A graphical presentation of ML^F types with a linear-time unification algorithm

Didier Rémy, Boris Yakobowski

INRIA Rocquencourt

A brief presentation of ML$^{\mathsf{F}}$

[Le Botlan-Rémy, ICFP 2003]

[Le Botlan, 2003]

Why ML^F?

ML

Outer $\forall$

$\forall \alpha\beta.\ (\alpha \to \beta) \to \alpha\ t \to \beta\ t$

Full type inference

(Good)

System F

Inner (1st class) $\forall$ (Good)

$\lambda(f : \forall\alpha.\alpha \to \alpha)(f\ [\mathrm{int}]\ 1, f\ [\mathrm{bool}]\ 'b')$

Explicitely typed

(undecidable type inference)

Fully annotated terms are

(too) verbose

$\Rightarrow$ Need for partial type inference

ML^{F} features

Conservative extension of both ML and System F

- ML programs need no annotations (type inference)
- F terms need fewer annotations

  type abstraction and applications are inferred
- Annotations are only required on λ-abstractions that are used polymorphically used

ML^{F} features

Conservative extension of both ML and System F

- ML programs need no annotations (type inference)
- F terms need fewer annotations

  type abstraction and applications are inferred
- Annotations are only required on λ-abstractions that are used polymorphically used

Principal types (taking user-provided annotations into account)

Robust to small program transformations

e.g. if $E[a_1\ a_2]$ is typable so is $E[\text{apply}\ a_1\ a_2]$

(where apply is $\lambda f.\lambda x.f\ x$)

Example: type of `choose id`

Term	Type	
$\texttt{id} = \lambda x.x$	$\forall\alpha.\ \alpha \to \alpha$	(τ_{id})
$\texttt{choose} = \lambda x.\lambda y.\textsf{if } b \textsf{ then } x \textsf{ else } y$	$\forall\gamma.\ \gamma \to \gamma \to \gamma$	

Example: type of `choose id`

In System F, two different typings for `choose id`:

$$\texttt{choose}\ [\forall\alpha\cdot\alpha\longrightarrow\alpha]\ \texttt{id} \quad : \quad \tau_{id}\longrightarrow\tau_{id} \qquad F_1$$

$$\Lambda\alpha\cdot\texttt{choose}\ [\alpha\longrightarrow\alpha]\ (\texttt{id}\ \alpha) \quad : \quad \forall\alpha\cdot(\alpha\longrightarrow\alpha)\longrightarrow(\alpha\longrightarrow\alpha) \qquad F_2$$

Example: type of `choose id`

In System F, two different typings for `choose id`:

$$\texttt{choose}\ [\forall \alpha \cdot \alpha \to \alpha]\ \texttt{id} \quad : \quad \tau_{id} \to \tau_{id} \qquad F_1$$

$$\Lambda \alpha \cdot \texttt{choose}\ [\alpha \to \alpha]\ (\texttt{id}\ \alpha) \quad : \quad \forall \alpha \cdot (\alpha \to \alpha) \to (\alpha \to \alpha) \qquad F_2$$

In ML^{F} (note the absence of type annotations):

$$\texttt{choose id} : \forall\, (\beta = \tau_{id})\ \beta \to \beta \qquad \tau_1$$

$$: \forall\, (\alpha)\ \forall\, (\beta = \alpha \to \alpha)\ \beta \to \beta \qquad \tau_2$$

But $\tau = \forall\, (\beta \geq \tau_{id})\ \beta \to \beta$ is another, principal, typing:

$$\tau \sqsubseteq \begin{cases} \forall\, (\beta \geq \texttt{int} \to \texttt{int})\ \beta \to \beta & (\textit{i.e.}\ (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})) \\ \forall\, (\beta = \forall\, (\eta = \tau_{id})\ \eta \to \eta)\ \beta \to \beta & (\textit{i.e.}\ (\tau_{id} \to \tau_{id}) \to (\tau_{id} \to \tau_{id})) \\ \tau_1,\ \tau_2 \end{cases}$$

Syntactic presentation

A lot of administrative rules

- Hides the underlying principles
- Heavy proofs
- Makes extensions difficult

Is the instance relation the best within the framework?

Expensive unification (and hence type inference) algorithms.

Would it scale up to large or automatically generated programs?

Contributions

Graphs are used instead of trees to represent types.

Graphs had already been proposed as a simpler representation,

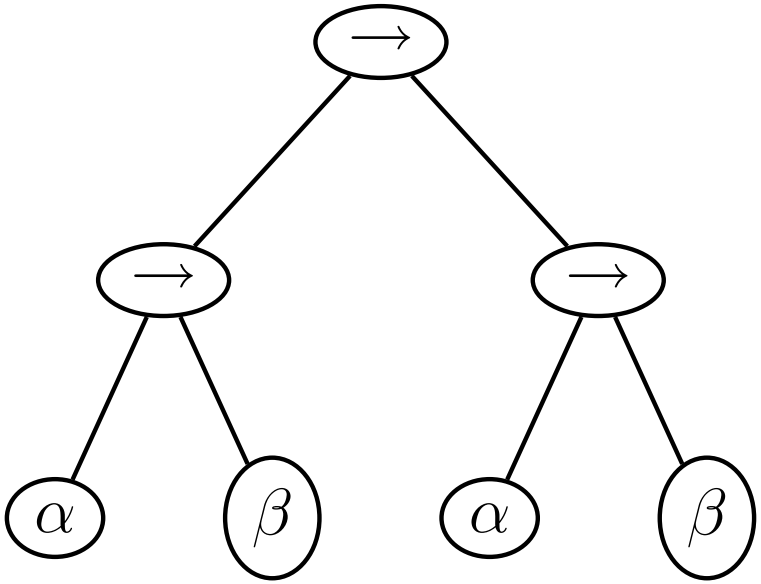
but were not formalized

- Simpler presentation, strongly related to first-order types
- Proofs are shorter and simpler
- Unification has good complexity

Representing first and second-order types

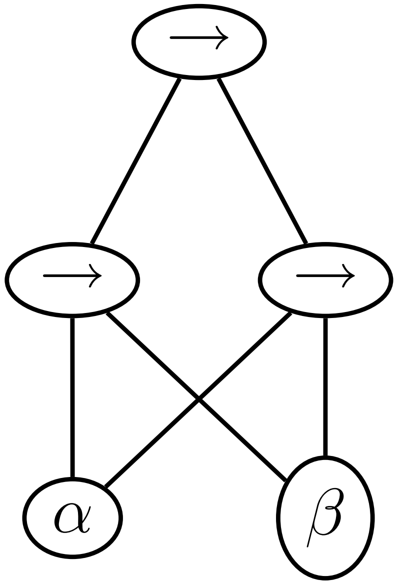
Representing first-order types

$(\alpha \to \beta) \to (\alpha \to \beta)$ as: a tree



Representing first-order types

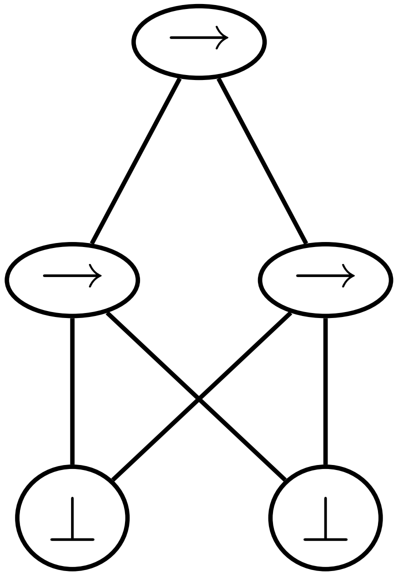
$(\alpha \longrightarrow \beta) \longrightarrow (\alpha \longrightarrow \beta)$ as: ~~a tree~~ a dag



All occurrences of a variable are shared.

Representing first-order types

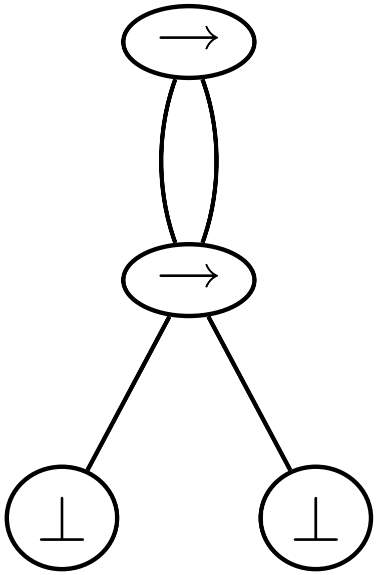
$(\alpha \to \beta) \to (\alpha \to \beta)$ as: ~~a tree~~ an anonymous dag



Variables can be α-converted and do not need to be named

Representing first-order types

$(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$ as: a ~~tree~~ an anonymous dag with sharing

