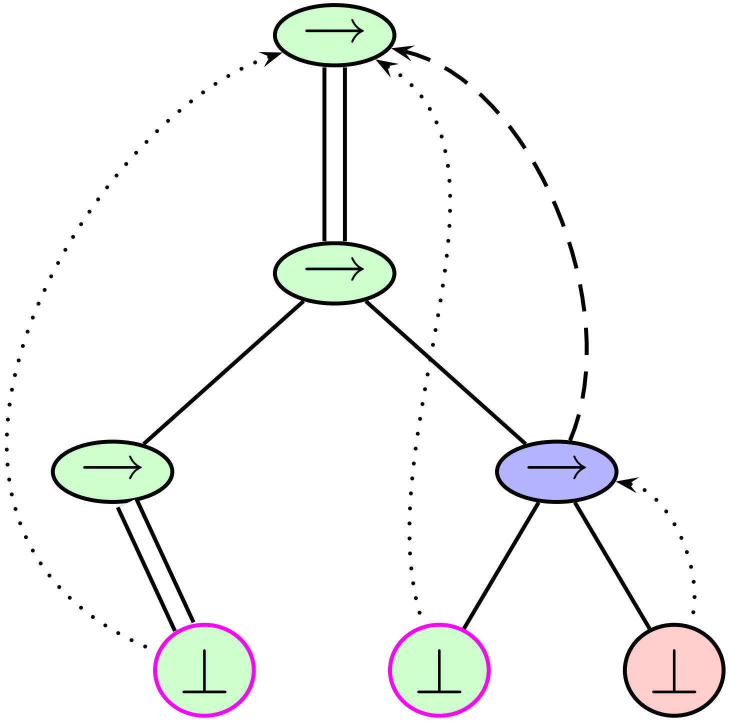


Deletion (implicit)

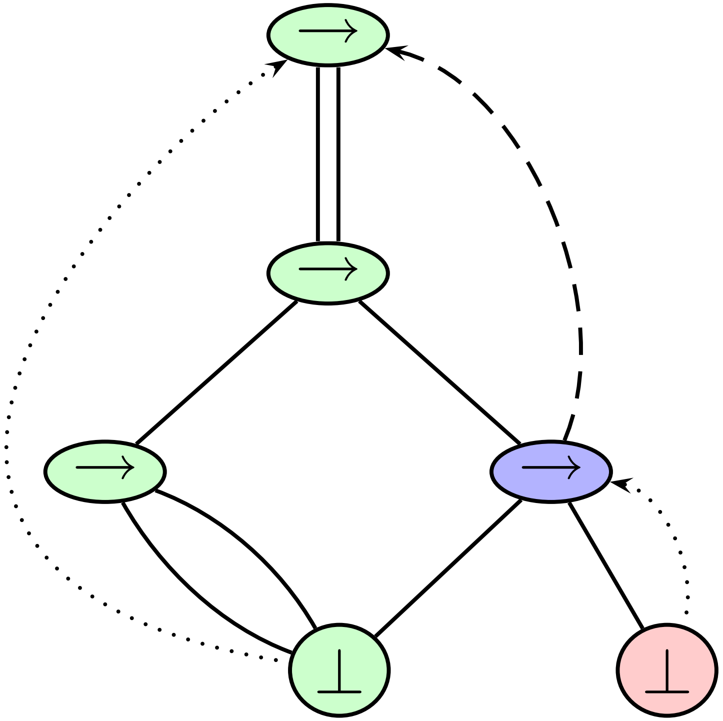


Instance relation

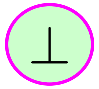
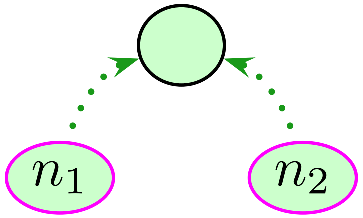
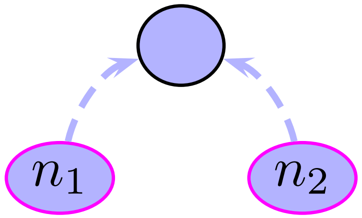
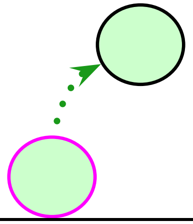
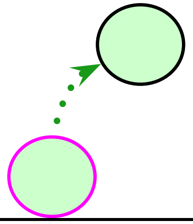
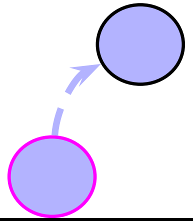
Merging



Instance relation



Instance relation $\leqslant$

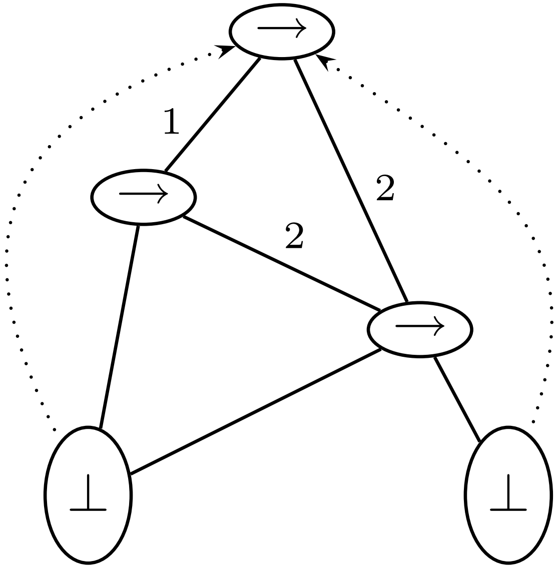
Operation	Relation	Conditions
Graft(τ'', n)	$\leqslant^G$	
Merge(n_1, n_2)	$\leqslant^M$	 or 
Weaken(n)	$\leqslant^W$	
Raise(n)	$\leqslant^R$	 or 

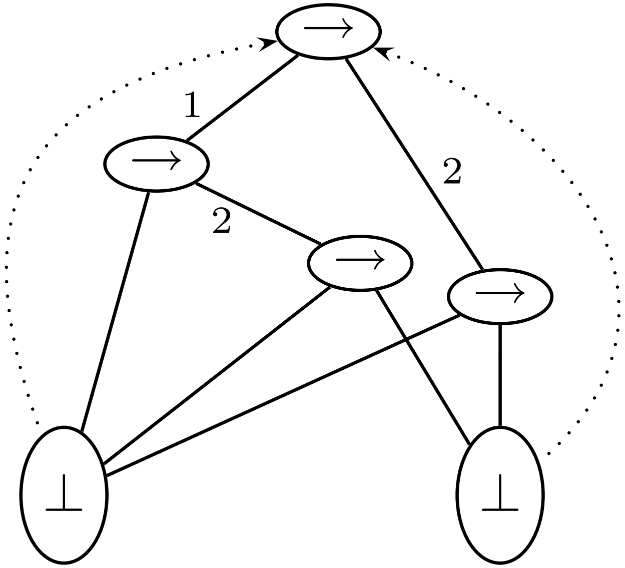
$$\leqslant \quad \triangleq \quad (\leqslant^G \cup \leqslant^M \cup \leqslant^W \cup \leqslant^R)^*$$

Similarity

$\leqslant^m$ is the subrelation of $\leqslant^M$ that merges monomorphic nodes.

Similarity is the relation $\approx$ is $(\leqslant^m \cup \geqslant^m)^*$.





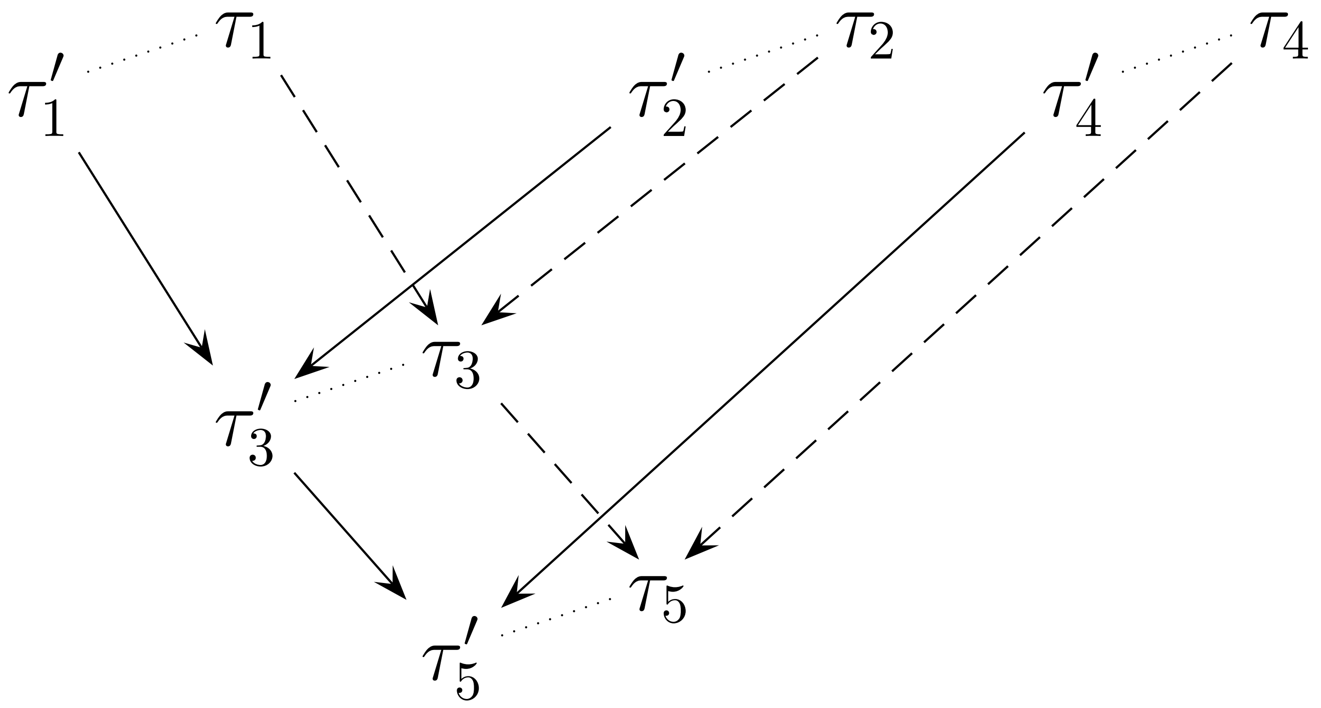
We are interested in instance modulo similarity $\leqslant_\approx$, which is $(\leqslant \cup \approx)^*$.

We compute instance up to deletion, but not up to similarity. . .

Commutation lemmas

Similarity is equal to $\leqslant^m ; \geqslant^m$.

Instance modulo similarity $\leqslant_\approx$ is equal to $\leqslant ; \geqslant^m$ are equal. Hence:



Instance is equal to $(\leqslant^G ; \leqslant^R ; \leqslant^{MW})$, where $\leqslant^{MW}$ is $(\leqslant^M \cup \leqslant^W)^\star$.

Unification algorithm

Definition A type τ' unifies nodes N of τ if τ' is an instance of τ and all nodes in N are merged in τ'.

Moreover τ' is a principal unifier is all other unifiers are an instance of τ'.

The algorithm proceeds in three steps:

1) Computes $\tilde{\tau_u}$ by performing first-order unification on the term-graph to merge all nodes of N.

2) Compute the binding tree $\overset{\succ}{\tau}_u$: Given a node n of $\tilde{\tau_u}$, let $n_1, ..., n_k$ be the nodes of τ that are merged into n. The binding edges of $n_1, ..., n_k$ are raised until they are all bound at the same level. The flag for n is the best flag common to $n_1, \ldots, n_k$.

3) Check permissions for all merges of $\tilde{\tau_u}$ that are still polymorphic in $\overset{\succ}{\tau}_u$.

Unification algorithm

Definition A type τ' unifies nodes N of τ if τ' is an instance of τ and all nodes in N are merged in τ'.

