A graph representation of ML^F types, and a simple, efficient unification algorithm

Didier Rémy, Boris Yakobowski

INRIA Rocquencourt

Why ML^{F}?

ML	System F
Full type inference (Good)	Explicitely typed (undecidable type inference)
Outer ∀ $\forall \alpha \beta\ (\alpha \to \beta) \to \alpha\ t \to \beta\ t$	Inner (1st class) ∀ (Good) $\lambda(f : \forall \alpha . \alpha \to \alpha)(f\ [\mathrm{int}]\ 1, f\ [\mathrm{char}]\ 'c')$

ML^{F}

- Type all ML programs (type inference)
- Encode all System F programs
- Annotations on λ-abstractions whose arguments are polymorphically used (and only them)

Example: type of `choose id`

	System F	ML^F
$\texttt{id} = \lambda x.x$	$\forall\alpha.\ \alpha \to \alpha$	$\forall\,(\alpha \geq \bot)\ \alpha \to \alpha$
`choose`	$\forall\gamma.\ \gamma \to \gamma \to \gamma$	$\forall\,(\gamma \geq \bot)\ \gamma \to \gamma \to \gamma$

In System F, two diffent typings:

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

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

In ML^F:

$$\texttt{choose id}: \quad \forall\,(\beta = \forall\,(\alpha)\ \alpha \to \alpha)\ \beta \to \beta \quad (\sigma_1)$$

$$: \forall\,(\alpha)\ \forall\,(\beta = \alpha \to \alpha)\ \beta \to \beta \quad (\sigma_2)$$

$\sigma = \forall\,(\beta \geq \forall\,(\alpha)\ \alpha \to \alpha)\ \beta \to \beta$ is another principal typing.

ML^{F} with syntaxic types

Syntax of types:

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

Translation from the types of System F:

$$\begin{aligned} [\![\alpha]\!] &= \alpha \\ [\![\forall\alpha \cdot t]\!] &= \forall\,(\alpha \geq \bot)\ [\![t]\!] \\ [\![t_1 \to t_2]\!] &= \forall\,(\alpha_1 = [\![t_1]\!])\ \forall\,(\alpha_2 = [\![t_2]\!])\ \alpha_1 \to \alpha_2 \end{aligned}$$

Instance relation

The instance $\prec$ relation should be as general as possible, while remaining sound and implicit[a]. But full generality and type inference are incompatible[b].

We split $\prec$ in two subrelations $\sqsubseteq$ and $\mathrel{-\!\!\!\sqsubset}$ named instance and abstraction such that :

- $\prec = (\sqsubseteq \cup \mathrel{\sqsupset\!\!\!-})^\star$ is sound
- $\mathrel{-\!\!\!\sqsubset} \subset \sqsubseteq$
- $\sqsubseteq$ is implicit
- $\mathrel{-\!\!\!\sqsubset}$ is explicitely reversible (and should be as small as possible)

Equivalence ($\equiv$) is the kernel of the instance relation. It deals

[a] Needed for type inference

[b] Otherwise, we would get a system as general as System Fwith decidable type inference

with commutations of binders, sharing of monotypes, removal of unnecessary binders ($\forall\,(\alpha = \sigma)\;\alpha \equiv \sigma$)

Difficulty with the current presentation

- Canonical form w.r.t the equivalence relation cannot be preserved by abstraction and instance. Equivalence[a] shows up during proofs.

- The abstraction[b] and instance[c] relations are defined by purely syntactic means, without much support for intuition. They are only justified by the properties of ML^F.

