INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A graphical presentation of MLF *types with a linear-time local unification algorithm.*

Draft version

Didier Rémy — Boris Yakobowski

N° ????

January 2008

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--????--FR+ENG

INRIA
ROCQUENCOURT

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

Draft version

Didier Rémy, Boris Yakobowski

Thème SYM — Systèmes symboliques
Projet Gallium

Rapport de recherche n° ???? — January 2008 — 40 pages

Abstract: ML^F is a language that extends ML and System F and combines the benefits of both. We propose a dag representation of ML^F types that superposes a term-dag, encoding the underlying term structure with sharing, and a tree encoding the binding structure. Compared to the original definition, this representation is more canonical, as it factors out most of the notational details; it is also closely related to first-order terms. Moreover, it permits a simpler and more direct definition of type instance that combines type instance on first-order term-dags, simple operations on the binding tree, and a control that allows or rejects potential instances. Using this representation, we build a linear-time unification algorithm for ML^F types, which we prove sound and complete with respect to its specification.

Key-words: System F, ML^F, Unification, Types, Graphs, Binders

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Une présentation par graphes de ML^F avec un algorithme d'unification local linéaire

Résumé : ML^F est un langage qui étend ML et le système F, tout en combinant les avantages des deux systèmes. Nous proposons une représentation par dag des types de ML^F, qui superpose un dag encodant la structure sous-jacente de terme avec partage explicite, et un arbre encodant la structure des lieurs. Comparée à la définition originelle, cette représentation est plus canonique, car elle évite la plupart des détails de notation; elle est également très proche de celle utilisée pour les termes du premier ordre. Par ailleurs, elle permet une définition plus simple et plus directe de la relation d'instance entre types, en combinant instance sur les types du premier ordre, des opérations simples sur l'arbre des lieurs, et un contrôle de permissions qui autorise ou rejette certaines instances. En utilisant cette représentation, nous construisons un algorithme d'unification de complexité linéaire. Nous prouvons celui-ci correct et complet vis-à-vis de sa spécification.

Mots-clés : Système F, ML^F, Unification, Types, Graphes, Lieurs

This report is an extended version of [13]. It is electronically available in both color and black-and-white versions[1]*. This is the black-and-white version.*

1 Introduction

The language MLF [7] has been proposed for smoothly combining the advantages of ML-style type inference [2] with the expressiveness of System-F first-class polymorphism [4]. MLF is a conservative extension of ML that allows to type all System-F terms [7]. MLF terms are partially annotated. All functions that use their parameter in a polymorphic way—and only those—need an annotation. In particular, ML terms do not.

MLF comes with a type inference algorithm: every well-typed source program provided with some annotations has a principal type—*i.e.* one of which all other correct types are *instances*. The typing rules of MLF are a simple generalization of those of ML, and are quite straightforward. Moreover, they can be presented as a particular instance of a simple generic type system that generalizes both ML and System F [8]. This system is parameterized by the exact language of types and a *type instance* relation between types. Unfortunately, while type instance and a subrelation called *abstraction* play a key role in MLF, they are defined by purely syntactic means and with little intuitive support. So far, these relations were mainly justified a posteriori by the properties of MLF. A more semantic-based definition has been proposed but only for a significant restriction of the language and only for the instance relation [8].

We propose an alternative definition of types based on an (acyclic) graph representation. More precisely, types have an underlying term-graph structure, similar to the representation of simple types with sharing, an additional binding tree, and further properties relating the two. The existence of a graphic presentation for MLF-types had already been suggested [6], but it was not sufficiently well-understood to be used formally. Graphic types are more canonical, as they factor a lot of the syntactical artifacts that can be found in the original syntactic definition of MLF. Both representations are isomorphic (and converting from one into the other has linear complexity), but reasoning on graphic types is easier.

We define instantiation on graphs as a combination of simple transformations that include the following three parts: instantiation of the first-order term-graph, simple transformations on the binding tree, and a control process based on flags attached to the binding tree.

We also present a sound and complete linear-time unification algorithm on graphic types that finds the smallest instance of two types (for the instance relation). The algorithm follows the same pattern as the instance relation: unification of the first-order underlying term graph, computation of the least binding tree that is an instance of the ones of the input types, and a control of permissions rejecting some unsound instances.

Outline The paper is organized as follows. We briefly reintroduce suntactic MLF types (§2) as well as the definitions of terms, term-graphs and first-order unification (§3). We introduce the graphic representation of MLF types (§4). We define the instance relation on graphic types (§5) and study some of its properties (§6). We describe a unification algorithm for graphic types and prove it sound and complete (§7). Finally, we present an informal comparison between the syntactic and the graphic presentations of MLF, and give algorithms translating between syntactic and graphic types (§8). A table of notations is included in the appendix (page 39).

Notations

If a is a meta-variable ranging over some set A, we write $\overline{a}$ for a sequence of elements of A.

Let G be an arbitrary graph with nodes N and edges E labeled in L, *i.e.* E is a subset of $N \times L \times N$. We write E^{-1} for the symmetric relation of E, obtained by reversing the arrows in E.

We write $n_1 \xrightarrow{a} n_2 \in E$ for $(n_1, a, n_2) \in E$. Often, E may be left implicit and we simply write $n_1 \xrightarrow{a} n_2$. We may fix a label $a \in L$ and see $\xrightarrow{a}$ as the binary relation $\{(n_1, n_2) \mid (n_1 \xrightarrow{a} n_2)\}$. If $\overline{a}$ is a string of labels $a_1 \dots a_k$, we write $n_1 \xrightarrow{\overline{a}} n_k$ for $n_1 \xrightarrow{a_1} \dots n_{k-1} \xrightarrow{a_{k-1}} n_k$. Using regular expressions syntax, we write $n \xrightarrow{*} n'$ if there exists a string of labels $\overline{a}$ such that $n \xrightarrow{\overline{a}} n'$, and $n \xrightarrow{+} n'$ if, moreover, this string is non-empty.

Fixing one side of the arrow to a particular set of nodes N, we write $(N \longrightarrow)$ (*resp.* $(\longrightarrow N)$) for the set of nodes leaving (*resp.* reaching) a node in N.

We often see some relations as rewriting systems. Consequently, in the following we write $f\,;g$ for the inverse composition $g \circ f$. The semicolon notation emphasizes the order in which the rewritings can be done. Similarly,

[1] See `http://gallium.inria.fr/~remy/mlf/`.

given two relations, we write $\mathcal{R} \,;\, \mathcal{R}'$ for the composition of relations defined by $x \; (\mathcal{R} \,;\, \mathcal{R}') \; y \iff \exists z,\; x \; \mathcal{R} \; z \wedge z \; \mathcal{R}' \; y$.

Contents

1 Introduction **3**

2 A brief introduction to (syntactic) ML$^\mathsf{F}$ **6**
2.1 ML$^\mathsf{F}$ syntactic types . . . 6
2.2 Type instance . . . 6

3 First- and second-order types **7**
3.1 First-order terms . . . 7
3.2 Term-graphs . . . 8
3.3 Instance and unification on term-graphs . . . 9
3.4 Representing second-order types . . . 10

4 ML$^\mathsf{F}$ graphic types **11**
4.1 Representing ML$^\mathsf{F}$ types . . . 11
4.2 Well-formedness . . . 11
4.3 Invariants induced by well-formedness . . . 12
4.4 Operators for building graphs . . . 13

5 The instance relation on graphic types **15**
5.1 Shaping the instance relation . . . 16
5.2 Permissions . . . 18
5.3 Instance operations . . . 19
5.3.1 Grafting . . . 19
5.3.2 Merging . . . 20
5.3.3 Raising . . . 22
5.3.4 Weakening . . . 22
5.4 The instance relation . . . 22
5.5 Similarity . . . 23
5.6 The abstraction relation . . . 24

6 Properties of instance **25**
6.1 Reorganizing relations . . . 25
6.2 Big step instance subrelations . . . 26
6.2.1 Big step raising . . . 26
6.2.2 Big step merging and weakening . . . 26
6.2.3 Grafting unconstrained graphs . . . 27
6.3 Performing an instance operation early . . . 27