Moreover τ' is a principal unifier is all other unifiers are an instance of τ'.

The algorithm proceeds in three steps:

1) Computes $\tilde{\tau}_u$ by performing first-order unification on the term-graph to merge all nodes of N. Cost $O(n)$ (ou $O(n\alpha(n))$).

2) Compute the binding tree $\overset{\succ}{\tau}_u$: Given a node n of $\tilde{\tau}_u$, let $n_1, ..., n_k$ be the nodes of τ that are merged into n. The binding edges of $n_1, ..., n_k$ are raised until they are all bound at the same level. The flag for n is the best flag common to $n_1, \ldots, n_k$. Cost $O(n)$: a top down visit. The most involved part of the algorithm. Uses a linear algorithm for computing least-common ancestors.

3) Check permissions for all merges of $\tilde{\tau}_u$ that are still polymorphic in $\overset{\succ}{\tau}_u$. Cost $O(n)$, simple visit of $\tilde{\tau}_u$.

Unification algorithm

Correction τ' is a unifier of τ.

Completeness If there is a unifier of τ, this algorithm finds one.

Principality The unifier return by the algorithm is a principal one.

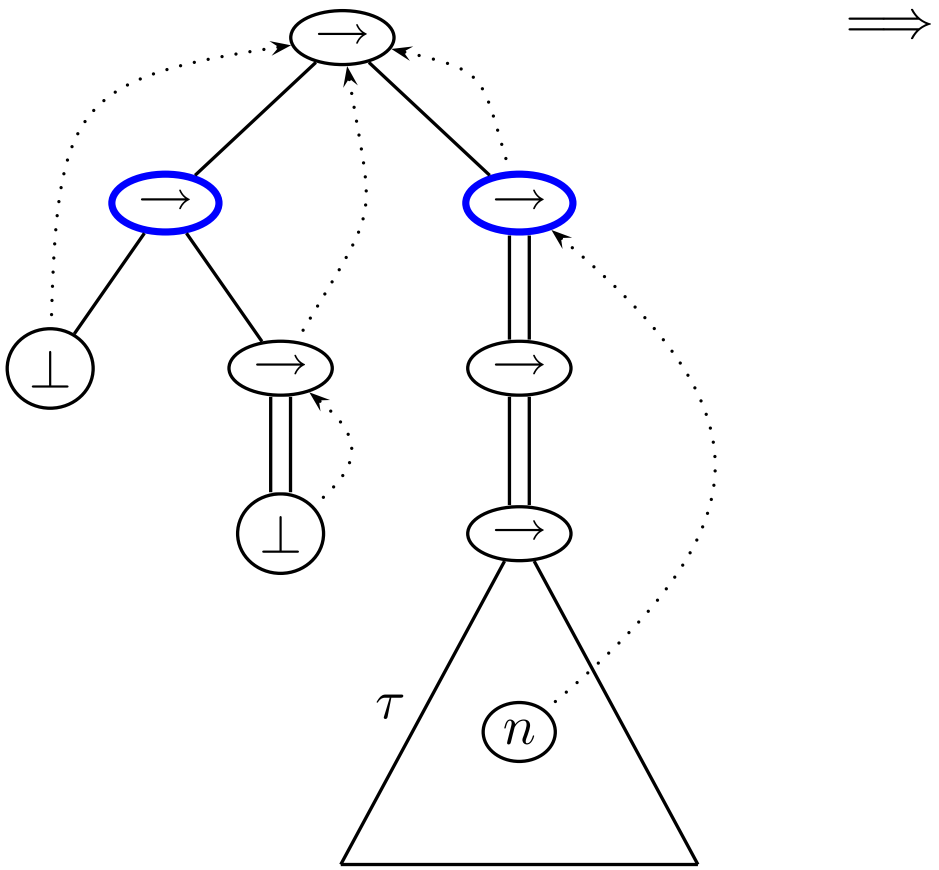
Proofs are involed. Relies a lot on commutation lemmas, but not only.

Unification algorithm

Principality

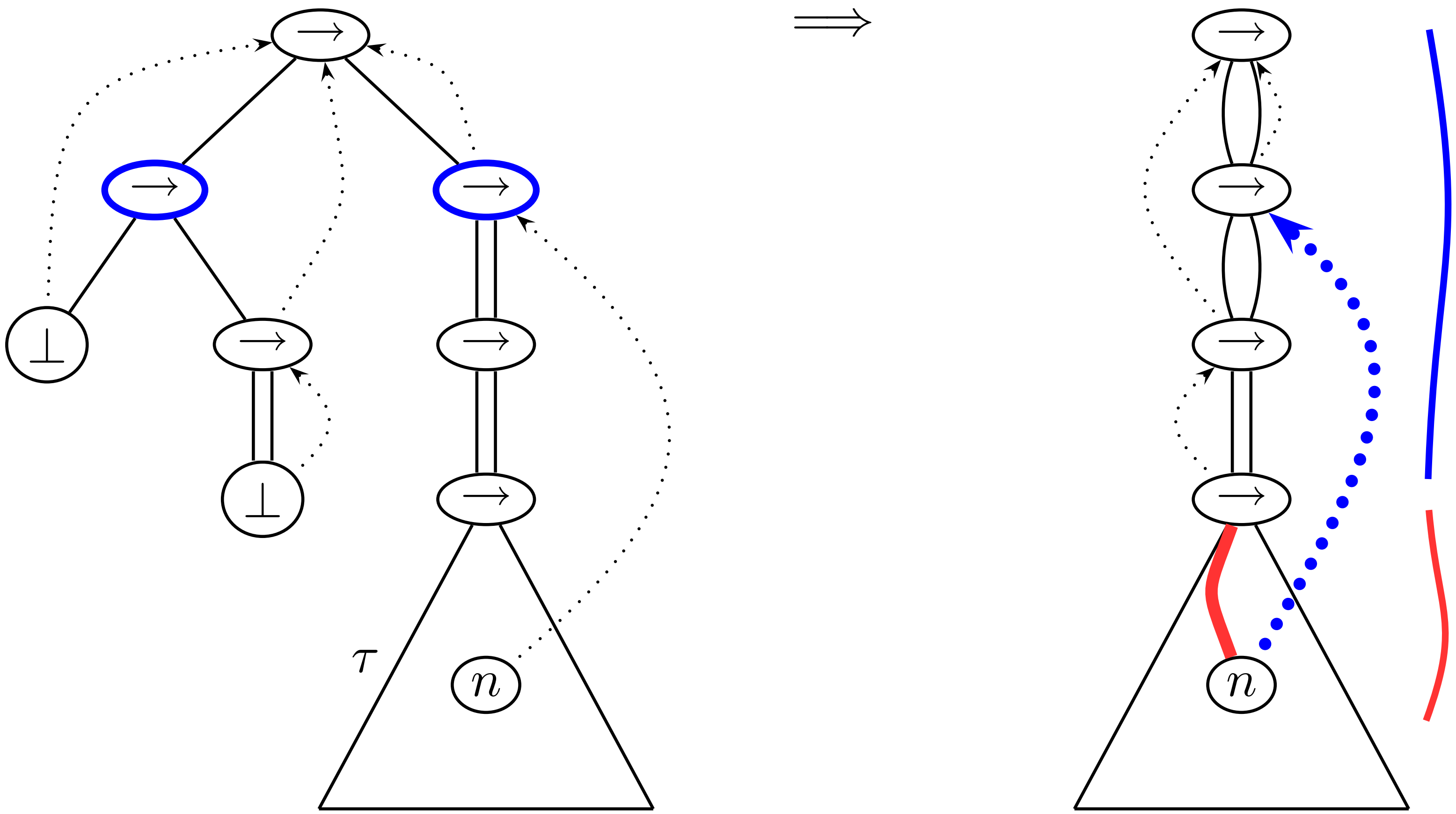
$$\begin{tikzcd}
\tau \arrow[r, "\leqslant^{G}"'] \arrow[dr, dashed, "\leqslant^{G}"'] \arrow[rrrddd, bend right, "\leqslant"'] & \tau_g \arrow[r, "\rho_1", "\leqslant^{R}"'] \arrow[d, dotted, "\leqslant^{G}"', "\gamma_1"] \arrow[drr, phantom] & \tau_r \arrow[r, "\eta_1", "\leqslant^{MW}"'] \arrow[d, dotted, "\leqslant^{G}"', "\gamma'_1"] & \tau_u \arrow[d, dotted, "\leqslant^{G}"', "\gamma''_1"] \\
 & \tau'_g \arrow[r, dotted, "\rho'_1", "\leqslant^{R}"'] \arrow[dr, dashed, "\leqslant^{R}"'] & \tau'_r \arrow[r, dotted, "\eta'_1", "\leqslant^{MW}"'] \arrow[d, dotted, "\leqslant^{R}"', "\rho_2"] & \tau'_u \arrow[d, dotted, "\leqslant^{R}"', "\rho'_2"] \\
 & & \tau''_r \arrow[r, dotted, "\eta''_1", "\leqslant^{MW}"'] \arrow[dr, dashed, "\leqslant^{MW}"'] & \tau''_u \arrow[d, dotted, "\leqslant^{MW}"', "\eta_2"] \\
 & & & \tau_v
\end{tikzcd}$$

