From ML to ML^F : Graphic Type Constraints with Efficient Type Inference

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 D.3.3 [*Programming Languages*]: Language Constructs and Features—Constraints; Polymorphism; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; D.3.2 [*Language Classifications*]: Applicative (functional) languages

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 have 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 $\mathit{apply}\ \ a_1\ \ a_2$ 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 original 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 [10]. In this work, we extend graphic types to address the question of type inference. We do not adapt the original type inference algorithm [2] by replacing its unification algorithm on syntactic types with the new, more efficient unification algorithm on graphic types [10] because 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 parametrizing our type inference system with the actual set of types that is being used, rediscovering a known efficient type inference algorithm for ML [8, 9]. 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 levels, 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.
- Our constraint system is in fact implicitly parametrized 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 identify a subset of constraints in *solved forms*, and use these to give a semantics to our constraints as sets of types.
- We identify a set of *acyclic* constraints, that include all typing constraints, and have decidable principal solved forms.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.
Copyright © 2008 ACM 978-1-59593-919-7/08/09. . . $5.00

- 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

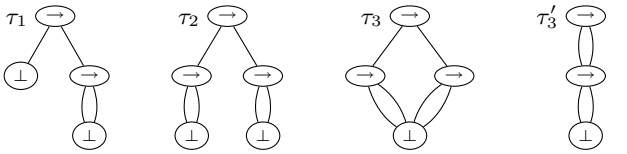


Figure 1. Graphic ML types

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 labeled 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 usually 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 from n to n' is written $n \circ\!\!\rightarrow n'$. For example, in τ_1, $\langle 2 \rangle \circ\!\!\rightarrow \langle 21 \rangle$.

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 reasoning[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 the graphic type representing $\gamma \rightarrow \gamma$ under $\langle 1 \rangle$. 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 must reflect this mechanism in graphic type inference. To this effect, types are extended into *constraints*. We first introduce a new type constructor G of arity one, so as to distinguish *types* from *type schemes*. Indeed, G-nodes indicate where polymorphism may be introduced. We then associate to each variable node a *binding edge*[2] which goes to the G-node where the variable is bound. Hence, G-nodes can be seen as introducing *generalization levels* (hence their names).

In particular, G-nodes are used to type let constructs and it is important that they can be nested. Moreover, we do not want them to appear inside types. Both requirements can be fulfilled by stratifying constraints: all G-nodes are in the top-most part, above the *type part* of the constraint. Each G-node but the root is bound to another G-node, and can only be accessed by its binding edge.

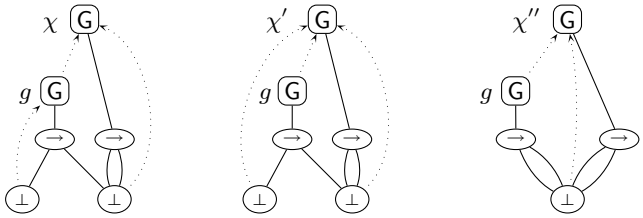


Figure 2. Constraints with G-nodes

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 $n \succ\!\!\rightarrow g$ to say that n is bound at g. Given a node n in χ, there is at most one g such that $n \succ\!\!\rightarrow g$, called the *binder* of n, and written $\overleftarrow{n}$. In all three constraints of Figure 2, we have $g \succ\!\!\rightarrow \langle \epsilon \rangle$ and $\langle 11 \rangle \succ\!\!\rightarrow \langle \epsilon \rangle$. 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 $\forall\,(\alpha)\ \alpha \rightarrow \beta$, 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 $\alpha \rightarrow \beta$, which

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

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

is monomorphic in the context of g. The root node represents the same type scheme $\forall(\beta)\ \beta \to \beta$ in all three constraints.

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 level. Readers familiar with rank-based ML type inference [8, 9] 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 modeled by *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 merged. (In drawings we do not represent unification edges whose two extremities are the same node.)
- 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. Being "an instance of" will be precisely defined 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 have an arrow type τ whose domain 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.