7 Unification **28**
7.1 Unification problem . . . 28
7.2 Admissible problems . . . 28
7.3 Unification algorithm . . . 29
7.4 Example of unification . . . 31
7.5 Correctness of the algorithms . . . 32
7.5.1 Auxiliary results . . . 32
7.5.2 Main correctness results . . . 32
7.6 Complexity . . . 33
7.7 Generalized unification problems . . . 33
7.8 Unification in restrictions of ML$^\mathsf{F}$ . . . 34

8 Relating the syntactic and graphic versions of ML^F 34
8.1 An informal comparison of syntactic and graphic instance 34
8.2 Translating graphic types to and from syntactic types 35
8.2.1 From graphic to syntactic types 35
8.2.2 From syntactic to graphic types 35

9 Conclusion 36

A Syntactic ML^F relations 38

B Table of notations 39

2 A brief introduction to (syntactic) MLF

2.1 MLF syntactic types

MLF types are parameterized by a set of of type symbols Σ, including at least the arrow symbol $\rightarrow$. We distinguish first-order types t from second-order types σ, which are both defined by the following grammar, in BNF form.

$$\begin{array}{lcl} t & ::= & \alpha \mid t \rightarrow t \mid \dots \\ \sigma & ::= & t \mid \bot \mid \forall\,(\alpha \diamond \sigma)\ \sigma \\ \diamond & ::= & \geqslant \mid = \end{array}$$

A first-order type t is defined as usual. A syntactic, second-order type σ is a first-order type t, a bottom type $\bot$ (which intuitively stands for the System-F type $\forall\alpha.\alpha$), or a quantified type $\forall\,(\alpha \diamond \sigma)\ \sigma'$. A difference with System-F is that quantification always assign bounds to variables. Bounds are themselves second-order types. Bounds are either *rigid* when introduced with the $=$ flag, or *flexible* when introduced with the $\geqslant$ flag. Intuitively, the meaning of $\alpha \diamond \sigma$ is that α ranges over types that are either equivalent to σ when the bound is rigid, or an instance of σ when the bound is flexible.

For example, the type $\forall\alpha.\ \alpha \rightarrow \alpha$ of System F can be represented in MLF as

$$\forall\,(\alpha \geqslant \bot)\ \alpha \rightarrow \alpha. \qquad (\sigma_{id})$$

We may omit trivial bounds and we often write $\forall\,(\alpha)\ \sigma$ for $\forall\,(\alpha \geqslant \bot)\ \sigma$. The System-F type $(\forall\alpha.\ \alpha \rightarrow \alpha) \rightarrow (\forall\alpha.\ \alpha \rightarrow \alpha)$ cannot be represented directly, as the grammar forbids such types as $\sigma_{id} \rightarrow \sigma_{id}$. We instead use an auxiliary variable with a rigid bound and write

$$\forall\,(\beta = \sigma_{id})\ \beta \rightarrow \beta. \qquad (\sigma_1)$$

One may still, at first, understand rigid bounds by expansion as if σ_1 standed for the ill-formed type $\sigma_{id} \rightarrow \sigma_{id}$.

In MLF, we can also write the type

$$\forall\,(\beta \geqslant \sigma_{id})\ \beta \rightarrow \beta. \qquad (\sigma_2)$$

Syntactically, it only differs from σ_1 by changing the rigid bound into a flexible one. This time however, expansion would be a misleading intuition—otherwise, rigid and flexible bounds would make no difference. Intuitively, σ_2 should rather be understood by the set of its instances, that is, all types $\forall\,(\beta = \sigma)\ \beta \rightarrow \beta$ such that σ is an instance of σ_{id}. In fact, σ_1 is itself an instance of σ_2.

The auxiliary variable β is used to share the two instances of σ on the left and right sides of the arrow. Thus, σ_2 is quite different from the type

$$\forall\,(\beta \geqslant \sigma_{id})\ \forall\,(\beta' \geqslant \sigma_{id})\ \beta \rightarrow \beta', \qquad (\sigma_3)$$

which stands for all types $\forall\,(\beta = \sigma)\ \forall\,(\beta' = \sigma')\ \beta \rightarrow \beta'$ such that σ and σ' are *independent* instances of σ_{id}. This is similar to the difference between types $\forall\gamma.\gamma \rightarrow \gamma$ and $\forall\gamma.\forall\gamma'.\gamma \rightarrow \gamma'$ in System-F.

Combining both forms of quantification, the type

$$\forall\,(\beta = \sigma_{id})\ \forall\,(\beta' \geqslant \sigma_{id})\ \beta \rightarrow \beta' \qquad (\sigma_4)$$

may be understood as the set of all types $\forall\,(\beta = \sigma_{id})\ \forall\,(\beta' = \sigma)\ \beta \rightarrow \beta'$ (*i.e.* intuitively $\sigma_{id} \rightarrow \sigma$) such that σ is an instance of σ_{id}.

2.2 Type instance

A peculiarity of MLF is its sophisticated instance relation $\sqsubseteq$ that can operate deeply under other quantifiers and, indirectly, under type structure, as illustrated with type σ_4 above.

While flexible and rigid bounds are often used in covariant and contravariant contexts, respectively, quantification in MLF also allows to instantiate the (flexible) bound of a variable that appears both covariantly and contravariantly, as in σ_2. This is actually a key to having principal types in MLF. This is made possible, while maintaining type-soundness, by enforcing all occurrences of the bound to simultaneously pick the same instance: the weaker the types in contra-variant position (typically of arguments), the weaker the types in co-variant position (typically of results).

Instantiation is always safe—and permitted—under flexible bindings, which *provide* some polymorphism but do not request it. Conversely, it is generally unsafe—and thus forbidden—under rigid ones, which *require* some polymorphism, and might have assumed it. While a function of type $\forall(\alpha)\ \alpha \to \alpha$ can be safely considered as a function of type $t \to t$ for any monotype t, it would be unsafe to consider a function of type $\forall(\beta = \sigma_{id})\ \beta \to \beta$ as a function of type $\forall(\beta = t \to t)\ \beta \to \beta$: the former requires its argument to be polymorphic (and returns a polymorphic result) while the latter only requires its argument to be of type $t \to t$. In the second case, this argument could then be erroneously applied to values of unexpected type.

While rigid bounds that occur in contravariant position cannot be instantiated for soundness of type-checking, it is a key design choice to forbid instances of all rigid bounds, so that type instantiation is then only driven by bound flags and never looks at variances. This makes type inference decidable, tractable, and actually relatively simple.

Still, it would always be sound and often useful to treat a function of type σ_1 as a function of type σ_4. To circumvent this limitation—and recover all uses of polymorphism—MLF introduces type annotations $(\_ : \sigma)$ that behave as explicit retyping functions of type $\forall(\alpha = \sigma, \alpha' \geqslant \sigma)\ \alpha \to \alpha'$. That is, $(a : \sigma)$ *explicitly* requires a to have type σ, and then allows it to be used with an instance of σ.

In fact, MLF still allows a very restricted form of instance under rigid bounds, called *abstraction* and written $\sqsubseteq$. Typically, abstraction may increase sharing by merging two variables with the same rigid bound, but may not instantiate flexible bounds. For instance, σ_1 is an abstraction of $\forall(\beta = \sigma_{id}, \beta' = \sigma_{id})\ \beta \to \beta'$—but not the converse. Abstraction may be distinguished from general instances, as its inverse relation $\sqsupseteq$ is sound and is only disallowed in order to keep type inference decidable. The remaining reversible part $\sqsubseteq \cap \sqsupseteq$, called type equivalence and written $\equiv$, captures syntactic artifacts such as renaming of bound variables, commutation of adjacent binders, removal of useless binders, and such.

In the original definition of type instance, places where inner instantiation or abstraction may actually occur are implicitly defined by contextual inference rules. Namely, instantiation may only occur under flexible quantifiers, called a flexible context, and abstraction may only occur under a sequence of rigid quantifiers itself in a flexible context. For example, abstraction is disallowed in the inner bound α_3 of $\forall(\alpha_1 = \forall(\alpha_2 \geqslant \forall(\alpha_3 = \sigma_3)\ \sigma_2)\ \sigma_1)\ \sigma$. While such a transformation appears to be sound from a semantic point of view, its naive integration would surprisingly break type soundness via ad hoc intricate interaction with type equivalence.