Incrementality



Merging

Incrementality

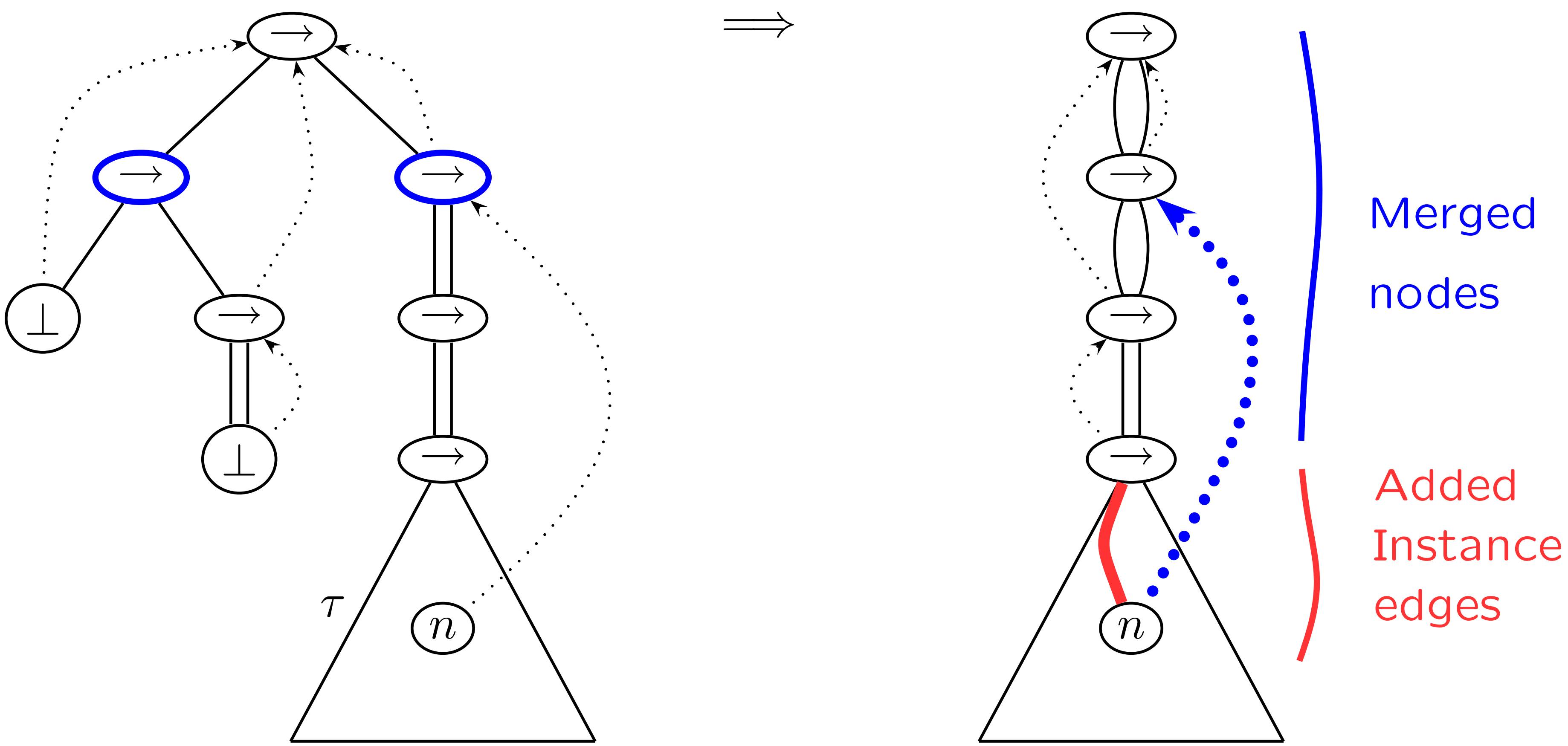


Merged nodes

Added Instance edges

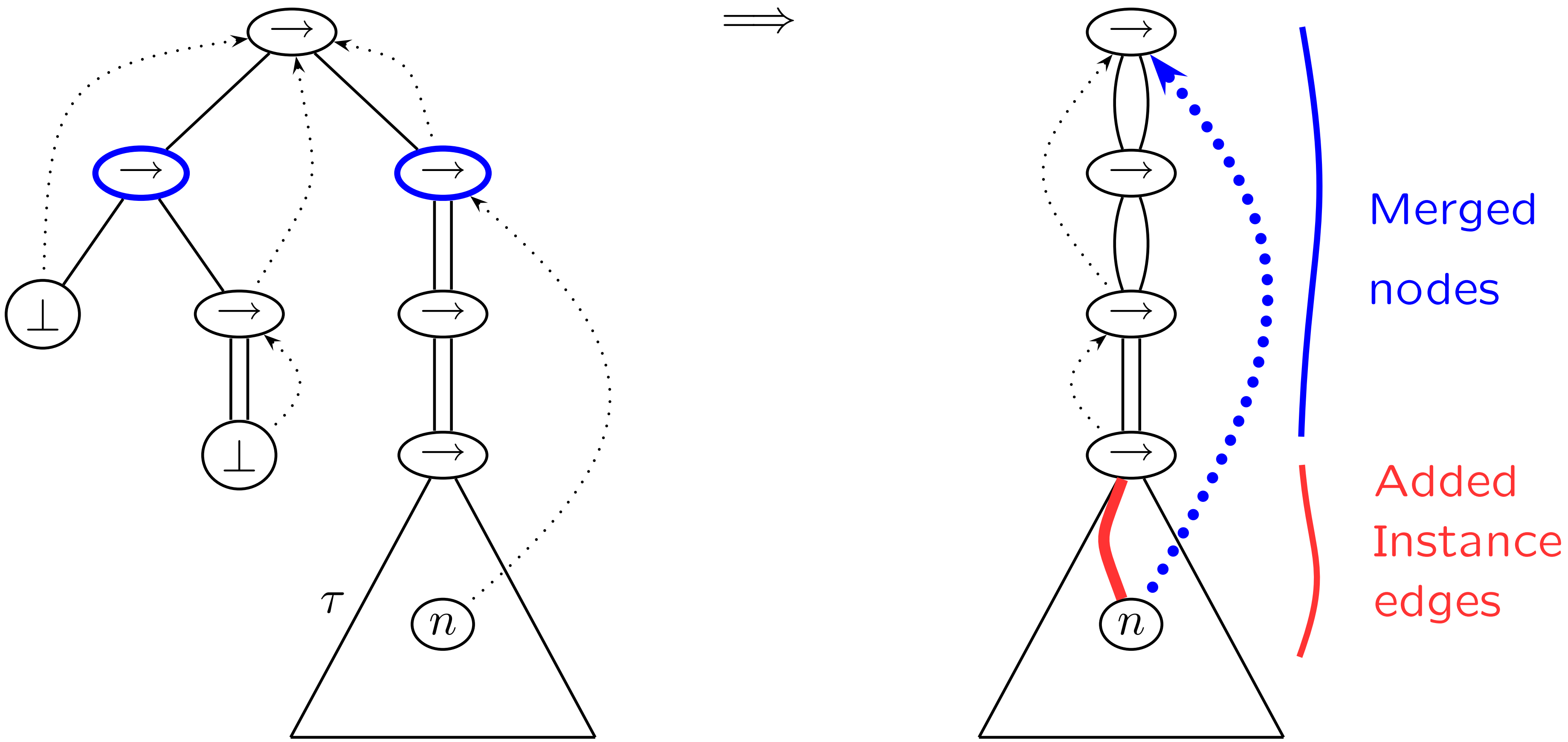
Escaping edge

Incrementality



Add virutal structure edge

Incrementality

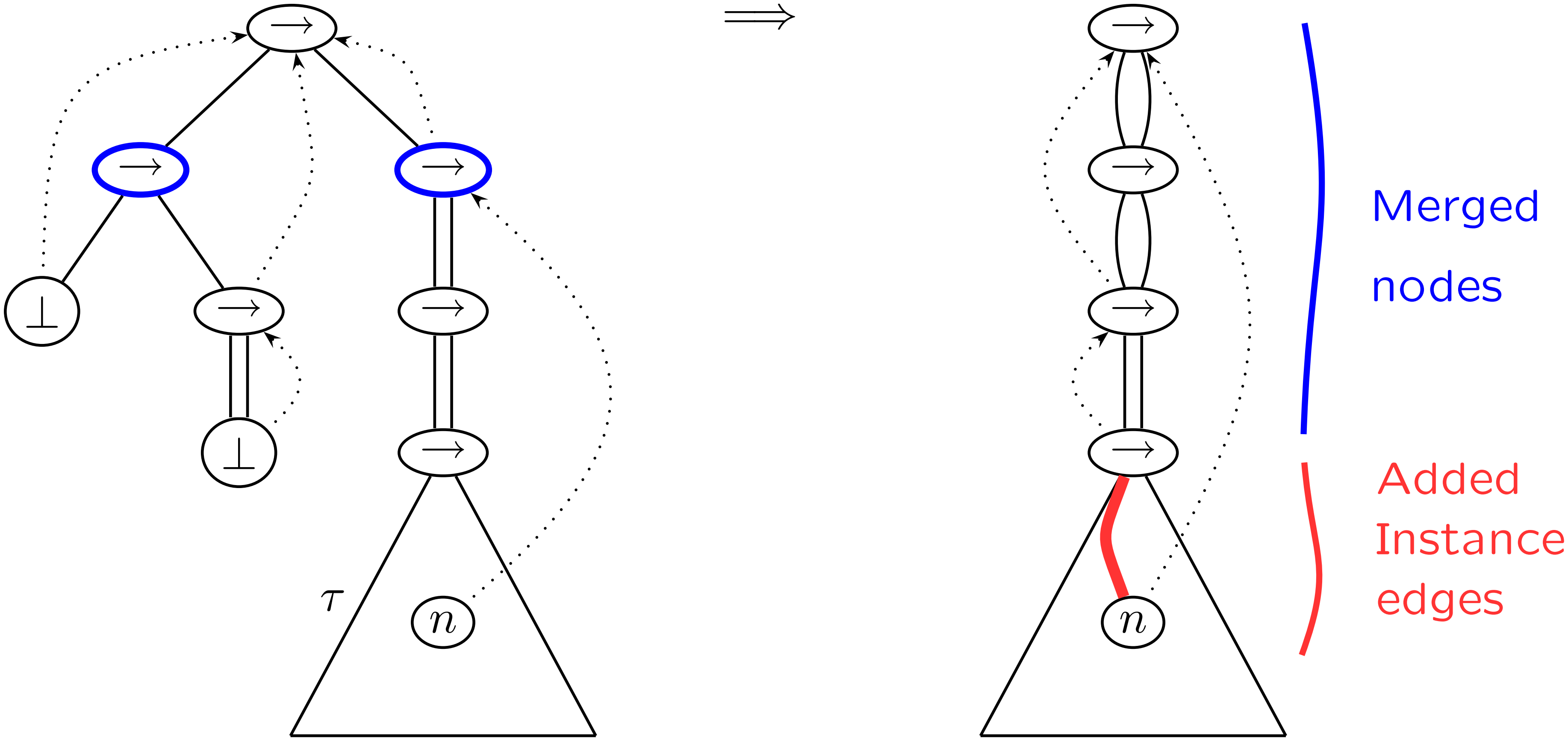


Merged nodes

Added Instance edges

Now correct

Incrementality



Cost linear in number of merged nodes plus number of added instance edges

From graphic types to graphic constraints

Key features

- Binding structure (and invariants)
- Instance relation

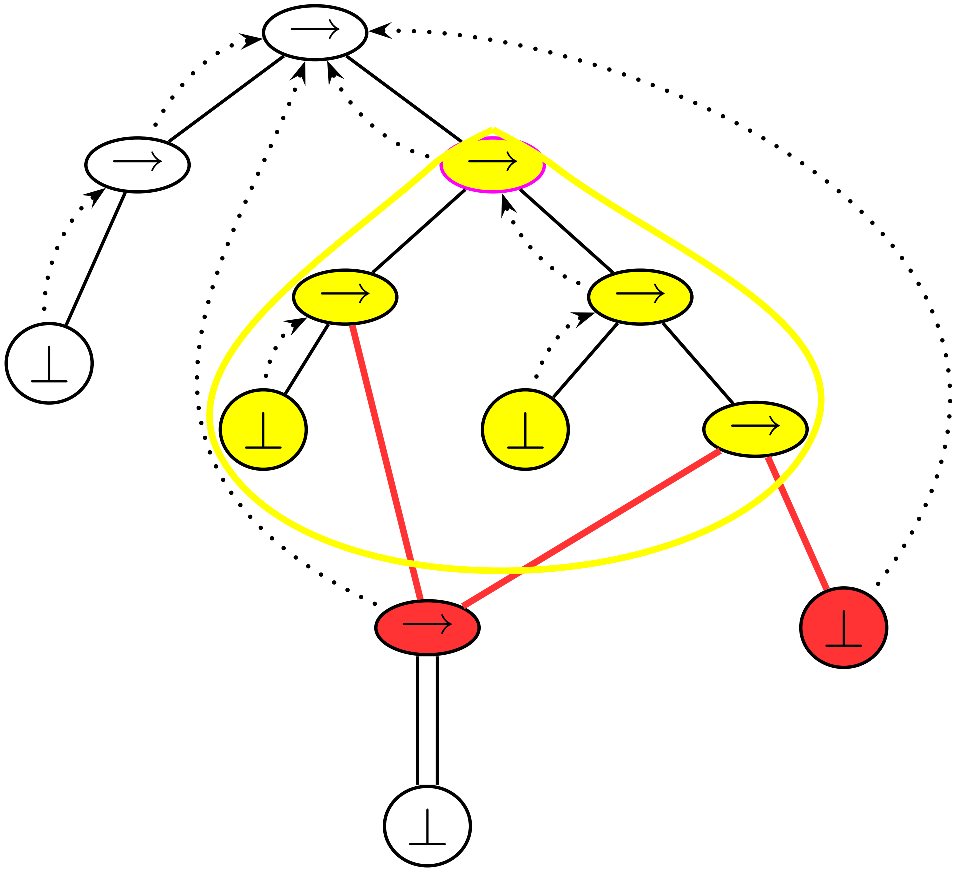
Type constraints

- Add new node to types, that are to be interpreted, especially as type constraints.
- Preserve the invariants
- Introduce new transformations (beyond instantiation) to simplify them.

Focussing at a node

The *interior* $\lceil n \rceil$ of a node n is the set of nodes dominated by n when inverse binding edges are added to structure edges.