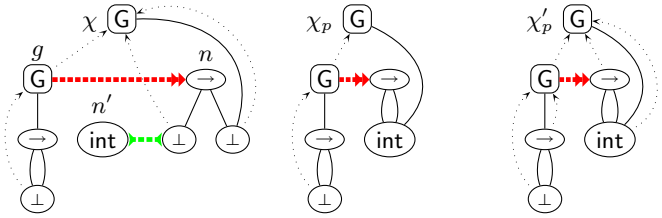


Figure 3. Typing id 1

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 g. 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, *i.e.* 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. Instead, we choose to bind all nodes explicitly to the enclosing G-node they belong to. A fully-bound version of the type χ_p of 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 a set of constraint edges from the typing environment, meant to constrain the nodes corresponding to the free variables of the expression. Each edge is labeled by the variable it constrains. In drawings we represent such a set of edges as an edge ⇢, generally omitting the labels.

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 its own generalization level, 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.

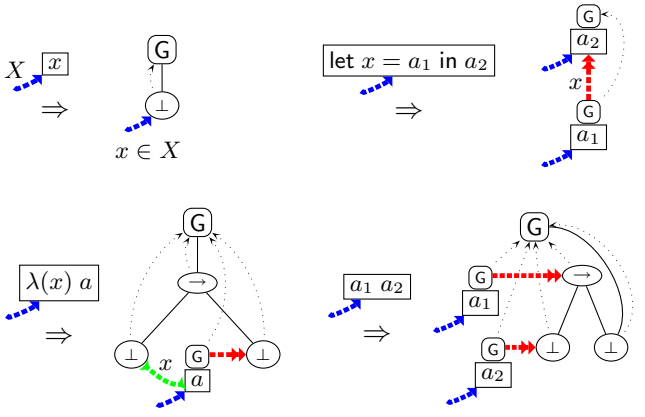


Figure 4. Typing of primitive expressions

- 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, with the additional constraint that x must be an instance of a_1. The other (free) variables of a_1 and a_2 are constrained by the typing environment.
- 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

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

the type 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 must be 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.

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.

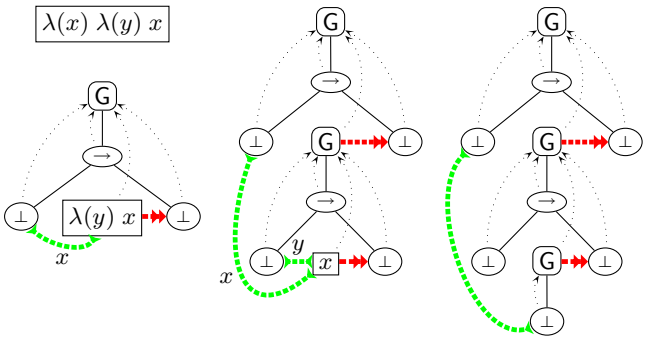


Figure 5. Typing constraints for $\lambda(x)\ \lambda(y)\ x$

2. An overview of ML$^\mathsf{F}$ and ML$^\mathsf{F}$ graphic types

2.1 ML$^\mathsf{F}$ 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 $\mathsf{choose\ id}$, 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, ML$^\mathsf{F}$ enriches types with a new form of (bounded) quantification: $\mathsf{choose\ id}$ receives the type $\forall(\alpha \geqslant \sigma_{id})\ \alpha \to \alpha$. The variable α is allowed to range over all possible instances 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 ML$^\mathsf{F}$ 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, ML$^\mathsf{F}$ 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. Interestingly, inlining rigid bounds as in $\sigma_{id} \to \mathsf{int} * \mathsf{char}$ provides a very good and intuitive approximation of types, correct from a semantic standpoint (albeit not from a type inference point of view).

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

Sharing inside types is of paramount importance in ML$^\mathsf{F}$. 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. ML$^\mathsf{F}$ graphic types have originally been introduced in part to directly capture these notions inside the representation of types [10]. They also provide a more canonical representation of types, and permit a straightforward definition of the type instance relation between types.