[a] 6 non-trivial rules
[b] 4 rules
[c] 6 rules

Contributions

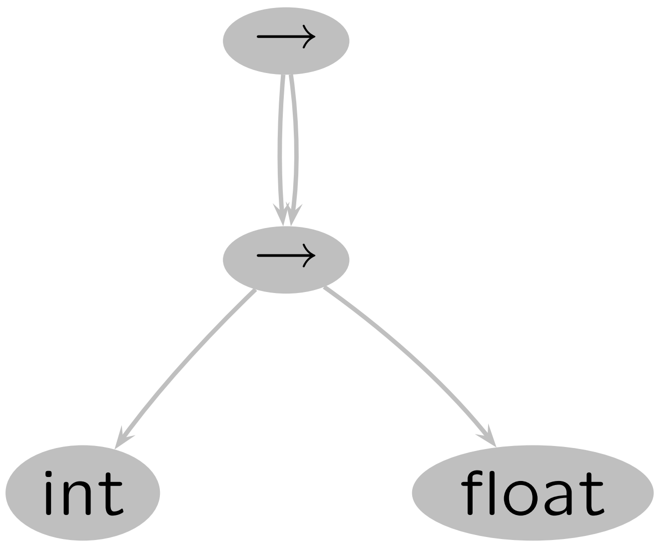
- Graphs have already been proposed as a simpler representation for types, but were not formalized
- Complexity of the unification algorithm is unknown
- Reasoning about ML^F is heavy[a], even though the presentation is not that complicated

[a] Didier Le Botlan's PhD thesis is 320 pages longs

ML^F with graphs

Types are represented by graphs. More precisely, a dag structure represents the skeleton of the type, and a tree the binding structure.

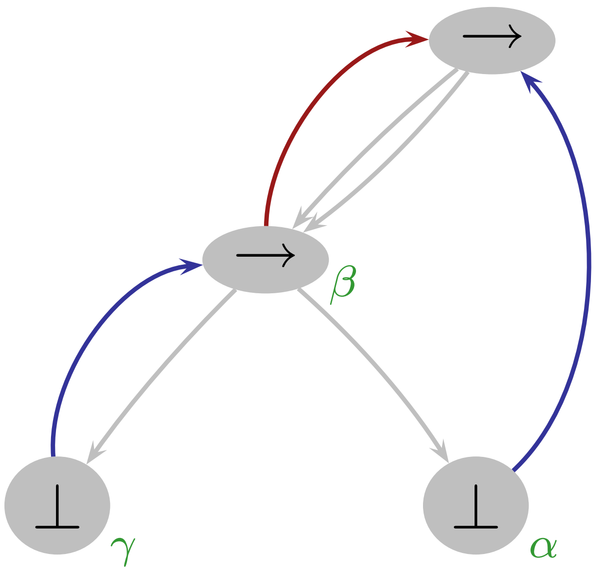
$$\sigma = (\mathtt{int} \to \mathtt{float}) \to (\mathtt{int} \to \mathtt{float})$$



ML^F with graphs

Types are represented by graphs. More precisely, a dag structure represents the skeleton of the type, and a tree the binding structure.

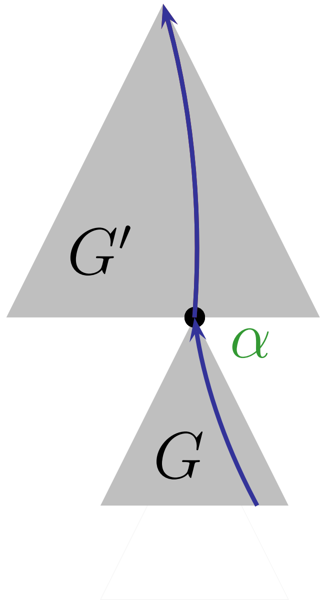
$$\sigma' = \forall\,(\alpha \geq \bot)\;\; \forall\,(\beta = \forall\,(\gamma \geq \bot)\;\; \gamma \to \alpha)\;\; \beta \to \beta$$



Translation from syntaxic graphs

- Translation of monotypes is straightforward
- Translation of $\forall\,(\alpha = \sigma)\ \sigma'$ is inductively defined.
  (same for flexible bounds)