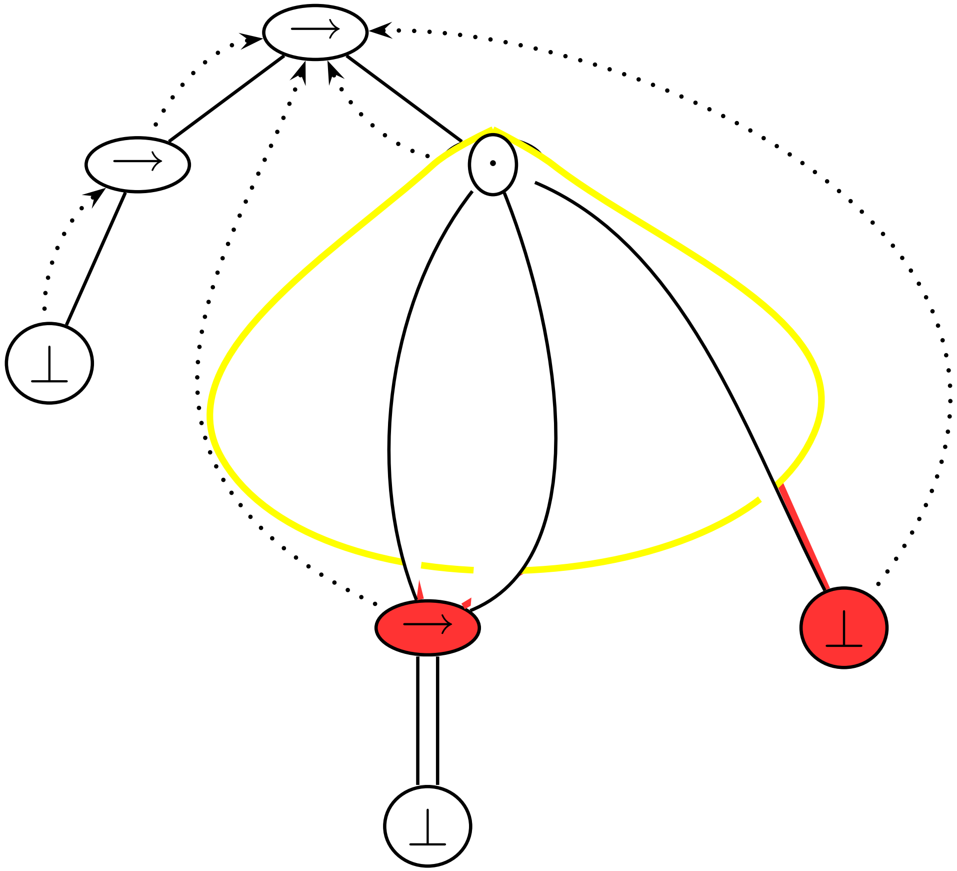
The *frontier* of n is the set of nodes that are not interior nodes but reached by structure edges from interior nodes.



Focussing at a node

The *interior* $\lceil n \rceil$ of a node n is the set of nodes dominated by n when inverse binding edges are added to structure edges.

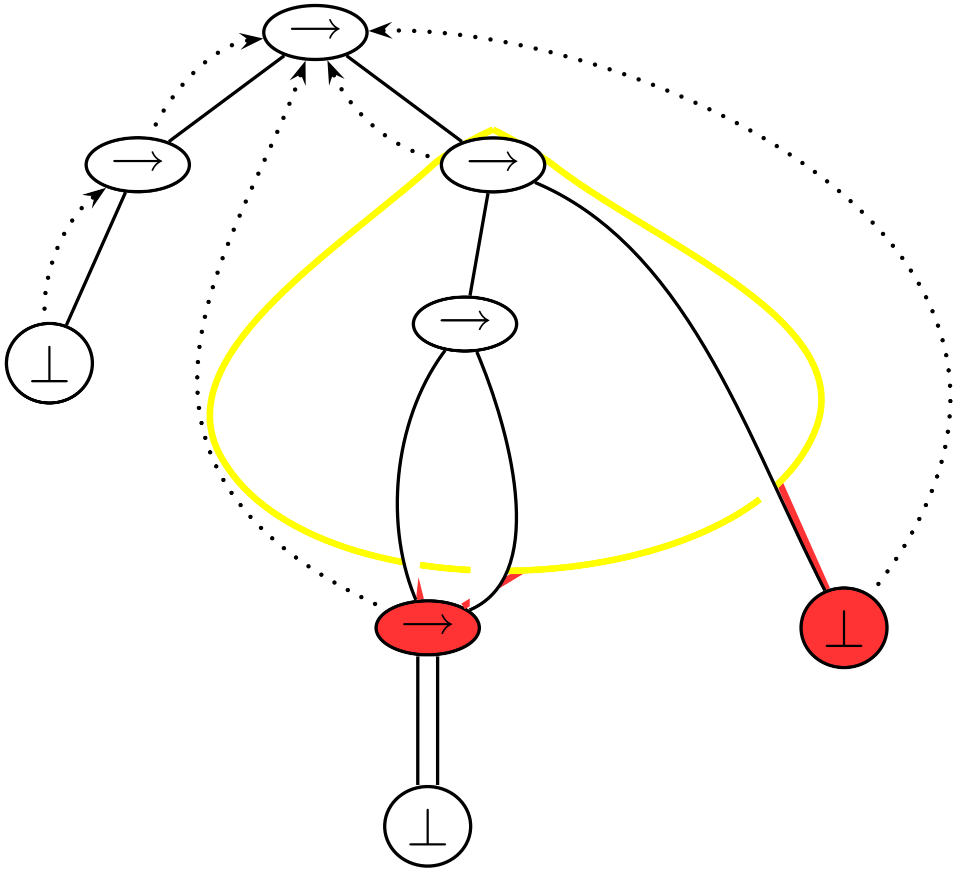
The *frontier* of n is the set of nodes that are not interior nodes but reached by structure edges from interior nodes.



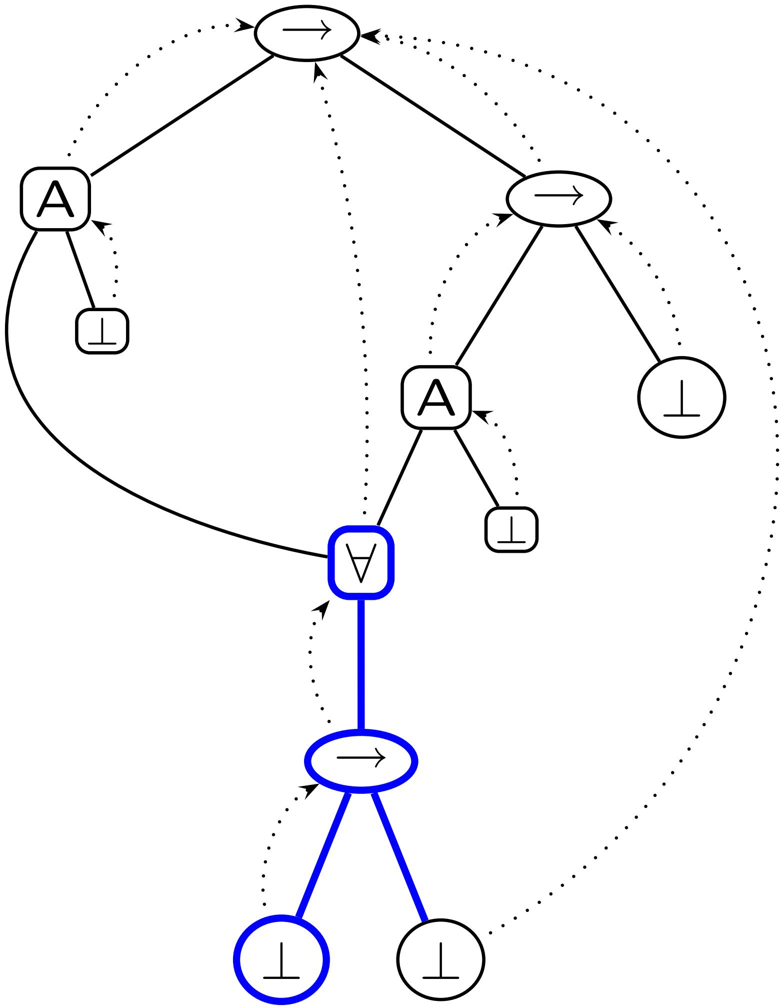
Focussing at a node

The *interior* $\lceil n \rceil$ of a node n is the set of nodes dominated by n when inverse binding edges are added to structure edges.

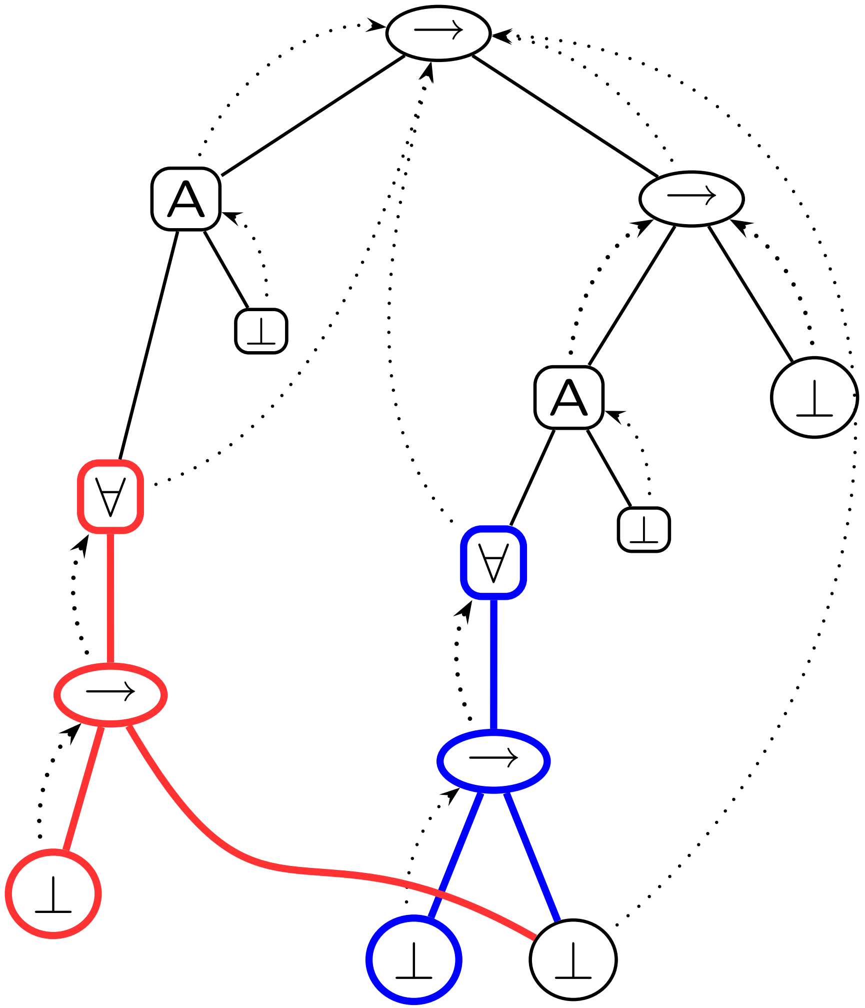
The *frontier* of n is the set of nodes that are not interior nodes but reached by structure edges from interior nodes.