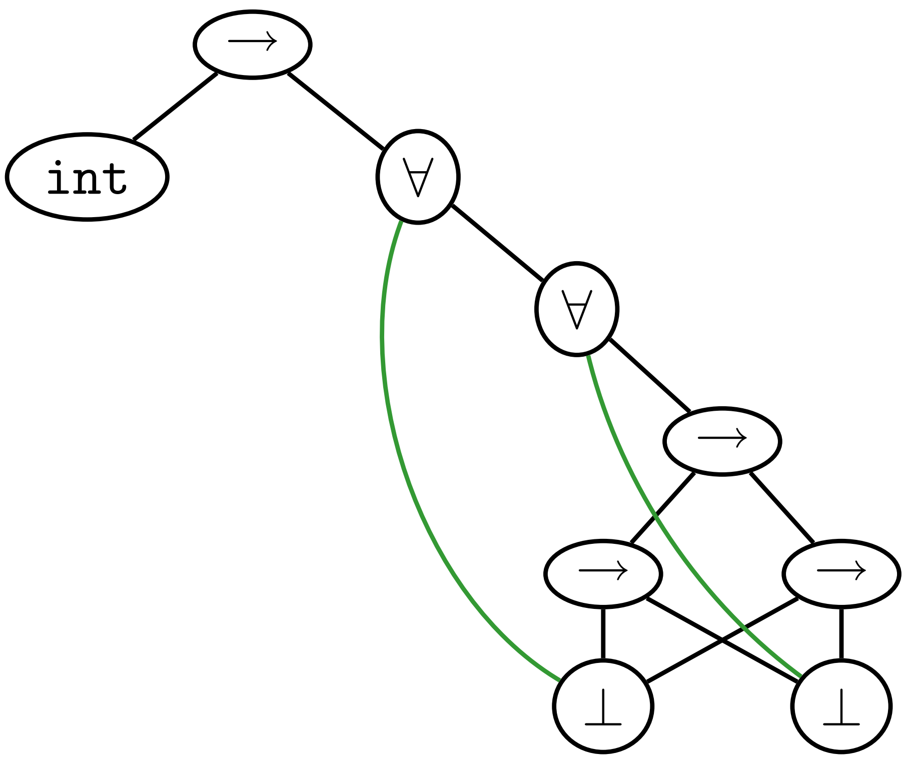


Non-variable nodes may be also shared

Representing terms with binders

Binders are represented with explicit $\forall$ nodes

$$\texttt{int} \to (\forall \alpha \beta. (\alpha \to \beta) \to (\alpha \to \beta))$$

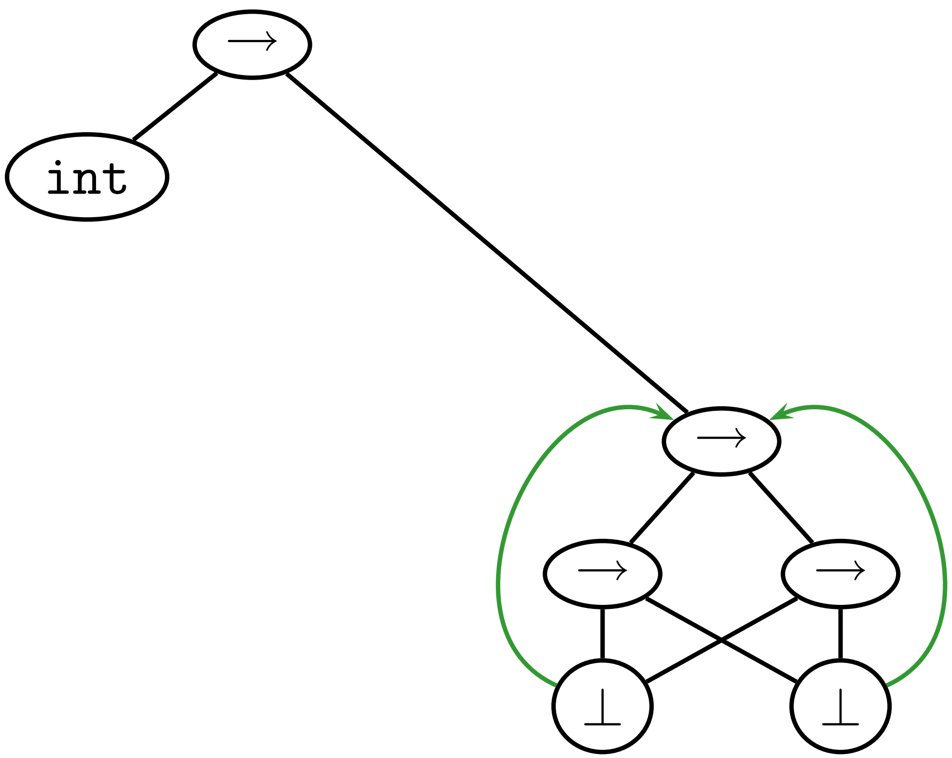


Problem: commuting or instantiating binders change the structure of the type

Representing terms with binders

With bindings edges, between a variable and the node where the variable is introduced.

$\texttt{int} \rightarrow (\forall \alpha \beta . (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta))$



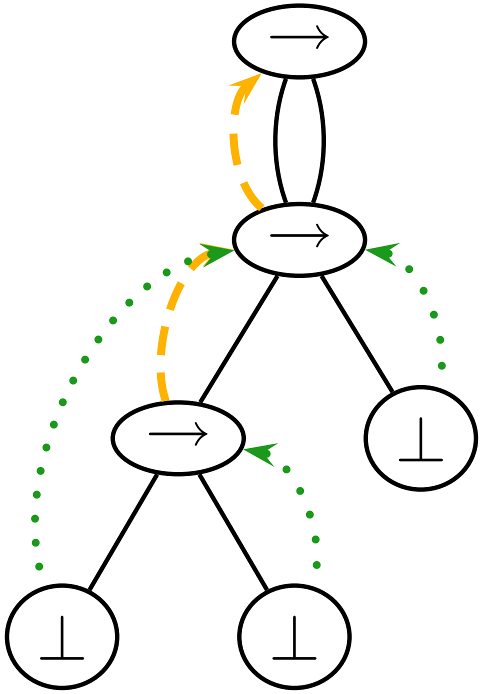
Commutation of binders comes for free!

ML^{F} types, graphically

$\mathsf{ML^F}$ graphic types

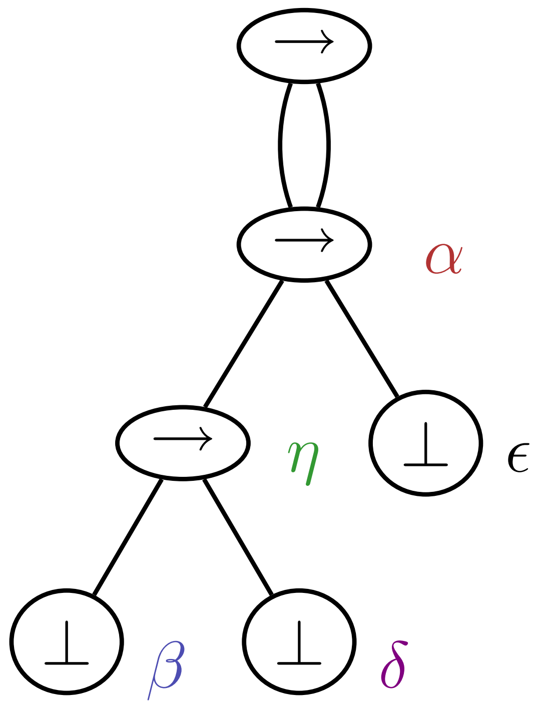
$$\forall (\alpha = \forall (\beta \geq \bot) \forall (\eta = \forall (\delta \geq \bot) \, \beta \to \delta) \forall (\epsilon \geq \bot) \, \eta \to \epsilon) \, \alpha \to \alpha$$

As a graphic type:



$$\underline{\alpha : \underline{\beta : \bot}\ \ \underline{\eta : \underline{\delta : \bot}\ \beta \to \delta}\ \ \underline{\epsilon : \bot}\ \eta \to \epsilon}\ \ \alpha \to \alpha$$

As a graphic type:

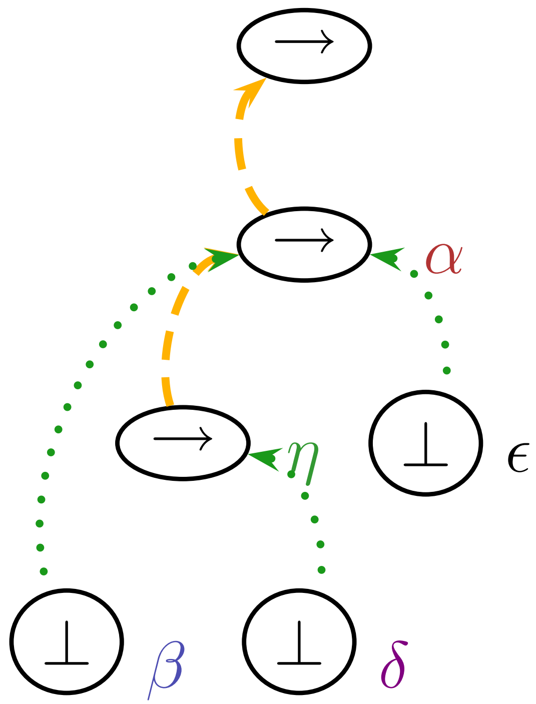


A first-order term graph...