One of our main contributions is to revisit the instance relation (§5) using a graph presentation of types (§4). This new presentation eliminates most of the syntactic artifacts and so is more direct, allows more support for intuition, and supports extensions of the abstraction relation just mentioned without endangering soundness.

3 First- and second-order types

This section presents a formal definition of first-order types, as well as of their graph representation. The latter is often used—behind the scene—in efficient first-order unification algorithms. We then introduce graphical notations on the well-known System-F types, as they offer a good support for intuitions.

3.1 First-order terms

Let paths, ranged over by π, be sequences of integers. We let ϵ designate the empty path, and $\pi\pi'$ the concatenation of π' after π. We extend concatenation to sets of paths by $\Pi \cdot \Pi' = \{\pi \cdot \pi' \mid \pi \in \Pi, \pi' \in \Pi'\}$.

A (first-order) *term* t over a signature Σ (a set of symbols with arities) and a set of variables $\mathcal{V}$ is a mapping from a non-empty set of paths to $\Sigma \cup \mathcal{V}$ that is prefix-closed and respect arities. That is, for all paths π in $\mathsf{dom}(t)$ (the domain of t) and all integers k, $\pi k \in \mathsf{dom}(t)$ is equivalent to $1 \leq k \leq \mathsf{arity}(t(\pi))$.

First-order types are usually understood as trees. For example the tree (a) of Figure 1 represents the type $(\alpha \to \beta) \to (\alpha \to \beta)$. This type is itself the function that maps ϵ and the paths 1 and 2 to $\to$, paths 11 and 21 to α, and paths 21 and 22 to β.

The *projection* of a term t at a path π is the partial function t/π that maps t to its subterm rooted at π; projecting type (a) at path 1 or 2 yields the type $\alpha \to \beta$. It is defined by $\pi' \mapsto t(\pi\pi')$. Conversely, we may treat a symbol C of Σ as a function on trees that maps a tuple of terms $\overline{t}$ of arity $\mathsf{arity}(C)$ to the term t that maps ϵ to C and paths $k_i\pi$ to $t_i(\pi)$ for i in $1..\mathsf{arity}(C)$.

A *substitution* φ is a mapping from variables to terms; it is extended to a mapping from terms to terms in the usual way.

A term t' is an instance of a term t, which we write $t \leqslant t'$, if it is the image of t by some substitution φ. Two terms t and t' are unifiable if there exists a substitution φ, called a unifier of t and t', that identifies them. The

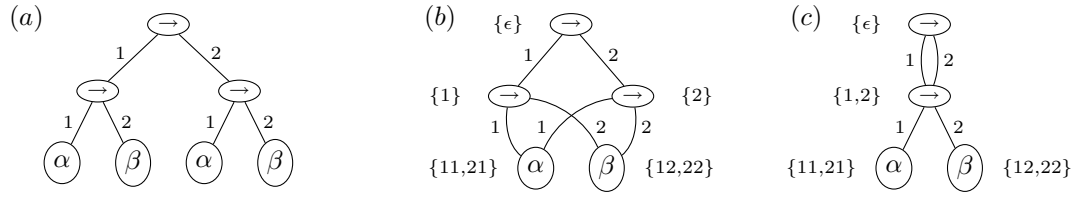


Figure 1: Several representations of $(\alpha \to \beta) \to (\alpha \to \beta)$.

unifier φ is said to be principal if any other unifier can be written as $\varphi' \circ \varphi$ for some substitution φ'. Similarly, t'' is a (principal) unifier of t and t' if it is of the form $\varphi(t)$ where φ is a (principal) unifier of t and t'.

Unification is a well-known problem on first-order terms that can be computed in linear time [11] using dags. Other algorithms use union-find structures and have $n\alpha(n)$ time complexity; however, they run faster in practice [5, 10] and are simpler to implement. Moreover, Huet's algorithm [5] can perform unification on regular terms as well. Interestingly, all three algorithms use a graph representation of types. In fact, they compute unification on graphs representations of terms, and reinterpret the resulting graphs as terms.

3.2 Term-graphs

When representing first-order types, it is sometimes convenient (and often more efficient) to identify all variables with the same name, as shown in the dag (b) of Figure 1. In fact, inner nodes with identical subtrees can also be shared, as illustrated on Figure 1 (c). This enables sharing of common suffixes, hence for a more compact, but also richer representation, where sharing of nodes asserts that the substrees are indeed equal. The use of a dag representation of terms for efficient first-order unification algorithms is standard. It may be explicit when algorithms are described imperatively, or left implicit as in Huet's algorithm [5]. Our presentation is also inspired by the notations used in graph rewriting [12].

Definition 1 (*Term-graphs)* An equivalence relation $\sim$ on the paths of a term t is a *congruence* if it is suffix-closed, *i.e.* $\pi \sim \pi'$ and πk and $\pi' k$ are in $\mathsf{dom}(t)$ implies $\pi k \sim \pi' k$. It is *weakly consistent* if the image by t of an equivalence class contains at most one symbol of Σ. It is *consistent* if the image by t of an equivalence class is a singleton.

A *term-graph* is a pair of a term t and a consistent congruence $\sim$ on $\mathsf{dom}(t)$ such that every variable appears in at most one equivalence class[2]. □

Intuitively, we may view an equivalence relation on t as "sharing" some "nodes" of t. Congruences enforces that when two paths are shared, (the paths at) their respective subterms remain shared. Consistent relations ensure that shared paths are mapped to the same symbol or variable. Weakly consistent relations do not impose a constraint on paths that are mapped to variables—their are used to reason about unification.

A consistent congruence on t partitions the paths of t into (disjoint) equivalence classes that represent the *nodes* of the term-graph. As t is constant on every node, we may extend t to nodes by mapping each node n to the common value of t on all paths of n. Consistently, we also write $\mathsf{dom}(t)$ for the equivalence classes of t, *i.e.* for the set of the nodes of t.

We use letter n to range over nodes. We also write r the root node $\{\epsilon\}$ of a graph. We use letter g to range over term-graphs and write $\hat{g}$ and $\tilde{g}$ for the term and the equivalence relation defining g.

For example, dags (b) and (c) on Figure 1 are two term-graphs representing the same term $(\alpha \to \beta) \to (\alpha \to \beta)$. In the dag (b), only variable nodes are shared. This term-graph has five nodes $\{\epsilon\}$, $\{1\}$, $\{2\}$, $\{11, 21\}$ and $\{12, 22\}$. Here, we have drawn node names; however, we usually leave them implicit.

Notice that the nodes $\{1\}$ and $\{2\}$ of (b) are congruent: for any path π, 1π and 2π are equivalent, and moreover, both nodes are labelled with the arrow symbol. Therefore, the equivalence relation of (b) could be enriched with $\{1, 2\}$ while remaining congruent and consistent, resulting in exactly the equivalence relation of dag (c). Intuitively, the subgraphs under $\{1\}$ and $\{2\}$, which were identical in (b), have been merged in (c). In this simple example, only the nodes $\{1\}$ and $\{2\}$ have been merged. The name of the merged nodes is the union of the names of the nodes to be merged.

[2]This last invariant is not strictly required, but there is no real advantage to use graphs if it is not enforced.

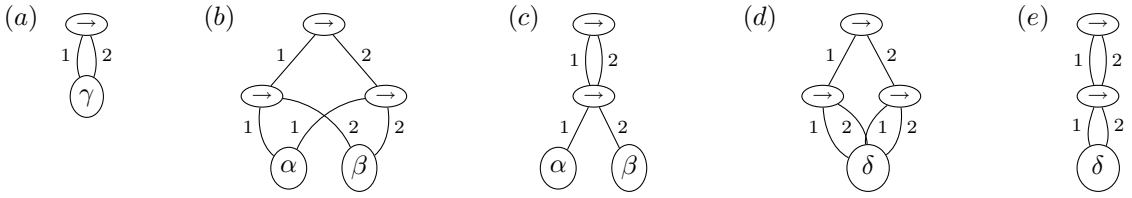


Figure 2: Term-graph instance.