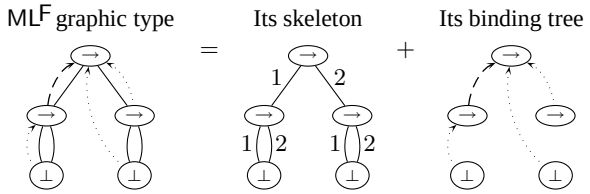


Figure 6. An example of ML$^\mathsf{F}$ graphic type

ML$^\mathsf{F}$ graphic types can be decomposed into a first-order quantifier free skeleton, and a *binding tree* that tells for every node *where* and *how* it is bound. In particular, we use edges rather than nodes for quantifiers, as it leaves the structure invariant by extrusion of quantifiers. Figure 6 shows an example of a type and its decomposition.

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 $\succ\!\!\longrightarrow$ 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 $n \overset{\geqslant}{\succ\!\!\longrightarrow} n'$ and $n \overset{=}{\succ\!\!\longrightarrow} n'$ for flexible and rigid edges respectively, or as $n \overset{\diamond}{\succ\!\!\longrightarrow} n'$ where $\diamond$ stands for either $\geqslant$ or $=$.

We write $\circ\!\!\prec$ for $(\circ\!\!\longrightarrow) \cup (\longleftarrow\!\!\prec)$, called *mixed edges*. Let $\longrightarrow$ range over $\circ\!\!\longrightarrow$, $\succ\!\!\longrightarrow$ and $\circ\!\!\prec$. We write $\overset{*}{\longrightarrow}$ for the reflexive transitive closure of $\longrightarrow$, and $\overset{+}{\longrightarrow}$ for the transitive closure. We write $(N \longrightarrow)$ for $\{n' \mid \exists n \in N, n \longrightarrow n'\}$.

All superpositions of a graphic type with a binding tree do not form an ML$^\mathsf{F}$ graphic type. Indeed, the resulting graph must be *well-dominated*: the binder of a node n must dominate n for the relation $\circ\overset{+}{\prec}$. 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 $\langle \epsilon \rangle \longleftarrow\!\!\prec \langle g1 \rangle \circ\!\!\longrightarrow \langle g11 \rangle$).

2.3 The instance relation

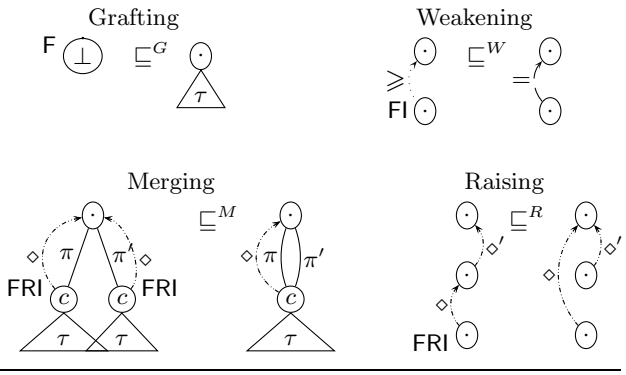


Figure 7. Instance operations

The instance relation on ML$^\mathsf{F}$ 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$^\mathsf{F}$ 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 is used to extrude polymorphism. If we consider the ML$^\mathsf{F}$ type $\forall\,(\alpha \geqslant \forall\,(\beta)\ \beta \to \beta)\ \alpha \to \alpha$ of 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 $\sqsubseteq$ that is too expressive would allow—and thus require for principality—guessing polymorphic types, making type inference undecidable [14]. In ML$^\mathsf{F}$, $\sqsubseteq$ is the restriction of such a larger instance relation $\leq$. The missing operations in $\leq \setminus \sqsubseteq$ are then made available *explicitly*, through the use of user-provided type annotations.

Permissions The instance operations presented in Figure 7 are only sound in certain contexts. For example, the graphic type of Figure 6 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 operation under the node $\langle 1 \rangle$, such as grafting the node $\langle 11 \rangle$, is unsound.

The operations allowed or forbidden on a node n mainly depend on its *permissions*, which are determined by the binding flags $\geqslant$ or $=$ on the binding edges above n. It is a key point of ML$^\mathsf{F}$ 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 $\langle \epsilon \rangle \xleftarrow{=} \langle 1 \rangle \xleftarrow{\geqslant} \langle 11 \rangle$; hence it is a L-node. Node $\langle 1 \rangle$ is a R-node, while all other nodes are F-nodes.

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. Conversely, rigid edges request polymorphism. Hence, on R or L-nodes, we must at least forbid the transformation of nodes with flexible edges, in order to remain sound.