ML$^{\mathsf{F}}$ graphic types

$$\forall\, \boxed{\alpha = \forall\, \boxed{\beta \geq \bot}\; \forall\, \boxed{\eta = \forall\, \boxed{\delta \geq \bot}\; \beta \to \delta}\; \forall\, \boxed{\epsilon \geq \bot}\; \eta \to \epsilon}\; \alpha \to \alpha$$

As a graphic type:

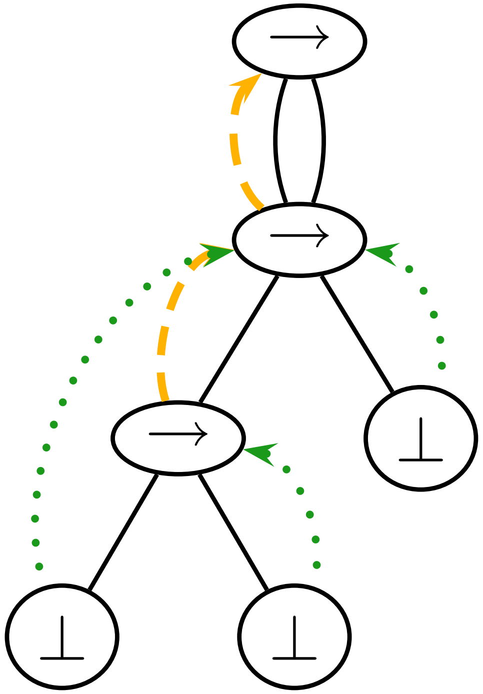


...plus a binding tree...

$\mathsf{ML^F}$ graphic types

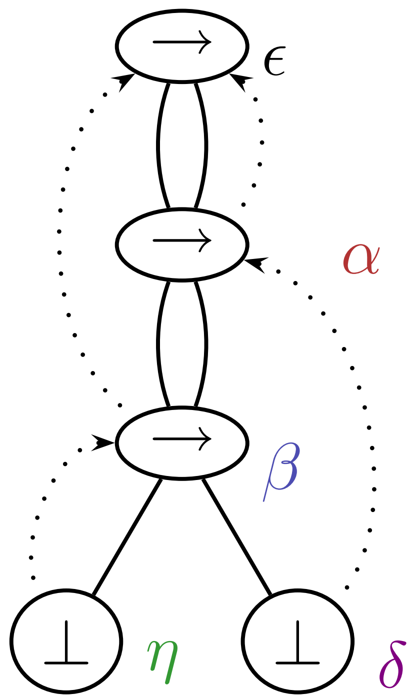
$$\forall\,(\alpha = \forall\,(\beta \geq \bot)\,\forall\,(\eta = \forall\,(\delta \geq \bot)\,\beta \to \delta)\,\forall\,(\epsilon \geq \bot)\,\eta \to \epsilon)\,\alpha \to \alpha$$

As a graphic type:



...superposed

Well-formedness of graphic types

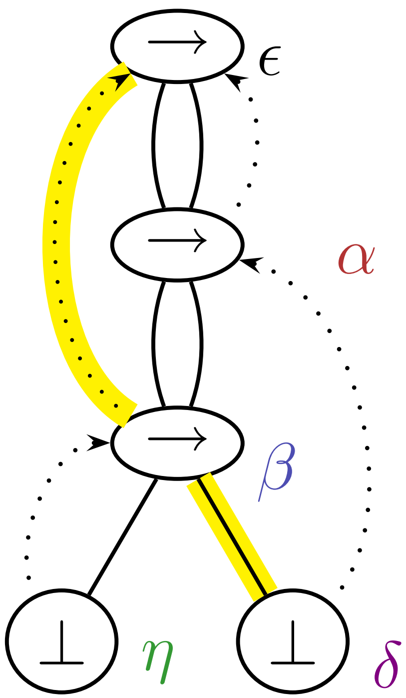


Syntactically:

- $\boxed{\forall\, (\alpha \geq \boxed{\forall\, (\delta)\ \beta \to \beta})}\ \boxed{\forall\, (\beta \geq \boxed{\forall\, (\eta)\ \eta \to \delta})}\ \alpha \to \alpha$

- There is a mutual dependency between α and β
  $\Rightarrow$ Not a ML$^{\mathsf{F}}$ type

Well-formedness of graphic types



Graphically:

- The binder of a node n must dominate n in all the mixed paths between n and the root ϵ
- There is a path between δ and ϵ which does not contain α

  $\Rightarrow$ This graph is not a type

Instance between graphic types

Instance

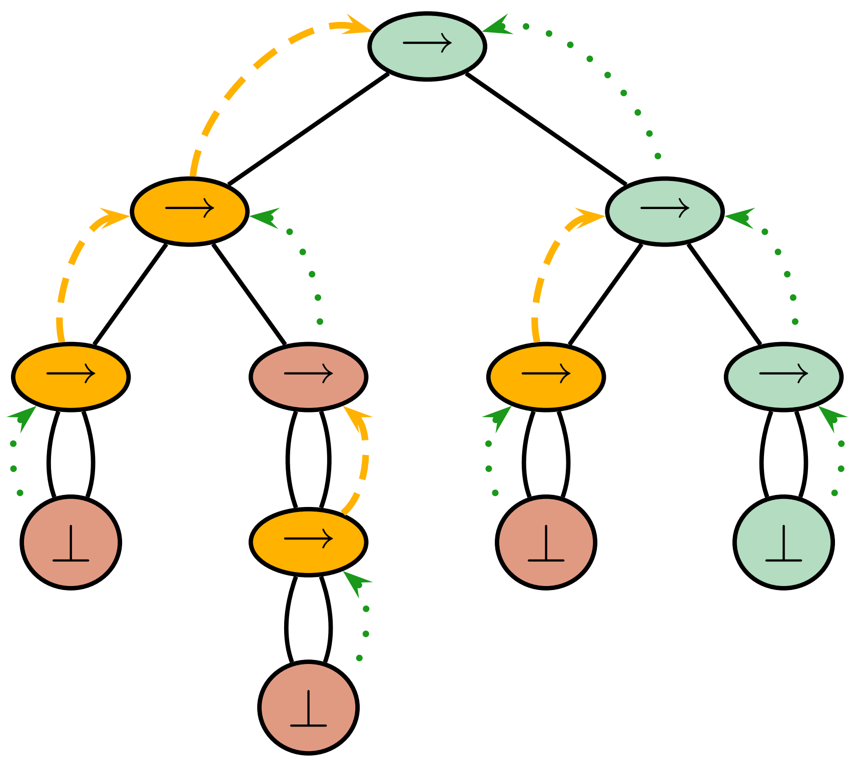
Only four different transformations on graphic types:

$$\left[\begin{array}{l} \left.\begin{array}{l}\text{Grafting}\\ \text{Merging}\end{array}\right\} \text{ change the structure of the type}\\ \left.\begin{array}{l}\text{Raising}\\ \text{Weakening}\end{array}\right\} \text{ change the binding tree of the type}\end{array}\right.$$

Plus some permissions on nodes governing the set of transformations that can be applied to a node

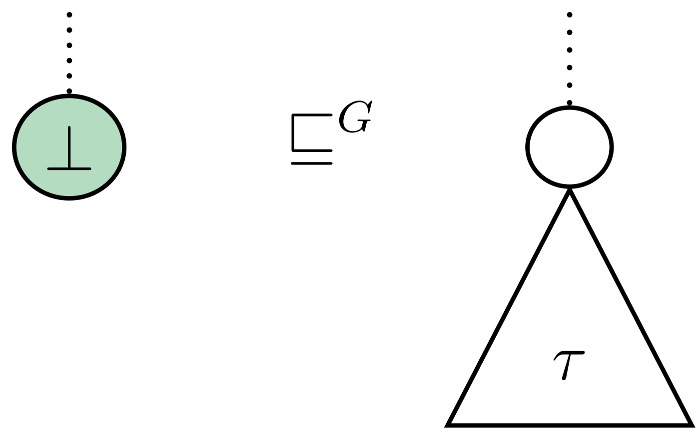
Flags and permissions

- Transformations whose inverse can be unsound: allowed on flexible nodes
- Transformations whose inverse is sound, but that cannot be made implicit while retaining type inference: allowed on rigid nodes:



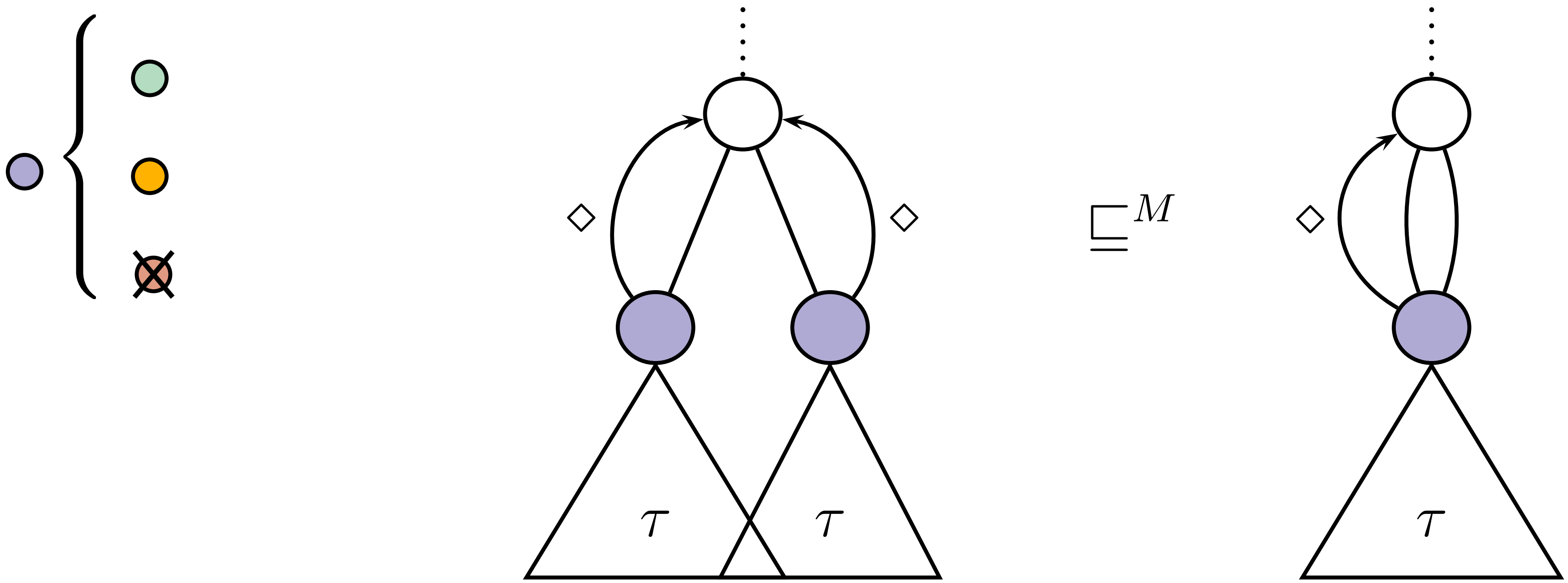
Binding path	Permissions
$\geq^*$	Flexible
$(\geq\|=)^*=$	Rigid
Others	Locked