Term-graphs as graphs A term-graph g may be read as an ordinary graph whose nodes are $\mathsf{dom}(g)$, labeled by the function g, and whose labeled edges $n \xrightarrow{k} n'$ are the set of triples such that there exists a path π in n with πk in n'. In this setting, we forget the underlying structure of nodes as sets of paths, and treat them as atoms. We use the term *standard graphs* to refer to this view; the two representations are isomorphic. The standard view is sometimes necessary for efficiency of algorithms, since otherwise maintaining names could be exponential in the size of the graph. However, the default, named view is more convenient for referring to nodes and to keep track of nodes during a sequence of graph transformations.

Designating nodes in graphs When a graph is known, we often use a single path π as a short-name for the unique node n to which π belongs, and write $\langle\pi\rangle$ for n. For example, in picture (c) of Figure 1, $\langle 12\rangle$ and $\langle 22\rangle$ refer to the same node $\{12, 22\}$. More generally, given two term-graphs t and t' such that $\tilde{t} \subseteq \tilde{t}'$ (*i.e.* t' shares more nodes than t), a node n of t can be translated unambiguously into a node n' of t', which is the only node of t' that is a superset of n. For brevity we often designate n' by n.

We usually leave arities implicit, as we always write outgoing edges downwards and from left to right.

3.3 Instance and unification on term-graphs

Unsurprinsingly, instance of term-graphs is two-fold: it is either an instance of the underlying first-order term $\hat{t}$, which changes the structure of the type, or an instance of the equivalence relation $\tilde{t}$, which merges more nodes.

Definition 2 A term-graph g' is an *instance* of a term-graph g, which we write $g \leqslant g'$, if $\hat{g} \leqslant \hat{g}'$ and $\tilde{g} \subseteq \tilde{g}'$. We say that g' is a *reversible instance* of g if moreover $\hat{g} = \hat{g}'$.

Two term-graphs are *equivalent* if they are instances of one another. Two term-graphs are *similar* if they are two reversible instances of a same term-graph. □

Two equivalent term-graphs are in fact equal up to the renaming of their variables. Reversible instance only changes the representation of a type, but not its meaning as a first-order type. It is thus "semantically" reversible, although it is usually not operationally allowed. Precisely, similarity abstracts over the notational details brought by term-graphs: two term-graphs are similar if they represent the same first-order type.

Let us give some examples. In figure 2, the term-graphs (c) and (e) are two instances of (a), through the substitutions $\gamma \mapsto \alpha \to \beta$ and $\gamma \mapsto \delta \to \delta$ respectively. The graph (c) is also a reversible instance of (b), as it shares more nodes than (b), and its underlying tree is equal to that of (b). Hence, (b) and (c) are similar—but not equal. Here, (c) is also an instance of (b). In general however, none of two similar types would be an instance of the other, as one could share could share more in one branch and less in another one. The term-graph (d) is also an instance of (b), through the substitution $\alpha, \beta \mapsto \delta$. Even though $\hat{c} \leqslant \hat{d}$ holds, (d) is *not* an instance of (c), as the nodes $\langle 1\rangle$ and $\langle 2\rangle$ are shared in (c) but not in (d). Similarly, (e) is an instance of (d); conversely, (d) is not an instance of (e). Finally, (e) is an instance of (b). Instance is indeed a transitive relation.

On term-graphs, unification can be *internalized*, that is, it may be defined on two nodes of a same term-graph instead of between two term-graphs. We say that a term-graph g' is a unifier of two nodes of a term-graph g if it is an instance of g that identifies both nodes (*i.e.* there exists a node n of g' that is a superset of both nodes). For example, the term-graph (c) is a unifier of the nodes $\langle 1\rangle$ and $\langle 2\rangle$ of the term-graph (b). A unifier g' of two nodes is *principal* if any other unifier of those nodes is also an instance of g'. Unification of two nodes of g can be computed as the smallest weakly consistent, congruent equivalence that contains $\tilde{g}$ and identifies both nodes [5].

Unification of term-graphs also computes their unification up to similarity, *i.e.* unification on terms. More precisely, if g' is a (principal) unifier of two nodes n_1 and n_2 in a term-graph g, then $\hat{g}'/n$ is a (principal) unifier

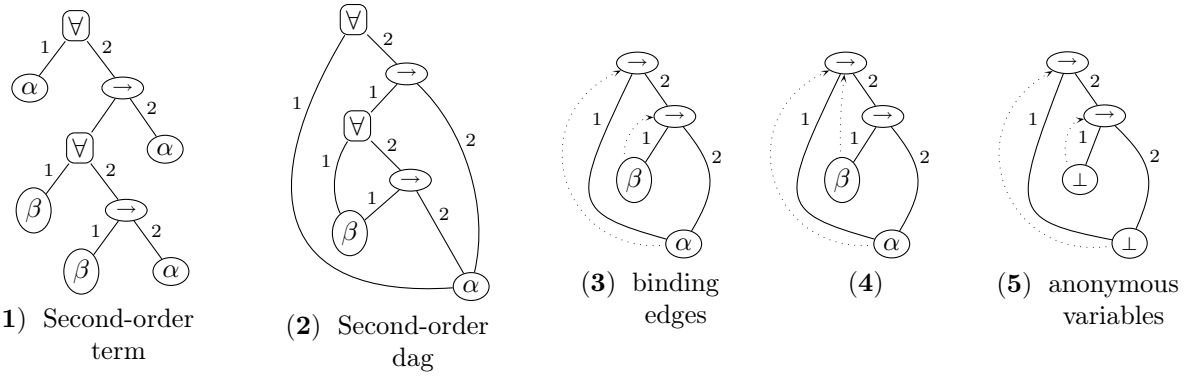


Figure 3: Representations of second-order types.

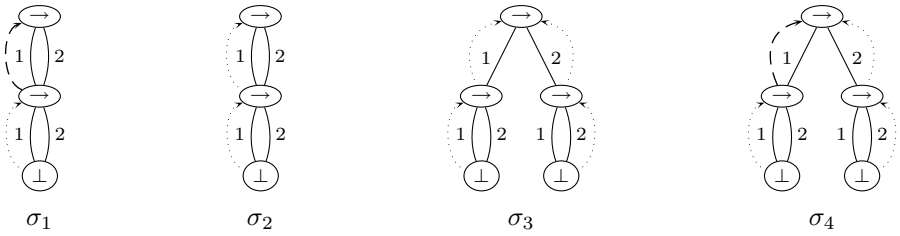


Figure 4: Examples of graphic ML^F types.

of $\hat{g}/n_1$ and $\hat{g}/n_2$ where n is the node of g' that is a superset of n_1 and n_2 (where the notation t/n stands for the (common) subterm of t at any path in n). This property, often overlooked in the literature, justifies the fact that term-graphs can be used instead of first-order terms to perform first-order unification.

3.4 Representing second-order types

Traditionally, binders are represented with an explicit node labeled with a special symbol $\forall$ of arity two. For example, the System-F type $\forall\alpha.(\forall\beta.\beta \to \alpha) \to \alpha$ is usually represented as the tree (1) of Figure 3. Using dags, we would rather represent it as in (2). However, representing quantifiers as special nodes inserted in the structure is counter-intuitive, as it hides the underlying common structure of all instances.

We may in fact remove the quantifier node and instead introduce a *binding edge* between the bound variable and the node just above which it is bound, as illustrated in graph (3). We orient the binding edge from the bound variable to its binding node. This is just a convention, and we could have chosen the opposite direction—our choice is slighly easier to think about, as each variable node is bound to a single node, but a single node could be a binding position for several variables.

Notice that this notation looses the order of adjacent binders and makes useless binders not representable—two artifacts of the syntactic notations that we are so happy to eliminate. For instance, $\forall\alpha.\forall\beta.(\beta \to \alpha) \to \beta$, $\forall\beta.\forall\alpha.(\beta \to \alpha) \to \beta$ and $\forall\gamma.\forall\alpha.\forall\beta.(\beta \to \alpha) \to \beta$ will all have the same representation (4).