However, there exists an exception. 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, which prevents its instantiation. Formally, a node n is said to be *inert* and called an I-node, if for any variable node n' such that $n' \rightarrowtail^* n$, there is at least one rigid edge between n' and n. Inert nodes include *monomorphic* nodes, on which no variable node is bound at all (for example all the nodes in a graphic representation of $\mathsf{int} \to \mathsf{int}$). Following the reasoning above, all operations are sound at inert nodes.

We can now reread the definition of $\sqsubseteq$ in Figure 7 with permissions in mind. Nodes with permission F allow all transformations, including the grafting or variables. Nodes with either permission R or I allow weakening, raising or merging, either because they contain no polymorphism (if they are inert), or because the polymorphism is protected by a rigid edge, which is preserved by the transformation.

A more thorough discussion of permissions can be found in [12].

2.4 Graphic constraints as an extension of graphic types

We can see graphic constraints as a small extension of ML$^\mathsf{F}$ graphic types, which allows reusing all the results already established on the latter.

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 following, 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 [12]. 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 on graphic types [12] for graphic constraints:

DEFINITION 1. We say that a unification edge n_1 ⟡ n_2 is *admissible* if either it is admissible on graphic types or $n_1 \xrightarrowtail{+} g$ and $n_2 \rightarrowtail g'$ where g and g' are G-nodes. □

We require unification constraints to relate two type nodes (the shape of G-nodes, which is in close correspondence with the λ-terms being typed, must be invariant), 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 $\vec{n}$ are not in the same partition for the relation $\circ\!\!\rightarrow$. 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.

Instantiation edges An instantiation edge $g \dashrightarrow n$ 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 $(\langle\epsilon\rangle \circ\!\!\prec) \setminus N$, *i.e.* all the nodes of χ but those in N. The *projection* of χ, written $proj(\chi)$, 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 $g \dashrightarrow n$ 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 levels. In essence, nodes bound above g are not generalizable at the level of g, 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 $(n \stackrel{*}{\prec\!\!\longleftarrow})$ 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.* $\mathcal{C}(n) \cap (n \stackrel{*}{\circ\!\!\longrightarrow})$.

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 $\mathcal{I}(n)$. □

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 also contains the leftmost existential arrow node.

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 trees 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.)

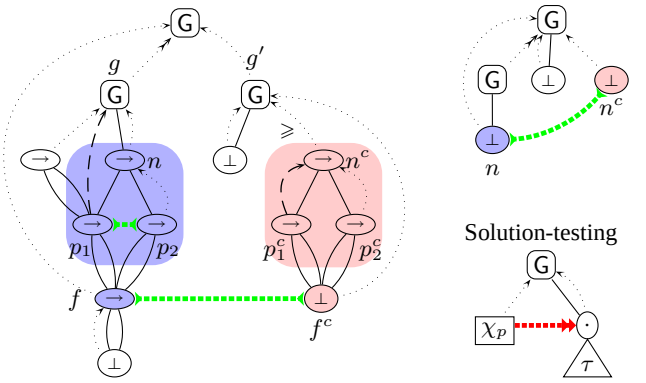


Figure 9. Examples of ML$^\mathsf{F}$ expansion and solution-testing

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;
- 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 $p \succ\!\!\stackrel{\diamond}{\longrightarrow} g$, adding the binding edge $p^c \succ\!\!\stackrel{\diamond'}{\longrightarrow} p'$, where
  - in ML, $(\diamond', p')$ is $(\geqslant, g')$ (notice that $\diamond$ is necessarily $\geqslant$)
  - in ML$^\mathsf{F}$, $(\diamond', p')$ is $(\diamond, n^c)$ if p is not n, or $(\geqslant, g')$ if p is n. □

An illustration of an ML$^\mathsf{F}$ 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) and their copies (n^c, p_1^c, p_2^c and f^c, on the right).

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*.