Grafting



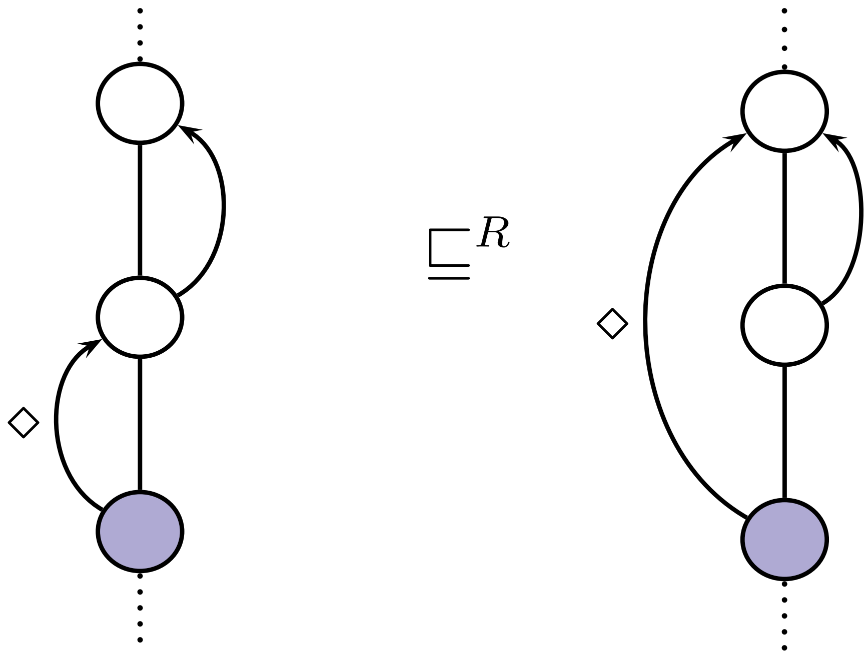
- Similar to the ML instance rule + generalization
  $\forall\, \alpha.\tau \leqslant \forall\, \overline{\beta}.\tau\,[\alpha/\tau']$
- Replaces a variable node by a type
- Irreversible transformation (the shape of the type changes), the node must be flexible

Merging



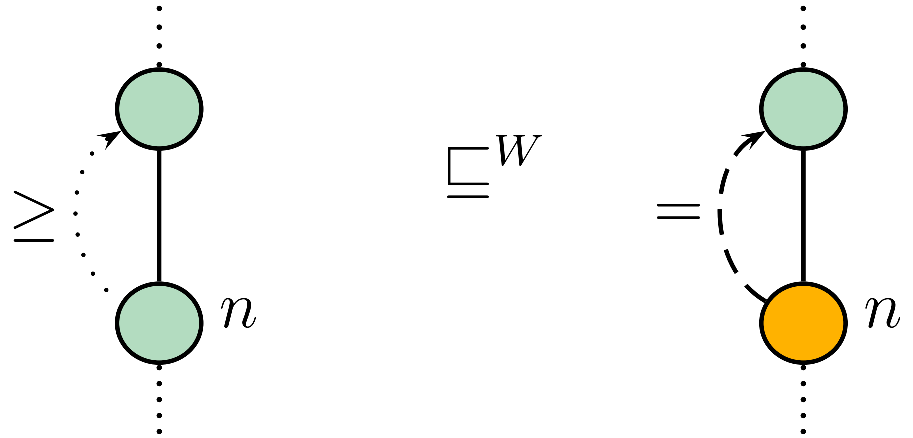
- Partly similar to the ML instance $\forall \alpha\beta.\, \alpha \to \beta \leqslant \forall \alpha.\, \alpha \to \alpha$
- Merges together two identical subgraphs bound on the same node with the same flag
- The nodes must be flexible or rigid

Raising



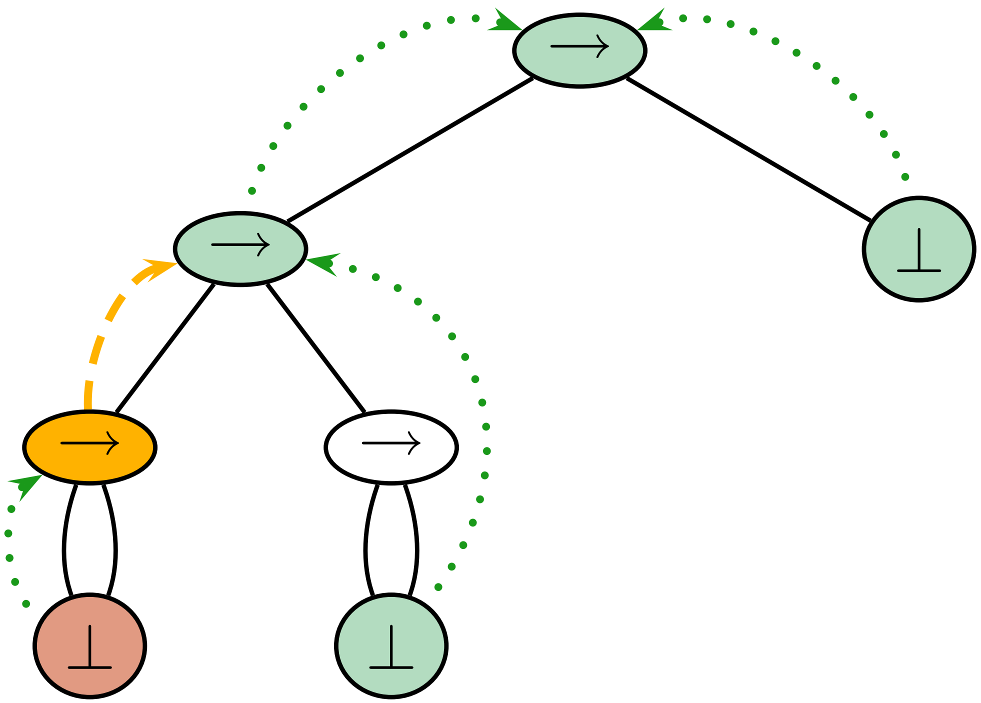
- Scope extrusion ($\tau \to (\forall \alpha.\, \tau') \leqslant \forall \alpha.\, \tau \to \tau'$, α not free in τ)
- Used to prove that the type $\forall\,(\beta \geq \forall\,(\alpha)\; \alpha \to \alpha)\; \beta \to \beta$ of `choose id` can be instantiated into $\forall\,(\alpha)\;\forall\,(\beta \geq \alpha \to \alpha)\; \beta \to \beta$
- The node must be flexible or rigid

