Graphic Type Constraints and Efficient Type Inference: from ML to ML^F

Didier Rémy
INRIA
http://gallium.inria.fr/~remy

Boris Yakobowski
INRIA
http://www.yakobowski.org

Abstract

ML^F is a type system that seamlessly merges ML-style type inference with System-F polymorphism. We propose a system of graphic (type) constraints that can be used to perform type inference in both ML or ML^F. We show that this constraint system is a small extension of the formalism of graphic types, originally introduced to represent ML^F types. We give a few semantic preserving transformations on constraints and propose a strategy for applying them to solve constraints. We show that the resulting algorithm has optimal complexity for ML^F type inference, and argue that, as for ML, this complexity is linear under reasonable assumptions.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism, constraints; D.3.2 [*Language Classifications*]: Applicative (functional) languages; E.1 [*Data Structures*]: Graphs

General Terms Algorithms, Design, Languages, Theory

Keywords System F, ML^F, ML, Unification, Type Inference, Types, Graphs, Type Constraints, Type Generalization, Type instantiation, Binders.

Introduction

ML^F [2] is a type system that combines the power of first-class, System-F-style polymorphism with the convenience of ML type inference. ML^F is a conservative extension of ML. In particular, all ML terms are typable in ML^F. Moreover, the full power of first-order polymorphism is also available, as any System-F term can be typed by using type annotations (containing second-order types). Still, as in ML, all typable expressions having principal types. Moreover, the set of well-typed programs is invariant under a wide class of program transformations, including let-expansion, let-reduction, η-expansion of functional expressions, reordering of arguments, curryfication, and also "abstraction of applications", which means that $a_1\ a_2$ is typable if and only if $apply\ a_1\ a_2$ is (where $apply$ is $\lambda(f)\ \lambda(x)\ f\ x$). Furthermore, only lambda-bound arguments that are used polymorphically need an annotation; this makes it very easy for the user to predict where and which annotations to write. Finally, ML^F is an impredicative type system, which allows embedding polymorphism inside containers; for example, $(\forall(\alpha)\ \alpha \to \alpha)\,\mathsf{list}$ is a valid type, quite different from the weaker $\forall(\alpha)\ ((\alpha \to \alpha)\,\mathsf{list})$. A full comparison between ML^F and other extensions of System F can be found in [3].

Unfortunately, the power of ML^F has a price. ML^F types are more general than System-F types, making them look unfamiliar. The original syntactic presentation of ML^F [2] is also quite technical, and most extensions of the system in this form would require a large amount of work. Finally, the type inference algorithm based on syntactic types has obvious sources of inefficiencies and we believe that it would not scale up well to large, possibly automatically generated, programs.

Graphic types have been introduced as a simpler alternative to the original syntactic types, in order to solve all three issues [9]. In this work, we extend graphic types to address the question of type inference. We do not adapt the existing type inference algorithm [2] by replacing its unification algorithm on syntactic types with the new, more efficient unification algorithm on graphic types [9]: repeatedly translating to and from graphic types would be both inelegant and inefficient, loosing the quite compact representation of graphic types. Moreover, we believe that the graphic presentation is better suited for studying the meta-theoretical properties of ML^F.

Instead, we propose an entirely graphical presentation of type inference. Additionally, we highlight the strong ties between ML^F and ML by parameterizing our type inference system with the actual set of types that is being used, rediscovering a known efficient type inference algorithm for ML. Our approach is also constraint-based, hence more general than just a particular type inference algorithm: we introduce a set of graphic constraint constructs, and define typing constraints in term of those.

Our contributions are as follows:

- We propose a small set of *graphic constraints*, featuring generalization scopes, existential nodes, unification and instantiation edges. We encode typing problems in terms of those, by defining a compositional translation from λ-terms to constraints.
- We show that this system can be seen as a small generalization of the formalism of graphic types.
- We show that our constraint system is in fact implicitly parameterized by the type system considered and the operation of taking an instance of a type scheme. We make this last operation explicit for both ML and ML^F, and (re)prove that ML is a subsystem of ML^F.
- We give a semantics to our constraints, first by defining what it means for a constraint to be solved, and then as a set of types. We link solved constraints and fully decorated λ-terms.

- We identify a set of *acyclic* constraints, that include all typing constraints, and have decidable principal solutions.
- We study the theoretical complexity of solving typing constraints and show that under reasonable assumptions, type inference in MLF has linear complexity—as in ML. We also observe that our algorithm has optimal complexity for both ML and MLF type inference.

Outline of the paper We introduce a graphic presentation of ML types, extend it to graphic constraints, and define a translation from source expressions to constraints (§1). We give a brief overview of MLF and graphic types, and show that graphic constraints are an extension of graphic types (§2). We define what it means for a graphic constraint to be solved, both in ML and MLF (§3), and present sound and complete transformations on constraints (§4). We show that a large class of constraints have principal solutions and introduce a strategy to reduce any such constraint to an equivalent one in solved form (§5). We discuss type annotations in MLF (§6). We show that our strategy for solving constraints leads to an efficient implementation of type inference (§7). We present a few examples of typings in §8 and discuss related works in §9.

An online prototype MLF typechecker and an extended version of this paper with all proofs are available online at `http://gallium.inria.fr/~remy/mlf/`.

1. Graphic types and constraints

1.1 ML Graphic types

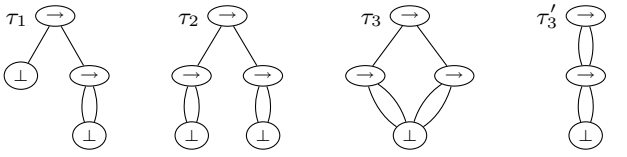


Figure 1. Graphic ML types

ML graphic types are first-order (quantifier free) *term dags*. As with first-order terms, every node is labeled with a symbol, the arity of which determines the number of its successors. Symbols contain at least the arrow $\rightarrow$ of arity 2. Variable nodes are labelled using a pseudo-symbol $\bot$ of arity 0. However, in first-order term dags (as opposed to first-order terms), nodes may also be shared, *i.e.* there may be different *paths* leading to the same node. Paths are sequences of integers that are used to designate nodes. The empty path ϵ designates the root node. If $\overline{k}$ designates node n, $\overline{k} \cdot j$ designates the j'th successor of n. We often leave $\cdot$ implicit and write 121 instead of $1 \cdot 2 \cdot 1$. For illustration, consider the type τ_1 of Figure 1. The rightmost lowermost node (which is labeled with $\bot$) can be designated by either path 21 or path 22: this is a shared node. We write $\langle \pi \rangle$ for the node designated by path π. An edge between nodes n and n' in τ is written $n \circ\!\!\longrightarrow_\tau n'$ or $n \circ\!\!\longrightarrow n' \in \tau$,r simply $n \circ\!\!\longrightarrow n'$ when τ may be left implicit from context. For example, $\langle 2 \rangle \circ\!\!\longrightarrow_{\tau_1} \langle 21 \rangle$.

In ML graphic types, only sharing of variable nodes is significant: sharing of inner nodes, such as $\langle 1 \rangle$ in type τ_3', is not. Thus, ML graphic types may always be unfolded and read back as trees. However, before doing so, bottom nodes must all be relabeled, each with a different type variable, so that all occurrences that were shared in the graph representation become the same type variable in the unfolding. For instance, the skeleton of the type τ_1 in Figure 1 represents the ML type $\alpha \rightarrow (\beta \rightarrow \beta)$. Similarly, τ_2 represents $(\alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \beta)$, while both τ_3 and τ_3' represent $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$.

Type instance $\leqslant$ on ML graphic types captures almost entirely the corresponding instance relation on ML types. In particular, $\tau_1 \leqslant \tau_2 \leqslant \tau_3 \leqslant \tau_3'$ holds. However, $\leqslant$ is oriented so that it allows only more sharing; thus $\tau_3' \leqslant \tau_3$ does not hold, even though the ML types they represent are equal. This permits a simpler definition of $\leqslant$, thus simpler reasonings[1]. We then prove that all our results hold when types are equal up to a *similarity* relation $\approx$ that captures sharing of inner nodes (*i.e.* $\tau_3 \approx \tau_3'$ holds).

Notice that $\leqslant$ can be decomposed into two more atomic relations, *grafting* and *merging*. Grafting adds a subgraph under a variable node. For example, τ_2 is obtained from τ_1 by grafting under $\langle 1 \rangle$ the graphic type representing $\gamma \rightarrow \gamma$. Merging shares some nodes, which need not be variable nodes. For example, τ_3 results from sharing nodes $\langle 11 \rangle$ and $\langle 21 \rangle$ in τ_2, while τ_3' is obtained by sharing $\langle 1 \rangle$ and $\langle 2 \rangle$ in τ_3.

1.2 (Graphic) type schemes and generalization

Central to ML type inference is the notion of generalization:

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \text{ does not appear free in } \Gamma}{\Gamma \vdash e : \forall\,(\alpha)\ \tau}\ \textsc{Gen}$$

We will use the same mechanism in graphic type inference. To this effect, we introduce a new type constructor G of arity one. A G-node has a double role:

- It is used to indicate where polymorphism is introduced. For that purpose, we introduce a *binding edge*[2] leaving from each bound variable to the G-node where it is bound. Thus a G-node can be seen as representing a *type scheme*.
- It will be safe to generalize any variable node introduced in the scope of a G-node at the level of this node. Thus, G-nodes can be seen as modeling *generalization scopes*.

G-nodes will in particular be used to type let constructs, and we need to be able to to nest them. However, they belong to the constraint and should not be part of the type structure. Both requirements can be fulfilled by binding every G-node to the outer G-node where it has been introduced and accessing it only through its binding edge (except for the toplevel node, at the root).

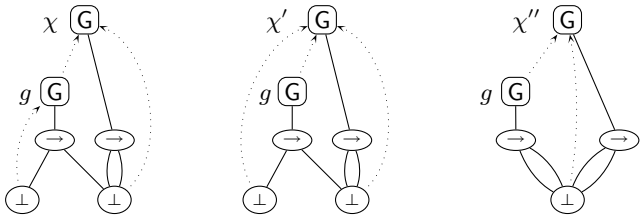


Figure 2. Constraints with generalization scopes

Figure 2 shows three constraints, each containing two G-nodes, the root $\langle \epsilon \rangle$ and the node g, bound at $\langle \epsilon \rangle$. We extend the syntax of paths to allow named nodes such as g. For example, in all three constraints the rightmost lowermost bottom node can be designated by either $\langle g \cdot 1 \cdot 2 \rangle$, $\langle 1 \cdot 1 \rangle$ or $\langle 1 \cdot 2 \rangle$. In the figures, binding edges are dotted oriented lines. In the text, we use $n \succ\!\!\longrightarrow_\chi g$ or $n \succ\!\!\longrightarrow g \in \chi$o say that n is bound at g in χ (we often leave χ implicit from context). Given a node n in χ, there is at most one g

[1] This also makes our definitions closer to (usual) implementations, which use a union-find based representations of types.

[2] Using binding edges instead of a sequence of explicit $\forall$ nodes have many advantages; in particular we gain commutation of adjacent binders and removal of useless quantification for free.

such that $n \succ\!\!\longrightarrow g$, called the *binder* of n, and written $\overleftarrow{n}$. In all three constraints of Figure 2, we have $g \succ\!\!\longrightarrow \langle\epsilon\rangle$ and $\langle 11\rangle \succ\!\!\longrightarrow \langle\epsilon\rangle$. Notice that binding edges do not count in arities: in χ, $\langle 1\rangle$ is the rightmost arrow node, not g.

The node g of constraint χ represents the type scheme $\forall(\alpha)\ \alpha \to \beta$, where β is a free variable represented by the node $\langle 11\rangle$ that is bound above g; conversely, the node $\langle g11\rangle$ representing α is bound at g. By contrast, in the constraint χ', both variables are bound above g, hence g represents the type $\alpha \to \beta$, which is monomorphic in the context of g. The root node represents the same type $\forall(\beta)\ \beta \to \beta$ in all three constraints.

G-nodes can only appear under the G-node to which they are bound. Constraints are therefore stratified: all G-nodes are at the top-most part of χ, above the *type* nodes of the constraint.

The instance relation $\leqslant$ on ML graphic types can be extended to an instance relation $\sqsubseteq$ on graphic constraints as follows: we allow any transformation along $\leqslant$ at every type node, except that nodes can only be merged if they have the same bound. In parallel, we introduce a third instance operation that consists in *raising* a binding edge along another one, *i.e.* replacing the bound s of a node n by the bound of s. This results in extruding the polymorphism to the enclosing generalization scope. Readers familiar with rank-based ML type inference [7, 8] can recognize the similarity between raising and adjusting the ranks of two variables about to be unified.

As an example, consider nodes $\langle g11\rangle$ and $\langle g12\rangle$ in Figure 2. In χ, they cannot be merged. However, node $\langle g11\rangle$ can be raised, resulting in the constraint χ'. The merging is now possible, and results in the constraint χ''. In summary, we have $\chi \sqsubseteq \chi'$ and $\chi' \sqsubseteq \chi''$, and therefore $\chi \sqsubseteq \chi''$ by transitivity.

1.3 Constraint edges and existential nodes

In order to perform type inference, we only need three more constructs: unification and instantiation constraints (both being modelized using *constraints edges*) and existential nodes.

A unification edge n_1 ⟨----⟩ n_2 links two type nodes and means that n_1 and n_2 should be unified. Unification edges whose two extremities are the same node are implicitly removed.

An instantiation edge g ----▸ n relates a G-node g to a type node n. It requires the type under n to be an instance of the type scheme represented by g. The exact definition of "being an instance" will be given in §3.1.

Existential nodes are type nodes that are only part of the constraint structure. Usually they are nodes in which we are not interested *per se*, but only indirectly, in order to constrain other nodes. For example, the typing of an application $a_1\ a_2$ requires a_1 to be arrow type τ such that the domain of τ is also the type of a_2. However, we are eventually only interested in the type resulting from the application, *i.e.* the codomain of τ. We thus introduce the arrow node of τ as an existential node.

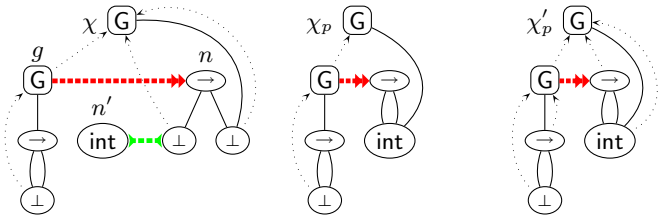


Figure 3. Typing id 1

Examples of constraints are given in Figure 3. The constraint χ is the typing of id 1, where id is the identity function. The leftmost G-node g represents the type scheme $\forall(\alpha)\ \alpha \to \alpha$ of id. The root G-node represents the typing constraint for an application, as explained above. In particular, n is an existential arrow node constrained (through an instantiation edge) to be an instance of the G-node s. Finally, n' is an existential node that represents the type int of 1 and constrains (through a unification edge) the domain of n to be an integer.

Neither the instantiation nor the unification constraints are solved in χ. The unification constraint can be satisfied by grafting the type int under $\langle n1\rangle$ and merging this node with n'. The instantiation constraint can be solved by taking as an instance of $\forall(\alpha)\ \alpha \to \alpha$, the identity $\beta \to \beta$ itself, and unifying this type with n (thus merging $\langle n1\rangle$ and $\langle n2\rangle$). The resulting constraint is depicted by χ_p. In particular, the type of the application is the type scheme represented by $\langle\epsilon\rangle$, in this case the ground type int.

About unbound nodes So far, we have only bound variable nodes and G-nodes. However, this approach lacks some homogeneity, and we instead choose to bind all nodes explicitly to the G-node they belong to. A fully-bound version of χ_p in Figure 3 is χ'_p.

1.4 Putting it all together: typing constraints

Let x range over a denumerable set of variables. Expressions are those of the λ-calculus enriched with let bindings. As usual, the expressions $\lambda(x)\ a$ and $\mathsf{let}\ x = a'\ \mathsf{in}\ a$ binds x in a but not in a'.

$$a ::= x \mid \lambda(x)\ a \mid a\ a \mid \mathsf{let}\ x = a\ \mathsf{in}\ a$$

To represent typing problems, we use a compositional translation from source terms to *typing constraints*. We introduce *expression nodes* as a meta-notation standing for the constraint the expression represents. An expression node is represented by a rectangular box in drawings. Expression nodes receive from the typing environment a set of constraint edges, meant to constrain the nodes corresponding to the free variables of the expression. Each edge is labelled by the variable it constrains. In drawings we represent such a set of edges as a blue edge ----▸, often leaving the labels implicit.

Expression nodes can be inductively transformed into simpler constraints using the rules presented in Figure 4. We follow the logical presentations of ML type inference, where generalization can be performed at every typing step, *i.e.* not only at let constructs[3]. Thus each basic expression is typed in a generalization scope, and the root of a basic constraint will always be a G-node. We have drawn those nodes in the right-hand sides of Figure 4 in order to disambiguate the origin of edges.