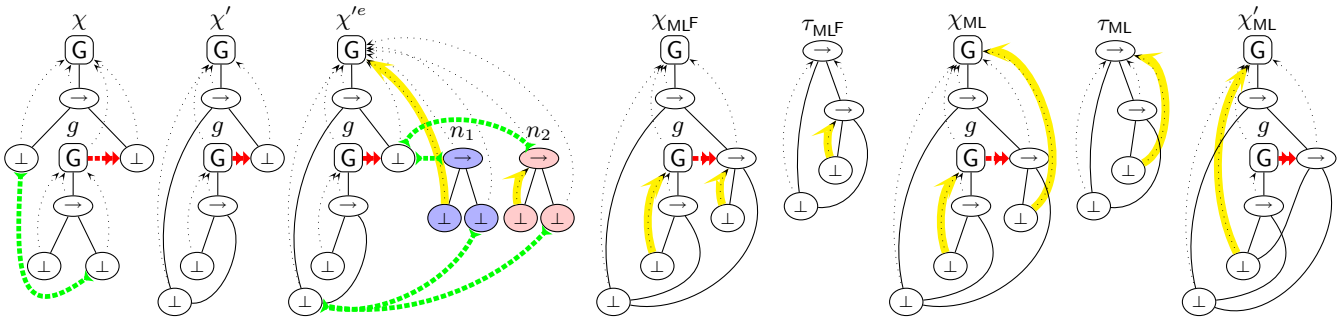


Figure 8. Typing $\lambda(x)\ \lambda(y)\ x$

ML *versus* ML$^\mathsf{F}$ *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), and then an ML$^\mathsf{F}$ expansion (under n_2). 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 ML$^\mathsf{F}$ expansion $\langle n_2 \cdot 1\rangle$ is bound on n_2, creating *inner* polymorphism, forbidden in ML.

Notice that, by definition, ML$^\mathsf{F}$ expansion is always more general than ML expansion: the former can be obtained from the latter by performing a few raisings afterward.

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 ML$^\mathsf{F}$ 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 types which a presolution expands to, plus all the instances of those types. In particular, the set of solutions is closed by instance. Notice that a solution can be witnessed by more than one 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 is contained in 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 $\chi_{\mathsf{ML}^\mathsf{F}}$, 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 $\chi_{\mathsf{ML}^\mathsf{F}}$ 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 ML$^\mathsf{F}$ *and* ML presolutions of χ. Interestingly, $\chi_{\mathsf{ML}^\mathsf{F}} \sqsubseteq \chi_{\mathsf{ML}} \sqsubseteq \chi'_{\mathsf{ML}}$ holds. In fact, $\chi_{\mathsf{ML}^\mathsf{F}}$ is the principal presolution of χ in ML$^\mathsf{F}$, as we will prove in §5.

The types corresponding to the expansions of $\chi_{\mathsf{ML}^\mathsf{F}}$, χ_{ML} and χ'_{ML} are $\tau_{\mathsf{ML}^\mathsf{F}}$, τ_{ML} and τ_{ML} respectively. Hence $\tau_{\mathsf{ML}^\mathsf{F}}$ 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 $\tau_{\mathsf{ML}^\mathsf{F}}$ 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 [11]; of course, different presolutions will give different decorations of a.

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 node 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

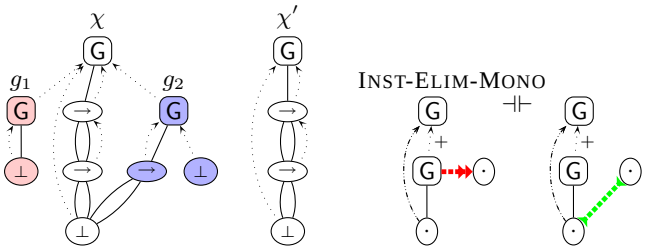


Figure 10. Simplifying unconstrained existential nodes and degenerate instantiation edges

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* ML$^\mathsf{F}$** It is immediate to prove that ML$^\mathsf{F}$ extends ML. Indeed, the ML instance relation is a subrelation of the one in ML$^\mathsf{F}$, and an instantiation edge solved in the ML sense is also solved in the ML$^\mathsf{F}$ sense (as ML$^\mathsf{F}$ expansions are more general).

PROPERTY 2. *All ML (pre)solutions are ML$^\mathsf{F}$ (pre)solutions.* □

Interestingly, ML$^\mathsf{F}$ 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.