Finally, as quantified variables are treated modulo α-renaming, we may advantageously draw them anonymously, as in (5). For that purpose, we introduce a new kind of node $\bot$, called a *bottom node* to mean "a variable". The bottom sign $\bot$ is not a true symbol (it is not an element of Σ) but a new pseudo-symbol that does not clash with other symbols during unification. We intendedly reuse the same notation as the bottom type of syntactic ML^F types, since the notation "$\forall\,(\alpha \diamond \bot)$" plays the same role as "$\forall\alpha.$" in System-F types.

4 MLF graphic types

4.1 Representing MLF types

Let us illustrate the graphic representation of MLF types on the four types σ_1, σ_2, σ_3, and σ_4 introduced earlier (§2.1) and drawn in Figure 4. As for System F, we draw binding edges from nodes to their binding node, but use two kinds of edges to distinguish between flexible and rigid bindings, represented by dotted and dashed lines respectively[3]. In addition, we represent the bound of the variable in place of the unique node representing that variable (hence, non-bottom nodes may now also have binding edges). For instance, the graph representing σ_3 contains at the node $\langle 1\rangle$ a subgraph representing the bound σ_{id} of the variable α. This node is itself bound at the root. For bottom bounds, we thus recover the representation of variables for System-F types. For example, node $\langle 11\rangle$ of σ_3 is a bottom node.

Thus defined, the binding edges of graphic types are upwards-closed. From a theoretical standpoint, some nodes (such as the type constructor int) need not be bound. However, this makes reasoning on the graphic types uneasy, so we require that all nodes be bound.

As in term-graphs, sharing of non-variable nodes is possible. However, it is most of the case semantically significant (unlike in term-graphs). In MLF, σ_4, in which the two occurrences of σ_{id} may be instantiated separately, is quite different from σ_3, in which both sides of the arrow must be instantiated simultaneously. This is partly reflected in the graphic presentation by the fact that there are copies of the graph representing σ_{id} in σ_4, but only one in σ_3. We are now able to characterize graphic types, *i.e.* graphs representing MLF types. We first define pre-types (§4.1), and then state well-formedness conditions they must satisfy in order to be types (§4.2).

Definition 3 A *pre-type* τ is a pair of:

1. A term-graph $\mathring{\tau}$, whose nodes are labelled by elements of $\Sigma \cup \{\bot\}$. The bottom nodes must be leaves and the other nodes must respect the arity of their symbol.
2. A binding tree $\overset{\succ}{\tau}$ for $\mathring{\tau}$. That is, a set of binding edges labelled with flags that form an upside-down tree rooted at $\langle\epsilon\rangle$, such that any node different from the root is in the tree. □

Notations In the text, we write $n \circ\!\!\longrightarrow n' \in \tau$ (*resp.* $n \overset{\diamond}{\succ\!\!\longrightarrow} n' \in \tau$) to mean that there is a structure edge (*resp.* a binding edge with flag $\diamond$) from n to n' in τ. We may drop the flag when it is unimportant. If $n \overset{\diamond}{\succ\!\!\longrightarrow} n' \in \tau$, we also write $\overset{\diamond}{\tau}(n)$ for $\diamond$ and $\overset{\succ}{\tau}(n)$ (or simply $\overset{\succ}{n}$) for n'; we call n' the *binder* of n and we say that n is bound at n'.

If π is a path, we write $n \overset{\pi}{\circ\!\!\longrightarrow} n'$ to denote the fact that n' is at path π from n. We write $\longleftarrow\!\!\prec$ the symmetric of the relation $\succ\!\!\longrightarrow$ and $\circ\!\!\prec$ the union $(\circ\!\!\longrightarrow) \cup (\longleftarrow\!\!\prec)$. Given some nodes $n_1, ..., n_k$, we say that the sequence $n_1 \circ\!\!\prec n_2 \ldots \circ\!\!\prec n_{k-1} \circ\!\!\prec n_k$ is a *mixed path* between n_1 and n_k; this path is said to *contain* n if $n = n_i$ for some $1 \leq i \leq k$. We write $\hat{\tau}$ and $\tilde{\tau}$ for the term and equivalence defining $\mathring{\tau}$. We simply write $\tau(n)$ for $\hat{\tau}(n)$, *i.e.* the symbol on the node n. We write $\overset{\circ}{\tau}_{foo}\!\!\longrightarrow$ and $\overset{\succ}{\tau}_{bar}\!\!\longrightarrow$ instead of $\mathring{\tau}_{foo}$ and $\overset{\succ}{\tau}_{bar}$ for wide arguments.

For example, consider the pre-types of Figure 5. We have $\langle 1\rangle \overset{=}{\succ\!\!\longrightarrow} \{\epsilon\} \in \tau_1$. Hence, the binder of $\langle 1\rangle$ is $\overset{\succ}{\tau}_1\langle 1\rangle = \{\epsilon\}$, and $\overset{\diamond}{\tau}_1(\langle 1\rangle)$ is $=$. We also have $\langle 1\rangle \circ\!\!\longrightarrow \langle 11\rangle \circ\!\!\longrightarrow \langle 112\rangle \in \tau_2$ or, leaving τ_2 implicit, $\langle 1\rangle \overset{12}{\circ\!\!\longrightarrow} \langle 112\rangle$. Following the first or second outgoing edge from $\langle 1\rangle$ leads to the same node $\langle 11\rangle$ (also named $\langle 12\rangle$). Finally, we also have $\{\epsilon\} \longleftarrow\!\!\prec \langle 11\rangle \overset{2}{\circ\!\!\longrightarrow} \langle 112\rangle$, which is a mixed path in τ_2 from $\{\epsilon\}$ to $\langle 112\rangle$.

4.2 Well-formedness

All pre-types are not types as the later must correspond to syntactic types. In particular, the binding tree must be compatible with lexical scoping of variables. Two examples of ill-formed binding trees are described in Figure 5:

- In pre-type τ_1, the node $\langle 21\rangle$ is bound at a node that is not among its parents. This is not permitted, as in a syntactic presentation, the variable should be bound on the left branch and used on the right branch, out of its scope.

[3] In generic diagrams where an edge can be indifferently flexible or rigid, we use dashed-dotted edges.

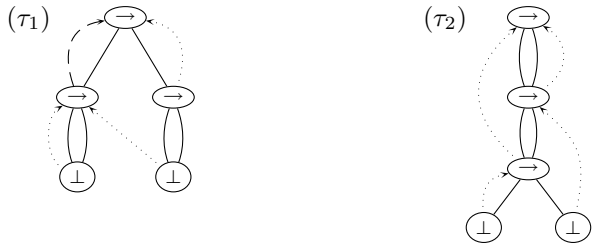


Figure 5: Invalid graphic types.

- In pre-type τ_2, both nodes $\langle 1\rangle$ and $\langle 11\rangle$ are bound at the root. However, the two bindings depend on one another: when attempting the translation to a syntactic type, if we try to bind $\langle 11\rangle$ first, we must refer to $\langle 112\rangle$ which is bound under $\langle 1\rangle$; conversely, if we choose to bind $\langle 1\rangle$ first, we must use $\langle 11\rangle$ in the bound of $\langle 1\rangle$ while this node is not bound yet.

Both invariants can be captured using the notion of *domination*.

Definition 4 (*Domination*) Let τ be a pre-type and n and n' two nodes of τ. We say that n dominates n' and we write $n \circ\!\!\rightarrow\!\!\prec n'$ if every mixed path from the root to n' contains n. □

It is well-known that domination is a partial order.

Consider again the pre-types of Figure 5. In τ_1, the mixed paths between $\{\epsilon\}$ and $\langle 11\rangle$ are

$$\{\epsilon\} \circ\!\!\longrightarrow \langle 1\rangle \circ\!\!\xrightarrow{1} \langle 11\rangle \qquad \{\epsilon\} \longleftarrow\!\!\prec \langle 1\rangle \circ\!\!\xrightarrow{1} \langle 11\rangle$$
$$\{\epsilon\} \circ\!\!\longrightarrow \langle 1\rangle \circ\!\!\xrightarrow{2} \langle 11\rangle \qquad \{\epsilon\} \longleftarrow\!\!\prec \langle 1\rangle \circ\!\!\xrightarrow{2} \langle 11\rangle$$

All four paths contain $\langle 1\rangle$. Hence node $\langle 1\rangle$ dominates node $\langle 11\rangle$.