- A variable x is typed as the universal type scheme $\forall(\alpha)\ \alpha$. That is, it is a G-node whose child is a bottom node bound on the G-node. The bottom node is constrained by the unique edge annotated by x in the typing environment (if there is no such edge, the constraint is not closed, thus untypable).
- A let-binding $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ is typed as a_2 plus some constraints on x. The generalization scope for a_1 is introduced at the level of a_2. The (free) variables of a_1 and a_2 are constrained by the typing environment, except for x in a_2 which must be an instance of a_1.
- An abstraction $\lambda(x)\ a$ is typed as a type scheme containing an arrow type. The codomain of the arrow must be an instance of a. The variables of a are constrained by the typing environment, except for x that must unify with the domain of the arrow.
- An application $a_1\ a_2$ is typed as the codomain of an arrow type existentially introduced. The domain of the arrow is constrained so that it is an instance of the type of a_2, while the arrow type itself must be an instance of the type of a_1. Both sub-expressions are constrained by the typing environment.

[3] It is well-known that, for ML, both presentations are equivalent. However, this is not the case for MLF.

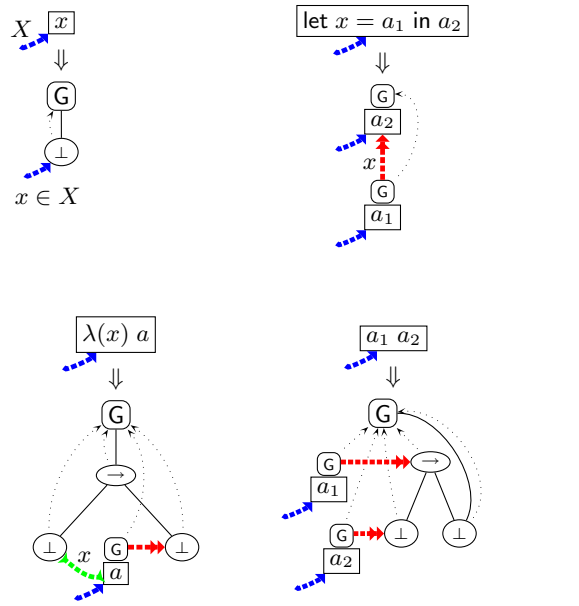


Figure 4. Typing of primitive expressions

Figure 5 shows the steps transforming the expression node for $\lambda(x)\ \lambda(y)\ x$ into a typing constraint. Notice that, in the middle constraint, the expression node for x receives two unification edges, one for x and one for y. However the unification edge for y is not useful, and is ultimately dropped since y is not free in x.

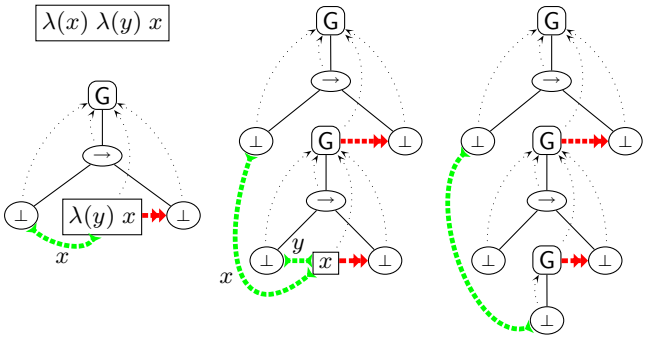


Figure 5. Typing constraints for $\lambda(x)\ \lambda(y)\ x$

2. An overview of MLF and MLF graphic types

2.1 MLF types

Combining ML-style type inference with System-F polymorphism is difficult, as type inference in the presence of first-class polymorphism leads to two competing strategies: should types be kept polymorphic for as long as possible, or conversely, for as short as possible? Unfortunately, those two paths are not confluent in general, leading to two correct but incomparable types for an expression (assuming equal types for their subexpressions).

As an example, consider the expression choose id, where id has type $\forall(\alpha)\ \alpha \to \alpha$ (which we refer to as σ_{id}) and where choose has type $\forall(\beta)\ \beta \to \beta \to \beta$. In System F, we can give this application both types $\forall(\gamma)\ (\gamma \to \gamma) \to (\gamma \to \gamma)$ and $\sigma_{id} \to \sigma_{id}$. Yet, neither one is more general than the other.

To solve this problem, MLF enriches types with a new form of (bounded) quantification: choose id receives the type $\forall(\alpha \geqslant \sigma_{id})\ \alpha \to \alpha$. Notice that both the inner polymorphism of $\sigma_{id} \to \sigma_{id}$ and the fact that both occurrences of σ_{id} are linked are retained. The variable α is allowed to range over all possible instance of its bound σ_{id}, as indicated by the sign $\geqslant$. We say it is *flexibly* bound. Of course, the two occurrences of α on both sides of the arrow must simultaneously pick the same instance: the weaker the argument, the weaker the result. The idea is to keep types as polymorphic as possible, in order to be able to recover later—just by (implicit) instantiation—what they would have been if some part had been instantiated earlier.

This form of quantification, while expressive, is not yet sufficient. For example, consider the term $\lambda(\mathsf{id} : \forall(\alpha)\ \alpha \to \alpha)\ (\mathsf{id}\ 1, \mathsf{id}\ \text{'c'})$. It is not typable in ML, as the variable id is used on two arguments with incompatible types, int and char. In System F, it can be given the type $\sigma_{id} \to \mathsf{int} \times \mathsf{char}$. However, it would be incorrect to give it the MLF type $\forall(\alpha \geqslant \sigma_{id})\ \alpha \to \mathsf{int} * \mathsf{char}$, as this type could be instantiated to $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int} * \mathsf{char}$, which would erroneously allow the application of the successor function to a character. Therefore, MLF introduces another form of quantification, called *rigidly*-bounded quantification and written with an "=" sign. The above term can be given the type $\forall(\alpha = \sigma_{id})\ \alpha \to \mathsf{int} * \mathsf{char}$. Rigid quantification is used when polymorphism is *required*, as rigid bounds will never be weakened by instantiation.

2.2 MLF graphic types

Sharing inside types is of paramount importance in MLF. For example, the types $\forall(\alpha \geqslant \sigma)\ \forall(\beta \geqslant \sigma)\ \alpha \to \beta$ and $\forall(\gamma \geqslant \sigma)\ \gamma \to \gamma$ are quite different—the former being more general than the latter as it can pick different instances of σ for α and β.

MLF graphic types have originally been introduced in part to directly capture these notions inside the representation of types [9]. They also provide a more canonical representation of types, and permit a straightforward definition of the type instance relation between types.

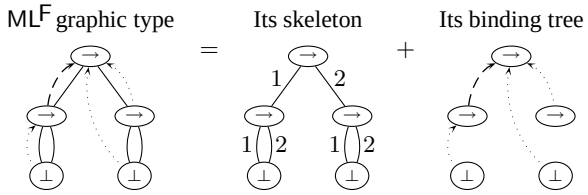


Figure 6. An example of MLF graphic type

MLF graphic types can be decomposed into a first-order quantifier free skeleton (*i.e.* an ML graphic type), and a *binding tree* that tells for every node *where* and *how* it is bound. Figure 6 shows an example of such a decomposition.

The binding tree represents the quantifiers. As in graphic constraints, we use edges rather than nodes for quantifiers, as it leaves the structure invariant by extrusion of quantifiers. All nodes have a binder. (Bottom nodes, which represent variables, must be bound. Binding non-bottom nodes that are themselves bounds of other nodes is important to keep precise track of sharing and instantiation permissions, as described in the next section. Binding nodes that are not themselves bounds of other nodes is not strictly necessary, but convenient for the regularity of the presentation.)

We use the notation $\succ\!\!\rightarrow$ for binding edges, as in graphic constraints. However, we must distinguish between flexible and rigid quantification. Flexible quantification allows instantiation, as in ML, so we (re)use dotted edges. Rigid quantification uses dashed edges, as for node $\langle 1 \rangle$ in Figure 6. When the nature of binding

edges is unimportant, we draw them as dotted-dashed lines. In the text we write $n \succ\!\!\xrightarrow{\geqslant} n'$ and $n \succ\!\!\xrightarrow{=} n'$ for flexible and rigid edges respectively, or as $n \succ\!\!\xrightarrow{\diamond} n'$ where $\diamond$ stands for either $\geqslant$ or $=$.

We write $\circ\!\!-\!\!\prec_\chi$ for $(\circ\!\!\rightarrow_\chi) \cup (\leftarrow\!\!\prec_\chi)$ called *mixed edges* and often leave χ implicit from context. Let $\longrightarrow$ range over $\circ\!\!\rightarrow$, $\succ\!\!\rightarrow$ and $\circ\!\!-\!\!\prec$. We write $\xrightarrow{*}$ and $\xrightarrow{+}$ for the reflexive, transitive, and transitive closures of $\longrightarrow$, respectively. (In drawings, we also use $\rightarrow\!\!\!\!\rightarrow$ and $\succ\!\!\rightarrow\!\!\!\!\rightarrow$ for $\xrightarrow{+}$ and $\xrightarrow{*}$, to be less intrusive.) We write $(N \longrightarrow)$ for $\{n' \mid \exists n \in N, n \longrightarrow n'\}$. The domain of the constraint χ, written $\mathsf{dom}(\chi)$ is the set of nodes $\langle\epsilon\rangle \xleftarrow{*}\!\!\prec_\chi$ reachable from the root node by inverse binding edges.

All superpositions of a graphic type with a binding tree do not form an ML^F graphic type. Indeed, the resulting graph must be *well-dominated*: the binder of a node n must dominate n for the relation $\circ\!\!\xrightarrow{+}\!\!\prec$. In essence, well-domination ensures that scopes are properly nested. The same property must actually hold in graphic constraints: in Figure 3, binding $\langle g1\rangle$ at the root in χ'_p would have been incorrect. Indeed, g, the binder of $\langle g11\rangle$, would not have dominated $\langle g11\rangle$ (as shown by the path $\langle\epsilon\rangle \longleftarrow\!\!\prec \langle g1\rangle \circ\!\!\rightarrow \langle g11\rangle$).

2.3 The instance relation

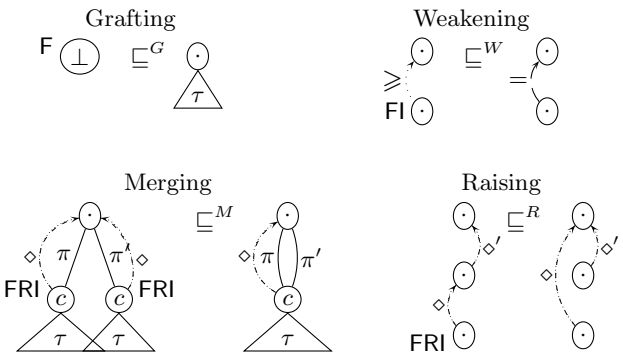


Figure 7. Instance operations

The instance relation on ML^F types $\sqsubseteq$ is defined as the composition of the atomic instantiation steps described schematically in Figure 7. That is, $\sqsubseteq$ is the relation $(\sqsubseteq^G \cup \sqsubseteq^M \cup \sqsubseteq^R \cup \sqsubseteq^W)^*$. The annotations F, FI, and FRI are explained next.

Grafting and *Merging* operate on the underlying term structure, as in ML graphic types. Grafting replaces a bottom node (*i.e.* a variable) by an arbitrary ML^F type. Merging fuses two isomorphic subgraphs, as in $\forall(\alpha \diamond \tau)\ \forall(\beta \diamond \tau)\ \alpha \to \beta \sqsubseteq^M \forall(\alpha \diamond \tau)\ \alpha \to \alpha$. *Raising* and *Weakening* operate on the binding tree. As in graphic constraints, raising extrudes a binder to the enclosing scope. If we consider the ML^F type $\forall(\alpha \geqslant \forall(\beta)\ \beta \to \beta)\ \alpha \to \alpha$ of $\mathsf{choose\ id}$, raising the variable β gives $\forall(\beta)\ \forall(\alpha \geqslant \beta \to \beta)\ \alpha \to \alpha$ (which is equivalent to the System-F type $\forall(\beta)\ (\beta \to \beta) \to (\beta \to \beta)$). Weakening turns a flexible binding edge into a rigid one, in order to require polymorphism.

Taking an instance of a type is *implicit*. Thus, $\sqsubseteq$ must not solely be sound with respect to the reduction of terms, but also permit type inference. Indeed, a relation $\sqsubseteq$ too expressive would allow—thus require, for principality—guessing polymorphic types, making type inference undecidable [13]. In ML^F, $\sqsubseteq$ is the restriction of such a larger instance relation $\leq$. The operations in $\leq \setminus \sqsubseteq$ are available *explicitly*, through the use of user-provided type annotations.

Permissions The instance operations presented in Figure 7 are only sound in certain contexts. The transformations allowed on a node depend mainly on the context in which the node appears, which we may abstract as the node *permission*. It is a key point of ML^F that permissions depend only on the binding tree—in particular, they are independent of the variances of type constructors. There are three permissions: *flexible*, *rigid*, and *locked*, abbreviated by their first letter. A node with permission x is said to be an x-node. The permission of a node n is obtained by following the binding edges linking the root to n in the automaton opposite. Notice that the automaton follows binding edges in the inverse direction of the one in drawings. For instance, for node $\langle 11\rangle$, the automaton starts in the initial state F and ends in the state L, since $\langle\epsilon\rangle \xleftarrow{=}\!\!\prec \langle 1\rangle \xleftarrow{\geqslant}\!\!\prec \langle 11\rangle$; hence this node is an L-node. Node $\langle 1\rangle$ is an R-node. All other nodes are F-nodes.

We may now give an exact description of which transformations are allowed at which node. We also provide intuitions, although a complete justification is beyond the scope of this paper.

Flexible edges are roughly the analogous of ML quantification and indicate where polymorphism is provided. Thus, by design, F-nodes allow all forms of instantiation.

In contrast, rigid edges are used to require polymorphism. Consider the graphic type of Figure 6. It corresponds to the System-F type $\forall(\alpha)\ (\forall(\gamma)\ \gamma \to \gamma) \to (\alpha \to \alpha)$. A function of this type cannot in general be treated as a function of type $\forall(\alpha)\ \tau \to \alpha \to \alpha$ where τ is an arbitrary instance of $\forall(\gamma)\ \gamma \to \gamma$, because at least this amount of polymorphism is required. Hence, an instance operation under the R-node $\langle 1\rangle$, such as grafting the L-node $\langle 11\rangle$, must not be allowed. More generally, grafting and merging of variables under R-nodes is unsound and forbidden, as well as any operation that would allow them indirectly.

On the contrary, the inverse of an instance operation could in principle be applied under node $\langle 1\rangle$. However, allowing operations in the inverse direction would not permit type inference based on first-order unification. Therefore, the allowed transformations are the restriction of sound transformations with $\sqsubseteq$ and we only allow $\geq \cap \sqsubseteq$, called abstraction and written $\sqsubseteq$, under rigid nodes. Abstraction only permits merging and raising of nodes with rigid bindings, *i.e.* R-nodes. In particular, abstraction does not permit grafting or weakening, as they would be unsound (or impossible) at R and L nodes.

There is however one exception to this definition of abstraction. An operation at a node n can be unsound only if there exists a variable node n' that is (transitively) flexibly bound to n. Otherwise, there is either no polymorphism at n, or it is protected by a rigid edge below n. Formally, a node n is said to be *inert* and called an I-node, if for any variable node n' such that $n' \succ\!\!\xrightarrow{*} n$, there is at least one rigid edge between n' and n; all operations are sound at inert nodes. (Notice however that inner nodes cannot be variables and thus cannot be grafted.) Inert nodes include *monomorphic* nodes, on which no variable node is bound at all (for example all the nodes in a graphic representation of $\mathsf{int} \to \mathsf{int}$).

One can now reread the definition of $\sqsubseteq$ in Figure 7 with permissions in mind. FI abbreviates F or I, FRI abbreviates F or R or I. For example, weakening can be performed at flexible or inert nodes.

2.4 Graphic constraints as an extension of graphic types

Instead of as an independent formalism, we choose to see graphic constraints as a small extension of ML^F graphic types. This avoids the introduction of a new framework for constraints, and allows reusing all the results already established on graphic types.

G-nodes We add G to the algebra of type constructors and introduce two sorts Scheme and Type. The symbol G has signature $\mathsf{Type} \Rightarrow \mathsf{Scheme}$ while all others have signature $\mathsf{Type}^n \Rightarrow \mathsf{Type}$ (where n is the symbol arity); thus G-nodes cannot appear under nodes of sort Type, called *type nodes*. All constraints must be well-sorted, and we require G-nodes to be flexibly bound. In the follow-

ing, the root of a constraint is always a G-node. We let the letter g range over G-nodes.

Unification edges A unification problem over graphic types is the pair of a graphic type and an equivalence relation on its nodes. A solution of a unification problem is an instance of the type that makes the nodes equivalent for this relation [10]. This subsumes the simpler problem of unifying two independent types. Unification edges are a graphic representation of a unification problem.

On a large class of problems, called *admissible*, unification is *principal*; *i.e.* an admissible problem admits a solution from which all other solutions are instances. We slightly extend the definition of admissibility for graphic constraints:

DEFINITION 1. We say that a unification edge $n_1 \blacktriangleright\!\!\cdots\!\!\blacktriangleleft n_2$ is *admissible* if either it is admissible on graphic types or $n_1 \succ\!\!\stackrel{+}{\longrightarrow} g$ and $n_2 \succ\!\!\longrightarrow g'$ where g and g' are G-nodes. □

We require unification constraints to relate two type nodes (the unification of two G-node would have no meaning), and to be admissible.