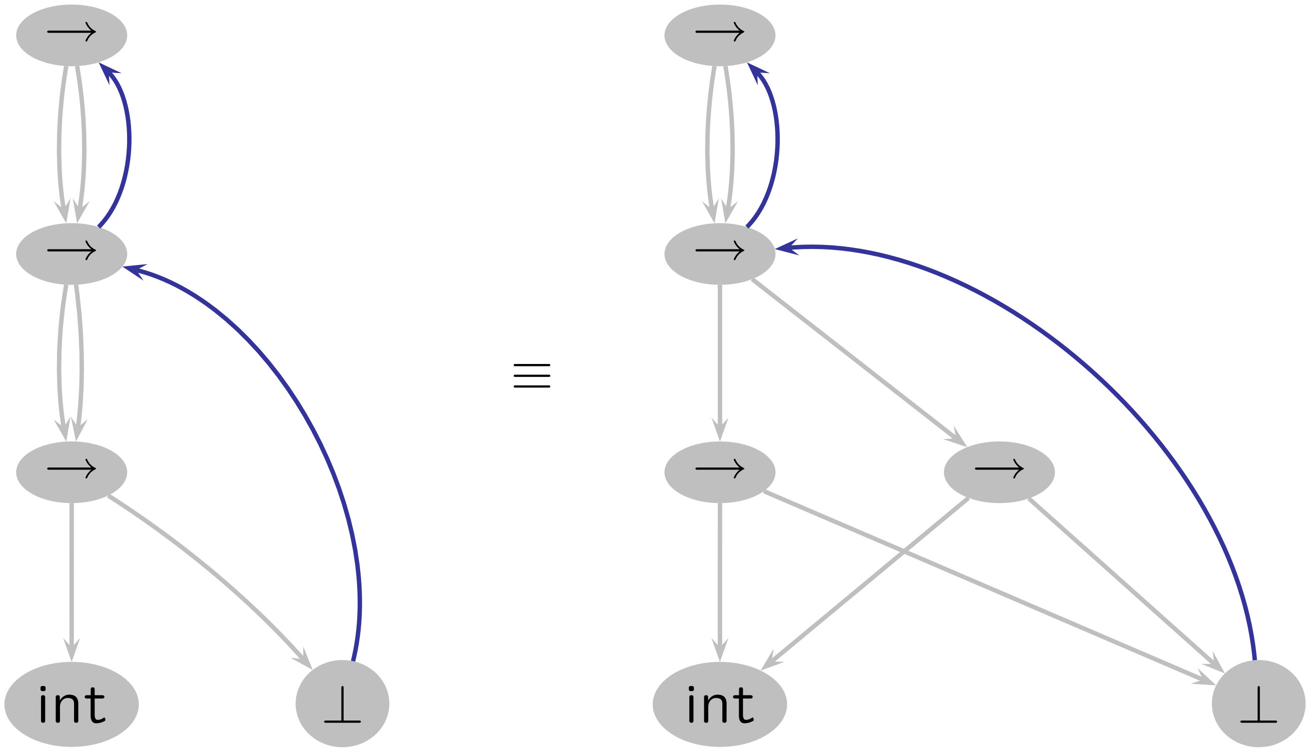
1. Convert σ to G.
2. Convert σ' to G'. α is a free variable of σ', and appears as a free node in G.
3. Join G and G'.
4. Bind the node corresponding to α to the root of G' (if there is polymorphism in G).



Induce some well-formedness conditions for the binding graph.

Equivalence of graphs

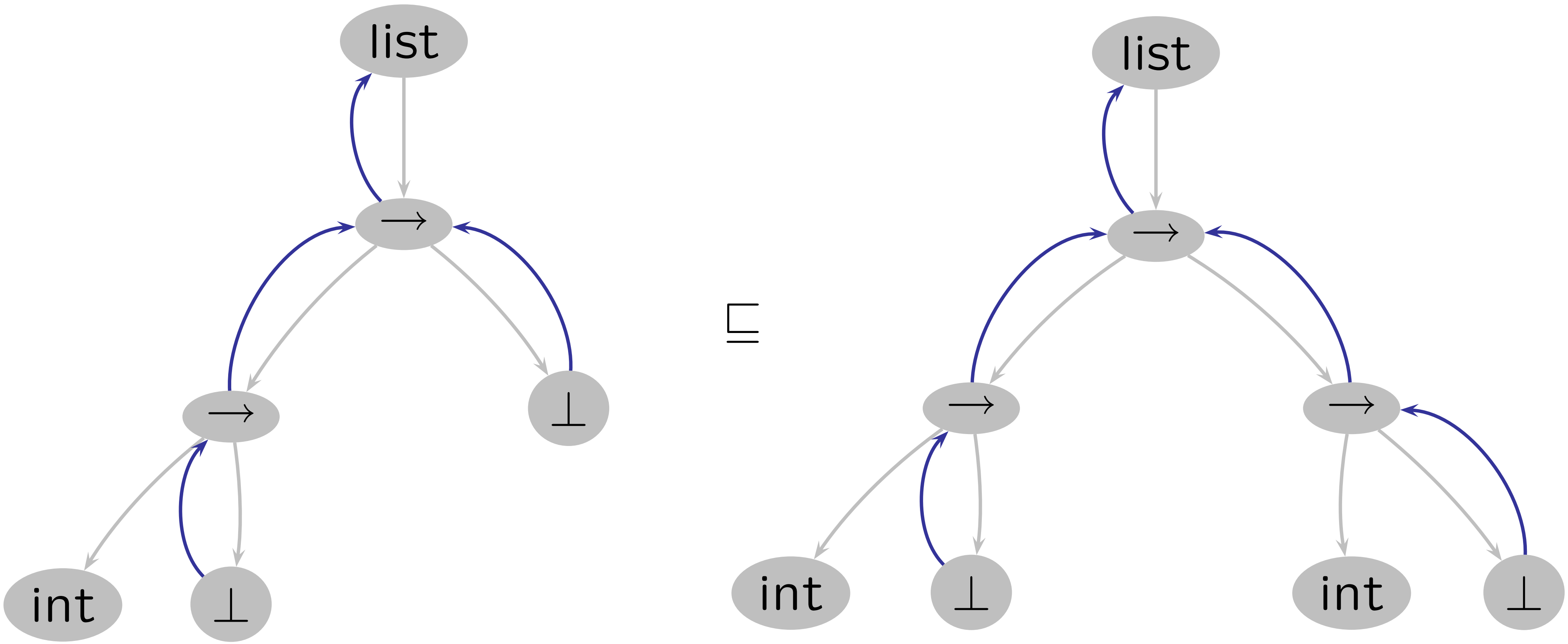
Equivalence on graphs is only sharing or unsharing of monomorphic nodes.