Conversely, node $\langle 1\rangle$ does not dominate $\langle 21\rangle$, as evidenced by the path $\{\epsilon\} \circ\!\!\xrightarrow{2} \langle 2\rangle \circ\!\!\xrightarrow{1} \langle 21\rangle$. Similarly, in τ_2, $\langle 1\rangle$ does not dominate $\langle 112\rangle$, since $\{\epsilon\} \longleftarrow\!\!\prec \langle 11\rangle \circ\!\!\xrightarrow{2} \langle 112\rangle$.

Well-formed types are simply types in which the binder of a node dominates the node itself. This result is just a generalization of the known case for types with explicit nodes encoding binders (type (2) of Figure 3), where $\forall$-nodes must dominate the variables they introduce. In our more general case, we must also take the binding tree into account.

Definition 5 (*Types*) The binding tree of a pre-type τ is *well-dominated* if every bound node is dominated[4] by its binder, *i.e.*, for all n and n' in τ, $n \succ\!\!\longrightarrow n'$ implies $n' \circ\!\!\rightarrow\!\!\prec n$, or, in short, $(\longleftarrow\!\!\prec) \subseteq (\circ\!\!\rightarrow\!\!\prec)$. A *(graphic) type* is a well-dominated pre-type. □

As seen in examples, neither τ_1 nor τ_2 are types, as they are not well-dominated. In particular, $\overset{\succ}{\tau_1}(\langle 21\rangle)$ does not dominate $\langle 21\rangle$ in τ_1 and $\overset{\succ}{\tau_2}(\langle 112\rangle)$ does not dominate $\langle 112\rangle$ in τ_2.

4.3 Invariants induced by well-formedness

Well-domination is a fairly strong property: it imposes several invariants on the relation between the structure and the binding tree of a type. We characterize some of them below.

We first introduce a notion of "*bubble*" that contains all the nodes that are "under" a given node.

Definition 6 (*Bubbles*) The *bubble* of a node n of a type τ, written $\mathcal{B}_\tau(n)$, is the set $\{n' \in \mathsf{dom}(\tau) \mid \overset{\succ}{n} \circ\!\!\xrightarrow{+} n' \circ\!\!\xrightarrow{*} n\}$ of nodes above n and strictly under the binder of n. □

We may omit τ in $\mathcal{B}_\tau(n)$ when it is clear from context. Notice that the bubble of n contains n, but not $\overset{\succ}{n}$. We call $\overset{\succ}{n}$ the *top* of the bubble. In Figure 6, we have drawn a type τ (on the left) and a schema of τ (on the right) in which the bubbles (and the edges connecting the nodes) of the three nodes $\langle 1\rangle$, $\langle 1111\rangle$ and $\langle 1122\rangle$ are colored. For example, $\mathcal{B}(1122) = \{\langle 11\rangle, \langle 112\rangle, \langle 1122\rangle\}$ is drawn in red.

Well-domination ensures the following properties.

[4]Since $\overset{\succ}{\tau}$ is a tree, it is acyclic and $\overset{\succ}{n}$ cannot be n. Hence we couldd also require every bound node to be *strictly* dominated by its binder.

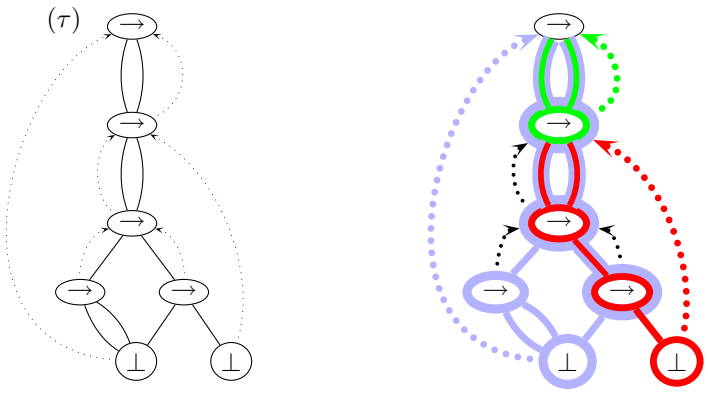


Figure 6: Illustration of bubbles

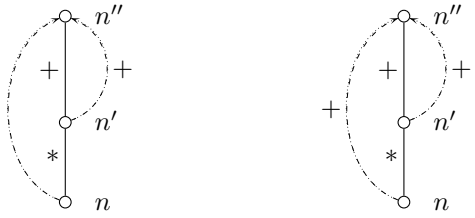


Figure 7: Diagrams for bubbles nesting

Property 1 *For any type τ and for all nodes n, n', and n'' in τ,*

1. *if $n \rightarrowtail n'$ then $n' \circ\!\!\xrightarrow{\pm} n$*
   (binding edges follow inverse structure edges);
2. *if $n' \in \mathcal{B}(n)$, then $\breve{n} \circ\!\!\triangleright\!\!\prec n'$*
   (the top of the bubble dominates all nodes of the bubble);
3. *if $n' \in \mathcal{B}(n)$ then $n' \succ\!\!\xrightarrow{\pm} \breve{n}$*
   (all bound nodes of a bubble are transitively bound at the top of the bubble);
4. *if $n \succ\!\!\xrightarrow{\pm} n'' \circ\!\!\xrightarrow{\pm} n' \circ\!\!\xrightarrow{*} n$, then $n' \succ\!\!\xrightarrow{\pm} n''$;*
5. *if $\mathcal{B}(n) \cap \mathcal{B}(n') \neq \emptyset$ then either $n \succ\!\!\xrightarrow{\pm} \breve{n}'$ or $n' \succ\!\!\xrightarrow{\pm} \breve{n}$.* □

Properties 3 and 4 are presented as diagrams in Figure 7, which shows that Property 4 is actually a subcase of 3. The conclusion is the rightmost binding edge. Property 3 can be checked (in a simple case) on node $\langle 1 \rangle$ of type τ of Figure 6. This node is in the bubble of node $\langle 1111 \rangle$, and is (directly) bound at the binder of $\langle 1111 \rangle$. Property 5 implies in particular that bubbles having a common node do not separate above that node. An example is nodes $\langle 1111 \rangle$ and $\langle 1122 \rangle$ of τ of Figure 6, whose bubbles intersect on $\langle 11 \rangle$ or $\langle 112 \rangle$.

Bubbles can also be used to order nodes along a "dependency" relation. Indeed, given two nodes n and n', if $n \in \mathcal{B}(n')$, we know that the subgraph under n contains n'. Thus n "uses" n' in its bound when the graphic type is written as a syntactic one.

Definition 7 (*Order on bound nodes*) We write $<_{\mathcal{B}}$ the partial order on bound nodes defined by $n <_{\mathcal{B}} n'$ if and only if $\breve{n} = \breve{n}' \wedge \mathcal{B}(n') \subset \mathcal{B}(n)$. □

In type τ of Figure 6, the relation $\langle 1111 \rangle <_{\mathcal{B}} \langle 1 \rangle$ holds. Nodes minimal for $<_{\mathcal{B}}$ are always lower in the type (the converse implication being false).

4.4 Operators for building graphs

In this section, we introduce several operations to build or transform graphs. The semantics of those transformations in terms of graphic types will be considered in the next section.