Existential nodes Existential nodes are nodes that are not reachable when following only structure edges. Formally, n is existential if n and $\check{n}$ are not in the same partition for the relation $\circ\!\!\stackrel{*}{\longrightarrow}$. Existential nodes can be of any sort. However, we require all existential nodes to be bound on G-nodes. Without this restriction, a transformation that could be applied to a constraint χ would not be applicable to a constraint χ' derived from χ by adding some unconstrained existential nodes, thus making reasoning in the system quite difficult.

The restrictions on G-nodes and existential nodes imply that the binding structure above an existential type node n is $n \succ\!\!\stackrel{\diamond}{\longrightarrow} (\mathsf{G} \succ\!\!\stackrel{\geqslant}{\longrightarrow})^* \langle\epsilon\rangle$, and all G-nodes have flexible permissions.

LEMMA 1. *Any node reachable (by a mixed path) from a type node is a type node.* □

√ (Proof p. 15)

Instantiation edges An instantiation edge $g \blacktriangleright\!\!\cdots\!\!\blacktriangleright n$ must connect a G node to a type node. We also require n to be bound on a G-node (otherwise our system would not be stable by the operation of taking the instance of a type scheme).

We introduce three operators for transforming constraints.

DEFINITION 2. Let χ be a constraint and N a subset of its nodes. The *restriction* of χ to N, written $\chi \restriction N$, is the subgraph composed of all the nodes of N and all edges between two nodes of N. The *removal* of N from χ, written $\chi \setminus N$, is the restriction of χ to $\mathsf{dom}(\chi) \setminus N$, *i.e.* all the nodes of χ but those in N.

The *projection* $proj(\chi)$ of χ is the constraint obtained by removing all unification and instantiation edges from χ. □

ML$^\mathsf{F}$ and ML constraints From now on, we distinguish ML$^\mathsf{F}$ constraints (that use the full range of ML$^\mathsf{F}$ graphic types), from ML constraints in which types are restricted to ML graphic types. That is, ML constraints are constraints in which all nodes have flexible binding edges, and all type nodes are bound on a G-node.

Typing constraints are the subset of constraints generated from λ-terms by the rules of Figure 4. It is straightforward to verify that they verify all the well-formedness conditions above. Moreover, they are ML constraints: the typing constraints are *exactly* the same in both systems.

PROPERTY 1. *Typing constraints are well-formed ML and ML$^\mathsf{F}$ constraints.* □

The instance relation on graphic constraints is essentially the instance relation $\sqsubseteq$ on graphic types, and we use the same symbol for both.

DEFINITION 3. Two constraints χ and χ' are such that $\chi \sqsubseteq \chi'$ if χ and χ', viewed as graphic types, are in instance relation, and the binding structure of G-nodes is the same in χ and χ'. □

Said otherwise, G-nodes, which encode the shape of the constraint, cannot be merged, raised or weakened.

3. Semantics of constraints

3.1 Expanding a type scheme

An instantiation constraint $g \blacktriangleright\!\!\cdots\!\!\blacktriangleright n$ requires n to be an instance of the type scheme under g; hence, we must define what are the instances of g. Of course, we must take into account generalization scopes. In essence, nodes bound above g are not generalizable, while those bound under are. We use a uniform characterization for both ML and ML$^\mathsf{F}$.

DEFINITION 4. The *constraint interior* of a node n, written $\mathcal{C}(n)$, is the set $(n \stackrel{*}{\longleftarrow\!\!\prec})$ of all nodes transitively bound to n. The *structural interior*, written $\mathcal{I}(n)$, is the restriction of the constraint interior to nodes structurally reachable from n, *i.e.* $\mathcal{C}(n) \cap (n \circ\!\!\stackrel{*}{\longrightarrow})$.

The *structural frontier* of a node n, written $\mathcal{F}(n)$, is the set $(\mathcal{I}(n) \circ\!\!\longrightarrow) \setminus \mathcal{I}(n)$ of the nodes outside $\mathcal{I}(n)$ with a structural immediate predecessor inside $\mathcal{I}(n)$. □

We write $\mathcal{C}_\chi(n)$, $\mathcal{I}_\chi(n)$ and $\mathcal{F}_\chi(n)$ when there is an ambiguity on χ. Notice that in an ML constraint, $n \in \mathcal{I}(g)$ implies in fact $n \succ\!\!\longrightarrow g$.

As an example, consider the first constraint of Figure 9. Let us focus at node n first. Its constraint interior is composed of itself and p_2. The node p_1 is not in the interior as it is bound above n. The structural frontier of n is composed of the nodes p_1 and f, reachable from n and p_2 respectively. If we consider g, its structural interior is composed of g, n, p_1, and p_2 while its constraint interior (in light green) also contains the leftmost existential arrow node.

Notice that $n \in \mathcal{I}(n)$ and $n \in \mathcal{C}(n)$; $\mathcal{I}(n)$ is reduced to $\{n\}$ when all the children of n are bound strictly above n.

The structural interior of a G-node g represents the nodes generalizable at the level of g. Conversely, it would be unsafe to generalize the nodes in the structural frontier or the nodes below. Thus, in order to take an instance of g:

- We copy the skeleton of the structural interior of g. The shape of the binding tree depends on whether we perform expansion in ML$^\mathsf{F}$ or in ML, as binding tress for ML are more restrictive than for ML$^\mathsf{F}$.
- For each node n in the structural frontier we introduce a fresh bottom node connected to the original node n by a unification edge. This ensures that all instances of g will share n. (Reusing n directly would result in ill-dominated constraints.)

The creation of a fresh instance of a type scheme is called *expansion*. It must be given a "destination" G-node where to bound the nodes created by the expansion. Expansion is slightly less general in ML than in ML$^\mathsf{F}$, as types in ML are more constrained than types in ML$^\mathsf{F}$. The difference will be explained through examples below.

DEFINITION 5 (*ML$^\mathsf{F}$ and ML expansion).* Let g and g' be two G-nodes of a constraint χ. Let n be $\langle g \cdot 1 \rangle$. The *expansion of g at g'* is derived from χ by:

- adding a copy of $proj(\chi \restriction (\mathcal{I}(g) \cup \mathcal{F}(g) \setminus \{g\}))$. The copy of a node p is called p^c;

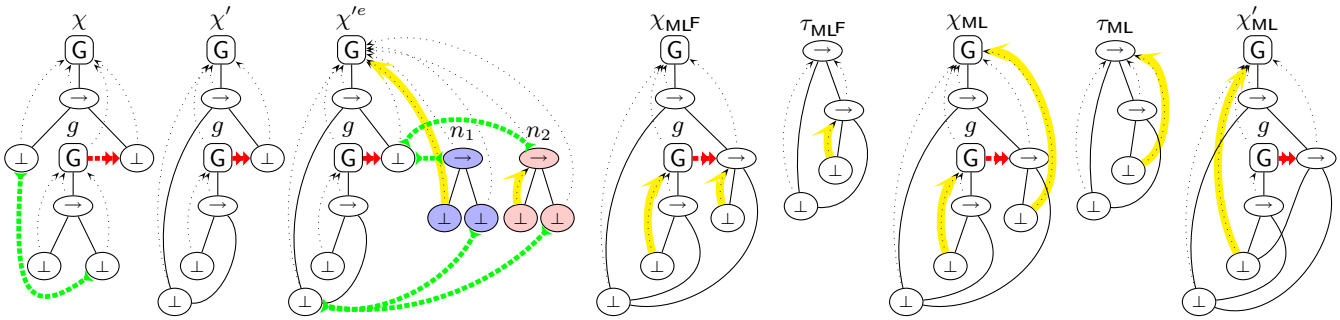


Figure 8. Typing $\lambda(x)\ \lambda(y)\ x$

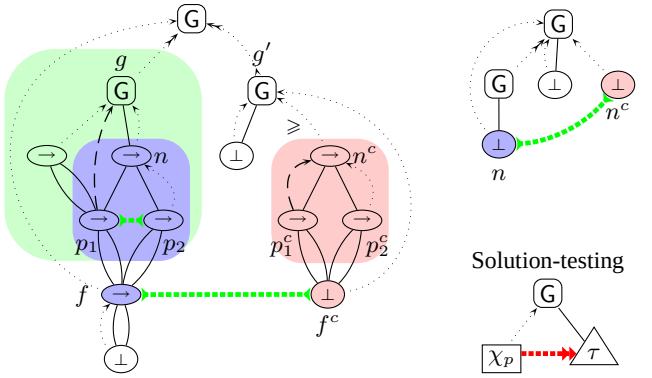


Figure 9. Examples of MLF expansion and solution-testing

- for every node f in $\mathcal{F}(g)$, changing f^c into a bottom node flexibly bound at g' and adding the unification edge $f \dashrightarrow f^c$;
- for every node $p \in \mathcal{I}(g)$ such that $p \succ\!\!\!\rightarrow g$, adding the binding edge $p^c \succ\!\!\!\xrightarrow{\diamond'}\!\!\!\rightarrow p'$, where
  - in ML, $(\diamond', p')$ is $(\diamond, g')$ (notice that $\diamond$ is necessarily $\geqslant$)
  - in MLF, $(\diamond', p')$ is $(\diamond, n^c)$ if p is not n, or $(\geqslant, g')$ if p is n.□

An illustration of an MLF expansion is given as the left constraint in Figure 9. The right-hand side of the constraint is the result of expanding the G-node g at g'. We have highlighted the nodes to be copied (n, p_1, p_2 and f, on the left) in dark blue and their copies (n^c, p_1^c, p_2^c and f^c, on the right) in red.

Notice that existential nodes and inner constraints are ignored during expansion (as is illustrated by the unification edge between p_1 and p_2 in Figure 9). Indeed, expansion is concerned with the type structure, not with the constraint structure.

Degenerate type schemes An interesting subcase occurs when n is not bound on g (which implies, by well-domination, that $\mathcal{I}(g)$ is reduced to $\{g\}$). In this case, g introduces no polymorphism, and there is no generic part to expand. Hence, only n is copied, but the copy will ultimately be unified with n itself (as illustrated in the top constraint on the right of Figure 9). We say that g is *degenerate*.

ML versus MLF expansion Consider the constraint χ' in Figure 8. Disregarding the unification edges on n_1 and n_2 for now, the constraint χ'^e shows the result of performing an ML expansion of g at $\langle\epsilon\rangle$ (under n_1, in blue), and then an MLF expansion (under n_2, in red). The difference lies in the binders of $\langle n_1 \cdot 1\rangle$ and $\langle n_2 \cdot 1\rangle$, which we have highlighted. In the ML expansion, $\langle n_1 \cdot 1\rangle$ is bound on $\langle\epsilon\rangle$. However, in the MLF expansion $\langle n_2 \cdot 1\rangle$ is bound on n_2, creating *inner* polymorphism, forbidden in ML.

Notice that, by definition, MLF expansion is always more general than ML expansion: the former can be obtained from the latter by performing a few raisings afterwards.

PROPERTY 2. *Consider an ML constraint χ, g and g' two* G*-nodes. Let χ_{ML}* (resp. χ_{MLF}*) be the result of performing an ML* (resp. MLF) *expansion of g at g^c in χ. Then $\chi_{\mathsf{MLF}} \sqsubseteq \chi_{\mathsf{ML}}$.* □

3.2 Meaning of constraints

We are now ready to give a meaning to constraints, and start by characterizing solved constraint edges. An instantiation edge is solved when a fresh instance of the type scheme *matches* the target of the edge, *i.e.* it unifies with the target without changing the constraint.

DEFINITION 6 (*Propagation*). Let e be an edge $g \dashrightarrow n$ of a constraint χ. We call *propagation* of e in χ, written χ^e, the constraint obtained by expanding g at $\widetilde{n}$, and adding a unification edge between n and the root of the expansion. □

Intuitively, propagation enforces the constraint imposed by an instantiation edge by forcing the unification of a copy of the type scheme with the constrained node. For example the constraint χ'^e in Figure 8 results from performing both an ML and an MLF propagation on the unique instantiation edge of χ'.

DEFINITION 7 (*Solved constraint edge*). A unification edge of χ is solved if its two extremities are merged. An instantiation constraint e of χ is solved if $\chi^e \sqsubseteq \chi$. □

DEFINITION 8. A *presolution* of a constraint χ is an instance χ_p of χ in which all constraint edges are solved. A *solution* of χ is a type τ, *witnessed* by a presolution χ_p of χ, for which the instantiation edge in the solution-testing constraint of Figure 9 is solved. □

In essence, solutions are all the possible types that are instances of the expansion of a presolution.

DEFINITION 9. The *meaning* of a constraint is the set of its solutions. A constraint χ *entails* a constraint χ' if the meaning of χ contains the meaning of χ'. Two constraints are *equivalent* if they have the same meaning. We write $\Vdash$ and $\dashv\Vdash$ for entailment and equivalence of constraints. □

It follows from the semantics of constraints that instantiation reduces the set of solutions, *i.e.* if $\chi \sqsubseteq \chi'$, then $\chi' \Vdash \chi$. Instantiation may sometimes preserve the meaning; however it usually does not, and a constraint may become unsolvable by instantiation. Conversely, many constraints not in instance relation may have the

same meaning—for example, constraints having different binding structure for G-nodes (*i.e.* constraint shape), as this structure is invariant by instantiation.

Examples Consider the constraint χ in Figure 8. (We will prove in the next section that it is equivalent to the last constraint presented in Figure 5. Hence, it encodes the typing of $\lambda(x)\ \lambda(y)\ x$.) In a first step, we can solve the unification edge by raising node $\langle g12\rangle$ and merging nodes $\langle 11\rangle$ and $\langle g12\rangle$, which results in χ'. However, this is not a presolution: the constraints imposed by the instantiation edge are not solved.

Further instantiations χ_{MLF}, χ_{ML} and χ'_{ML} *are* presolutions of χ, as can be verified by performing an instantiation test. (We have highlighted the differences between the three constraints.) Notice that χ_{MLF} is not a presolution in ML, as it contains inner polymorphism: node $\langle 121\rangle$ is not bound on $\langle\epsilon\rangle$. However, both χ_{ML} and χ'_{ML} are MLF *and* ML presolutions of χ. Interestingly, $\chi_{\mathsf{MLF}} \sqsubseteq \chi_{\mathsf{ML}} \sqsubseteq \chi'_{\mathsf{ML}}$ holds. In fact, χ_{MLF} is the principal presolution of χ in MLF (as we will prove in §5).

The types corresponding to the expansions of χ_{MLF}, χ_{ML} and χ'_{ML} are τ_{MLF}, τ_{ML} and τ_{ML} respectively. Hence τ_{MLF} and τ_{ML} are solutions of χ (as are all their instances). The graphic type τ_{ML} corresponds to the syntactic (ML) type $\forall(\alpha)\ \forall(\beta)\ \alpha \to \beta \to \alpha$, while τ_{MLF} represents $\forall(\alpha)\ \forall(\gamma \geqslant \forall(\beta)\ \beta \to \alpha)\ \alpha \to \gamma$. This second type corresponds roughly to the System-F type $\forall(\alpha)\ \alpha \to (\forall(\beta)\ \beta \to \alpha)$, with the additional possibility of instantiating β.

Presolutions and explicitly typed terms In our formalism, presolutions are interesting objects in their own right. Indeed, they can be seen as encoding an entire typing derivation. Given a λ-term a and a presolution χ_p of the typing constraint corresponding to a, χ_p can be used to obtain a version of a where all type information is fully explicit. Of course, different presolutions will give different decorations of a. (This correspondance should help with the definition of a Church-style version of MLF, which is ongoing work.)

Notice that the typing of $\lambda(y)\ x$ in Figure 8 is quite different in χ_{ML} and χ'_{ML}. In χ_{ML} it is polymorphic in its argument, while it is not in χ'_{ML}: node $\langle g11\rangle$ is bound on g (*i.e.* to the generalization scope corresponding to $\lambda(y)\ x$) in χ_{ML}, and to $\langle\epsilon\rangle$ in χ'_{ML}. This difference is reflected in the corresponding λ-terms in System F:

$$\left.\begin{array}{ll} \chi_{\mathsf{ML}}: & \Lambda\alpha.\,\Lambda\beta.\,\lambda(x:\alpha) \\ & \qquad (\Lambda\gamma.\,\lambda(y:\gamma)\ x)\,[\beta] \\ \chi'_{\mathsf{ML}}: & \Lambda\alpha.\,\Lambda\beta.\,\lambda(x:\alpha) \\ & \qquad \lambda(y:\beta)\ x \end{array}\right\} \forall(\alpha)\ \forall(\beta)\ \alpha \to \beta \to \alpha$$

Notice that, by construction, each type variable introduced by a Λ corresponds to a node bound on a G-node. For example, in χ_{ML}, α is $\langle 11\rangle$, β is $\langle 121\rangle$ and γ is $\langle g11\rangle$. In this simple case, the two λ-terms are β-convertible (at the level of types). Of course, this does not hold for all presolutions. For example, another typing for χ is $\forall(\beta)\ \mathsf{int} \to \beta \to \mathsf{int}$ (obtained by grafting int under $\langle 11\rangle$ in χ_{ML}), resulting in a λ-term that is not β-convertible to the ones above.