Weakening

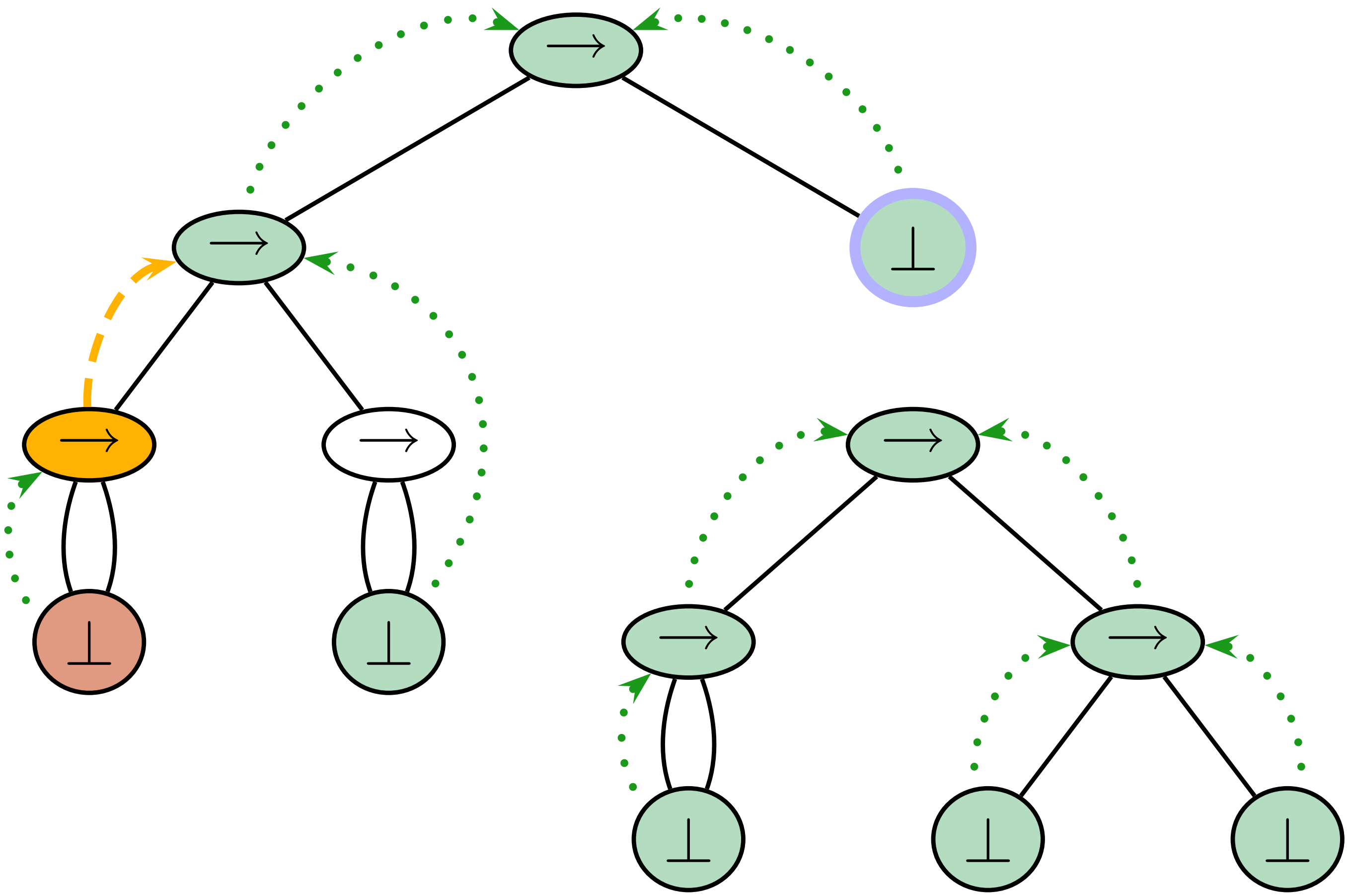


- Forbids some (irreversible) transformations under a node
- Used to require some polymorphism