PROPERTY 3. *Consider an ML constraint χ with an ML$^\mathsf{F}$ presolution χ_p in which all binding edges are flexible. Then there exists ML$^\mathsf{F}$ solutions of χ witnessed by χ_p that are ML types, and those types are also ML solutions of χ.* □

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.

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 $\chi \setminus \mathcal{C}(n)$. □

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'.

LEMMA 1. *Existential elimination preserves solutions.* □

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 this interior leaves the expansion of g unchanged. We can in fact lift this property to propagation, and by extension, to solved instantiation edges:

LEMMA 2. *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.* □

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

Unification edges The level of generalization we brought to our graphic representation is small enough that the unification algorithm on unconstrained graphic types [10] can be reused unchanged. The principality of unification on graphic types also ensures that unification edges can always be solved eagerly.

LEMMA 3. *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.* □

Interestingly, unification on ML graphic types can be solved with the unification algorithm for ML$^\mathsf{F}$ graphic types. This follows from the facts that type instance for ML is a subrelation of type instance for ML$^\mathsf{F}$ and that the unification algorithm of ML$^\mathsf{F}$ 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 [8, 9].

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.

LEMMA 4. INST-ELIM-MONO *preserves solutions.* □

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 3, 4 and 1.

Eager propagation A crucial property of our framework is that scheme expansion and propagation are essentially[4] monotonic w.r.t. to instance $\sqsubseteq$. An important consequence of this property is that we may propagate any instantiation edge in any constraint without changing its presolutions.

LEMMA 5. *Propagation preserves presolutions.* □

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

DEFINITION 11. A G-node g *depends* on another G-node g' if g' constrains the constraint interior of g, or if g is in the scope of g', *i.e.* $(\exists n \in \mathcal{C}(g), g' \dashrightarrow n) \vee (g \succ\!\!\stackrel{+}{\longrightarrow} g')$. □

[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.

DEFINITION 12. 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: instantiation edges follow the scopes of the variables of the expression, which are nested.

5.1 Finding a principal presolution

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

LEMMA 6. *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.* □

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. Afterward, 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 6). Moreover, since the constraint is acyclic, all instantiation edges already visited (hence solved) remain solved (Lemma 2).

The preservation of presolutions follows from Lemma 3 for steps 1 and 2b and from Lemma 5 for step 2a.

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

A corollary of this result is the fact that the (structural) interior of an unconstrained G-node g will never be instantiated in the principal presolution of a constraint. Hence, after having propagated an instantiation edge e leaving from g, it is safe to remove e.

COROLLARY 1. *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 $\chi^e \setminus e$.* □

***Typability in unannotated* ML^F *and* ML** 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 3, 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.* □

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.

6. Type annotations

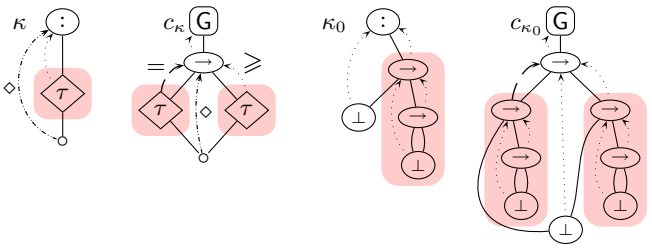


Figure 11. Types of coercion functions

Type annotations are a key to ML^F. 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 11. 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 11. 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-

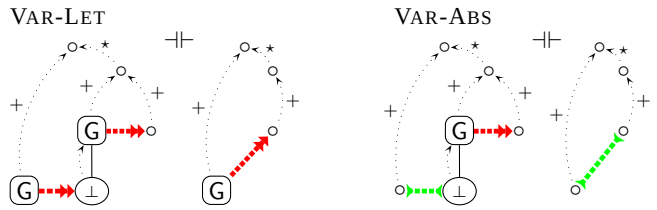
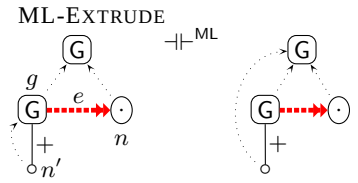


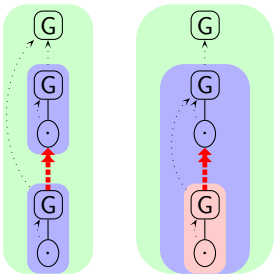
Figure 12. Simplifying the typing of variables



Condition:

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

Figure 13. 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 14. 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 12.