Relating ML and MLF It is immediate to prove that MLF extends ML. Indeed, the ML instance relation is a subrelation of the one in MLF, and Property 2 implies thatn instantiation edge solved in the ML sense is also solved in the MLF sense (as MLF expansions are more general).

PROPERTY 3. *All ML (pre)solutions are MLF (pre)solutions.* □

Interestingly, MLF presolutions containing only flexible edges can always be transformed by raising into ML presolutions. Thus flexible quantification alone is not significantly more expressive than ML quantification; it just gives more general types—and more opportunities to use rigid quantification.

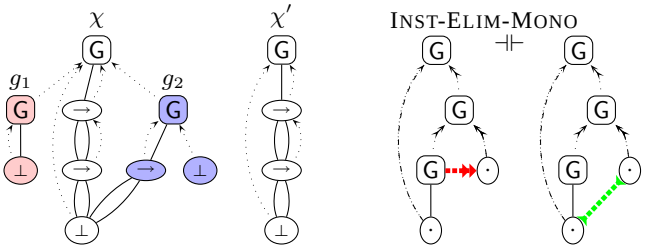


Figure 10. Simplifying unconstrained existential nodes and degenerate instantiation edges

PROPERTY 4. *Consider an ML constraint χ with an MLF presolution χ_p in which all binding edges are flexible. Then there exists MLF solutions of χ witnessed by χ_p that are ML types, and those types are also ML solutions of χ.* □

(Proof p. 15)

4. Reasoning on constraints

We now present a few transformations on constraints that preserve sets of solutions; most of them also preserve sets of presolutions—a much stronger result.

4.1 Preserving presolutions

While we are ultimately interested in proving that constraints have the same set of solutions, we often show the stronger result that *presolutions* are preserved by instantiation. We write $\chi \Vdash^p \chi'$ to mean that every presolution of χ is a presolution of χ'.

LEMMA 2. *Consider χ and χ'. Then $\chi \Vdash^p \chi'$ iff $\chi' \sqsubseteq \chi_p$ holds for any presolution χ_p of χ.* □

(Proof p. 16)

4.2 Unconstrained existential nodes

Existential nodes are meant to introduce constraint edges. Once those edges have been solved, the existential nodes become useless, and can be eliminated. Implementation-wise, this allows saving memory; it also permits to reason on simpler constraints.

DEFINITION 10. Let n be an existential node of a constraint χ such that no node in $\mathcal{C}(n)$ is the origin or the target of a constraint edge. We call *existential elimination of n in χ* the constraint $\chi \setminus \mathcal{C}(g)$. □

We refer to this operation as EXISTS-ELIM. An example is shown in Figure 10, where existentially eliminating the nodes g_1 and g_2 in χ (whose constraint interiors are highlighted) gives χ'.

Let us write $\exists^E$ for a an existential elimination, and $\exists^I$ for the inverse operation.

LEMMA 3. *Atomic instance $\sqsubseteq_1$ and existential introduction commute.* □

(Proof p. 16)

Note that this property would *not* hold if G-nodes could be raised, as examplified in Figure 11.

This result is also of particular importance to us, as it means we can locally reason on constraints without existential nodes—hence reusing all results obtained on graphic types [10]

For existential elimination, we must distinguish the cases where the instance transformation occurs inside the part being eliminated. Also, if the interior of the node being eliminated is changed (by a raising), more than one elimination might be needed.

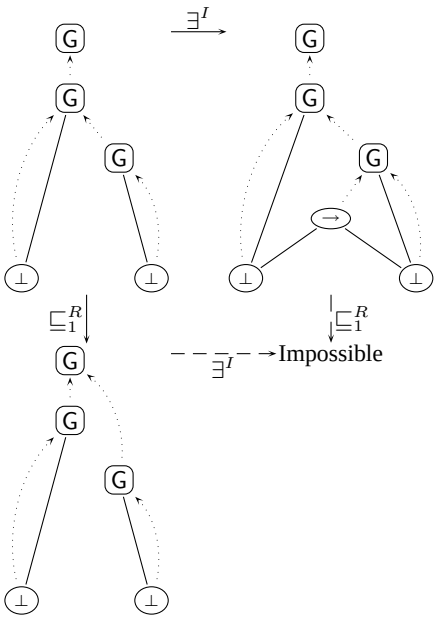


Figure 11. Non-commutation of instance and existential introduction when G-nodes can be raised

LEMMA 4. *Let χ be a constraint, χ' and χ'' such that $\chi \sqsubseteq_1 \chi'$ and $\chi \ \exists^E \ \chi''$ Then either $\chi' = \chi''$ or $\chi' \ (\exists^E)^+ \ \chi''$ or the two operations commute. Moreover the two operations commute if the instance operation does not change the nodes being eliminated.* □

(Proof p. 16)

LEMMA 5. *Consider a constraint χ which is a presolution, and χ' derived from χ by performing an existential introduction or elimination (EEI). Then χ' is a presolution witnessing the same solutions as χ.* □

(Proof p. 16)

As a direct consequence of the previous lemmas:

LEMMA 6. *Existential elimination preserves solutions.* □

(Proof p. 16)

4.3 Solved instantiation edges

Expansion is concerned only with the nodes of the structural interior of a G-node g. A transformation that does not change the interior of g leaves its expansion unchanged. We can in fact lift this property to propagation, and by extension, to solved instantiation edges:

LEMMA 7. *An instantiation edge $g \dashrightarrow d$ that is solved in a constraint χ remains solved in any instance of χ that leaves $\mathcal{I}(g)$ unchanged.* □

(Proof p. 16)

This property is quite important for reasoning, as it ensures that unrelated changes will not break solved edges.

4.4 Unification edges

The level of generalization we brought to our graphic representation is small enough that the unification algorithm on unconstrained graphic types [9] can be reused unchanged. Moreover, unification remains principal.

LEMMA 8. *The unification algorithm on graphic types is a sound, complete and principal transformation on graphic constraints.* □

(Proof p. 17)

The principality of unification on graphic types also ensures that unification edges can always be solved eagerly.

LEMMA 9. *Let e be a unification edge of χ. If unifying e in χ fails, χ has no solution. Otherwise, let χ' be the principal unifier of e in χ. Then χ and χ' have the same (pre)solutions.* □

(Proof p. 17)

Interestingly, unification on ML graphic types can be solved with the unification algorithm for ML^F graphic types. This follows from the facts that type instance for ML is a subrelation of type instance for ML^F and that the unification algorithm of ML^F applied to ML graphic types returns ML graphic types. In fact, the unification algorithm needs not check for permissions when the input types are ML constraints, since in this case all nodes have flexible permissions. Moreover, the raisings it performs amount to updating generalization levels when variables are merged, exactly as done in efficient implementations of ML type inference based on ranks and term dags [7, 8].

4.5 Degenerate instantiation edges

A degenerate G-node contains no polymorphism, as witnessed by the fact that no "real" fresh node is created when it is expanded. An instantiation edge leaving from a degenerate G-node is itself degenerate, in the sense that it is equivalent to an unification edge. This is described by rule INST-ELIM-MONO on the right of Figure 10.

DEFINITION 11. Let e be an instantiation edge a constraint χ leaving from a degenerate G-node. We call INST-ELIM-MONO the rule transforming χ into $\chi^e \setminus e$. □

LEMMA 10. INST-ELIM-MONO *preserves solutions.* □

(Proof p. 18)

Notice that this rule does not preserve presolutions strictly speaking, as it changes the number of instantiation edges. However, it preserves the "shape" of presolutions, *i.e.* the structure of G-nodes and type nodes.

We can now prove that the constraint χ of Figure 8 is equivalent to the typing constraint of $\lambda(x)\ \lambda(y)\ x$ given in Figure 3. Indeed the former is obtained from the latter by successively:

1. solving by unification the constraint edge on node $\langle 11 \rangle$;
2. performing INST-ELIM-MONO on the G-node corresponding to the variable x (which we call g), as it is degenerate after the unification;
3. existentially eliminating g (whose interior is reduced to $\{g\}$).

Thus the equivalence is by Lemmas 9, 10 and 6.

4.6 Eager propagation

A crucial property of our framework is that scheme expansion and propagation are essentially[4] monotonic w.r.t. to instance $\sqsubseteq$.

LEMMA 11. *Consider a constraint χ' such that $\chi \sqsubseteq_1 \chi'$, and e an instantiation edge of χ. Then $\chi'^e \Vdash^p \chi^e$.* □

(Proof p. 18)

An important consequence of this property is that we may propagate any instantiation edge in any constraint without changing its presolutions.

[4] The property $\chi \sqsubseteq \chi' \implies \chi^e \sqsubseteq \chi'^e$ does not hold. However, if $\mathcal{U}(\chi^e)$ and $\mathcal{U}(\chi'^e)$ are the constraints resulting from solving the unification edges generated by the propagation in χ^e and χ'^e, $\chi \sqsubseteq \chi' \implies \mathcal{U}(\chi^e) \sqsubseteq \mathcal{U}(\chi'^e)$ does hold.

LEMMA 12. *Propagation preserves presolutions.* □

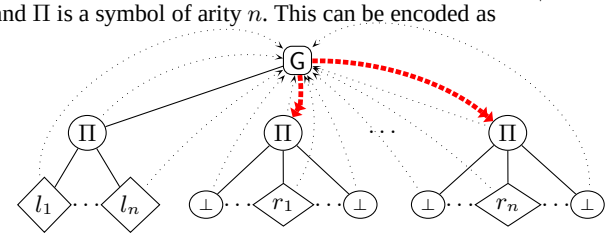
(Proof p. 18)

This result provides a good test when designing the relation $\sqsubseteq$. Indeed, if it did not hold, it would be impossible to reduce type inference to propagation (*i.e.* type scheme instance) and unification.

5. Solving acyclic constraints

In their full generality, our constraints may be used to encode typing problems with polymorphic recursion, which are already undecidable in ML.

Alternatively, we can also encode semi-unification problems (that are also undecidable). Consider a semi-unification problem $(l_i \leq r_i)_{i=1}^n$. We may build the equivalent problem $(L \leq R_i)_{i=1}^n$ where L is $\Pi(l_1, ...l_n)$ and R_i is $\Pi(\alpha_1^i, ..., \alpha_{i-1}^i, r_i, \alpha_{i+1}^i, ...\alpha^n)$ and Π is a symbol of arity n. This can be encoded as



where all nodes are bound at the root (and in general l_i's and r_i's share some leaves).

Thus we restrict our attention to constraints in which the instantiation edges induce an *acyclic* relation.

DEFINITION 12. A G-node g *directly depends* on another G-node g' if g' constrains the constraint interior of g, *i.e.* $\exists n \in \mathcal{C}(g), g' \dashrightarrow n$. The *dependency* relation between G-nodes is the transitive closure of the "directly depends on" relation. □

DEFINITION 13. A constraint χ is *acyclic* if the dependency relation on its G-nodes is a strict partial order. □

Notice that the typing constraints presented in Figure 4 are acyclic, as instantiation edges follow the scopes of the variables of the expression, which are nested.

Importantly, acyclicity is stable by instantiation.

LEMMA 13. *If $\chi \sqsubseteq \chi'$ then the dependency relation in χ' is a subrelation of the one in χ.* □

(Proof p. 18)

5.1 Finding a principal presolution

In acyclic constraints, propagating-then-unifying an instantiation edge solves that edge.

LEMMA 14. *Let e be an instantiation edge $g \dashrightarrow d$ of a constraint χ where d is not in $\mathcal{C}(g)$. Let χ' be the principal unifier of the unification edges introduced in χ^e (if this unifier exists). Then χ' is an instance of χ in which e is solved.* □

(Proof p. 19)

The condition on n and $\mathcal{C}(g)$ vacuously holds on acyclic constraints. It ensures that the interior of g will not be changed by the unification. Afterwards, the conclusion is simply by idempotency of propagation-unification.

Acyclic constraints admit a principal presolution, which can be built using the following strategy.

1. Solve all unification edges by unification.
2. Visit the instantiation edges in an order compatible with the dependency relation. On each edge e:
   (a) perform a propagation on e;
   (b) unify the resulting unification edges.

   Those operations solve e (Lemma 14). Moreover, since the constraint is acyclic, all instantiation edges already visited (hence solved) remain solved (Lemma 7).

The preservation of presolutions follows from Lemma 9 for steps 1 and 2b and from Lemma 12 for step 2a.

We introduce one more definition, in order to characterize G nodes of a constraint that remain solved throughout the traversal of the instantiation edges.

DEFINITION 14. A G-node g is *recursively-solved* if its interior is not the target of a unification edge and for any edge $e = g' \dashrightarrow d$ with $d \in \mathcal{C}(g)$, e is solved and g' is recursively solved. □

THEOREM 1. *Acyclic constraints have principal decidable presolutions.* □

(Proof p. 19)

The proof of this result implies a few interesting properties.

The first one is that G-nodes with no escaping edges can be solved locally.

DEFINITION 15. We say that a G-node g is *closed* if any edge $n \longrightarrow n'$ (with $\longrightarrow$ ranging over $\circ\!\!\rightarrow$, $\succ\!\!\rightarrow$, $\dashleftarrow$ and $\dashrightarrow$), $n \in \mathcal{C}(g)$ implies $n' \in \mathcal{C}(g)$ or $n = g$ and $\longrightarrow$ is $\dashrightarrow$ or $\succ\!\!\rightarrow$. □

COROLLARY 1. *If s is closed in an acyclic constraint χ, and χ admits a presolution χ_p, there exists a presolution χ'_p of χ, witnessing the solutions of χ_p, such that s is closed in χ'_p.* □

This result relies on the fact that unification and propagation "follow" edges. Hence, no binding edge will ever be raised above g in the principal presolution.

Another key consequence is that once a G-node is recursively solved, its interior will never need to be instantiated more. Thus, after its outgoing instantiation edges have been propagated, we can remove them.

COROLLARY 2. *Consider a recursively solved G-node g of an acyclic constraint χ. For any edge $e = g \dashrightarrow d$, the constraints χ and $\chi^e \setminus e$ are equivalent.* □

Of course, this also holds for unconstrained G-nodes, which are trivially recursively solved.

COROLLARY 3. *Let e be an edge $g \dashrightarrow n$ of an acyclic constraint χ. If $\mathcal{C}(g)$ is not the target of a constraint edge, then χ and $\chi^e \setminus e$ are equivalent. Under those hypotheses, we call* INST-EXPAND *the replacement of χ by $\chi^e \setminus e$.* □

This is another proof of the correctness of rule INST-ELIM-MONO. However, Lemma 10 is more general, as it does not require the constraint to be acyclic.

Typability in unannotated ML^F and ML The strategy solving an acyclic constraint gives us some hindsight on the expressiveness of ML^F. Consider a typing constraint. It is an ML constraint (Property 1). If it is solvable in ML^F, its principal presolution will contain only flexible edges, as propagation and unification do not introduce new rigid edges. Then, by Property 4, it will have an ML solution. Thus, a program without type annotations is typable in ML^F if and only if it is typable in ML. (However, in general its principal type in ML will be a strict instance of its principal type in ML^F).

THEOREM 2. *Any expression typable without type annotations in ML^F is typable in ML.* □

This result is a direct consequence of the following result:

LEMMA 15. *Consider an ML acyclic constraint. It is typable in ML if and only if it is typable in ML^F.* □

(Proof p. 19)

Inconsistent constraints We have so far ignored the possibility that a constraint might become inconsistent while simplifying it. This situation is in fact implicitly dealt with by our formalism: an inconsistent constraint (such as a unification edge that would lead to a constructor clash or a cyclic type) cannot be solved. Thus it cannot be removed by existential elimination, and will remain unsolved. Consequently, the constraint has no presolution. Of course, an implementation can fail as soon as an inconsistency is found.

Efficiency Using the order induced by the dependency relation ensures that an instantiation edge never needs to be propagated more than once. Hence, the number of unification steps that can be performed is bounded by the number of instantiation edges plus the number of initial unification edges.

One potential source of inefficiency in the strategy used to find the principal presolution is that the resulting constraint can be much bigger than the solution itself. Hence a better approach (if we are interested only in the solutions) is to apply INST-EXPAND to perform the propagation, then existential elimination to the nodes that are no longer constrained. While this does not change time complexity, it ensures that constraints remain as small as possible and improves space complexity.

5.2 Splitting G-nodes

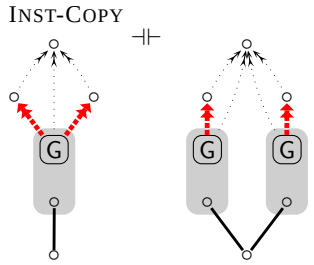


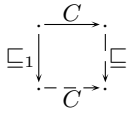
Figure 12. Rule INST-COPY

An interesting rule to consider is INST-COPY, presented in Figure 12. It can be applied whenever a G-node has one or more outgoing instantiation edge. The edges may be arbitrarily partitioned into two sequences, and the interior nodes and edges of the G-nodes are duplicated (as well as edges between the interior and the frontier).