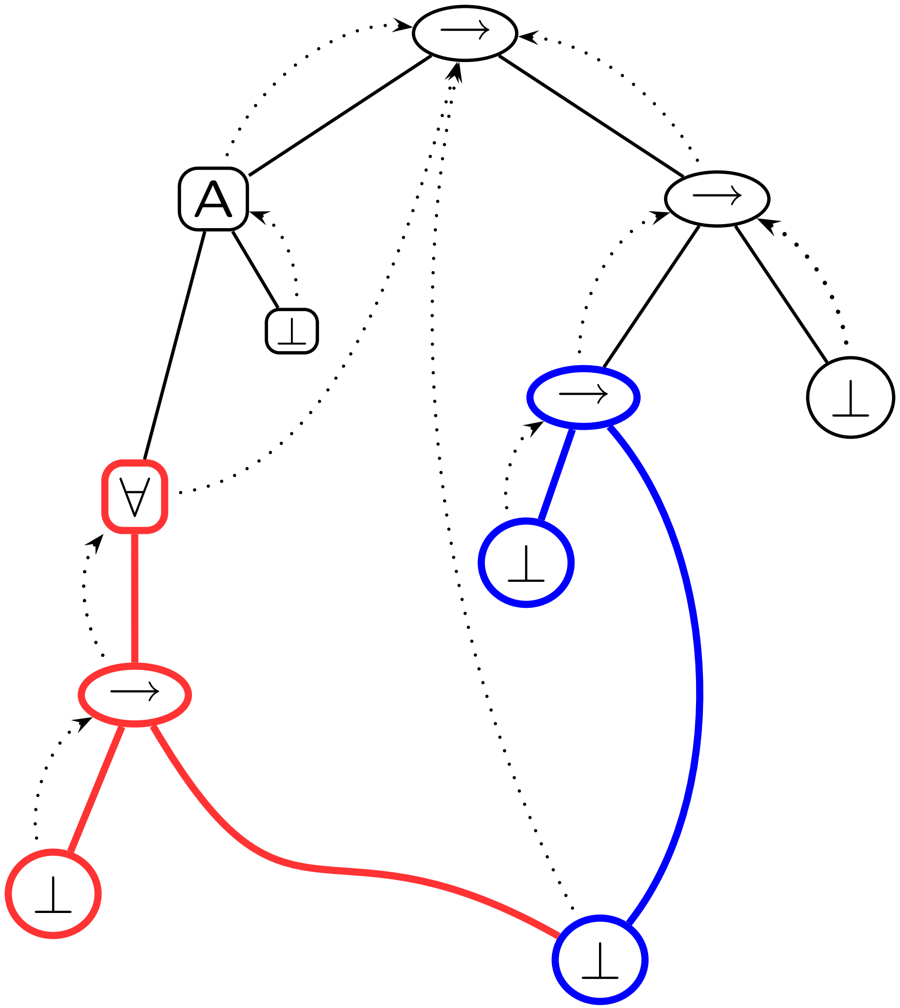
Copying



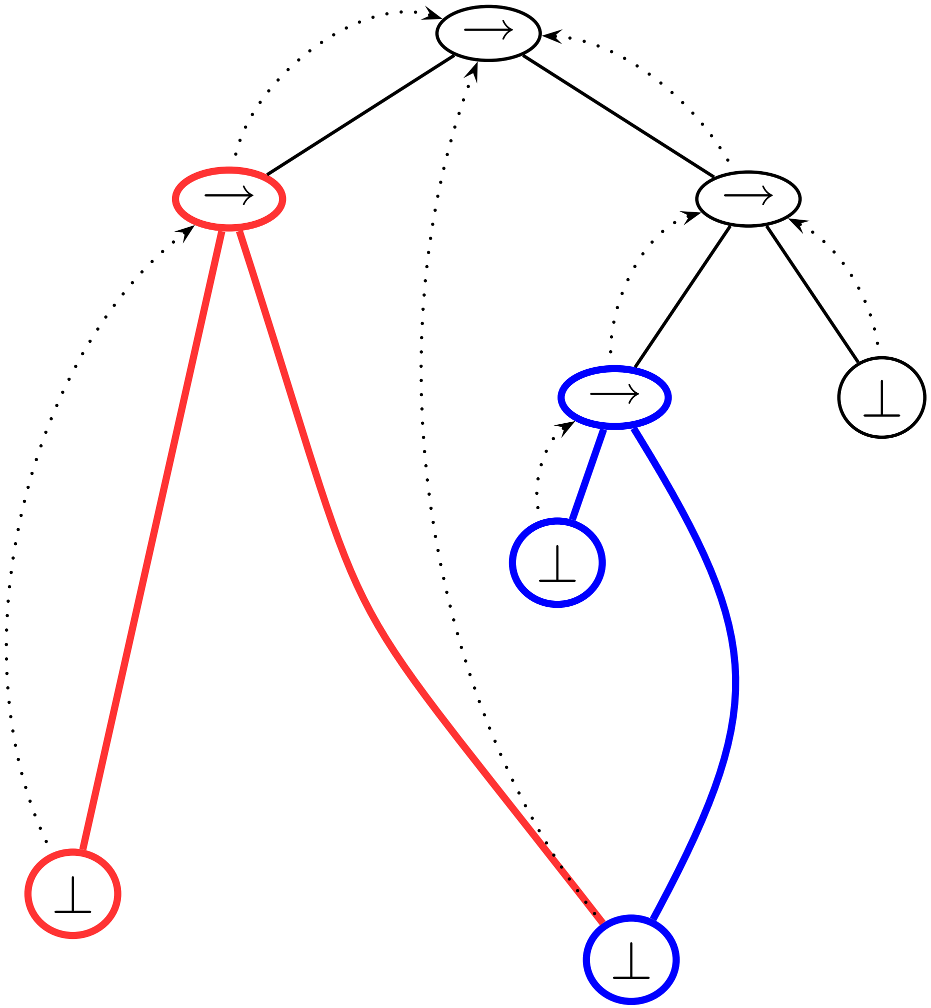
Copying



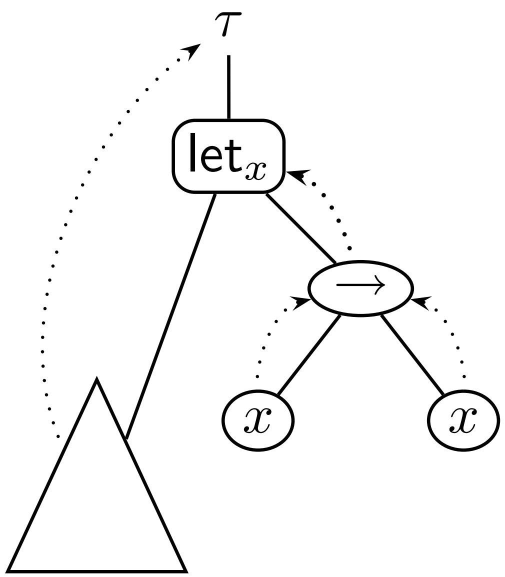
Copying



Copying

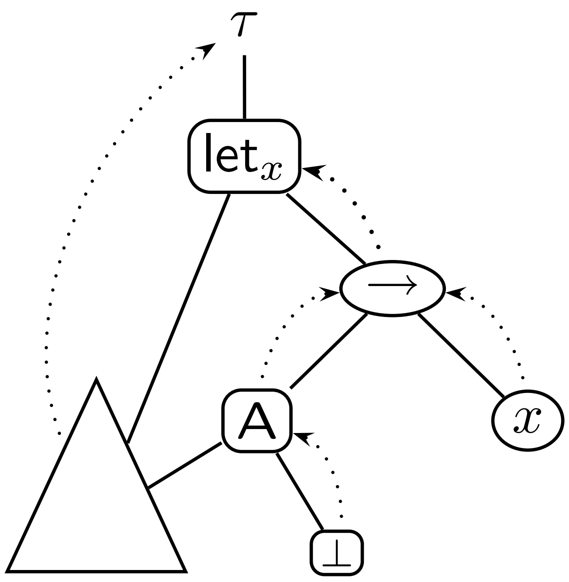


Abbreviations



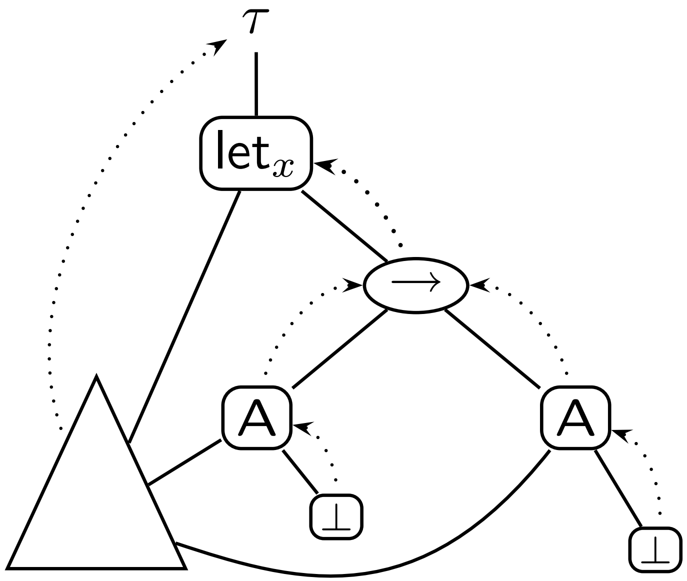
Replace any occurrence of x by a copy.

Abbreviations



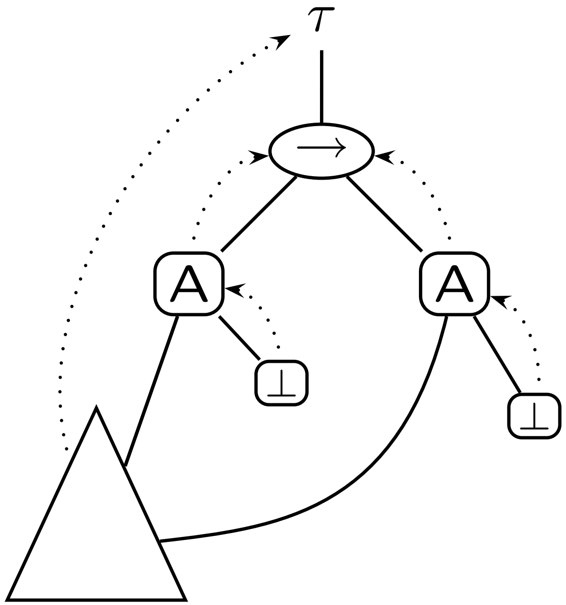
Replace any occurrence of x by a copy.

Abbreviations



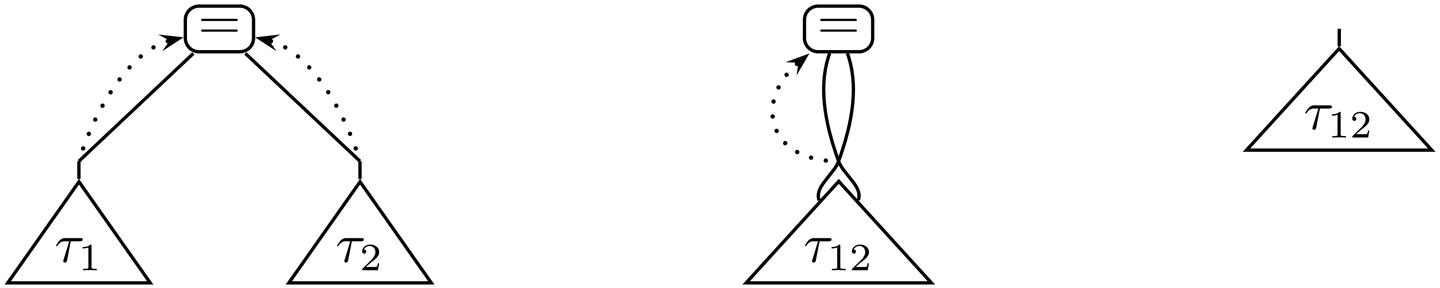
Remove unused let$_x$ —provided left-hand branch is consistent Reduce copies as before