Full example of instance

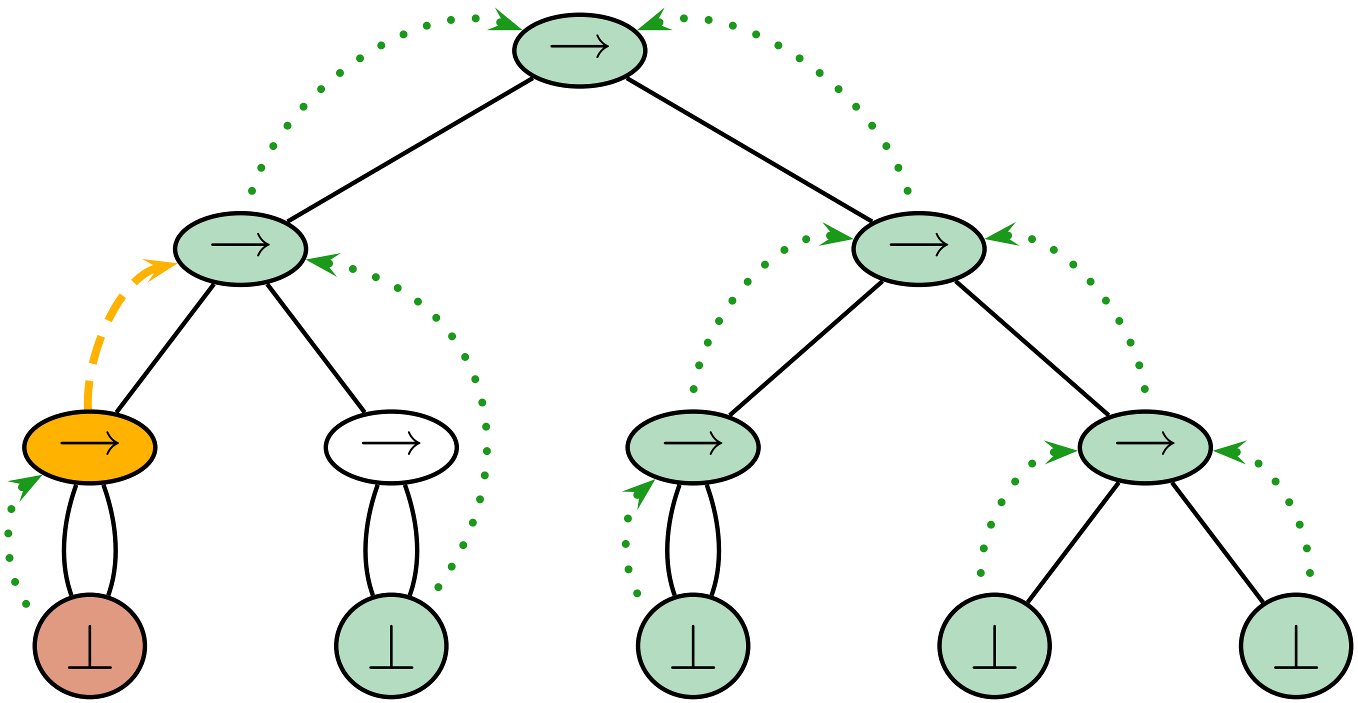


Full example of instance

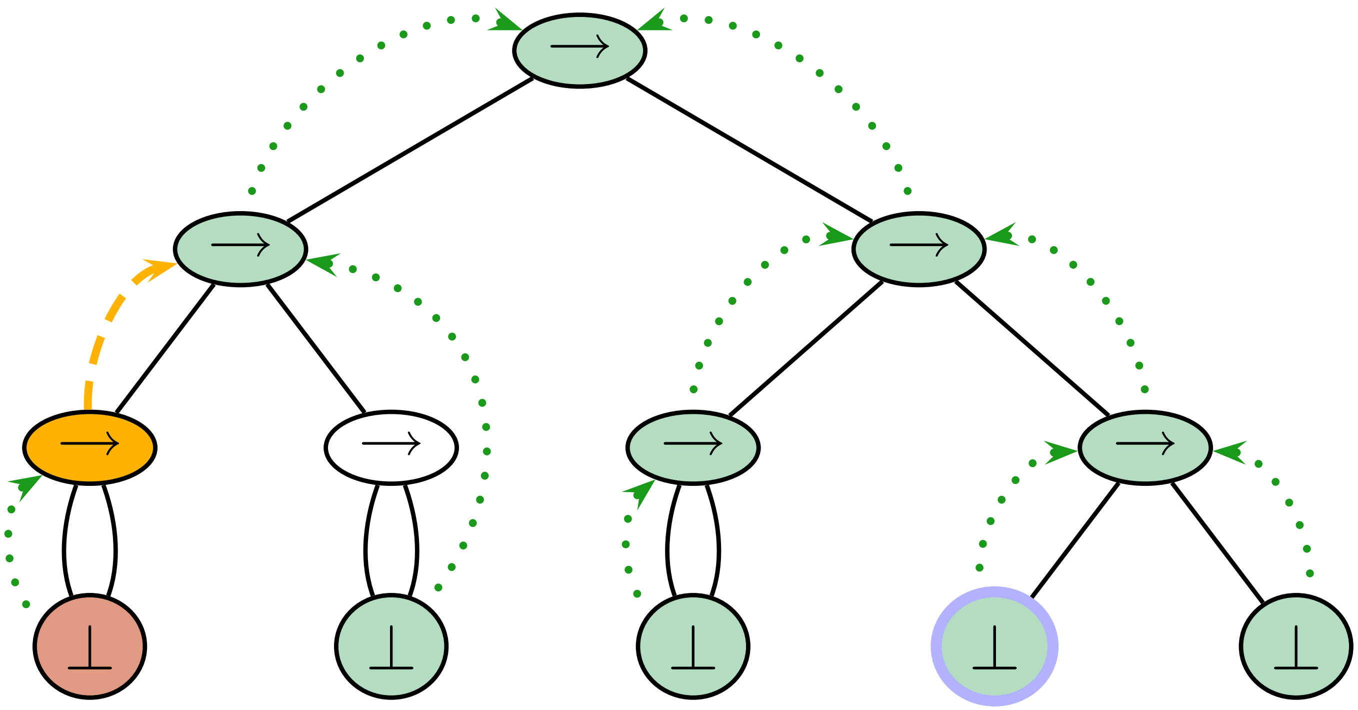


Grafting

Full example of instance

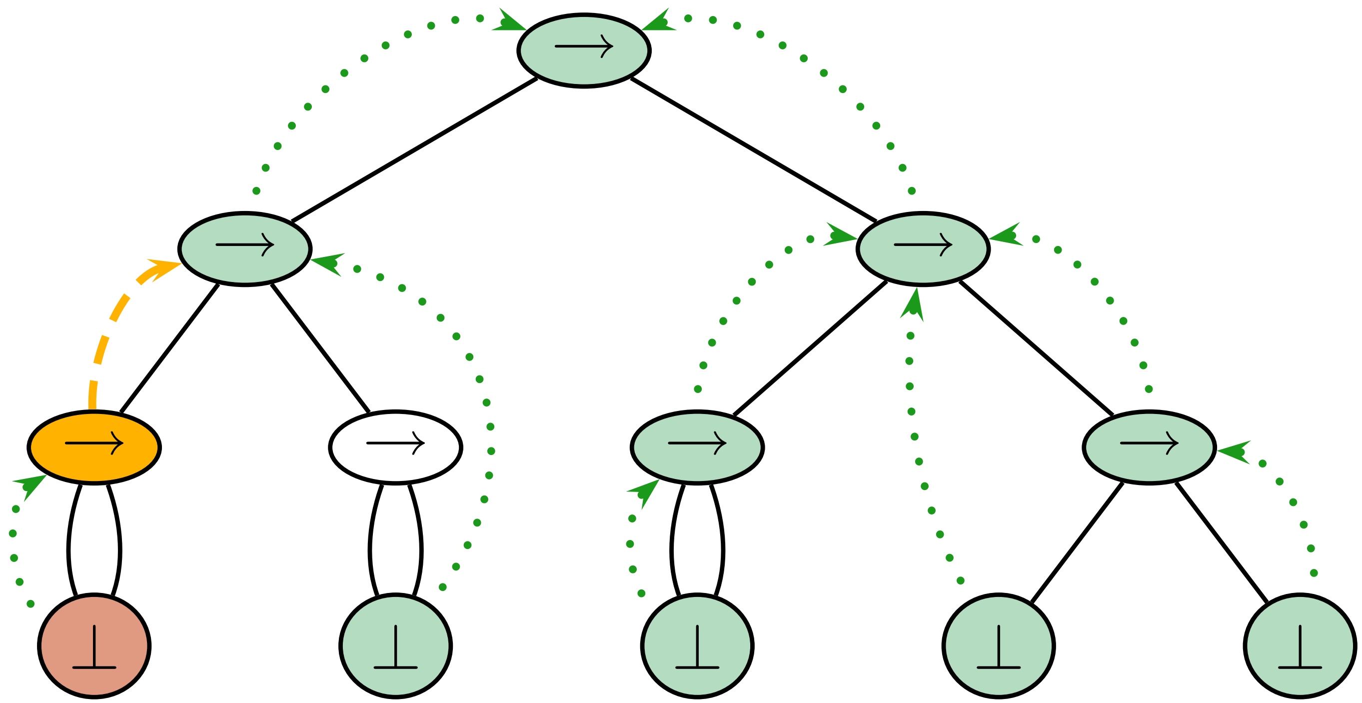


Full example of instance

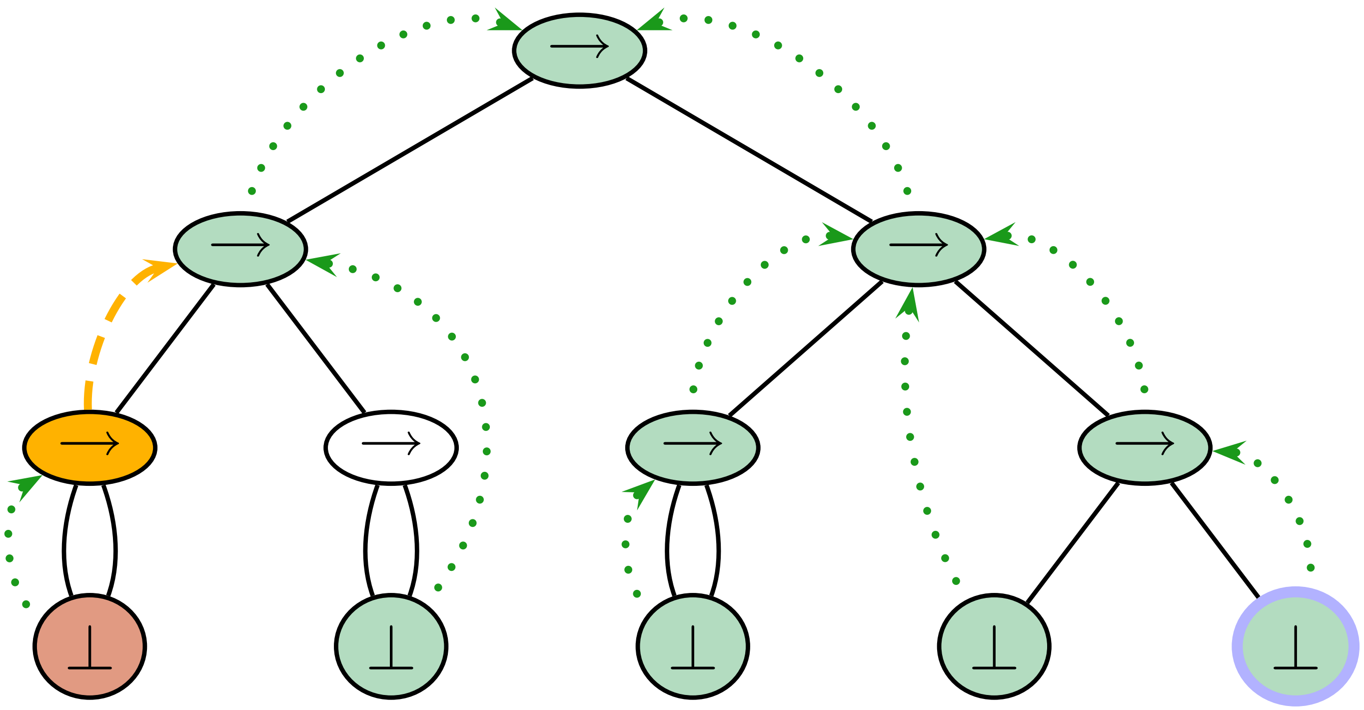


Raising

Full example of instance

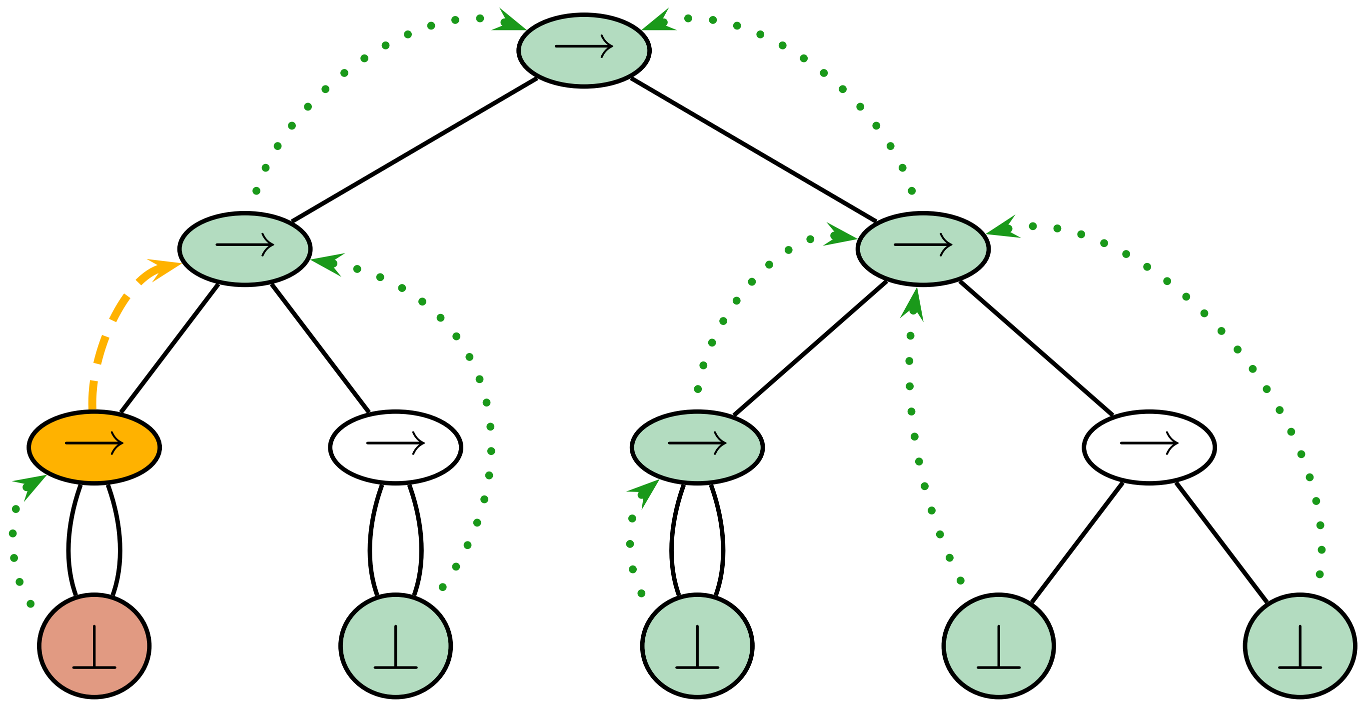


Full example of instance

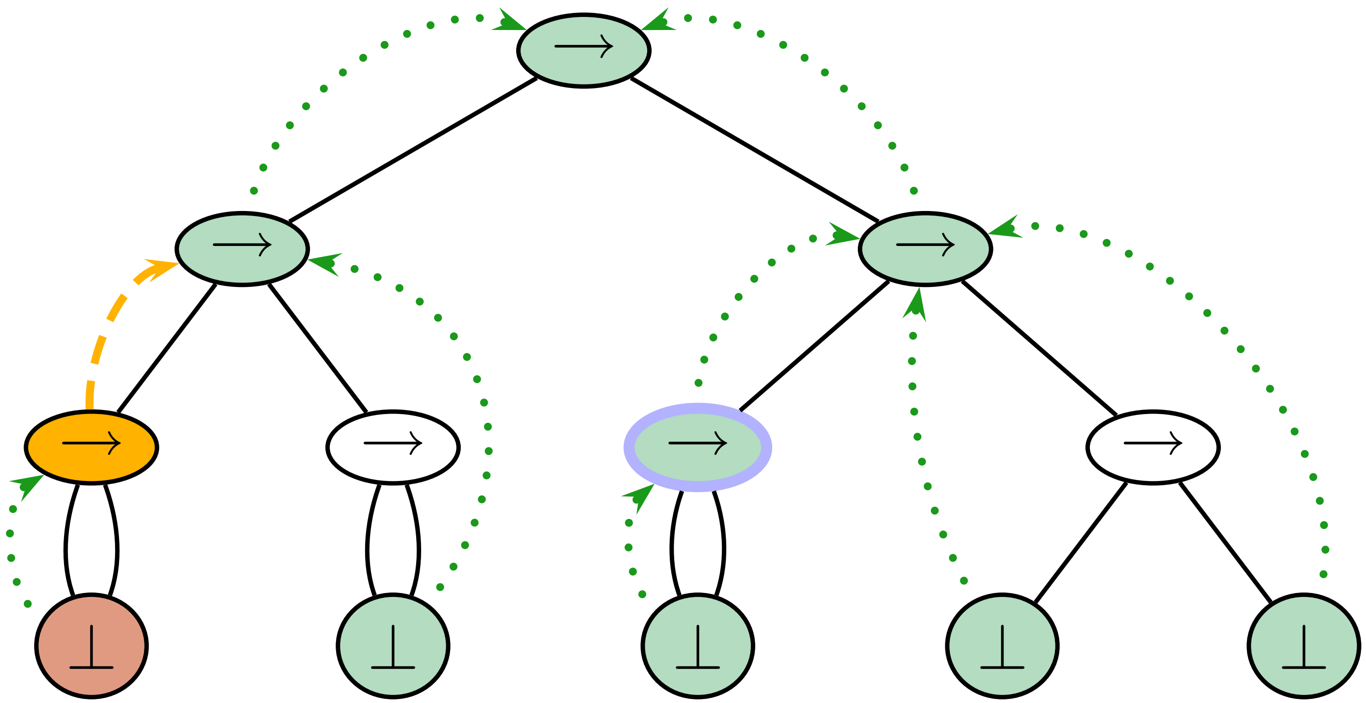


Raising

Full example of instance

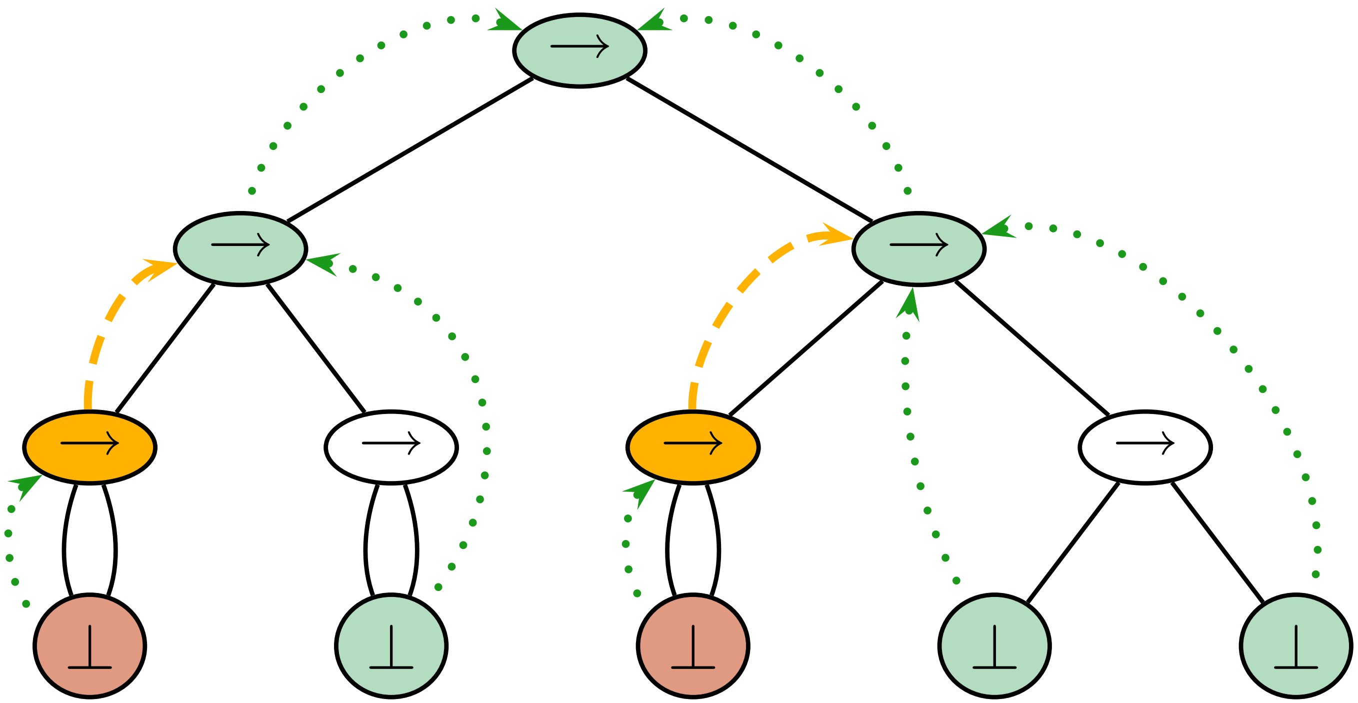


Full example of instance

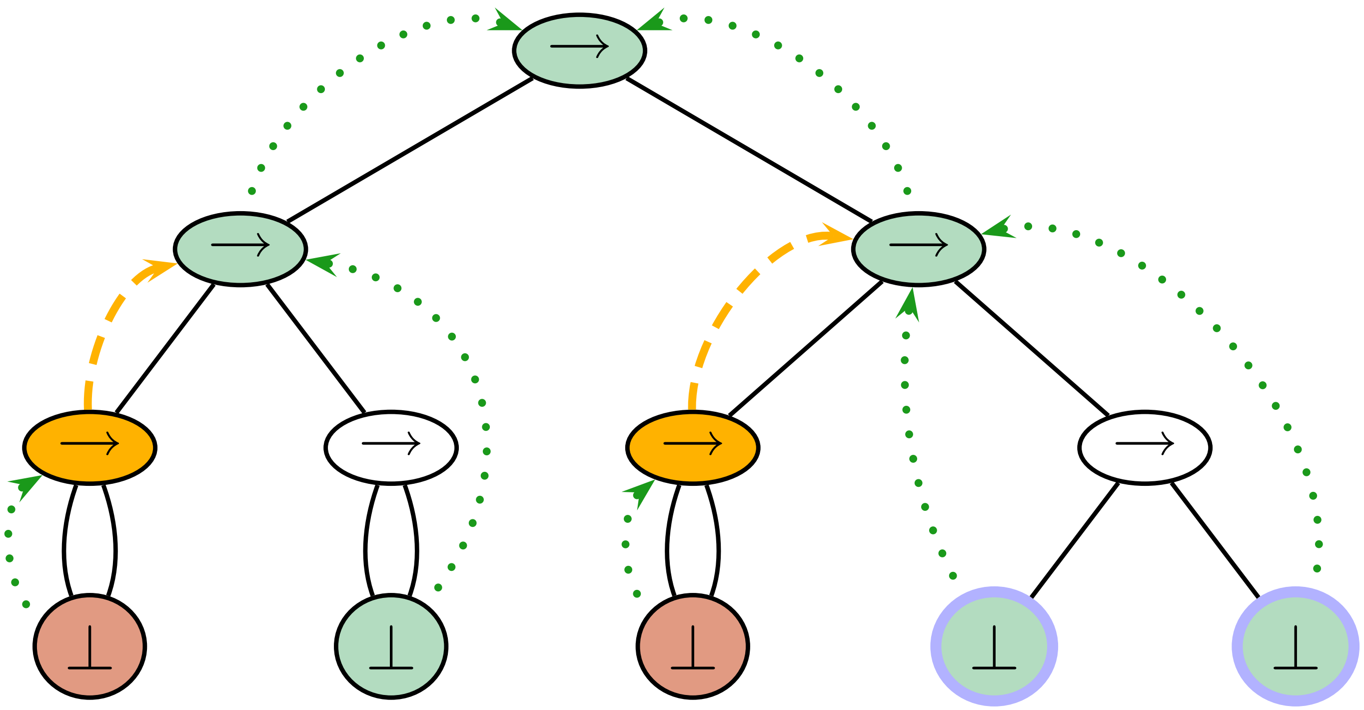


Weakening

Full example of instance

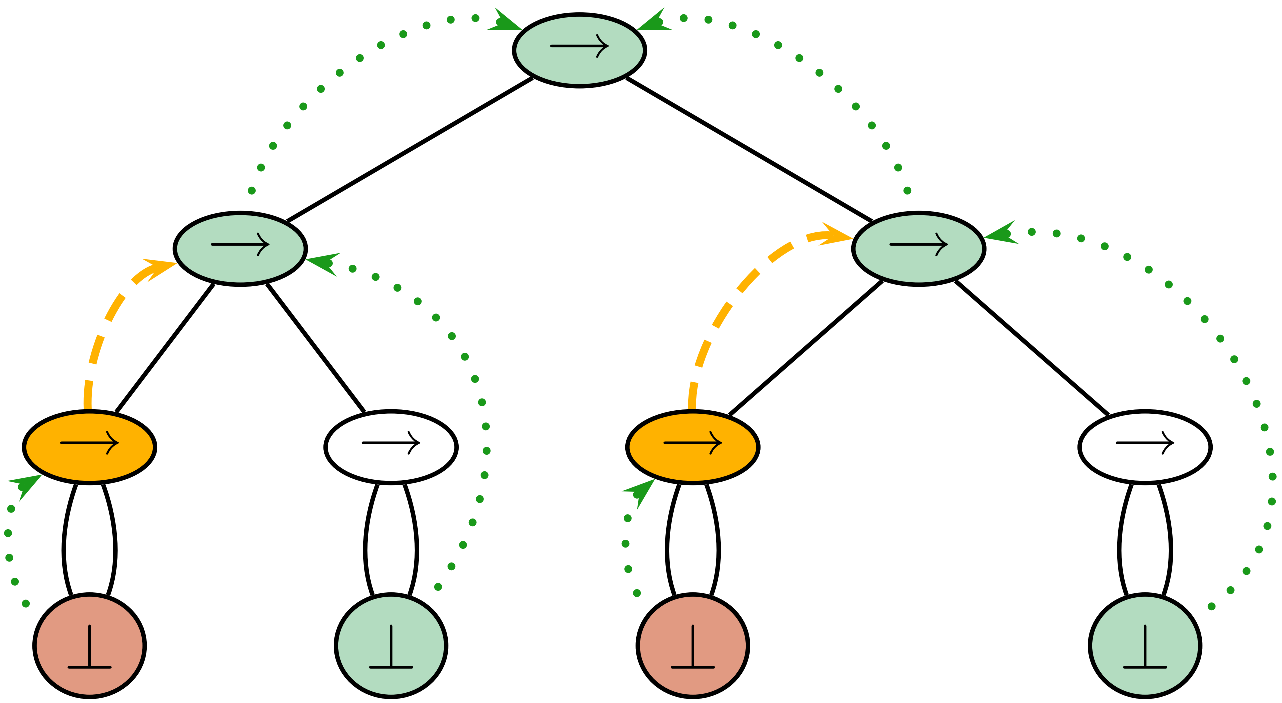


Full example of instance

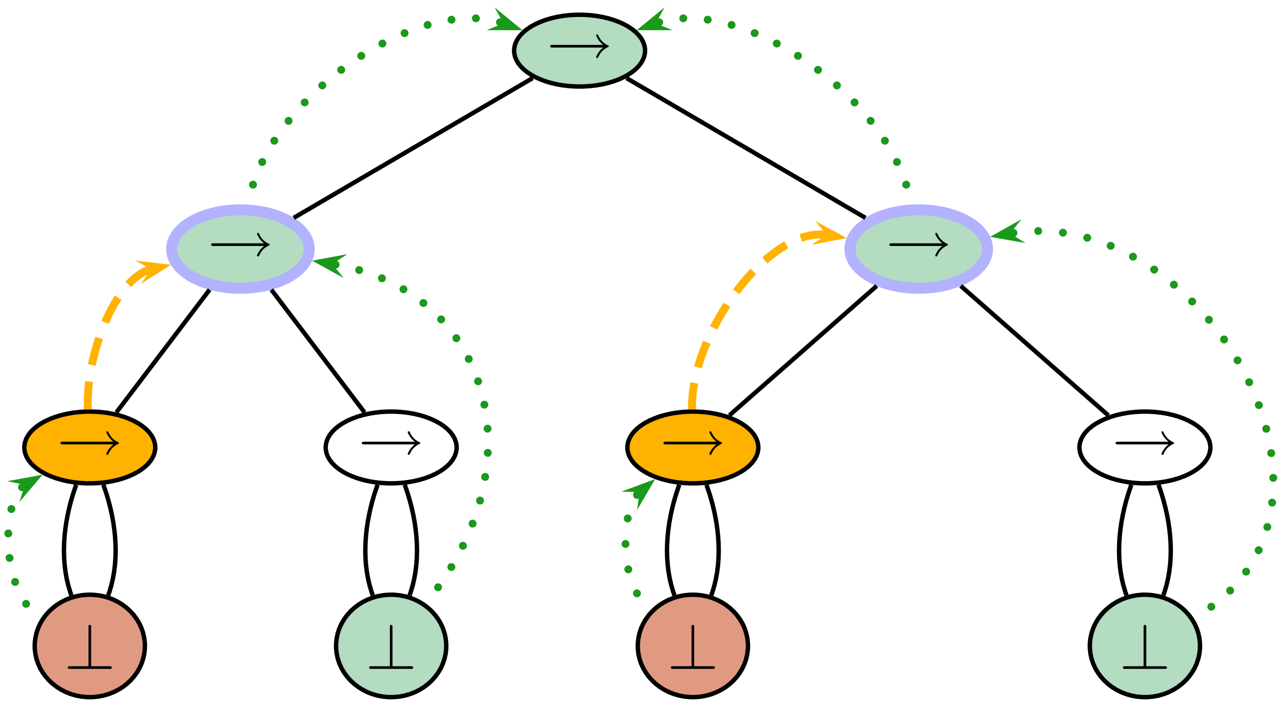


Merging

Full example of instance

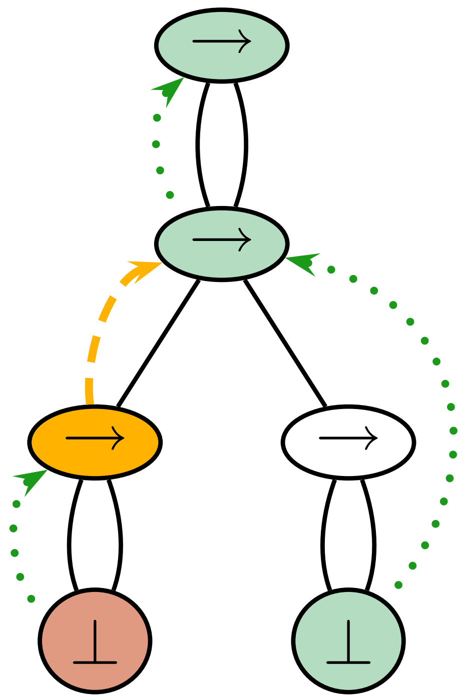


Full example of instance



Merging

Full example of instance



Instance properties

Definition: The instance relation $\sqsubseteq$ is $(\sqsubseteq^G \cup \sqsubseteq^M \cup \sqsubseteq^R \cup \sqsubseteq^W)^*$

Commutation: $\sqsubseteq$ is equal to $\sqsubseteq^G ; \sqsubseteq^R ; \sqsubseteq^{MW}$ ($\sqsubseteq^{MW}$ is $(\sqsubseteq^M \cup \sqsubseteq^W)^*$)