Intuitively, one could think that the constraint in which the G-node has been split has more solutions. Indeed, each scheme could seemingly pick a different type. However, this is not the case on acyclic constraints. Indeed, since they have principal presolutions, the two schemes can pick only instances of the most general solution.

We first prove two intermediate commutation lemmas.

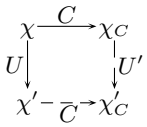
LEMMA 16. *Let χ and χ' be two constraints such that $\chi \sqsubseteq \chi'$. Let g be a G-node. Let C be one application of* INST-COPY *on g. The following holds:*



□

(Proof p. 19)

LEMMA 17. *Let χ be a constraint, g one of its G-nodes, C an application of* INST-COPY*. Let U be the application of* Unif *on a given unification edge e, U' the application of this algorithm to e if it is not duplicated by C, and to e and its copy otherwise. The following holds:*



□

LEMMA 18. *Rule* INST-COPY *preserves the meaning of acyclic constraints.* □

(Proof p. 20)

6. Type annotations

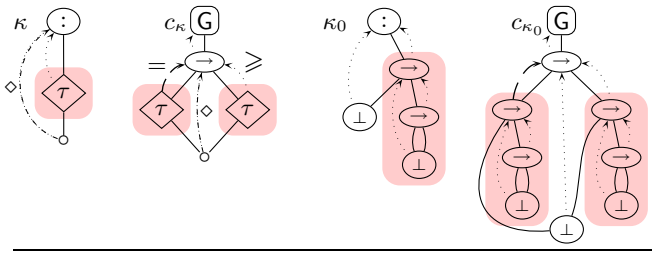


Figure 13. Types of coercion functions

Type annotations are a key to MLF. Interestingly, we do not use primitive typing constructs to type them. Instead, we add a denumerable set of *coercion functions* to the typing environment.

As an example, consider the annotation $(a : \exists\beta\forall(\alpha)\ \beta \to (\alpha \to \alpha))$. It contains both *universal* and *existential* quantification, and expresses that a must be a function, the type of its first argument being left unspecified, and its return type being exactly $\alpha \to \alpha$. This annotation can be represented by the type κ_0 of Figure 13. The existential part is bound at the root ":" node, while the nodes inside the universal part are bound on $\langle 1\rangle$ or under (in this simple case they are all bound on $\langle 1\rangle$).

More general annotations are depicted by the pseudo-type κ of the same figure. In the annotation $(a : \kappa)$, the type τ at node $\langle 1\rangle$ inside κ is *universally* quantified. However, the other nodes of κ, represented by the $\circ$ meta-node notation and bound on the root, are *existentially* quantified: they can be instantiated during type inference.

The annotation $(a : \kappa)$ is desugared as the application $c_\kappa\ a$, where the type of the coercion c_κ is also shown on Figure 13. Each side of the arrow is a copy of τ. Hence, they could a priori be instantiated independently. However, the domain is rigidly bound, meaning that the polymorphism is requested, and thus cannot actually be weakened by instantiation: a must be of type τ. On the contrary, the codomain is flexibly bound, meaning that the polymorphism is provided, and can freely instantiated. The nodes corresponding to the existential part of κ are not duplicated: they are shared between the domain and the codomain, and will be instantiated simultaneously on both sides. An example is given by the type c_{κ_0}.

Similarly, the expression $\lambda(x : \kappa)\ a$ is also syntactic sugar, for $\lambda(x)$ let $x = (x : \kappa)$ in a; an example is given in §8. Notice that type annotations are part of expressions. Hence, two terms with different annotations are really different terms and do not usually have a common, most general type.

7. Complexity of type inference

7.1 Simplifying typing constraints

For homogeneity, typing constraints introduce a G-node for every sub-expression, including variables. However, those are superflu-

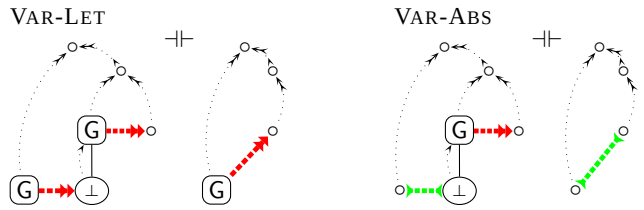
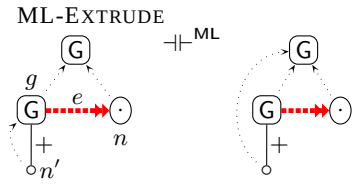


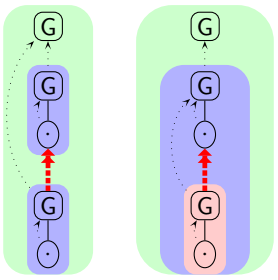
Figure 14. Simplifying the typing of variables



Condition:

- e is the only instantiation edge leaving g
- $g \circ\!\!\xrightarrow{\pm} n'$

Figure 15. Simplifying ML constraints



On the left-hand side, both type schemes are introduced at the same level, thus the embedding is 2. On the right-hand side, one is inside the other and the embedding is 3.

Figure 16. Type schemes embedding

ous. Indeed, let-bound variables only generate indirections, while the G-node for a λ-bound variable will ultimately be degenerate. The corresponding simplifications rules are presented in Figure 14.

In ML typing constraints, the G-nodes for abstractions and application are also superfluous (hence G-nodes are in fact only needed for let-bound expressions). Indeed, as shown in Figure 15, a type node inside a type scheme that is "used" only once and at the nearest generalization scope can be extruded entirely.

All simplifications can be performed in linear time either after the generation of constraints, or on-the-fly during their generation.

LEMMA 19. *Rules* VAR-ABS *and* VAR-LET *preserve solutions.* □
(Proof p. 20)

LEMMA 20. *Rule* ML-EXTRUDE *preserves the solutions of ML constraints.* □
(Proof p. 21)

7.2 Complexity analysis

While type inference for ML is DEXP-TIME complete (when types need not be output), McAllester has shown [6] that type inference has complexity $O(kn(\alpha(kn) + d))$ where α is the inverse of the Ackermann function, k is the maximum size of type schemes and d the maximum embedding of type schemes. (Figure 16 describes what is meant by embedding of type schemes.) In McAllester's analysis, d corresponds to the maximum left-nesting of let constructs, *i.e.* nestings of the form let $x = (\text{let } y = \ldots \text{ in } \ldots)$ in $\ldots$.

As argued by McAllester, d is almost always bounded by 5, and k does not increase with the size of the program. Under those assumptions, type inference in ML has $O(n\alpha(n))$ complexity, which is almost linear (the term $\alpha(n)$ is negligible).

Our strategy for solving constraints is quite similar to the one used in efficient implementations of type inference for ML [6, 7]. In particular, type schemes are also simplified in an innermost fashion. Unification in ML$^\mathsf{F}$ can also be performed in time $O(n\alpha(n))$ and the complexity analysis of McAllester for ML can be transferred to our constraints setting—provided we reason on the embedding of G-nodes instead of the embedding of let constructs. More precisely, for our typing constraints, the function d verifies:

$$\begin{aligned} d(x) &= 1 \\ d(\lambda(x)\ a) &= d(a) + 1 \\ d(a\ b) &= \max(d(a), d(b)) + 1 \\ d(\text{let } x = a \text{ in } b) &= \max(d(a) + 1, d(b)) \end{aligned}$$

When applying VAR-LET and VAR-ABS, d verifies $d(x) = 0$.

Importantly d does not increase with right-nesting of let bindings. In particular, a large upper bound of d is the height of the biggest function of the program (when written as an abstract syntax tree). Under the two assumptions that **(1)** large programs are composed of cascades of right-nested toplevel let declarations, and **(2)** k does not increase with the size of the program, type inference in our constraints system (thus in ML$^\mathsf{F}$) has linear complexity.

Notice that, if we restrict ourselves to ML, using the constraint simplification of Figure 15 will eliminate G-nodes for all subexpressions but the left-hand side of let constructs. We therefore obtain exactly the same complexity as McAllester.

Our analysis also provides an upper bound for the complexity of type inference. In the worst case, the maximum size of type schemes k is bounded by $2^{O(n)}$ and the maximum depth of G-nodes d is bounded by n. The complexity is thus in $2^{O(n)} \times n \times (\alpha(2^{O(n)} \times n) + n)$, *i.e.* in $2^{O(n)}$. As ML programs are typable in ML$^\mathsf{F}$ if and only if they are typable in ML, the complexity bound for ML$^\mathsf{F}$ cannot be better that the one for ML. We thus have established the exact complexity bound $2^{O(n)}$ for type inference in ML$^\mathsf{F}$.

8. Examples of typings

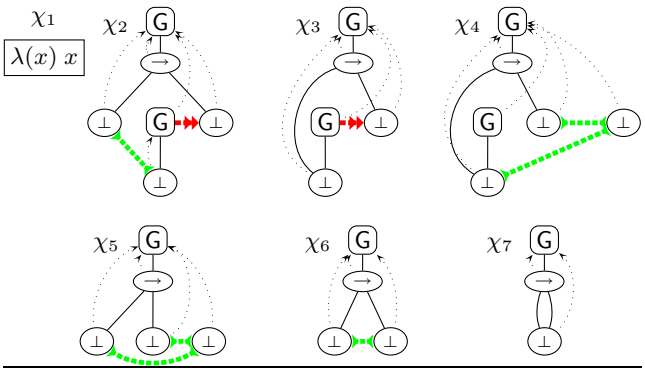


Figure 18. Typing $\lambda(x)\ x$

Figure 18 presents the typing of the identity, valid in both ML and ML$^\mathsf{F}$. The first step (from χ_2 to χ_3) is by unification, χ_4 is by INST-EXPAND on the instantiation edge. χ_5 is by EXISTS-ELIM on g, χ_6 is by unification on the rightmost edge. (The steps χ_2 to χ_6 could have been directly proven by VAR-ABS.) χ_7 is by unification. The resulting principal type is $\forall(\alpha)\ \ \alpha \to \alpha$, abbreviated as σ_{id}.

Figure 17 presents the typing of let $y = \lambda(x)\ x$ in $y\ y$ in ML$^\mathsf{F}$. In χ_3 we have developed the expression node for $y\ y$. In χ_4 we have replaced $\lambda(x)\ x$ by its principal typing and applied VAR-LET

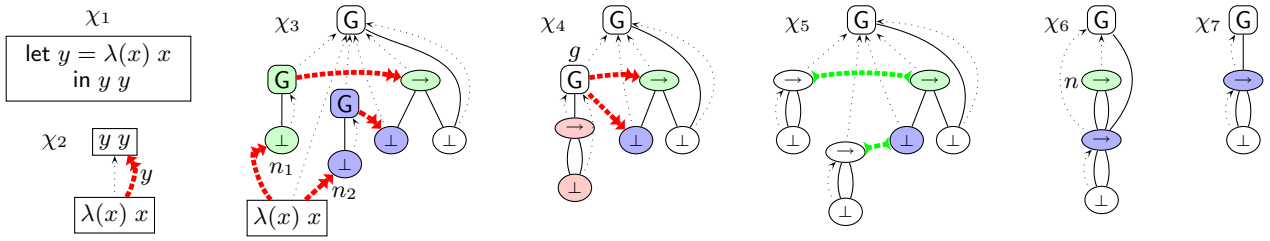


Figure 17. Typing let $y = \lambda(x)\ x$ in $y\ y$

to both n_1 and n_2. χ_5 is by INST-EXPAND on each instantiation edge, then by EXISTS-ELIM on g. χ_6 is by unification and χ_7 by EXISTS-ELIM on n. The result is σ_{id}. The derivation is essentially the same in ML, up to a few nodes bound at $\langle\epsilon\rangle$ in χ_5 to χ_7.

The last example (Figure 19) uses a type constraint on a parameter. As explained in Section 6, it expands into the expression described in constraint χ_2. In χ_3 we have expanded the expression nodes for both the abstraction and the application $c_{\kappa_{\mathsf{id}}}\ x$. We have also simplified on the fly the instantiation edge on $c_{\kappa_{\mathsf{id}}}$ into a unification one; this is possible by INST-EXPAND and EXISTS-ELIM. χ_4 is by VAR-ABS on n, then by unification on the redirected unification edge. χ_5 is by unification on the remaining edge. χ_6 is by EXISTS-ELIM on n. Up to a few unimportant differences, the highlighted nodes correspond to the constraint χ_3 of Figure 17. Simplifying those nodes thus results in χ_7. χ_8 is by INST-EXPAND on the instantiation edge, then by EXISTS-ELIM. χ_9 is by unification. The result is the type $\forall\,(\alpha = \sigma_{id})\ \forall\,(\beta \geqslant \sigma_{id})\ \alpha \to \beta$, corresponding roughly to the System-F type $\sigma_{id} \to \sigma_{id}$ in which instantiating the occurrence of σ_{id} on the right of the arrow is allowed.

Implementation An MLF type checker (which faithfully implements the algorithm presented in §5.1) can be found at `http://gallium.inria.fr/~remy/mlf/`. Although graphic types are used internally, we print the types in syntactic form. Using a simple syntactic sugar (discussed below) this nearly always results in quite readable, System-F looking, types. In particular, this should alleviate doubts that MLF types are too complicated to be presented to the programmer. An interactive mode in which the user can graphically select which constraint edge to solve is also available (and can be used to solve the examples above step by step).

Much of the difficulty in implementing graphic constraints lies in finding a good representation for graphic types, and implementing the unification algorithm. Notice that in MLF graphic types the graph structure and the binding tree are interwoven and their edges go in inverse directions. Finding an efficient functional representation of such a structure is not obvious, so we use an imperative implementation. This only causes problems when unification fails, *i.e.* when a type inference error occurs. In this case we type the expression a second time, and explain the error in terms of the last valid constraint.

The generation of typing constraints from the λ-term is entirely straightforward, using a single recursive function. Moreover, typing constraints are simple enough that the dependency relation on instantiation edges needs not be computed. Instead, we sort instantiation edges on-the-fly during constraints generation.

Explaining type inference errors raises two challenges: (**1**) explaining unification clashes caused by MLF polymorphism; (**2**) associating a type inference error with the corresponding part of the source term.

The difficulty in point (2) is only apparent. Indeed, it is straightforward to associate a constraint edge with the expression that resulted in its creation (and our prototype already prints exact source location in error messages). Our constraints-based approach also allows choosing a strategy to solve the instantiation edges[5]; for example, the function in an application can be typed before the argument. While our implementation already gives quite readable error messages (see Figure 20), we are also experimenting with other strategies.

Point (1) is trickier. However, in simple examples that do not result from encodings, a message such as "function f expects an argument of type τ but receives a value of type τ'; type τ' is probably not polymorphic enough" is often sufficient. An interesting step will be to preserve type variables used inside type annotations through inference—an easy, if tedious, task.

Displaying syntactic types MLF syntactic types are often difficult to read because of the bounded quantification they feature. For example, the term $K \triangleq \lambda(x)\ \lambda(y)\ x$ has for principal type

$$\forall\,(\alpha)\ \forall\,(\gamma \geqslant \forall\,(\beta)\ \beta \to \alpha)\ \alpha \to \gamma$$

A way to relieve this tension is to introduce a syntactic sugar that inlines bounds. Then K gets for principal type

$$\forall\,(\alpha)\ \alpha \to (\forall\,(\beta)\ \beta \to \alpha)$$

This type is a type of System F, hence much more familiar looking.

Of course, we cannot just inline all bounds, as we would lose sharing, the fact that the bound is flexible or rigid, and the place where the bound is introduced. In consequence, we follow three restrictions:

1. Bounds used strictly more than once are never inlined (except of course for monomorphic types).
2. Rigid bounds are inlined on the left of an arrow; flexible ones on the left. Thus our convention follows the variance of the arrow constructor, hence the intuition. For the other type constructors than the arrow, the choice is left to the user.
3. Bounds are inlined only when they are used immediately where they are introduced. For example, α is inlined in $\forall\,(\alpha \geqslant \sigma)\ \mathsf{int} \to \alpha$, but not in $\forall\,(\alpha \geqslant \sigma)\ \forall\,(\beta \geqslant \mathsf{int} \to \alpha)\ \mathsf{int} \to \beta$.

In spite of these restrictions, nearly all terms get principal types that look like System-F types. Indeed, MLF binds types "as low as possible", hence Condition 3 is rarely restrictive. Condition 2 is actually the heart of the sugar, and follows the intuitions given by the variance; following another convention would probably confuse the programmer. Condition 1 is also rarely used. Indeed, shared bounds nearly only appear in partial applications, which are themselves unusual.

Using our convention, deciding whether an inlined variable α can be instantiated is done by finding α in the sugarified type, and checking whether we have followed an arrow on the left. If this is the case, α has locked (or rigid) permissions. Otherwise it has flexible permissions and can be instantiated. Thus the criterion is quite simple. If we consider the principal type of K, β appears at

[5] In general, there exists different strategies with optimal complexity.