Abbreviations



Encoding unification problems

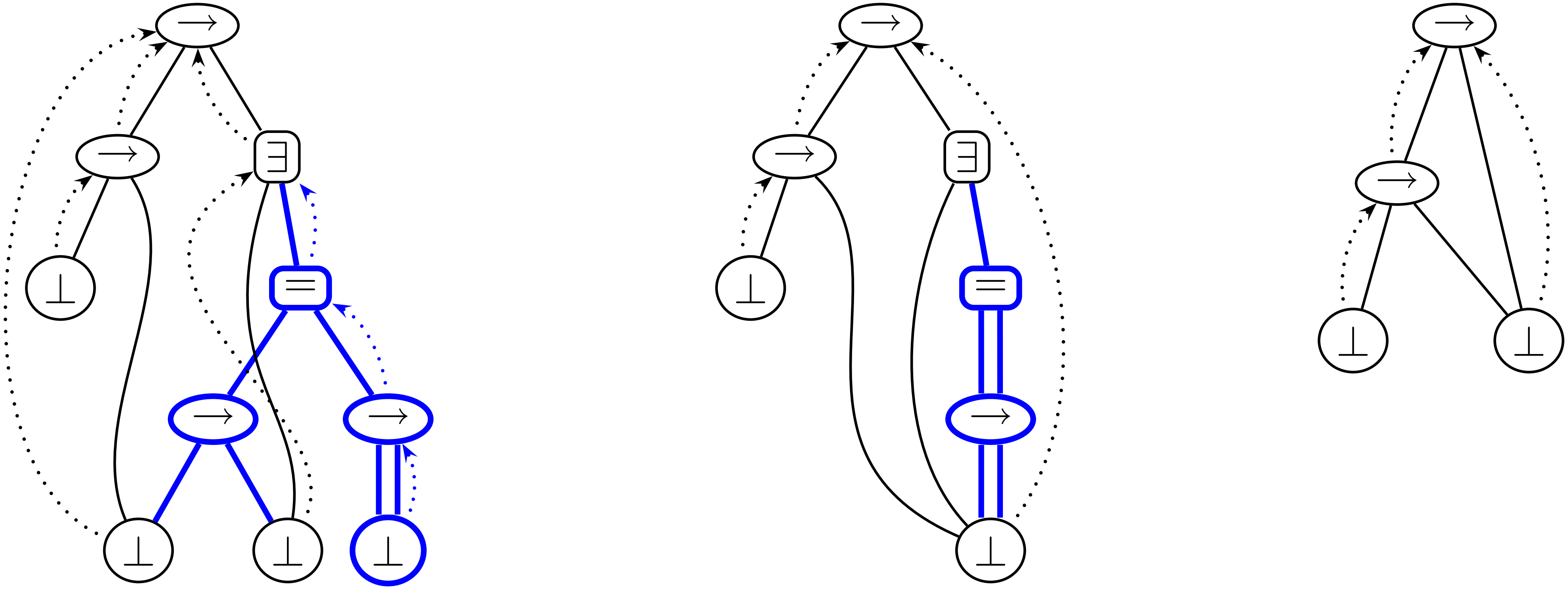
Unification



Encoding unification problems

Unification

More general constraints



Type inference constraints.

Syntactically

$$(Q)\ \Gamma \vdash a : \tau$$

Find pairs Q', τ' such that $Q' \leqslant \tau'$ and $(Q')\ \tau \leqslant \tau'$ and $(Q')\ \Gamma \vdash a : \tau'$.

Type inference constraints.

Syntactically

$$(Q)\ x_1 : \tau_1, \ldots x_n : \tau_n \vdash a : \alpha$$

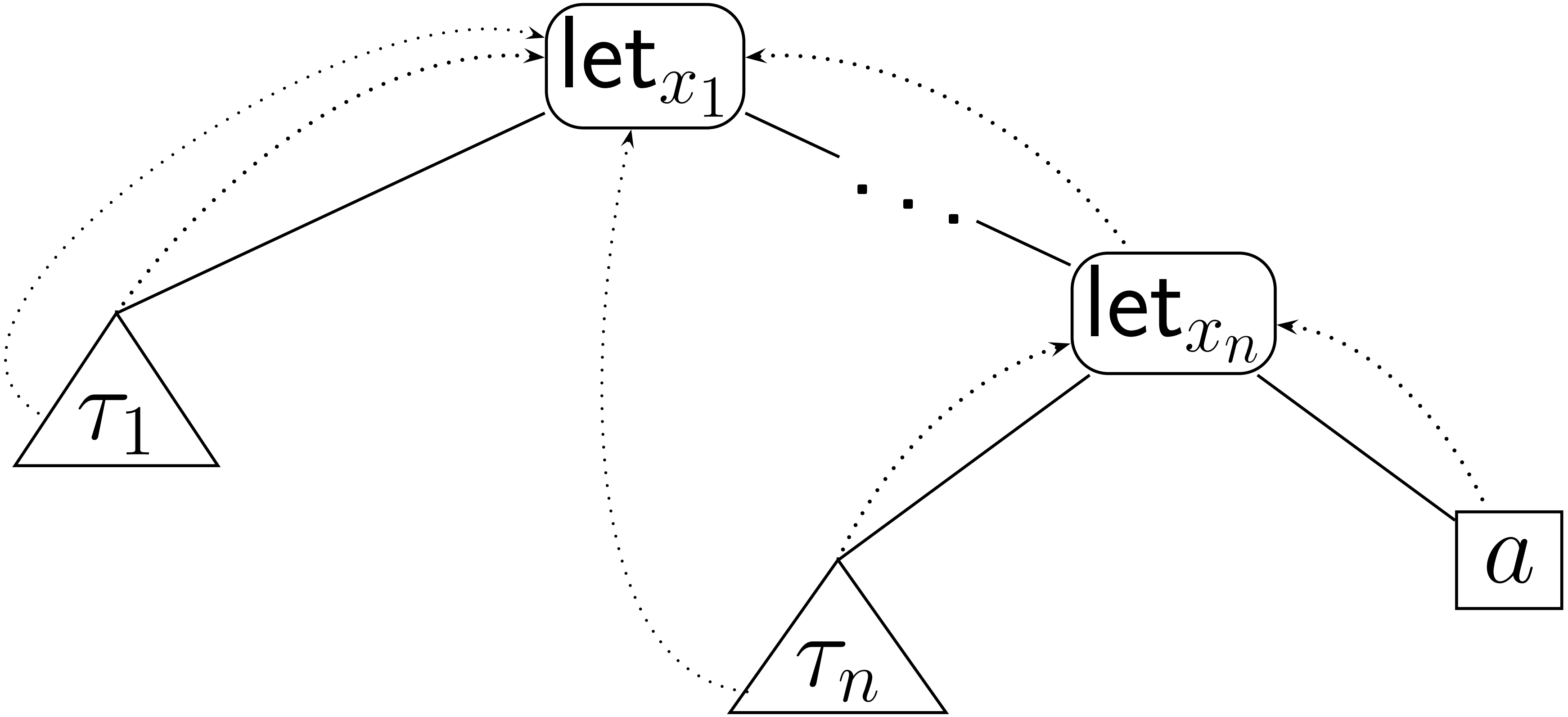
Find pairs Q', τ' such that $Q' \leqslant \tau'$ and $(Q')\ \tau \leqslant \tau'$ and $(Q')\ \Gamma \vdash a : \tau'$.

Type inference constraints.

Syntactically

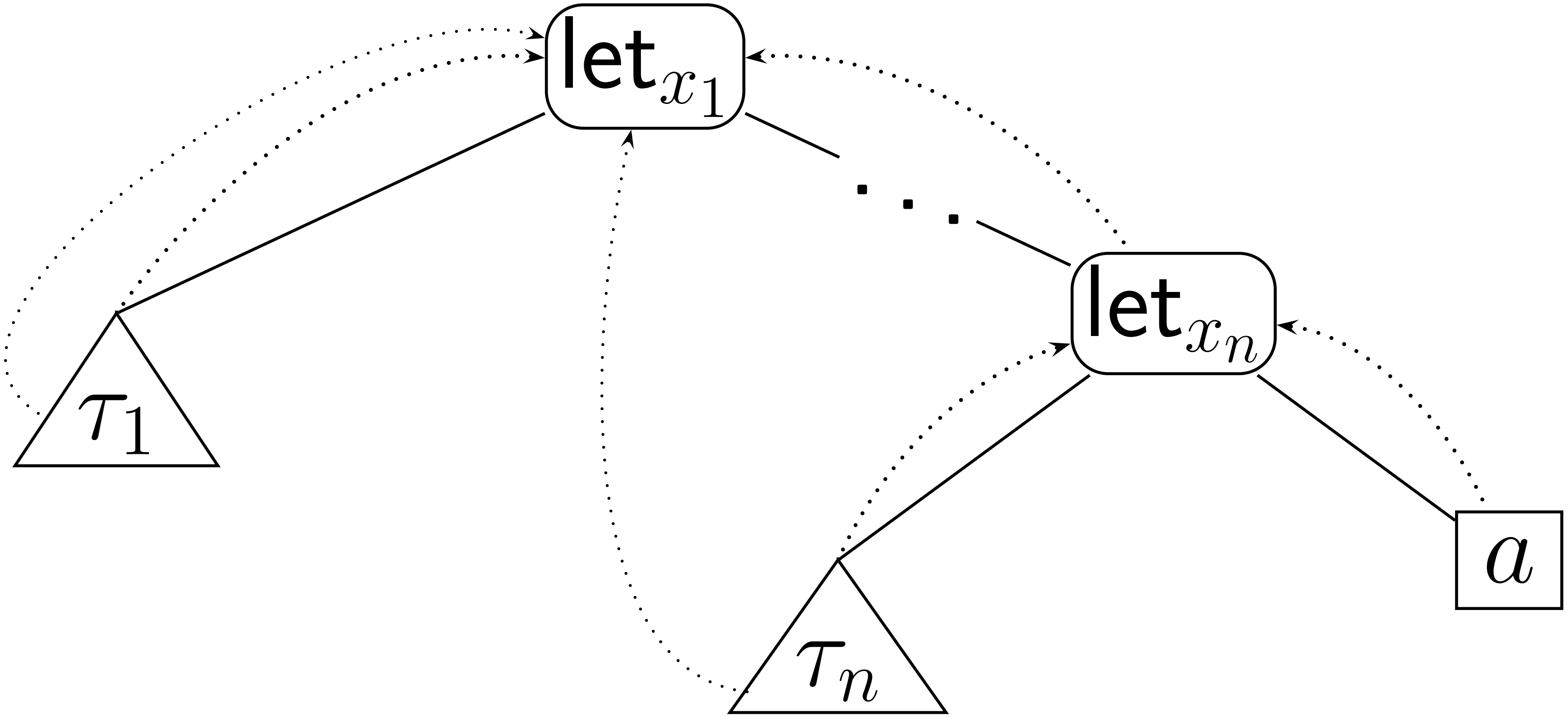
$$(Q)\ x_1 : \tau_1, \ldots x_n : \tau_n \vdash a : \alpha$$