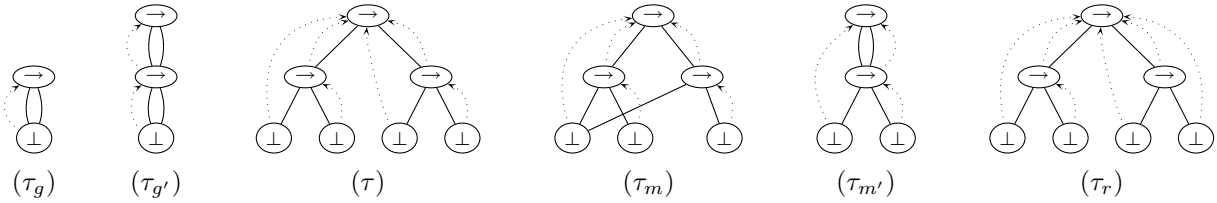


Figure 8: Operations on graphs

Grafting We write $\tau[\tau'/n]$ for the grafting of a type τ' at a bottom node n of a type τ; the resulting type is described by τ' for nodes below n and by τ for other nodes. In Figure 8, grafting type τ_g at node $\langle 1\rangle$ in itself yields type $\tau_{g'}$. Formally:

- $\widehat{\tau[\tau'/n]}$ maps nm to $\tau'(m)$ for $m \in \mathsf{dom}(\tau')$, and maps m in $\mathsf{dom}(\tau) \setminus \{n\}$ to $\tau(m)$;
- $\widetilde{\tau[\tau'/n]}$ is equal to $\tilde{\tau} \cup n \cdot \tilde{\tau}'$ where $n \cdot \tilde{\tau}'$ means the set of pairs $(n \cdot m, n \cdot m')$ such that $m\ \tilde{\tau}'\ m$;
- $\overset{\succ}{\longrightarrow}_{\tau[\tau'/n]}$ is $\overset{\succ}{\tau}$ extended with all edges $nm \overset{\diamond}{\succ\!\!\longrightarrow} nm'$ such that $m \overset{\diamond}{\succ\!\!\longrightarrow} m' \in \tau'$.

Projection .

Let us call *closed* a node n such that all nodes of the subgraph under n are transitively bound under n itself (that is $n \overset{\pm}{\circ\!\!\longrightarrow} n'$ implies $n' \overset{\pm}{\succ\!\!\longrightarrow} n$). Given such a node, we write τ/n for the projection of τ at n, obtained by removing all nodes not under n and all dangling edges, and renaming nodes accordingly (thus making n the root node of the resulting graph). For example, projecting at node $\langle 1\rangle$ in τ'_g yields type τ_g. Projecting at nodes $\langle 1\rangle$ or $\langle 2\rangle$ in τ is impossible, as $\langle 11\rangle$ is not bound under $\langle 1\rangle$ (and the resulting graph would be ill-bound). Formally:

- $\widehat{\tau/n}$ is $\hat{\tau}/n$.
- $\widetilde{t/n}$ is such that $\pi\ \widetilde{\tau/n}\ \pi'$ if and only if $n\pi\ \tilde{\tau}\ n\pi'$.
- $\overset{\succ}{\longrightarrow}_{t/n}$ is defined by $m \overset{\diamond}{\succ\!\!\longrightarrow} m' \in \tau/n$ if and only if $nm \overset{\diamond}{\succ\!\!\longrightarrow} nm' \in \tau$.

Fusion Provided the subgraphs under n_1 and n_2 are structurally equal and their binding trees can be fused, we write $\tau[n_1 = n_2]$ the type obtained by fusing those two subgraphs in τ. For example, fusing the nodes $\langle 11\rangle$ and $\langle 21\rangle$ in τ yields type τ_m. More interestingly, the nodes $\langle 1\rangle$ and $\langle 2\rangle$ can be fused in both τ and τ_m, resulting in type $\tau_{m'}$. Notice that the binding edges $\langle 11\rangle \succ\!\!\longrightarrow \{\epsilon\}$ and $\langle 21\rangle \succ\!\!\longrightarrow \{\epsilon\}$ of τ are fused in $\tau_{m'}$, as a side-effect of fusing $\langle 1\rangle$ and $\langle 2\rangle$.

We say that an equivalence relation on nodes is *consistent* if it is compatible with the binding structure. (That is, whenever $n_1 \sim n_2$, then n_1 and n_2 are both bound, to nodes n'_1 and n'_2 such that $n_1 \sim n'_2$.) The fusion $\tau[n_1 = n_2]$ is defined on all the graphs τ such that τ/n_1 and τ/n_2 are equal and the smallest congruence $\sim$ which merges n_1 and n_2 in τ is consistent with $\overset{\succ}{\tau}$. It is defined by:

- $\widehat{\tau[n_1 = n_2]}$ is $\hat{\tau}$
- $\widetilde{\tau[n_1 = n_2]}$ is $\tilde{\tau} \cup \{\, (n_1\pi, n_2\pi) \,|\, n_1\pi \in \mathsf{dom}(\hat{\tau}) \,\}$.
- $\overset{\succ}{\longrightarrow}_{\tau[n_1 = n_2]}$ is the quotient of $\overset{\succ}{\tau}$ by $\widetilde{\tau[n_1 = n_2]}$, which is well-defined by construction.

Raising Consider a node n of a type τ such that $n \overset{\diamond}{\succ\!\!\longrightarrow} n' \overset{\diamond'}{\succ\!\!\longrightarrow} n''$ holds. A simple transformation on the binding tree of τ is to "lift" the binding edge $n \succ\!\!\longrightarrow n'$ above the edge $n' \succ\!\!\longrightarrow n''$, resulting in the edge $n \overset{\diamond}{\succ\!\!\longrightarrow} n''$. The resulting pre-type is called the *raising* of n in τ, written $\tau \uparrow n$. In Figure 8, the type τ_r is the raising of the node $\langle 22\rangle$ in τ. Formally:

- $\widehat{\tau \uparrow n}$ is $\hat{\tau}$;

- $\widetilde{\tau \uparrow n}$ is $\tilde{\tau}$;
- $\overset{\succ}{\tau \uparrow n}$ is $\overset{\succ}{\tau}$, except on n where it is $n \overset{\diamond}{\succ\!\!\longrightarrow} n''$ (where $\diamond = \overset{\diamond}{\tau}(n)$, and $n'' = \overset{\succ}{\tau}(\overset{\succ}{\tau}(n))$).

Well-formedness of operations Grafting, fusion and projection are "well-behaved" operations that transform a type into a (well-dominated) type, but this is not the case for raising.

Property 2 *Let τ be a type.*

- *Let τ' be a type and n a bottom node of τ; the grafting $\tau[\tau'/n]$ is a type.*
- *Let n_1 and n_2 be two nodes that can be fused in τ. The fusion $\tau[n_1 = n_2]$ is a type.*
- *Let n be a closed node of τ. The projection τ/n is a type.* □

Raising at arbitrary nodes can result in ill-dominated pre-types. Consider the raising of $\langle 11\rangle$ in type τ of Figure 6, and let us call g the result. The mixed path $\{\epsilon\} \longleftarrow\!\!\prec \langle 11\rangle \circ\!\!\longrightarrow \langle 112\rangle \circ\!\!\longrightarrow \langle 1122\rangle$ of g does not contain the node $\langle 1\rangle = \overset{\succ}{g}(\langle 1122\rangle)$. Thus g is not well-dominated.

We can however characterize the set of nodes n which can raised while preserving well-domination. Intuitively, the bubble of n must not be contained in the bubble of a node n' bound at the same node as n.

Definition 8 (*Raisable node)* Given a type τ and a bound node $n \in \tau$, n is *raisable* in τ if it is minimal for $<_{\mathcal{B}}$ in τ. That is $\forall n' \in \tau$, $n' \not<_{\mathcal{B}} n$. □

Given the definition of $<_{\mathcal{B}}$, the condition can be relaxed into $\forall n' \in (\succ\!\!\longrightarrow \overset{\succ}{n}), n' \not<_{\mathcal{B}} n$, which may be checked locally. In Figure 6, $\langle 1122\rangle$ is minimal for $<_{\mathcal{B}}$, so it is raisable. Conversely, $\langle 11\rangle$ is not, as $\mathcal{B}_\tau(\langle 11\rangle) \subset \mathcal{B}_\tau(\langle 1122\rangle)$.

Lemma 1 *$\tau \uparrow n$ is a type (*i.e. *is well-dominated) iff n is raisable in τ.* □

5 The instance relation on graphic types

This section presents the instance relation on graphic types. The relation can be decomposed into local atomic transformations on types, each of them transforming either the underlying term-graph of the graphic type, or its binding tree, as seen in the previous section:

Grafting replaces a bottom node (*i.e.* a variable) by a type, as in first-order terms;

Merging fuses variables or inner nodes, as in the dag representation of first-order types;

Raising performs a scope extrusion. In spirit, transforming the System-F type $\tau' \to (\forall\alpha.\tau)$ into $\forall\alpha.(\tau' \to \tau)$ (whenever α does not appear free in τ') is a raising.