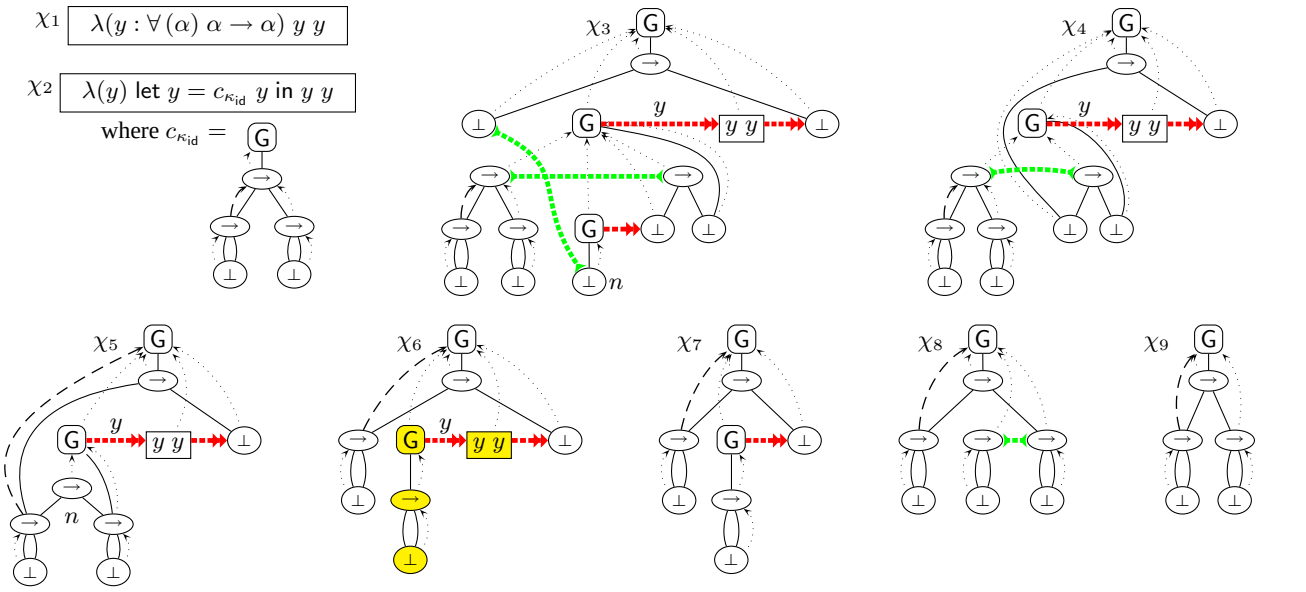


Figure 19. Typing $\lambda(y : \forall(\alpha)\ \alpha \to \alpha)\ y\ y$

```
# fun x → if x = 1 then True else x
Both branches of this 'if' have incompatible types. The 'then' part has type bool while the 'else' part
has type int.

# (fun x : 'a. 'a → 'a) → x x)  succ
Cannot apply the first expression to the second: the argument is probably not polymorphic enough.
The first expression has type ('a. 'a →' a) → ('c. 'c →' c) while the second has type int → int.

# fun x ⇒ x x
Cannot apply the first expression to the second: the resulting type would be cyclic.
The first expression has type 'a while the second has type 'a. The context is ('a > ⊥)
```

Figure 20. Examples of error messages

path $\langle 2 \rangle$. Hence we have not followed an arrow on the left (there would be 1 on the the path in this case), and β can be weakened by instantiation.

Our syntactic sugar is actually a variation on an idea proposed by Leijen[5, Section 2.6]. However, in this work, the third condition was ommitted, and the convention was not bijective (resulting in types with really different polymorphism being displayed the same way).

9. Comparison with other works

A detailed comparison between ML$^\mathsf{F}$ and other extensions of System F can be found in [3]. The most closely related work [4] propose a restriction of ML$^\mathsf{F}$ where non System-F types only appear internally during type inference and can always be instantiated into System-F types afterwards. Another related work [12] introduces a notion of "boxy types" that resembles our flexible bindings. Both works aim at finding a type system with second-order polymorphism that assigns System-F types to expressions. Since ML$^\mathsf{F}$ types are more expressive than System-F ones, we believe that our graphic presentation of type inference would help explore such systems more systematically. Hopefully, our inference algorithm could also be adapted to those works.

Efficient type inference for ML Efficient type inference algorithms for ML have many similarities with our graphic type inference algorithm. Of course, they all use an efficient graph-based unification algorithm and reduce type schemes in an inner-outer fashion. More interestingly, they also use a notion of ranks (or frames) to keep track of generalization levels and perform generalization more efficiently [6, 7, 8]. Merging two multi-equations in [8] requires them to have the same rank, hence lowering their rank to the smallest of the two beforehand. Similarly, merging two nodes in graphic types requires them to have the same bound, hence raising them to their lowest common binder. Raising binding edges has also strong similarities with Rule S-LET-ALL of [7].

Type inference as typing constraints To the best of our knowledge Henglein has first expressed type inference as the satisfaction of type-inference constraints, which led him to semi-unification problems [1]. Hence, the obvious similarity between our constraints and his. However, his constraints are interpreted over simple types while ours are interpreted over graphic types, that generalize System-F types. Our constraints are therefore more expressive. His constraints avoid the explicit representation of G-nodes, and instead read types as type schemes according to the context. We cannot make this simplification in ML$^\mathsf{F}$ because ML$^\mathsf{F}$ expansion is more complicated than the ML one.

Typing constraints for ML have been explored in detail [7]. There are many similarities between this work and ours. Typing constraints are introduced first, independently of the underlying language; then a set of sound and complete transformations on

typing constraints are introduced; the type inference algorithm is finally obtained by imposing a strategy on applications of constraint transformations. Moreover, some important steps of both frameworks can be put in correspondence (solving unification constraints, expansion of type-schemes, *etc.*). However, our constraints are more concise, for two reasons. Firstly, the graphical representation of types is more canonical: for instance, we need no rule for commutation of adjacent binders. Secondly, the underlying binding structure of graphic types is reused for describing the binding constructs of graphic constraints. Hence, the representation of constraints requires fewer extension to the representation of types, as the latter is already richer.

Semi-unification As shown by Henglein [1], type inference for ML reduces to semi-unification problems that are trivially acyclic by construction—in the absence of polymorphic recursion. Hence, we should be able to see our constraints as encoding a form of acyclic graphic-type semi-unification problems. It would certainly be worth further exploring this point of view. Possibly, we could enable implicit polymorphic recursion in MLF by allowing some incompleteness in type inference. (Explicit polymorphic recursion is already available through type annotations.)

Other versions of MLF There are two syntactic presentations of MLF [2, 3]. In the original one [2], the instance relation on types is not as general as the one proposed when graphic types were introduced [9] (and further increased later [10]). Hence, the type inference system we have presented is slightly more general; however, we do not know of a short, non-artificial λ-term typable in our system but not in the original one.

The extended instance relation has been transferred back to the syntactic presentation [3], albeit at some technical cost. However, this work does not address type inference. Moreover, it is based on a stratified, restricted version of MLF, in which types are not as general as those presented here or in the original presentation.

Conclusion

We have extended the initial presentation of graphic types [9] to represent typing constraints, for both ML and MLF. Graphic constraints are simpler than the syntactic constraints that have been developed for ML; in particular they sidestep tedious issues such as α-renaming or commutations of binders. We obtain a new, fully graphical presentation of MLF, where both the specification and the type inference algorithm are done graphically. This presentation highlights the very strong ties between ML and MLF. We have also shown that type inference for MLF has linear-time complexity under reasonable assumptions.

By lack of space, type soundness is deferred to another paper [11]. We have proven the soundness of the (larger) system that uses $\leq$ instead of $\sqsubseteq$ as the type instance relation (§2.3).

In spite of the overhead inherent to using a slightly uncommon formalism, reasoning on the meta-theoretical properties of the system has shown to be significantly simpler on graphic types than on syntactic ones. Hence, we believe our graphical approach is a very good basis for exploring further extensions of MLF with richer type structure, such as recursive types, primitive existentials, higher-order types, dependent types, or some form of subtyping. This new presentation of MLF typechecking as solving of typing constraints is also a significant simplification of MLF and a significant step towards its possible use in a full-scale programming language.

Acknowledgments We would like to thank Didier Le Botlan, Yann Régis-Gianas, Zaynah Dargaye, Jade Alglave and anonymous referees for helpful suggestions on previous versions of this work.

References

[1] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[2] Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of system-F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 27–38, August 2003.

[3] Didier Le Botlan and Didier Rémy. Recasting MLF. Research Report 6228, INRIA, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, June 2007.

[4] Daan Leijen. A type directed translation of MLF to System F. In *The International Conference on Functional Programming (ICFP'07)*. ACM Press, October 2007.

[5] Daan Leijen and Andres Löh. Qualified types for MLF. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 144–155, New York, NY, USA, September 2005. ACM Press.

[6] David McAllester. A logical algorithm for ML type inference. In *International Conference on Rewriting Techniques and Applications (RTA), Valencia, Spain*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer-Verlag, June 2003.

[7] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[8] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, INRIA, 1992.

[9] Didier Rémy and Boris Yakobowski. A graphical presentation of MLF types with a linear-time unification algorithm. In *TLDI'07: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 27–38, Nice, France, January 2007. ACM Press.

[10] Didier Rémy and Boris Yakobowski. A graphical presentation of MLF types with a linear-time incremental unification algorithm. Extended version, of [9], January 2008.

[11] Didier Rémy and Boris Yakobowski. An implicitly typed version of graphical MLF. Draft, available at `http://gallium.inria.fr/~remy/mlf`, 2008.

[12] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: inference for higher-rank types and impredicativity. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 251–262, New York, NY, USA, 2006. ACM Press.

[13] Joe B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

A. Proofs

Proof of Lemma 1

Let n be a type node. Consider a node n' such that $n \circ\!\!\xrightarrow{\pi}\!\!\prec n'$. The proof is by induction on π.

If π is the empty path, $n' = n$ which is indeed a type node.

If π is $n \circ\!\!\longrightarrow n'' \circ\!\!\xrightarrow{\pi'}\!\!\prec n'$: n'' cannot be a G-node, by well-sortedness. Hence n'' is a type node. Conclusion by induction hypothesis.

If π is $n \longleftarrow\!\!\prec n'' \circ\!\!\xrightarrow{\pi'}\!\!\prec n'$: n is a type node, hence no G-node is bound on it. Consequently, n'' is a type node. Conclusion by induction hypothesis. ■

Proof of Property 4

Let us call χ_r the constraint derived from χ_p by binding any type node n on the first G n' node such that $n \succ\!\!\xrightarrow{+}\!\!\rightarrow n'$ (if the binder of

n is already a G-node, its binder is unchanged). Notice that χ_r is an ML constraint.

Notice that χ_r can be derived from χ_p by applying a *multi-raise* operator to all the type nodes. This operator raises all nodes bound on a given node; it preserves well-domination by [10, Lemma 1]. Thus $\chi_p \sqsubseteq^R \chi_r$ holds **(1)**, since all nodes have flexible permissions.

Consider now an instantiation edge e. Let χ'_p be the MLF propagation of e in χ_p, χ'_r the ML one. By hypothesis, $\chi'_p \sqsubseteq \chi_p$ holds **(2)**.

Notice that $\chi'_p \sqsubseteq^R \chi'_r$ holds **(3)**. Indeed, given a G-node g, $\mathcal{I}_{\chi_p}(g) = \mathcal{I}_{\chi_r}(g)$, thus the expanded nodes in both χ'_p and χ'_r have the same shape. Then χ'_r is also the result of multi-raising all type nodes in χ'_p. Consider a node n of χ'_p. It is bound at the same node in χ'_r as in χ_r. Thus we can apply [10, Lemma 10] to $\chi'_p \sqsubseteq \chi_r$ (proven by (1) and (2)) and to each node raised in (3). We obtain $\chi'_r \sqsubseteq \chi_r$, which proves that χ_r is a presolution.

We have thus proven that χ_r is an ML presolution of χ. Since χ_r is an instance of χ_p, all instances of the expansion of χ_r are instances of the expansion of χ_p. In particular, all solutions witnessed by χ_r are also witnessed by χ_p. This is the desired result. ■

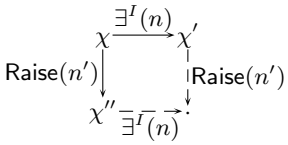
Proof of Lemma 2

Assume $\chi \Vdash^p \chi'$. By definition, any presolution χ_p of χ is a presolution of χ'. This implies $\chi' \sqsubseteq \chi_p$, which is exactly the result. Assume now that $\chi' \sqsubseteq \chi_p$ holds for any χ_p presolution of χ **(1)**. Consider such a presolution χ_p. By (1), $\chi' \sqsubseteq \chi_p$. By construction, χ_p is a presolution **(2)**. By (1) and (2) χ_p is a presolution of χ'. This proves $\chi \Vdash^p \chi'$. ■

Proof of Lemma 3

The proof is by case disjunction on the instance operation. If the operation is in $\sqsubseteq_1^G$, $\sqsubseteq_1^W$ or $\sqsubseteq_1^M$, the commutation is immediate.

If the operation is $\sqsubseteq_1^R$, we use the following names:



We prove that $\mathsf{Raise}(n')$ can be applied to χ'. This will also prove that $\exists^I(n)$ can be applied to χ'', as the result will be a well-formed graph.

By definition, n' is raisable in χ. [10, Lemma 1] implies that, $\forall n'', n'' \succ\!\!\!\rightarrow \overleftarrow{\chi}(n'), n'' \neq n' \implies n' \notin \mathcal{B}_\chi(n'')$ **(1)** (indeed, $n' \in \mathcal{B}(n)$ is trivially equivalent to $\mathcal{B}(n') \subset \mathcal{B}(n)$). This is equivalent to $\forall \pi, \neg(n' \circ\!\!\!\!\!\!\rightarrow^{\pi} n'' \in \chi)$ **(2)**.

Consider now *new* paths in χ', *i.e.* paths that link two nodes in χ' but not χ. Necessarily they contain the edge $\overleftarrow{n} \longleftarrow\!\!\!\!\!\!\cdot\ n$. Consider a path π starting from n'. Since n' is raised, it is a type node. Lemma 1 shows that all nodes reachable from n' are also type nodes. Hence there are no new paths starting from n'. Hence (2) still hold in χ', which proves that n' is raisable. ■

Proof of Lemma 4

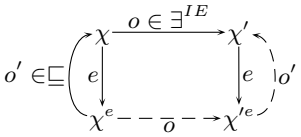
Let n the root of the existential elimination, o the instance operation. The proof is by case disjunction on o.

- $o = \mathsf{Graft}(\tau, n')$ **or** $o = \mathsf{Weaken}(n')$**:** If $n' \in \mathcal{C}(n)$, eliminating n in χ' gives χ''. Otherwise the two operations commute.
- $o = \mathsf{Raise}(n')$**:** If $n' \in \mathcal{C}(n)$ and n' is not bound on n, eliminating n in χ' gives χ'. If n' is bound on n, eliminating n' and n (the order is unimportant) in χ' gives χ''. If $n' \notin \mathcal{C}(n)$, the two operations commute.
- $o = \mathsf{Merge}(n_1, n_2)$**:** If n_1 and n_2 are both in $\mathcal{C}(n)$, eliminating n in χ' gives χ''. If none are in $\mathcal{C}(n)$, the two operations commute. If one node is in $\mathcal{C}(n)$, and the other is not, either n_1 or n_2 is n. Then $\chi' = \chi''$.

■

Proof of Lemma 5

- χ' and χ expand to the same type, by definition of expansion and existential elimination; hence they witness the same solutions.
- χ' is solved. Consider indeed a constraint edge e of χ'. It is also an edge of χ, as EEI preserve constraint edges. This edge is solved in χ, since χ is a presolution.
  - If e is a unification edge, it is still trivially solved in χ'.
  - If e is an instantiation edge, consider χ^e and χ'^e. We prove the dashed lines of the diagram below.



Propagation and EEI commute, as existential nodes are not copied during expansion. Hence χ'^e can be obtained from χ^e by the same EEI o which transforms χ into χ'.

Consider now the instance operation o' solving the unification edge introduced in χ^e:

- If the EEI is an introduction, Lemma 3 ensures that we can apply the same steps to χ'. This solves the edge and the resulting constraint is χ plus the EEI, which is exactly χ'. Hence e is solved in χ'.
- If the EEI operation was an elimination, notice that the instance steps solving the edge are such that $\chi^e \sqsubseteq \chi$. Consequently, they do not change the existential part that is being removed by o (the kernel of $\sqsubseteq$ is the identity relation). Hence, in this case, existential elimination and instance commute (Lemma 4), and the same argument as in the previous case proves that e is solved. ■

Proof of Lemma 6

Consider a constraint χ, and χ' obtained by performing an existential elimination from χ.

Suppose that χ has a presolution χ_p. Lemma 4 shows that there exists χ'_p such that $\chi_p \exists^E \chi'_p$ and $\chi' \sqsubseteq \chi'_p$ Lemma 5 ensures that χ'_p is a presolution witnessing the same solutions as χ_p. Thus χ'_p is a presolution of χ'. Hence any solution of χ is a solution of χ'.

Conversely, suppose χ' has a presolution χ'_p. Lemma 3 shows that there exists χ_p such that $\chi \sqsubseteq \chi_p$ and $\chi'_p \exists^I \chi_p$. Lemma 5 ensures that χ_p is a presolution witnessing the same solutions as χ'_p. In particular, χ_p is a presolution of χ. Hence any solution of χ' is a solution of χ. ■

Proof of Lemma 7

The proof is by induction on the length of $\chi \sqsubseteq \chi'$. If it is 0, the result is immediate. Otherwise, it is of the form $\chi \sqsubseteq_1 \chi'' \sqsubseteq \chi'$. We suppose without loss of generality that all grafting performed only graft one constructor, as in the hypotheses of [10, Lemma 9]. This is possible by the results of [10, Section 6.2.3] and [10, Theorem 1].