Graphically



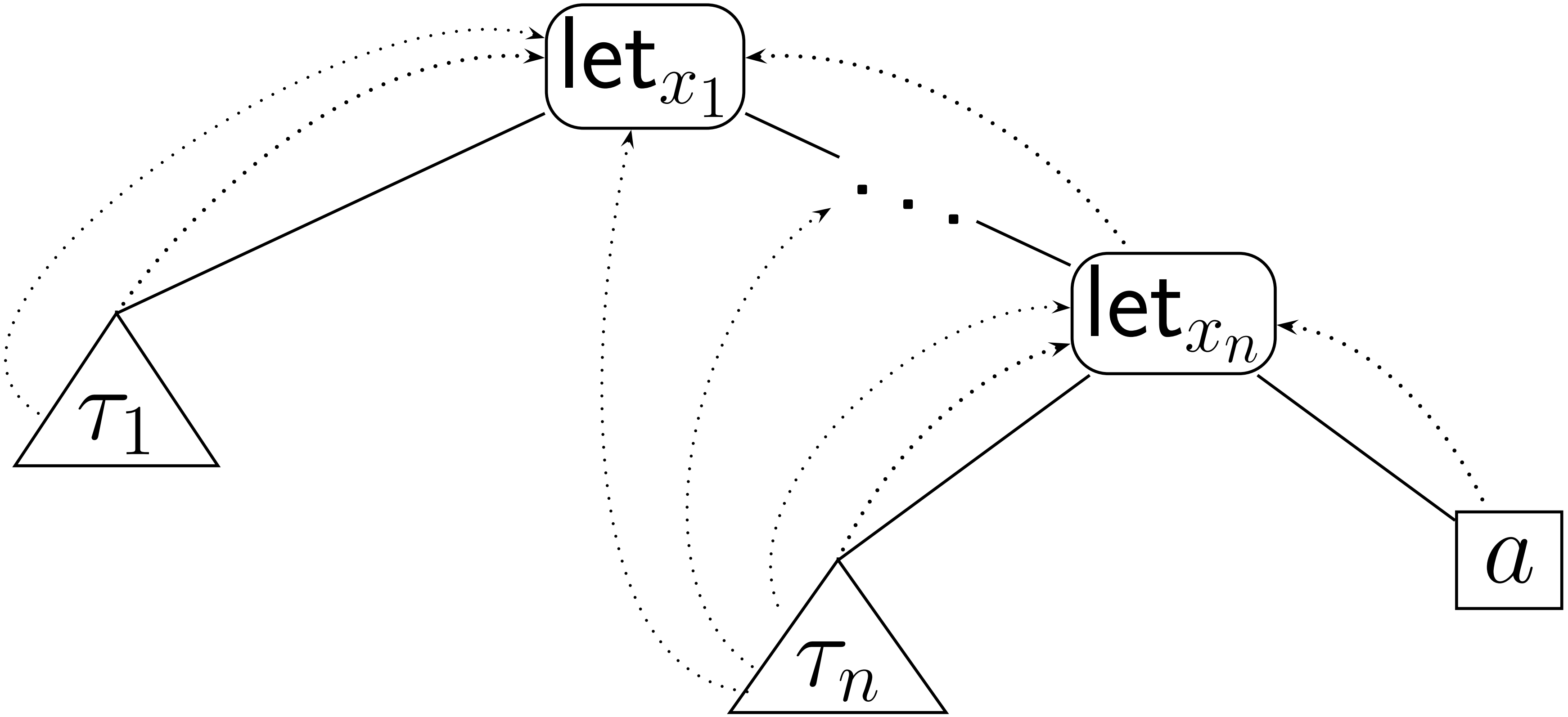
Type inference constraints.

Graphically



Type inference constraints.

Graphically

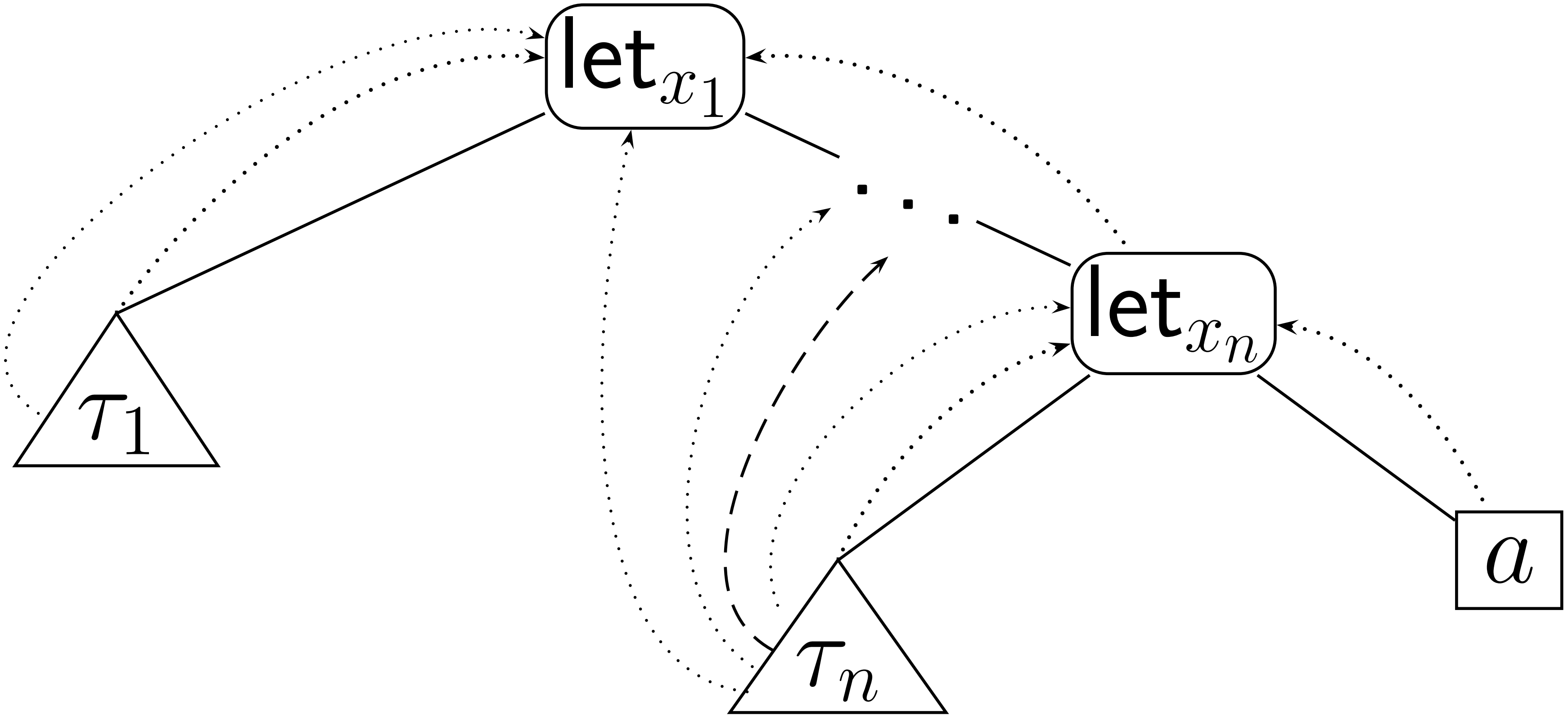


Key

Some nodes of τ_n may actually be bound tighter, just as tightly as permitted.

Type inference constraints.

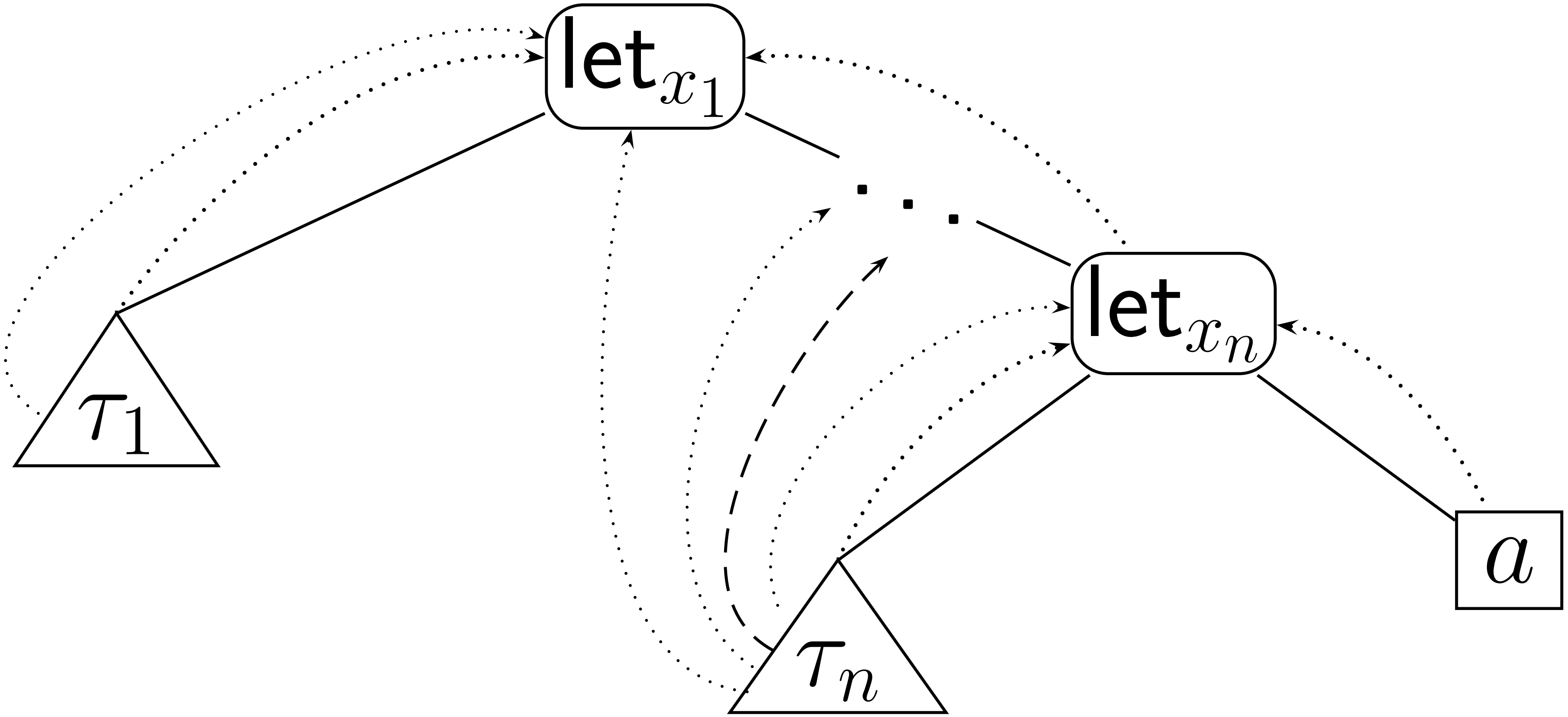
Graphically



Of course, some bindings may also be rigid.

Type inference constraints.

Graphically

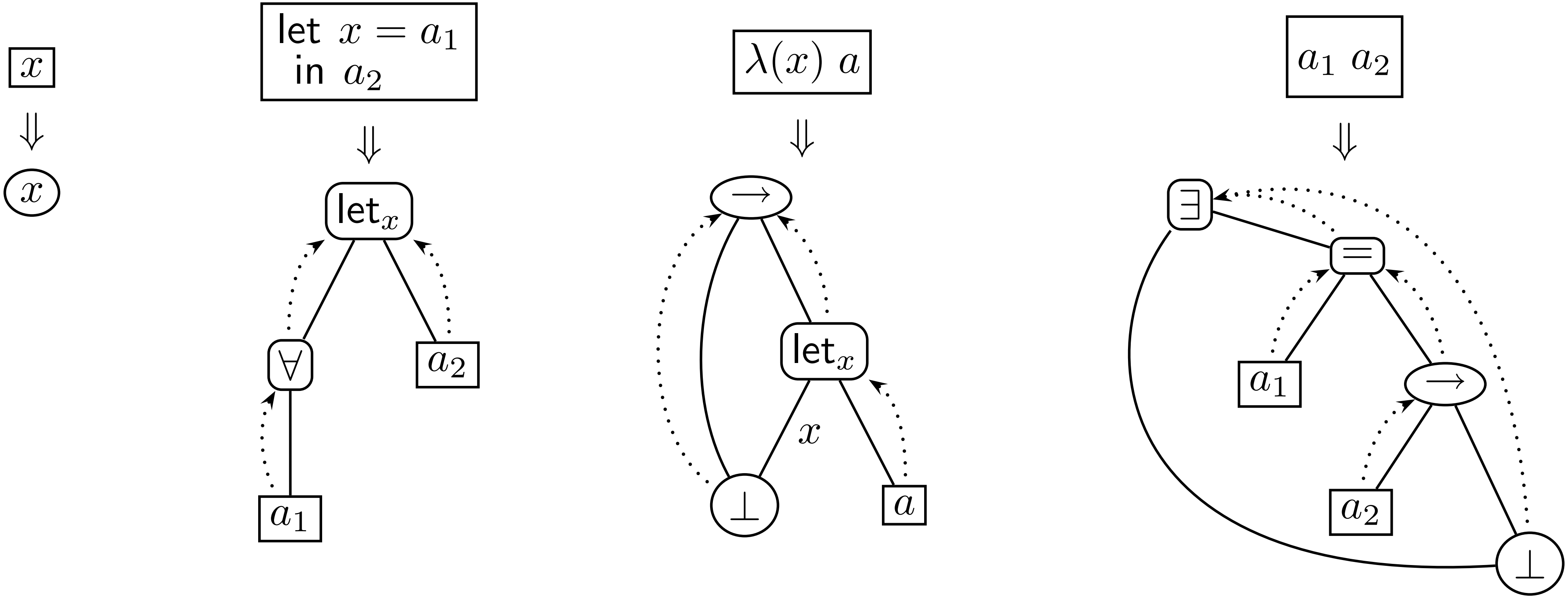


Find instances of the graph so that constraints are satisfied.