Equivalent syntaxic types are translated to equivalent graphs, and reciprocally.

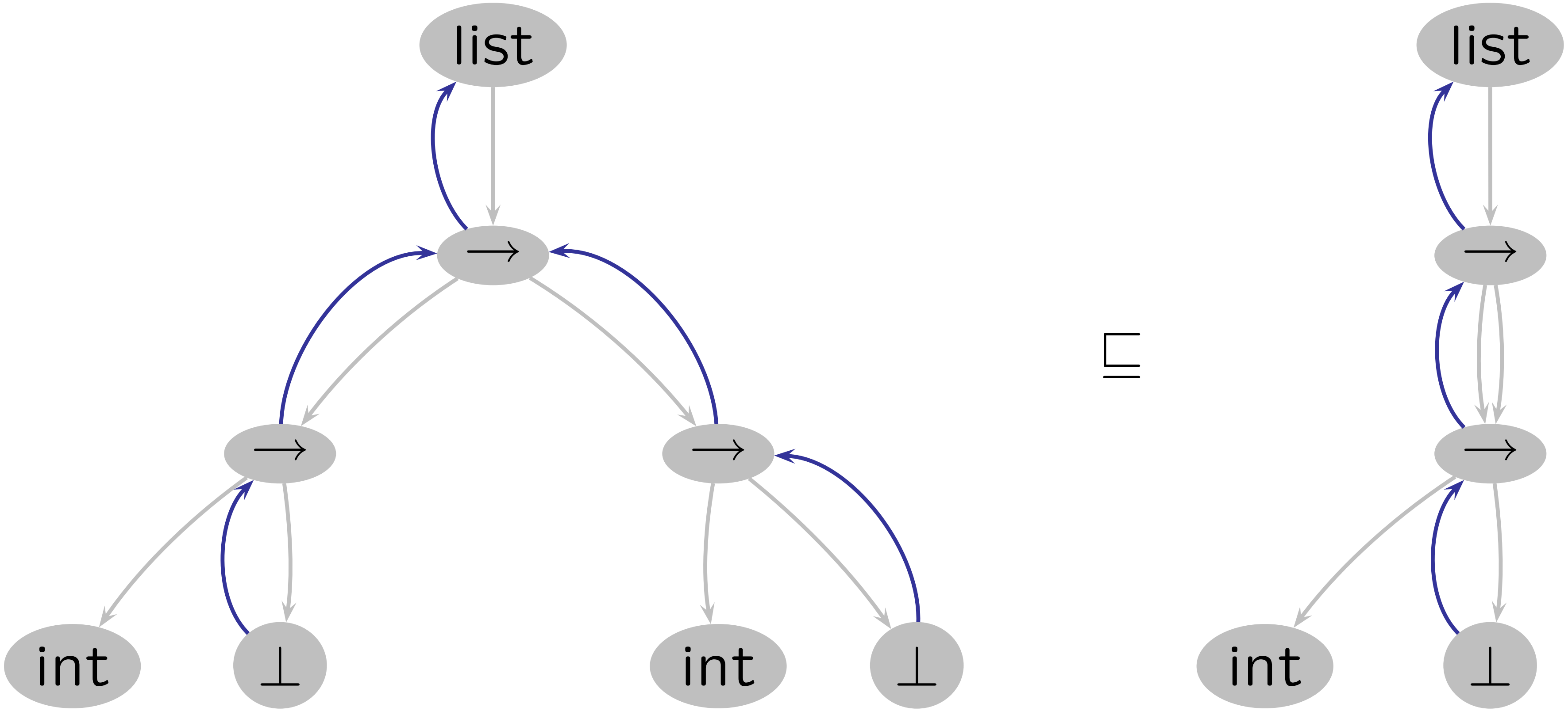
Instance on graphs: Inst rule

The Inst rule allows to replace a ⊥ node by a new graph. It is similar to the standard ML