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 13, a type node inside a type scheme that is "used" only once and at the nearest generalization node 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.

7.2 Complexity analysis

While type inference for ML is DEXP-TIME complete (when types need not be output), McAllester has shown [7] 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 14 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 [7, 8]. In particular, type schemes are also simplified in an innermost fashion. Unification in MLF 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 MLF) has linear complexity.

Notice that, if we restrict ourselves to ML, using the constraint simplification of Figure 13 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 MLF if and only if they are typable in ML, the complexity bound for MLF cannot be better that the one for ML. We thus have established the exact complexity bound $2^{O(n)}$ for type inference in MLF.

8. Examples of typings

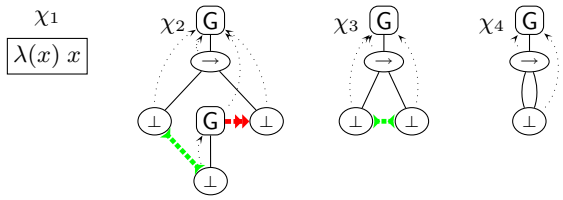


Figure 16. Typing $\lambda(x)\ x$

Figure 16 presents the typing of the identity, valid in both ML and MLF. The step χ_2 to χ_3 is by VAR-ABS. χ_4 is by unification. The resulting principal type is $\forall\,(\alpha)\ \alpha \to \alpha$, abbreviated as σ_{id}.

Figure 15 presents the typing of let $y = \lambda(x)\ x$ in $y\ y$ in MLF. 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 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 17) 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 $c_{\kappa_{\text{id}}}\ x$. We have also simplified on the fly the instantiation edge on $c_{\kappa_{\text{id}}}$ 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 15. 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