We prove that e is solved in χ'', by case disjunction on the operation o such that $\chi'' = o(\chi)$. (Afterwards the conclusion is by induction hypothesis applied to χ''.) Notice that, since e is solved, a derivation of $\chi^e \sqsubseteq \chi \sqsubseteq \chi''$ exists **(1)**.

If o is not $\mathsf{Raise}(d)$*:* Notice that, since $\mathcal{I}(g)$ is unchanged (hence $\mathcal{F}(g)$), the expansion creates the same nodes in χ and χ''. Hence $\chi''^e = o(\chi^e)$ holds (**2**), by Lemma 3.

- $o = \mathsf{Graft}(\tau, n)$:

  By (2) and the shape of o, $\chi^e \sqsubseteq_1^G \chi''^e$. Moreover, we have supposed that the graftings between χ and χ'' are atomic. Hence, by [10, Lemma 9] and (1), we obtain $\chi''^e \sqsubseteq \chi''$.

- $o = \mathsf{Raise}(n)$:

  By (2) and the shape of o, $\chi^e \sqsubseteq_1^R \chi''^e$; moreover, n is bound at the same node in χ''^e and χ''. By [10, Lemma 10] and (1), we obtain $\chi''^e \sqsubseteq \chi''$.

- $o = \mathsf{Merge}(n_1, n_2)$

  By (2) and the shape of o, $\chi^e \sqsubseteq_1^M \chi''^e$. By definition, n_1 and n_2 are merged in χ''. By [10, Lemma 11] and (1), we obtain $\chi''^e \sqsubseteq \chi''$.

- $o = \mathsf{Weaken}(n)$

  We consider an ordered derivation of $\chi^e \sqsubseteq \chi''$, in which weakenings and merging are performed bottom-up, a node n' being weakened before it is merged (this is the strategy used in the proof of [10, Theorem 4]). The operation $\mathsf{Weaken}(n)$ necessarily appears in this derivation. Also, all other operations only involve nodes of the expansion, while n is outside it. Hence, this weakening operation commutes with all previous operations in the derivation. By (2), this proves $\chi''^e \sqsubseteq \chi''$.

If $o = \mathsf{Raise}(d)$ In this case, $\chi''^e = (\mathsf{Raise}(d) ; \mathsf{Raise}(F^c) ; \mathsf{Raise}(r))(\chi^e)$, where $F = \mathcal{F}(g)$ and r the root of the expanded part. This proves that $\chi^e \sqsubseteq^R \chi''^e$ (**3**).

Let us next prove that all nodes of F are bound strictly above $\overset{\succ}{d}$ in χ (**4**). Consider a node $n \in F$. Let π be such that $\langle g \cdot 1 \rangle \circ\!\!\xrightarrow{\pi}\!\! n$. Since e is solved, $d \circ\!\!\xrightarrow{\pi}\!\! n$ also holds. By well-domination, all nodes of $\mathcal{F}(g)$ must be bound at least as high as d. Now, since d can be raised in χ, no node structurally under d can have been bound on $\overset{\succ}{d}$ in χ (by [10, Lemma 1]). This proves the desired subresult.

By (4), the nodes of F^c are bound lower in χ''^e than in χ''. By construction, r and d are bound at the same node in χ''^e and χ''. By applying [10, Lemma 10] $|F|+2$ times to (1) and (3), we obtain $\chi''^e \sqsubseteq \chi''$. ■

Proof of Lemma 8

We distinguish the instance relation on graphic types, written $\overset{g}{\sqsubseteq}$, and the one on constraints, written $\overset{\chi}{\sqsubseteq}$. Disregarding existential nodes, $(\overset{\chi}{\sqsubseteq}) \subset (\overset{g}{\sqsubseteq})$ holds, as G-nodes cannot be merged, raised or weakened. However, given two correct constraints χ and χ' with the same binding structure for G-nodes, the relation $\chi \overset{g}{\sqsubseteq} \chi'$ trivially implies $\chi \overset{\chi}{\sqsubseteq} \chi'$ (**1**).

One (apparent) difference between graphic types and graphic constraints is that some mixed paths disappear when existential nodes are raised. This is examplified by the constraints χ and χ_2 in Figure 21. By raising the $\bot$ node, the mixed path $\langle g \rangle \circ\!\!\prec \langle g \cdot 1 \rangle$ disappears. However, this difference is unimportant because we can add some structure edges, as we prove below.

In the following, we consider a constraint χ containing an unification edge $e = n_1 \blacktriangleright\!\!\cdots\!\!\blacktriangleleft n_2$. By definition, both n_1 and n_2 are type nodes.

Let S be the set of existential nodes of sort Type of χ. Let f_S be the function adding a structure edge $\overset{\succ}{\chi}(n) \circ\!\!\rightarrow n$ for any node n of S. (We temporarily add to our formalism the needed family of constructors G_k for $k \geq 1$.)

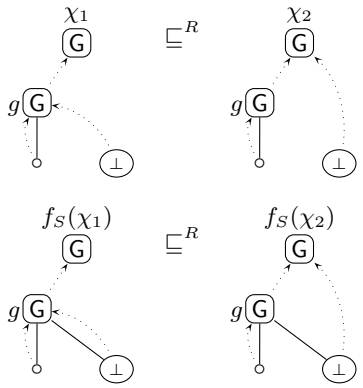


Figure 21. Mixed paths and raisings of existential nodes

Any instance χ' of χ (for $\overset{\chi}{\sqsubseteq}$) is in the domain of f_S. Conversely, the codomain of f_S is the set of instances of $f_S(\chi)$. Thus defined, f_S is a bijection (**2**).

Notice that the relation $\chi \overset{\chi}{\sqsubseteq} \chi' \iff f_S(\chi) \overset{\chi}{\sqsubseteq} f_S(\chi')$ holds (**3**). The result is immediate for grafting, merging and weakening. For raising, it is straightforward to adapt the proof of Lemma 3. (Indeed, we could have existentially introduced $g \longleftarrow\!\!\prec \mathsf{G} \circ\!\!\rightarrow n$ instead of the structure edge $g \circ\!\!\rightarrow n$.)

Notice also that $f_S(\chi)$ can be viewed as a graphic type, as the only existential nodes it contains are G-nodes, for which we could also add structure edges. Finally, notice that Unif behaves exactly the same way on χ or $f_S(\chi)$ (**4**), as it does not "see" the new edges, which are introduced on nodes above n_1 and n_2.

Soundness: Suppose Unif finds a unifier χ' for e. Then $\chi \overset{g}{\sqsubseteq} \chi'$ holds. However, χ' has the same binding structure for G nodes as χ (**5**), since Unif will never try to merge, weaken or raise nodes above n_1 and n_2, and there are no G-nodes under n_1 or n_2 (Lemma 1). Thus (1) implies $\chi \overset{\chi}{\sqsubseteq} \chi'$, and Unif is sound.

From now on, we suppose there exists an unifier χ_s of e in χ for $\overset{\chi}{\sqsubseteq}$. Then $f_S(\chi) \overset{\chi}{\sqsubseteq} f_S(\chi_s)$ holds by (3). Moreover $f_S(\chi_s)$ is a unifier of e in $f_S(\chi)$ (for $\overset{g}{\sqsubseteq}$ or $\overset{\chi}{\sqsubseteq}$) (**6**).

Completeness: Unif is complete on graphic types. Hence, by (6), it returns a unifier when called on $f_S(\chi)$. Hence, by (4) it returns a unifier when called on χ, and is complete.

Principality: Unif is principal on graphic types. Hence, by (6) it returns a principal unifier when called on $f_S(\chi)$. This type is an instance of $f_S(\chi)$ for $\overset{g}{\sqsubseteq}$. In fact, by (1) and (5), it is also an instance for $\overset{\chi}{\sqsubseteq}$. By (2), let χ_p be the constraint such that this unifier is $f_S(\chi_p)$.

By principality on graphic types, $f_S(\chi_p) \overset{g}{\sqsubseteq} f_S(\chi_s)$ holds. Hence $f_S(\chi_p) \overset{\chi}{\sqsubseteq} f_S(\chi_s)$ also holds by (1), (5) and the definition of χ_s. Finally, $\chi_p \overset{\chi}{\sqsubseteq} \chi_s$ holds by (3). This proves that χ_p is a principal unifier of e in χ for $\overset{\chi}{\sqsubseteq}$. This is exactly the desired result, as Unif returns χ_s when called on χ (by (4)). ■

Proof of Lemma 9

Suppose that the unification of e fails. Suppose now that χ has a presolution χ_p. By definition, e is solved in χ_p, hence χ_p is a unifier of e in χ. Contradiction with the completeness of the unification algorithm (Lemma 8).

Consider now a presolution χ_p of χ. By definition, $\chi \sqsubseteq \chi_p$ holds. Since e is solved in χ_p (by definition of presolutions), we have $\chi \sqsubseteq \chi' \sqsubseteq \chi_p$ by principality of unification (Lemma 8). This ensures that χ and χ' have the same presolutions. ■

Proof of Lemma 10

The proof steps are given in Figure 22.

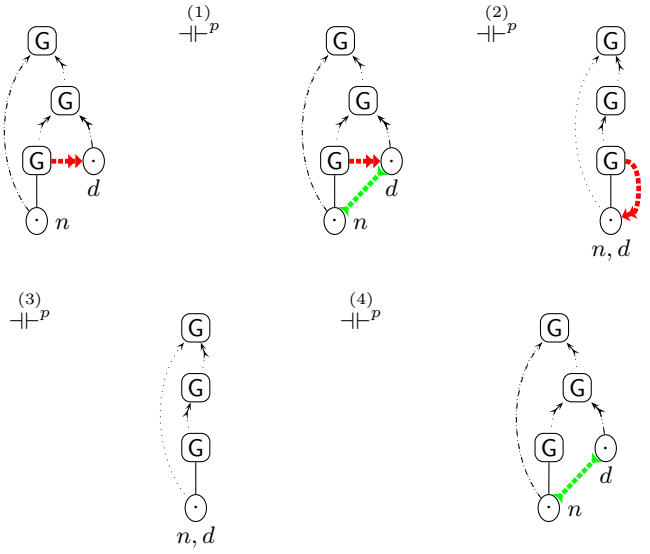


Figure 22. Proofs steps for INST-ELIM-MONO

Step (1) is by equality of presolutions. Indeed, the G-node is degenerate in any instance of both constraints, so the only way to solve the instantiation edge is by merging n and d. (More precisely, by merging d with n^c, which is itself merged with n.)

Step (2) is by unification (Lemma 9).

Step (3) is by equality of presolutions, with the same reasoning as for Step (1).

Step (4) is by UNIF again. ■

Proof of Lemma 11

Let $e = g \dashrightarrow d$. We proceed by case analysis on the operation p such that $\chi' = p(\chi)$. In many cases, the relation $\chi^e \sqsubseteq \chi'^e$ directly holds, which implies the result. We implicitly use the fact that the copy n^c of a node has always more permissions than n.

- p is $\mathsf{Graft}(\tau, n)$
  - If $n \in \mathcal{I}(g)$, then $\chi'^e = (p\,;\mathsf{Graft}(\tau, n^c))(\chi^e)$
  - If $n \notin \mathcal{I}(g)$, then $\chi'^e = p(\chi^e)$
- p is $\mathsf{Weaken}(n)$
  - If $n \in \mathcal{I}(g)$ and $n \neq \langle g \cdot 1\rangle$, then $\chi'^e = (p\,;\mathsf{Weaken}(n^c))(\chi^e)$.
  - If $n \notin \mathcal{I}(g)$ or $n = \langle g \cdot 1\rangle$, then $\chi'^e = p(\chi^e)$.
- p is $\mathsf{Merge}(n_1, n_2)$
  - The case $n_1 \in \mathcal{C}(g)$, $n_2 \notin \mathcal{C}(g)$ (or the symmetrical case) is impossible. Indeed, the only possibility would be that $n_1 = g$ (or $n_2 = g$), but G-nodes cannot be merged.
  - If $n_1, n_2 \in \mathcal{I}(g)$: $\chi'^e = (p\,;\mathsf{Merge}({n_1}^c, {n_2}^c))(\chi^e)$
  - If $n_1, n_2 \notin \mathcal{C}(g)$ or $n_1 \in \mathcal{I}(g)$ and $n_2 \in \mathcal{C}(g) - \mathcal{I}(g)$ (or the symmetrical case), then $\chi'^e = p(\chi^e)$
- p is $\mathsf{Raise}(n)$, with $n \neq d$
  - If $n \in \mathcal{I}(g)$ and $n \succ\!\!\not\longrightarrow g$ and $\neg(n \succ\!\!\longrightarrow\!\!\succ\!\!\longrightarrow g)$, then $\chi'^e = (p\,;\mathsf{Raise}(n^c))(\chi^e)$.
  - If $n \notin \mathcal{I}(g)$, or $n \in \mathcal{I}(g)$ and $n \succ\!\!\longrightarrow\!\!\succ\!\!\longrightarrow g$, $\chi'^e = p(\chi^e)$.
  - $n \in \mathcal{I}(g)$, $n \succ\!\!\longrightarrow g$. In this case, the commutation is not immediate. Indeed, in χ, n is in the interior, while it is in the exterior in χ'. The idea is to merge the copied nodes under n^c in χ^e with the nodes under n. However, this cannot be done until n has been raised high enough for this.

    Consider indeed χ^e. The node n^c can be raised in this constraint, because n could be raised in χ. Then let u' be a unification edge between n^c and n in χ^e. This edge is admissible, as $\overset{\succ}{n^c}$ is a G-node after the raising. Let us call χ'' this constraint. We have $\chi'' \Vdash \chi^e$ by dropping of constraints and instance.

    Moreover, the principal unifier of this edge in χ^e is exactly χ'', which shows $\chi^e \sqsubseteq \chi''$, hence $\chi'' \Vdash^p \chi^e$ **(1)**.

    Notice next that n is in the structural frontier of g in χ'. The only difference between χ'^e and χ'' is that there are some nodes under n^c in χ''. However, those nodes are exactly the same nodes as under n, and n and n^c are linked by a unification edge. Thus χ'' and χ'^e are equivalent for $\dashv\vdash^p$, which concludes with (1).
- p is $\mathsf{Raise}(d)$
  - $d \notin \mathcal{I}(g)$: Then $\chi'^e = (\mathsf{Raise}(d);\mathsf{Raise}(F);\mathsf{Raise}(n^c))(\chi^e)$ where $F = \mathcal{F}(g)$ and $n = \langle g \cdot 1\rangle$.
  - $d \in \mathcal{I}(g)$: notice that be definition of instantiation edges, $d \succ\!\!\longrightarrow g$ must hold. Then $\chi'^e = (\mathsf{Raise}(d)\,;\,\mathsf{Raise}(F)\,;\,\mathsf{Raise}(n^c)\,;\,\mathsf{Raise}(d^c)\,;\,\mathsf{Merge}(d^c, d))(\chi^e)$ with the same notations as above. ■

Proof of Lemma 12

Let χ be a constraint, e one of its instantiation edges.

Consider a presolution χ_p of χ. By definition, it is an instance of χ. By applying Lemma 11, we obtain $\chi_p^e \Vdash^p \chi^e$. By definition of presolutions, we also have $\chi_p^e \sqsubseteq \chi_p$, hence $\chi_p \Vdash^p \chi_p^e$, and $\chi_p \Vdash^p \chi^e$ by transitivity of $\Vdash^p$. Hence, χ_p is a presolution of χ^e, and $\chi \Vdash^p \chi^e$.

Consider now a presolution χ_p of χ^e. By definition, it solves all the unification edges of χ^e, in particular those resulting from the expansion. All introduced existential nodes are thus merged in χ_p, and $\chi \sqsubseteq \chi_p$ holds. This proves $\chi_p \Vdash^p \chi$. Thus, $\chi^e \Vdash^p \chi$ also holds. ■

Proof of Lemma 13

We show the property for atomic instantiation steps. The general case follows by induction. In each case, we show that direct dependency may only decrease.

Weakening: The dependency relation remains unchanged.

Grafting: The interiors of G-nodes are enlarged, but with new nodes that do not contain instantiation edges. Hence, the direct dependency relation remains unchanged.

Raising: The interiors of G-nodes may only decrease (if the node raised is bound on a G-nodes, its subgraph leaves the interior of this node, otherwise all G-nodes interiors are left unchanged). Hence the relation diminishes or remains unchanged.

Merging: The interiors of G-nodes are left unchanged, as G-nodes are never merged and so merging can only occur *inside* the interior of G-nodes. ■

Proof of Lemma 14

Let E be the set of unification edge resulting from the propagation.

1. χ' is an instance of χ:

   The existential structure introduced during the propagation is merged with the existing structure under d when the unification edges are solved.