rule for instantiation, except it can introduce more polymorphism.



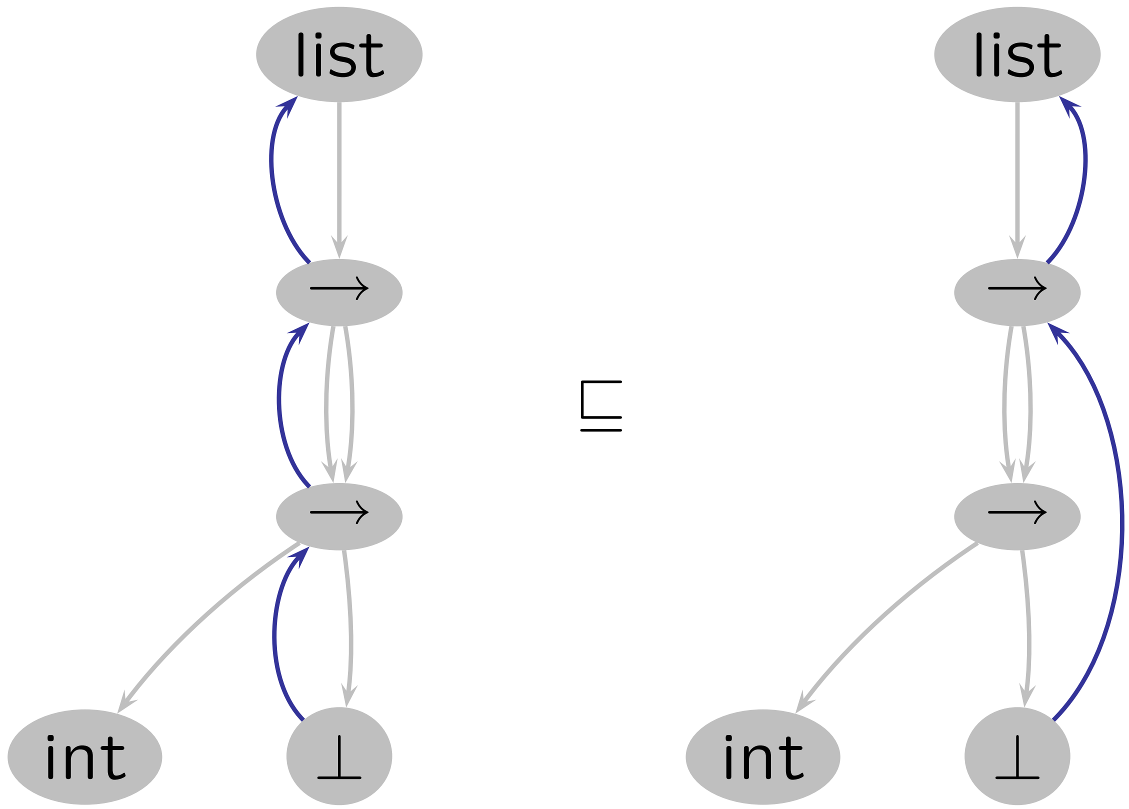
Instance on graphs: Merge rule

The Merge rule allows to merge together two subgraphs bound to the same node which are identical.