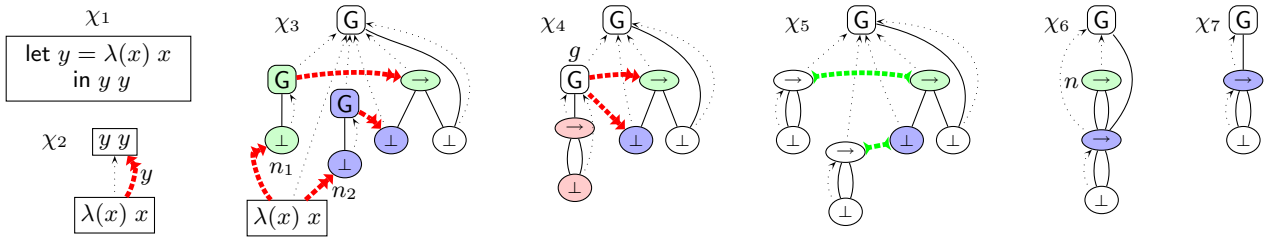


Figure 15. Typing let $y = \lambda(x)\ x$ in $y\ y$

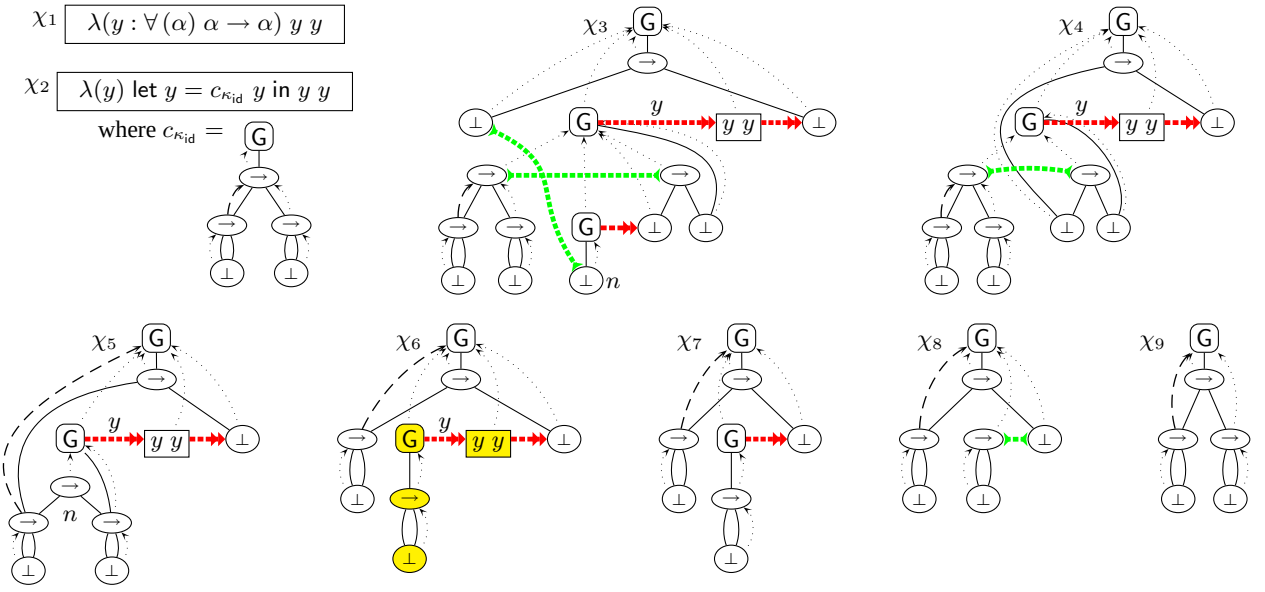


Figure 17. Typing $\lambda(y : \forall(\alpha)\ \alpha \to \alpha)\ y\ y$

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 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.

9. Comparison with other works

A detailed comparison between MLF and other extensions of System F can be found in [3]. The most closely related work [5] proposes an interesting restriction of MLF, called HML, in which rigid quantification is treated up to sharing and thus inlined, but which requires more type annotations—namely, all parameters of functions that are polymorphic need a type annotation. Interestingly, this restriction seems to be expressible directly in our framework, which should thus be easily applicable to perform type inference in HML. This would provide HML with an efficient type inference algorithm. Notice that while sharing of rigidly bound nodes is unnecessary in the definition of HML, it should remain essential in the implementation to maintain efficiency.

Another system, called FPH [13], uses System-F types externally. However its specification introduces "boxes" inside types to keep track of impredicative instantiations. Since type inference in FPH is internally performed using the MLF type inference mechanisms, and since FPH seems to be a subset of MLF (in fact of HML), we believe that FPH and type inference for FPH could also be expressed in our framework.

More generally, many recent works [4, 13, 6, 5] aim at finding a type system with second-order polymorphism that assigns System-F (or simplified MLF types) to expressions. All of those systems are less expressive than MLF, and our graphic presentation of type inference should help compare these alternatives—and perhaps explore others more systematically.

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 [7, 8, 9]. Merging two multi-equations in [9] 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 [8].

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 studying 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 MLF because MLF expansion is more complicated than the ML one.

Typing constraints for ML have been explored in detail [8]. 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 [10] (and further increased later [12]). Hence, the type inference system we have presented is slightly more general; however, we do not know of a short, simple, and uncontrived 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, and only in a stratified, restricted version of MLF in which types are not as general as those presented here or in the original presentation. Moreover, type inference has not been addressed in this revised syntactic version of MLF.

Conclusion

We have extended the initial presentation of graphic types [10] 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 use presolutions to interpret terms into a fully explicit, Church-style language, which is itself proved sound.

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, our graphical approach might be a 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 and Yann Régis-Gianas for numerous 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 *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07), Freiburg, Germany*, pages 111–122, October 2007.

[5] Daan Leijen. Flexible types: robust type inference for first-class polymorphism. Technical Report MSR-TR-2008-55, Microsoft Research, March 2008.

[6] Daan Leijen. HMF: Simple type inference for first-class polymorphism. In ICFP'08 [15].

[7] 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.

[8] 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.

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

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

[11] Didier Rémy and Boris Yakobowski. A Church-style intermediate language for MLF. Submitted, available at `http://gallium.inria.fr/~remy/mlf`, 2008.

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

[13] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. FPH: First-class polymorphism for Haskell. In ICFP'08 [15].

[14] 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.

[15] *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08), Victoria, British Columbia, Canada*. ACM Press, September 2008.