Their is a smaller solution if any, of which all other solutions are instances.

Type inference constraints.

Simplification



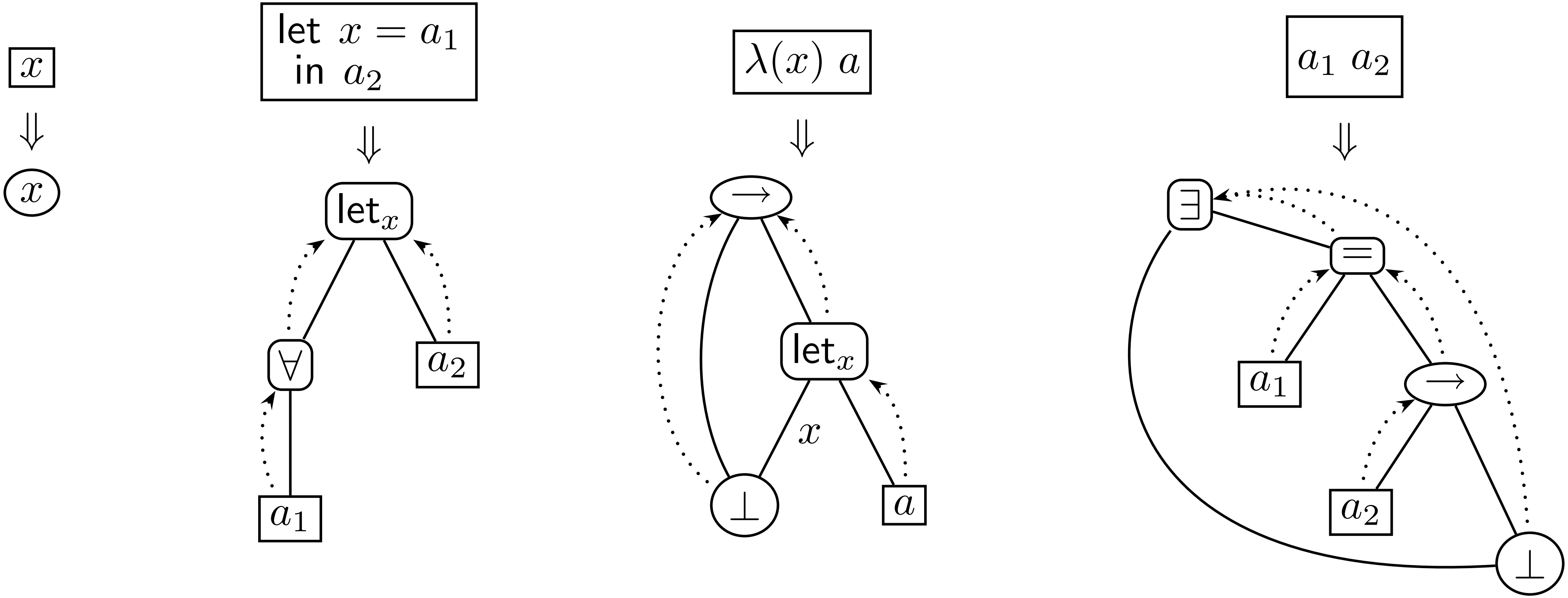
Key feature

Types always kept as polymorphic as possible.

Interior application nodes will remain bound to interior nodes (hence polymorphic) unless unified with some exterior node. [possible optimization]

Type inference constraints.

Simplification



Type abbreviations

A key in ML^F, but technically treated as coercion functions.

Futur works

Unification is all formalized. (See papers on the web)

Type constraints need to be formlaized

Subject reduction: calls for a direct proof using graphical constraints.

Extensions of the core language

- Recursive types
- F^ω (ı.e. allow quantification over type operators)
- Existential types:
  - Encoding via universal types: encapsulation is explicitly, opening is explicit but with no type information
  - Can we infer positions of openings? (See work by Daan Leijen)

Appendices

Syntactic instance

Type Equivalence

Eq-Refl

$$(Q)\ \sigma \equiv \sigma$$

Eq-Trans

$$\frac{(Q)\ \sigma_1 \equiv \sigma_2 \qquad (Q)\ \sigma_2 \equiv \sigma_3}{(Q)\ \sigma_1 \equiv \sigma_3}$$

Eq-Context-R

$$\frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \equiv \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \equiv \forall\,(\alpha \diamond \sigma)\ \sigma_2}$$

Eq-Context-L

$$\frac{(Q)\ \sigma_1 \equiv \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma_1)\ \sigma \equiv \forall\,(\alpha \diamond \sigma_2)\ \sigma}$$

Eq-Free

$$\frac{\alpha \notin \mathsf{ftv}(\sigma_1)}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \equiv \sigma_1}$$

Eq-Comm

$$\frac{\alpha_1 \notin \mathsf{ftv}(\sigma_2) \qquad \alpha_2 \notin \mathsf{ftv}(\sigma_1)}{(Q)\ \forall\,(\alpha_1 \diamond_1 \sigma_1)\ \forall\,(\alpha_2 \diamond_2 \sigma_2)\ \sigma \equiv \forall\,(\alpha_2 \diamond_2 \sigma_2)\ \forall\,(\alpha_1 \diamond_1 \sigma_1)\ \sigma}$$

Eq-Var

$$(Q)\ \forall\,(\alpha \diamond \sigma)\ \alpha \equiv \sigma$$

Eq-Mono

$$\frac{(\alpha \diamond \sigma_0) \in Q \qquad (Q)\ \sigma_0 \equiv \tau_0}{(Q)\ \tau \equiv \tau[\tau_0/\alpha]}$$

Type Abstraction

A-Equiv

$$\frac{(Q)\ \sigma_1 \equiv \sigma_2}{(Q)\ \sigma_1 \sqsubseteq \sigma_2}$$

A-Trans

$$\frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2 \qquad (Q)\ \sigma_2 \sqsubseteq \sigma_3}{(Q)\ \sigma_1 \sqsubseteq \sigma_3}$$

A-Context-R

$$\frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \forall\,(\alpha \diamond \sigma)\ \sigma_2}$$

A-Hyp

$$\frac{(\alpha_1 = \sigma_1) \in Q}{(Q)\ \sigma_1 \sqsubseteq \alpha_1}$$

A-Context-L

$$\frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall\,(\alpha = \sigma_1)\ \sigma \sqsubseteq \forall\,(\alpha = \sigma_2)\ \sigma}$$

Type Instance

I-Abstract

$$\frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \sigma_1 \leqslant \sigma_2}$$

I-Trans

$$\frac{(Q)\ \sigma_1 \leqslant \sigma_2 \qquad (Q)\ \sigma_2 \leqslant \sigma_3}{(Q)\ \sigma_1 \leqslant \sigma_3}$$

I-Context-R

$$\frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \leqslant \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \leqslant \forall\,(\alpha \diamond \sigma)\ \sigma_2}$$

I-Hyp

$$\frac{(\alpha_1 \geq \sigma_1) \in Q}{(Q)\ \sigma_1 \leqslant \alpha_1}$$

I-Context-L

$$\frac{(Q)\ \sigma_1 \leqslant \sigma_2}{(Q)\ \forall\,(\alpha \geq \sigma_1)\ \sigma \leqslant \forall\,(\alpha \geq \sigma_2)\ \sigma}$$

I-Bot

$$(Q)\ \bot \leqslant \sigma$$

I-Rigid

$$\frac{}{(Q)\ \forall\,(\alpha \geq \sigma_1)\ \sigma \leqslant \forall\,(\alpha = \sigma_1)\ \sigma}$$

Syntactic instance

Type Equivalence

Eq-Refl

$$(Q)\ \sigma \equiv \sigma$$

Eq-Trans

$$\frac{(Q)\ \sigma_1 \equiv \sigma_2 \qquad (Q)\ \sigma_2 \equiv \sigma_3}{(Q)\ \sigma_1 \equiv \sigma_3}$$

Eq-Context-R

$$\frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \equiv \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \equiv \forall\,(\alpha \diamond \sigma)\ \sigma_2}$$

Eq-Context-L

$$\frac{(Q)\ \sigma_1 \equiv \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma_1)\ \sigma \equiv \forall\,(\alpha \diamond \sigma_2)\ \sigma}$$

Eq-Free

$$\frac{\alpha \notin \mathsf{ftv}(\sigma_1)}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \equiv \sigma_1}$$

Eq-Comm

$$\frac{\alpha_1 \notin \mathsf{ftv}(\sigma_2) \qquad \alpha_2 \notin \mathsf{ftv}(\sigma_1)}{\begin{array}{c}(Q)\ \forall\,(\alpha_1 \diamond_1 \sigma_1)\ \forall\,(\alpha_2 \diamond_2\ \sigma_2)\ \sigma \\ \equiv \forall\,(\alpha_2 \diamond_2 \sigma_2)\ \forall\,(\alpha_1 \diamond_1 \sigma_1)\ \sigma\end{array}}$$

Eq-Var

$$(Q)\ \forall\,(\alpha \diamond \sigma)\ \alpha \equiv \sigma$$

Eq-Mono

$$\frac{(\alpha \diamond \sigma_0) \in Q \qquad (Q)\ \sigma_0 \equiv \tau_0}{(Q)\ \tau \equiv \tau[\tau_0/\alpha]}$$

Syntactic instance

Type Abstraction

A-Equiv

$$\frac{(Q)\ \sigma_1 \equiv \sigma_2}{(Q)\ \sigma_1 \sqsubseteq \sigma_2}$$

A-Trans

$$\frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2 \qquad (Q)\ \sigma_2 \sqsubseteq \sigma_3}{(Q)\ \sigma_1 \sqsubseteq \sigma_3}$$

A-Context-R

$$\frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \forall\,(\alpha \diamond \sigma)\ \sigma_2}$$

A-Hyp

$$\frac{(\alpha_1 = \sigma_1) \in Q}{(Q)\ \sigma_1 \sqsubseteq \alpha_1}$$

A-Context-L

$$\frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall\,(\alpha = \sigma_1)\ \sigma \sqsubseteq \forall\,(\alpha = \sigma_2)\ \sigma}$$

Syntactic instance

Type Instance

I-Abstract

$$\frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \sigma_1 \leqslant \sigma_2}$$

I-Trans

$$\frac{(Q)\ \sigma_1 \leqslant \sigma_2 \qquad (Q)\ \sigma_2 \leqslant \sigma_3}{(Q)\ \sigma_1 \leqslant \sigma_3}$$

I-Context-R

$$\frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \leqslant \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \leqslant \forall\,(\alpha \diamond \sigma)\ \sigma_2}$$

I-Hyp

$$\frac{(\alpha_1 \geq \sigma_1) \in Q}{(Q)\ \sigma_1 \leqslant \alpha_1}$$

I-Context-L

$$\frac{(Q)\ \sigma_1 \leqslant \sigma_2}{(Q)\ \forall\,(\alpha \geq \sigma_1)\ \sigma \leqslant \forall\,(\alpha \geq \sigma_2)\ \sigma}$$

I-Bot

$$(Q)\ \bot \leqslant \sigma$$

I-Rigid

$$\frac{}{(Q)\ \forall\,(\alpha \geq \sigma_1)\ \sigma \leqslant \forall\,(\alpha = \sigma_1)\ \sigma}$$