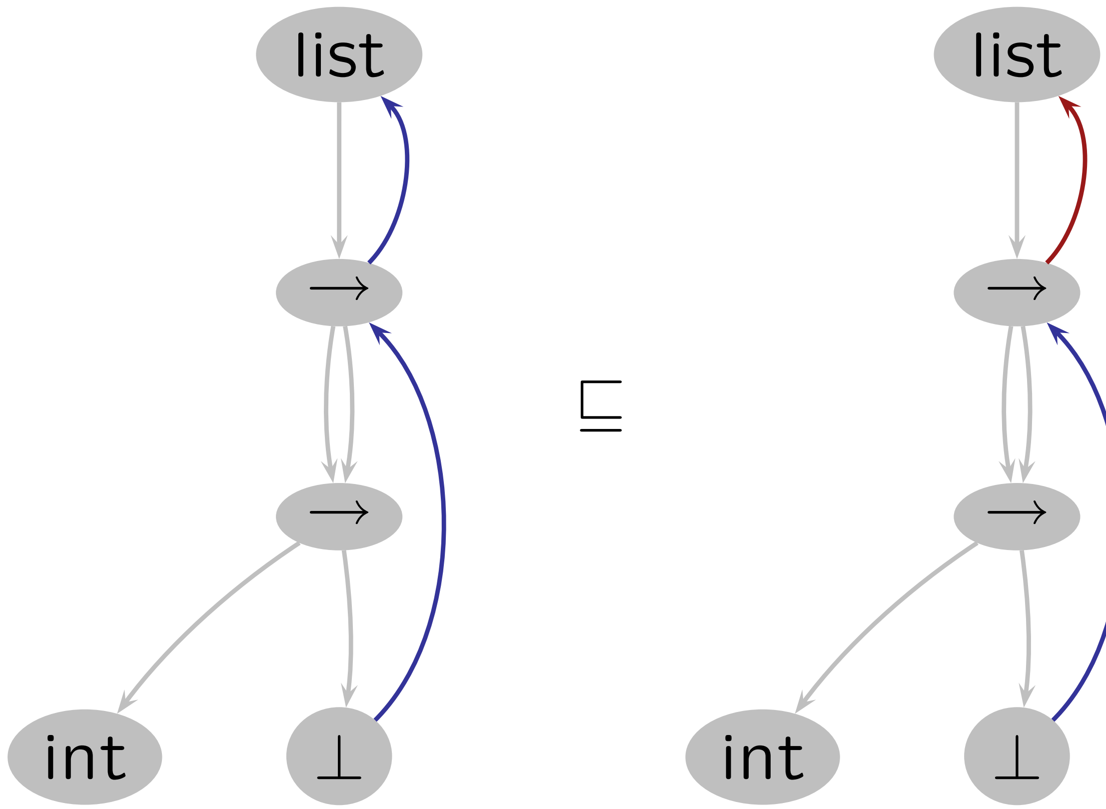
Instance on graphs: Extrude rule

The Extrude rule permits to lift a binder on top of another.
This decreases the rank of the polymorphism of the type.



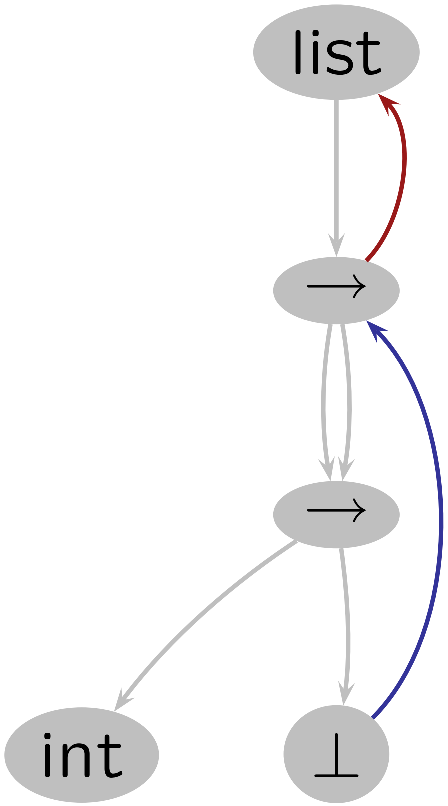
Instance on graphs: Rigid rule

A flexible binder can be turned into a rigid one through the Rigid rule.