Drastically simplifies proofs and reasonings on instance derivations

Unification

Unification

Unification problem:

Given two types τ_1 and τ_2, find τ_u such that $\tau_1 \sqsubseteq \tau_u$ and $\tau_2 \sqsubseteq \tau_u$

Unification

The unification algorithm proceeds in three steps:

1: Computes the structure of τ_u, by performing first-order unification on the structure of τ_1 and τ_2.

Cost $O(n)$ (or $O(n\alpha(n))$, depending on the algorithm).

Unification

The unification algorithm proceeds in three steps:

1: Computes the structure of τ_u, by performing first-order unification on the structure of τ_1 and τ_2.

2: Computes the binding tree of τ_u.

If the nodes $n_1, ..., n_k$ of τ_1 and τ_2 are merged into n in τ_u:

- 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 least permissive flag on $n_1, \ldots, n_k$.

Cost $O(n)$: a top down visit.

Quite involved step. Uses an amortized $O(1)$ algorithm for computing least-common ancestors.

Unification

The unification algorithm proceeds in three steps:

1: Computes the structure of τ_u, by performing first-order unification on the structure of τ_1 and τ_2.

2: Computes the binding tree of τ_u.

3: Checks the permissions for the merging operations performed in step 1.

Cost $O(n)$, slightly involved visit of τ_1, τ_2 and τ_u.

Unification algorithm

- Sound: τ_u is always an instance of τ_1 and τ_2
- Complete:
  - always returns an unifier if one exists
  - the unifier returned is principal (*i.e.* more general for $\sqsubseteq$) than any other unifier.

  Thus it computes all unifiers
- Good complexity: linear in $\max(|\tau_1|, |\tau_2|)$

  Extension to linear in $\min(|\tau_1|, |\tau_2|)$ in practice

Conclusion

- Simpler relations and proofs
- Presentation more semantic, thanks to permissions.
  - New (relaxed) instance relation.
  - Not easily transposable on syntactic types
- Good complexity for unification

Conclusion

- Simpler relations and proofs
- Presentation more semantic, thanks to permissions.
  - New (relaxed) instance relation.
  - Not easily transposable on syntactic types
- Good complexity for unification

Future works

- Revisit type inference using graphs
- Recursive types
- ...