2. The nodes of $\mathcal{C}_\chi(g)$ have not been instantiated in χ':

   By definition of the unification algorithm, the nodes changed are those reachable by $\circ\!\xrightarrow{*}$ from a node constrained by an edge in E. These nodes are either fresh (*i.e.* created by the propagation), under d, or under a node in the frontier (hence the exterior) of g. Hence, we just need to prove that no node under d is in $\mathcal{C}(g)$.

   By contradiction, suppose there exists n such that $n \in \mathcal{C}(g)$ and $d \circ\!\xrightarrow{*} n$. By well-domination, either $g \succ\!\xrightarrow{*} d$, or $d \succ\!\xrightarrow{*} g$. The second case is forbidden by the hypothesis on d and g. The first case is impossible because d is a type node, and g a G-node.

3. e is solved in χ'

   The diagram presented in Figure 23 holds.

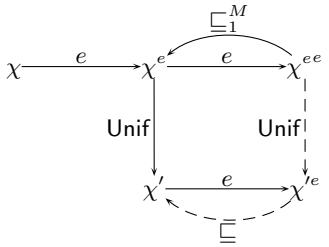


Figure 23. Propagation and unification

   Let us first justify the edge $\chi^{ee} \xrightarrow{\text{Unif}} \chi'^e$. Propagating e a second time in χ^e is similar to doing an existential introduction. The new nodes are not considered by the unification algorithm, so the first expanded nodes are merged exactly as in χ'. The square closes because the interior of g have not changed, so the propagation creates the same nodes in χ' and χ^e.

   Next, $\chi^{ee} \sqsubseteq^M \chi^e$ holds (by merging the copies of the nodes of the structural frontier, and the root of the two propagations). This implies $\chi^{ee} \sqsubseteq \chi'$. Hence χ' is an unifier of the edge introduced by the second introduction in χ^{ee}. This means that it is an instance of the principal unifier of this edge, which is χ'^e. (Indeed, the two propagated parts being equal, unifying one unification edge or the other yields the same graph.) Hence the relation $\chi'^e \sqsubseteq \chi'$ holds, which is exactly the desired result. ■

Proof of Theorem 1

We reason on constraints without unification edges, as those can be solved by unification without loss of generality (**1**) (Lemma 9).

Let χ be a constraint. We extend the relation "depends on" to two instantiation edges by $f \dashrightarrow\!\!\!\!\!\rightarrow d$ depends on $f' \dashrightarrow\!\!\!\!\!\rightarrow d'$ iff f depends on f'. This relation is a strict partial order, since the dependency relation on breakpoints is acyclic. Moreover, it only decreases (as a relation) by instance, as does the dependency relation (proof of Lemma 13).

We say that an instantiation edge $e = g \dashrightarrow\!\!\!\!\!\rightarrow d$ is recursively solved if g is recursively solved and e is solved. The proof is by induction on the number of instantiation edges not recursively-solved.

If this number is 0, χ is solved: all instantiation edges are recursively solved (hence solved), all unification edges have been solved (1). Hence we choose χ as its own more general presolution.

Otherwise, let $e = g \dashrightarrow\!\!\!\!\!\rightarrow d$ be a non recursively solved instantiation edge minimal for the dependency order. Since all unification edges are solved (1), this implies that g is recursively solved and e is not (**2**) (otherwise, we would consider one of the unsolved edges constraining g instead).

One step: We propagate e in the resulting graph and solve the resulting unification edges.

If there is no solution, χ is unsolvable (Lemmas 12 and 9). Hence, we now suppose that the unification has succeeded (otherwise there are no presolutions to reason on).

If the unification has succeeded, the resulting graph χ' has again the same presolutions as χ (Lemmas 12 and 9 again). Moreover, it is an instance of χ and e is solved in χ' (Lemma 14) (**3**).

All the instantiation edges recursively solved in χ are still solved in χ' (and hence recursively solved (**4**)):

The nodes of χ changed by the unification are those structurally under d. Now d is in at most one structural interior of a G-node (as those nodes are existentially introduced). Let us consider each case:

d **is in no structural interior:** all the nodes under d for $\circ\!\xrightarrow{\pm}$ are in no G structural interior either. Hence solving the unification did not change any such interior. Lemma 7 ensures that all edges previously solved are solved.

d **is in** $\mathcal{I}(g')$**:** (for a G-node g'). All the nodes under d for $\circ\!\xrightarrow{\pm}$ are in $\mathcal{I}(g')$ or outside of any structural interior of a G-node. Hence, the only such interior changed by unification is g'. Since d is in $\mathcal{I}(g')$, s' depends on s. By hypothesis (2), e is not solved, hence g' is not recursively solved. In particular, the edges originating from g' are not recursively solved. Hence, by Lemma 7 again, all edges recursively solved are still solved in χ'.

Conclusion: The edge e is recursively solved, as g is recursively solved (2) and e is solved (3). By hypothesis, e was not recursively solved. Hence, using (4), at least one more edge is recursively solved in χ'. Moreover, we have already proved that χ' is an instance of χ, and that it has the same presolutions as χ. Hence we can conclude by induction hypothesis.

Remark: the most general presolution is unique: Let χ_p and χ'_p be two potentially most general presolution (obtained by choosing the edges to propagate in different ways). We obtain immediately $\chi_p \sqsubseteq \chi'_p$ and $\chi'_p \sqsubseteq \chi_p$. The kernel of instance is equality [10, Lemma 2], hence the result. ■

Proof of Lemma 15

Let χ be the constraint. The direction "typable in ML implies typable in MLF" is proven in Property 3. Suppose then that χ is typable in MLF. The proof of Theorem 1 show that its principal presolution χ_p contains only flexible edges, as unification does not introduce fresh binding edges, and propagation only copies existing subgraphs. Thus we can apply Property 4 to χ_p, which gives us an ML solution to χ. ■

Proof of Lemma 16

Let g' be the name of the copy of g. Let χ_C be $C(\chi)$. The proof is by case disjunction on the instance operation o. In each case we describe the constraint χ'_C closing the diagram.

- $o = \mathsf{Graft}(\tau, n)$ with $n = \langle g \cdot \pi \rangle \in \mathcal{C}(g)$
  $\chi'_C = (\mathsf{Graft}(\tau, n)\ ;\ \mathsf{Graft}(\tau, g' \cdot \pi))(\chi_C)$

- $o = \mathsf{Weaken}(n)$ with $n = \langle g \cdot \pi \rangle \in \mathcal{C}(g)$
  $\chi'_C = (\mathsf{Weaken}(n)\,;\mathsf{Weaken}(g' \cdot \pi))(\chi_C)$
- $o = \mathsf{Raise}(n)$ with $n = \langle g \cdot \pi \rangle \in \mathcal{C}(g)$, n not bound on g
  $\chi'_C = (\mathsf{Raise}(n)\,;\mathsf{Raise}(g' \cdot \pi))(\chi_C)$
- $o = \mathsf{Raise}(n)$ with $n = \langle g \cdot \pi \rangle \in \mathcal{C}(g)$, n bound on g
  $\chi'_C = (\mathsf{Raise}(n)\,;\mathsf{Raise}(g' \cdot \pi)\,;\mathsf{Merge}(n, g' \cdot \pi))(\chi_C)$
- $o = \mathsf{Merge}(n_1, n_2)$ with $n_1 = \langle g \cdot \pi_1 \rangle$, $n_2 = \langle g \cdot \pi_2 \rangle$, $n_1, n_2 \in \mathcal{C}(g)$
  $\chi'_C = (\mathsf{Merge}(n_1, \langle g' \cdot \pi_1 \rangle)\,;\mathsf{Merge}(n_2, \langle g' \cdot \pi_2 \rangle))(\chi_C)$
- In all the other cases: $\chi'_C = C(\chi_C)$
  Note that, for the case $o = \mathsf{Merge}(n_1, n_2)$ and $g \dashrightarrow n_1$, $g \dashrightarrow n_2$, we either have to allow multiple instantiation edges between nodes, or to allow a single instantiation edge to be split into two edges. ■

Proof of Lemma 18

Let χ be a constraint. Let C be an application of INST-COPY. Let χ_C be $C(\chi)$.

Consider a presolution χ_p of χ. Let χ_{C_p} be the constraint obtained by applying Lemma 16 to χ_C and χ_p. By hypothesis, χ_p is solved, hence it does not contain unification edges, and all instantiation edges are solved. By its definition, χ_{C_p} does not contain unification edges. Moreover, all its instantiation edges can be solved, by using the steps that solve them in χ_p. Finally, χ_p and χ_{C_p} witness the same solutions, as the copy does not change the expansion. Hence all solutions of χ are solutions of χ_C.

In the other direction, we cannot apply the same approach. Indeed, the two copies of g could seemingly be instantiated in different ways. However, this is in fact never useful, because the most general presolution of χ' will instantiate the two copies of the G-node similarly.

Let χ_p be the most general presolution of χ. We show by induction on the number of steps leading to χ_p in the proof of Theorem 1 that the most general presolution of χ_C is $C(\chi_p)$.

If there are 0 steps, χ is solved. Then χ_C is solved, applying INST-COPY to a presolution yields a presolution. Moreover, both constraints witness the same solutions, as the copy does not change the expansion.x

Otherwise, there are two possible kinds of steps:

1. The unification of a unification edge e; let χ' be the resulting constraint. We unify e and its eventual copy in χ_C. Lemma 17 ensures that the resulting constraint is $C(\chi')$. This constraint is more general than χ_{C_p} (Lemma 9).
2. The propagation of an instantiation edge e. Then we propagate e and its eventual copy in χ_C. The result is trivially equal to $C(\chi^e)$ and is more general than χ_{C_p} (Lemma 12).

In both cases, conclusion is by induction hypothesis applied to the obtained constraint. ■

Proof of Lemma 19

For VAR-ABS, the result is by unification, INST-ELIM-MONO and existential elimination.

For VAR-LET, the proof steps are shown in Figure 24. We have $\chi_4 \dashv\vdash \chi_3$ by EXISTS-ELIM and $\chi_2 \Vdash \chi_1, \chi_3$ by dropping of constraints.

$\chi_1 \Vdash \chi_2$: let χ_p be a presolution of χ_1, plus the additional edge $g_1 \dashrightarrow n$ (thus χ_p is an instance of χ_2). We will prove that χ_p is a presolution of χ_2.

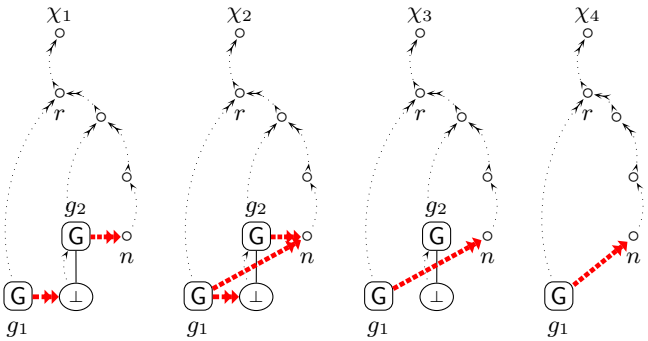


Figure 24. Steps proving VAR-LET

Consider the constraint χ'_p obtained by propagating $g_1 \dashrightarrow n$ in χ_p. Let T be the sequence of instance operations solving the propagation for the edge $g_1 \dashrightarrow \langle g_2 \cdot 1 \rangle$ (since χ_p is a presolution of χ_1). We suppose without loss of generality that T is done bottom-up, and that the last operation is $\mathsf{Merge}({g_1}^c, \langle g_2 \cdot 1 \rangle)$; let T' be T without this last operation.

Next, consider a path π such that $\langle g_1 \cdot 1 \cdot \pi \rangle \in \mathcal{F}(g_1)$. Then by definition of presolutions and propagation, $\langle g_1 \cdot 1 \cdot \pi \rangle = \langle g_2 \cdot 1 \cdot \pi \rangle$. Moreover, given the shape of the binding tree in χ_1, we have $\langle g_2 \cdot 1 \cdot \pi \rangle \in \mathcal{F}(g_2)$. Next, again by definition of presolutions and propagation, we have $\langle g_2 \cdot 1 \cdot \pi \rangle = \langle n \cdot \pi \rangle$.

Thus T' can almost directly be used to act on the expansion in χ'_p, as the nodes in the frontier of g_2 are shared with n. The only difference is that we must change the way nodes leaving $\mathcal{I}(g_2^c)$ are raised. Indeed, the binding tree above $\breve{n}$ and g_2 is not the same. Without loss of generality, we suppose that T' is such that all nodes ultimately outside g_2^c are first raised so that they are all bound at $\breve{g}_2$, then multi-raised until they are all bound at r (since they must be bound above the least common ancestor of g_1, g_2 and n), and then freely raised (this last step is unimportant for us). We then adapt T' so that the multi-raising are changed to account for the difference in binding heights between $\breve{g}_2$ and n. This preserves well-domination, as multi-raising preserves well-domination. We call T'' the resulting transformations, and χ''_p the constraint $T(\chi'_p)$.

Let us examine χ''_p. By construction, g_2^c could be merged with $\langle g_2 \cdot 1 \rangle$ if the expansion had been done on χ_2. In particular, up to expansion (*i.e.* binding and flag reset), $\mathcal{I}(\langle g_2 \cdot 1 \rangle)$ and $\mathcal{I}(g_2^c)$ are equal (**1**). In χ''_p however, the merging is not possible yet, because the nodes under n might be further instantiated.

Consider now χ''_2, where e is $g_2 \dashrightarrow n$. Since χ_p is a presolution of χ_1, this propagation can be solved. Let χ'''_p be the constraint obtained by solving only the unification edges resulting from the nodes on the frontier. By (1), it is in fact the case that $\chi'''_p = \chi''_p$. Indeed, in both constraints, the nodes in the frontier of the expansions are the nodes in the frontier of g_2, and the nodes in the interior of the expansion are the interior of g_2, up to expansion (in χ'''_p, unifying the nodes of the frontier has not changed the interior of the expansion). Thus the remainder of the propagation in χ''_p can be solved.

We have thus proven that the edge $g_1 \dashrightarrow n$ is solved. Thus χ_p is a presolution, in particular of χ_2, and $\chi_1 \Vdash \chi_2$ (in fact, in this case we have proven equality of the shape of the presolutions).

$\chi_4 \Vdash \chi_2$: let χ_p be a presolution of χ_4. Let χ'_p be χ_p plus g_2, the bottom node under it, and the two remaining instantiation edges. The unification edges introduced by propagating $g_1 \dashrightarrow \langle g_2 \cdot 1 \rangle$ can be solved:

- the bottom node under g_2 creates no constraint for the skeleton or the term-graph.
- since χ_p is a presolution, all nodes in $\mathcal{F}(g_1)$ are already bound above the least common binder of g_1 and n, which is the root of the graph here. Thus solving the unification edge does not raise a node already in χ_4.

We call χ''_p the resulting constraint. Notice that in this constraint, g_1 and g_2 have exactly the same frontier, and their structural interior is the same (up to expansion). Thus $g_2 \dashrightarrow n$ is solved, and χ''_p is a presolution. Since it witnesses the same solutions as χ_4, this proves the subresult. ■

Proof of Lemma 20

Let χ and χ' be the left and righ-hand sides of the rule respectively. One implication is obvious, as $\chi \sqsubseteq \chi'$

Suppose χ_p is an ML presolution of χ. Let us call m the node $\langle g \cdot 1 \rangle$. We argue will prove that m and n can be merged in χ_p, and that the result is a presolution.

Consider indeed χ^e_p. By hypothesis, $\chi^e_p \sqsubseteq \chi_p$. Consider χ^e_p in which the unification edges resulting from the frontier are solved. We call χ'_1 this constraint. Let m^c be the copy of m in the propagation, *i.e.* the root of the expansion. In χ'_1 there is exactly the same structure under m and m^c. Thus those two nodes can be unified, and we call χ'_2 the resulting constraint. Notice that the nodes under m^c are unchanged by this unification (**1**). Indeed, the unification will not change the nodes in $\mathcal{F}(g)$ (they are already merged in χ'_1), while the copies in the nodes of $\mathcal{I}(g)$ are all bound on $\overset{\vee}{n}$. Thus the nodes under m will be raised, while those under m^c will not. It remains to unify the edge between m^c and n. This is possible by (1), by following the same steps that those solving this edge in χ'_1. We call χ'_3 the resulting constraint.

Let us prove that χ'_3 is a presolution. We use Lemma 7, and consider the structural interiors that have been changed by unifying m and n. Unification only changes the nodes under the nodes unified, so we must only consider the structural interiors of the nodes under m and n, *i.e.* the structural interior of g and of g' (where g' is the G-node whose structural interior contains n. Notice that g' might not exist if n is existential in χ'_3).

By (1), the nodes under n have not been changed by the unification. Thus all instantiation edges leaving g' (if it exists) are still solved. Moreover, e is the only instantiation edge leaving g, and it is solved by construction of the unification. Thus all instantiation edges are solved in χ'_3, and it is a presolution.

Next, notice that unifying m and n raises n' so that it is bound at least at $\overset{\vee}{g}$ in χ'_3. Thus $\chi' \sqsubseteq \chi'_3$ holds, by [10, Lemma 10]. Moreover, χ and χ'_3 witness the same solutions, as the nodes under $\langle \epsilon \rangle$ are unchanged by the unification of m and n. This proves the other direction of the result. ■