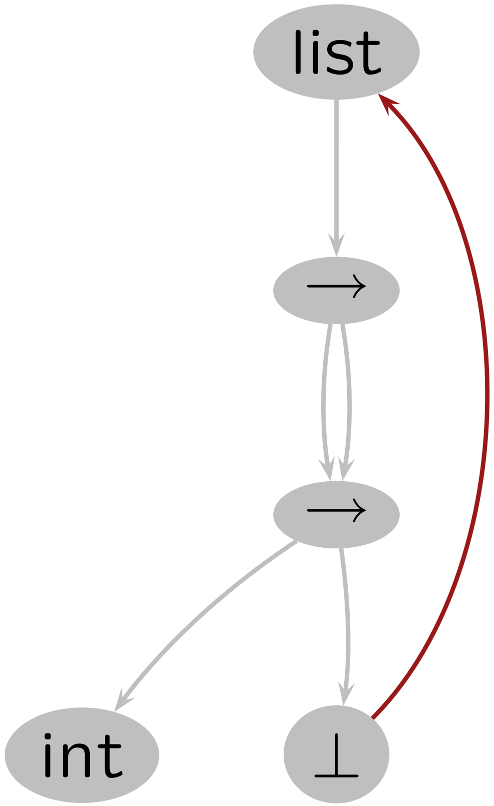


Restrictions to instance

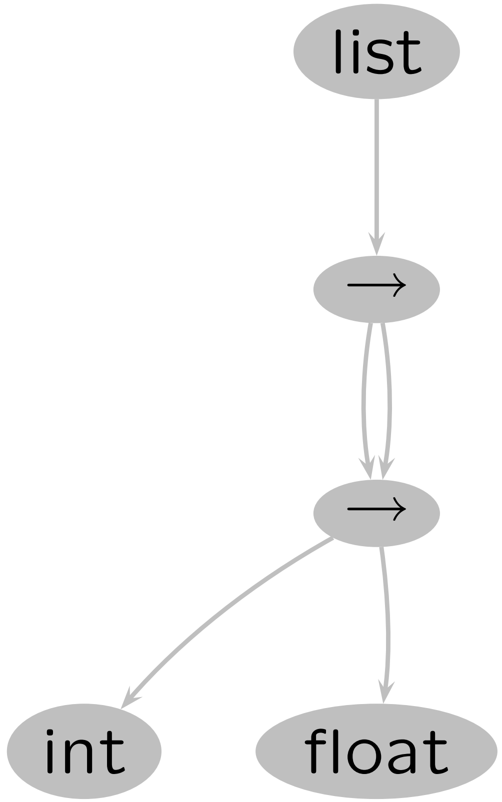
Instantiation through any of the previous rules can only happen on nodes which are at a flexible path from the root of the graph. Instantiation under a rigid bound is forbidden.



$\not\sqsubseteq$



$\not\sqsubseteq$



Instance: properties

Instance also includes Abstraction, which permits Merge and Extrude for nodes which are at a path allowing abstraction.

$\Rightarrow$ Instance includes exactly 6 different rules, which can be seen as conditional rewriting steps on graphs.

Instance on graphs and instance on syntaxic types permit to derive the same judgments (w.r.t the conversion functions).

Rules of an instance derivation can be ordered:

1. All Inst steps
2. All Extrude steps
3. Merge and Rigid intermingled

Unification algorithm

Given G_1 and G_2, we want to find the most general graph G such that $G_1 \sqsubseteq G$ and $G_2 \sqsubseteq G$.

1. First-order unification of the structure graph of G_1 and G_2. Gives the skeleton S of G.
2. Bind the nodes of S, using the binders of G_1 and G_2. This gives a possible G.
3. Check that G is indeed an instance of G_1 and G_2 (in fact, only the uses of the Merge rules). If not, there is no unifier.

Unification: properties

- Sound algorithm (always returns an instance of G_1 and G_2)
- We are currently proving principality
- Good complexity: linear in the sizes of the input graphs.
- For ML types, the algorithm simplifies to the standard 1st-order unification algorithm.

Conclusion

- More readable types
- Simpler proofs and rules
- Presentation more intuitive[a] and more canonical
- Complexity of the unification algorithm is now known

[a] Rules are what one would expect on graphs