Weakening changes a flexible binder into a rigid one.

However, this description is not complete. Firstly, we must adapt the transformations on first-order terms or dags to the richer structure of binders of ML^F. Secondly, some transformations going in the inverse direction of the transformations presented above are possible on first-order dags; for example the types d and e of Figure 2 are equivalent, hence in instance relation. We must decide whether those transformations are possible for the ML^F instance relation. Thirdly, all transformations should not be allowed at every node. Indeed, the idea behind rigid bounds is to freeze polymorphism, hence preventing some transformations.

In the next section we examine some instances that should—and should not—hold on graphic types. The formal definition of the instance relation will be given afterwards.

5.1 Shaping the instance relation

The instance relation $\sqsubseteq$ of ML^F is *implicit*: if two types τ and τ' are such that $\tau \sqsubseteq \tau'$, any expression with type τ can be used in a context where an expression of type τ' is required. Thus $\sqsubseteq$ must be shaped so that:

1. It is sound. Naturally, only instances ensuring type soundness should be allowed.
2. It is as large as possible. The larger the relation, the bigger the set of typable programs.
3. It allows type inference. If $\sqsubseteq$ is too large, type inference will likely be undecidable, as in System F [14].

Of course, the last two points are incompatible, and account for a large part of the design space. The last point must be studied together with type inference, which we leave for future work. Thus, in this section we concentrate on the first two points. (However, the type instance relation we propose *do* allow type inference.)

In the next examples, we identify some instances we find desirable, and others that must be flagged as unsound. In order to do this, we present of few terms, together with some types they can be given (Figure 9). Of course, we have a "chicken-and-egg" problem: the set of types a term can be given depend on the instance relation! However, it is our belief that the intuitive semantics of types we have given so far should provide intuitions on why the proposed types are correct.

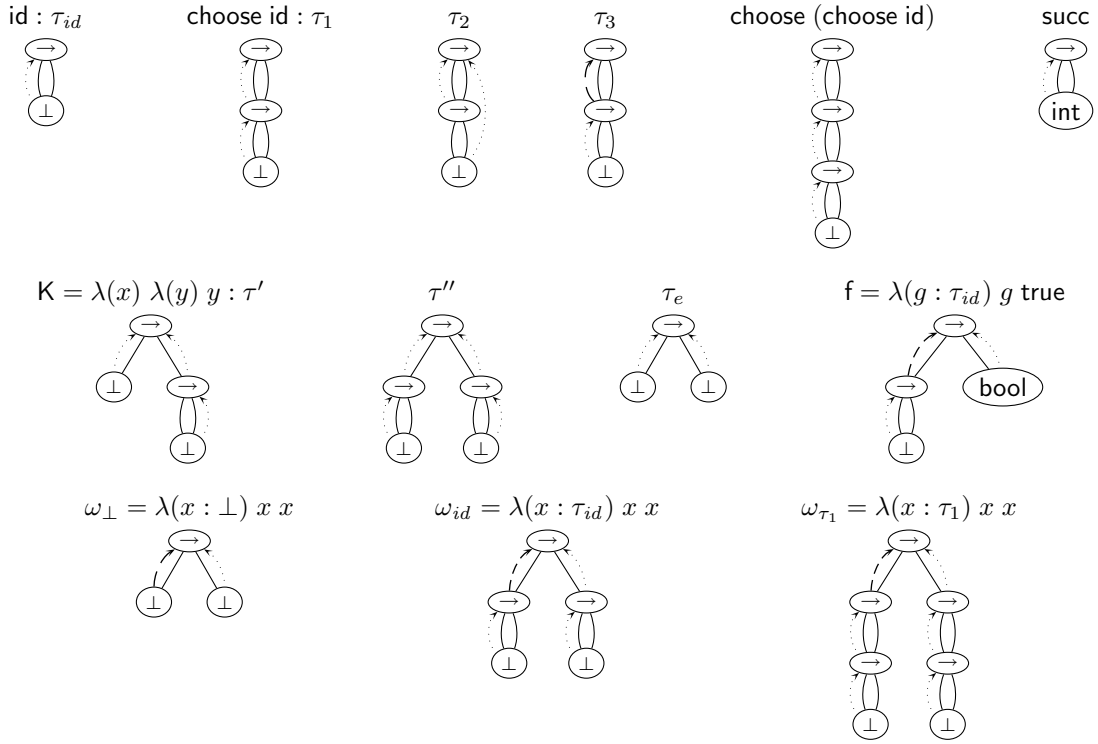


Figure 9: Examples of terms

Binding paths It is a key design point of ML^F not to take into account the variances of constructors in order to determine what operations are allowed on a given node. Moreover, the arrow constructor $\rightarrow$ has no special meaning, and is treated as all other type constructors. Instead, the transformations allowed are determined by the binding tree alone. More precisely, the transformations allowed on a node n are determined almost entirely by its *binding path*, which is defined as the sequence of flags on the binding edges from the root to n (thus followed in the reverse direction). For example, in Figure 4, the binding path of nodes $\langle 11 \rangle$ and $\langle 2 \rangle$ of type σ_4 are $(=\geqslant)$ and $(\geqslant)$ respectively. We write $\overline{\diamond}_\tau(n)$ the binding path of n in τ.

Nodes at flexible bindings paths Consider first the type τ_1 of $\mathsf{choose\ id}$. The types τ_2 and τ_3 correspond to the F-types of $\mathsf{choose\ id}$, *i.e.* $\forall(\alpha)\ (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ for τ_2 and $(\forall(\alpha)\ \alpha \rightarrow \alpha) \rightarrow (\forall(\alpha)\ \alpha \rightarrow \alpha)$ for τ_3. Thus we want $\tau_1 \sqsubseteq \tau_2$ and $\tau_1 \sqsubseteq \tau_3$ to hold. Notice that $\tau_1 \sqsubseteq \tau_2$ is a raising, while $\tau_1 \sqsubseteq \tau_3$ is a weakening.

Thus weakening a node n such that $\overline{\diamond}_n = (\geqslant)$ must be allowed. Similarly, raising a node n verifying $\overline{\diamond}_n = (\geqslant\geqslant)$ (provided that n is raisable) must be possible.

If we now consider the type of choose (choose id), raising and weakening must likewise be allowed at every node, in order to capture all possible derivations. More generally, this examples shows that adding a flexible binding at the beginning of a binding path should not change the set of allowed transformations. A node n such that $\overline{\diamond}_n = (\geqslant^+)$ is said to have *flexible permissions*.

Let us look next at grafting. Of course, we want the type of id to instantiate into the type of the successor function (as it does in ML). So grafting at a node with flexible permissions must be allowed. Notice that we can also instantiate the variable of τ_{id} by a polymorphic type (such as τ_{id} itself), as polymorphism is first-class: $\tau_{id} \sqsubseteq \tau_1$ holds.

Consider next the second projection K, whose most general type is τ' (which is slightly more general than the ML type $\forall(\alpha)\,\forall(\beta)\,\alpha \to \beta \to \beta$). We have already seen that $\tau' \sqsubseteq \tau''$ must hold. But we also want $\tau'' \sqsubseteq \tau_1$ to hold: we just forget that the two instances of τ_{id} are distinct (this allows for example to combine K and id in a list). Thus merging two nodes with flexible permissions bound at the same node must be allowed.

Finally, let us consider the inverse transformations. Those already forbidden on first-order dags remain forbidden in MLF: "ungrafting" the type of succ into τ_{id} would allow applying succ to (for example) a boolean, an operation which is unsound. Similarly, "unmerging" a variable, is unsound. It would allow transforming τ_{id} into τ_e, and would make the term id true true well-typed. But, unlike in ML, unmerging inner (polymorphic) nodes is also unsound. If we unshare $\langle 1\rangle$ in τ_1 (which is a valid type for id), we obtain τ''. This erroneously makes the term id succ true well-typed. (Note however that unsharing $\langle 1\rangle$ in τ_2 would be sound, as it would not unshare the variable at node $\langle 22\rangle$.) Finally, making a rigid edge flexible is also unsound in general. If we consider the term f (which applies an identity-like function to a boolean), making node $\langle 1\rangle$ flexible would make the unsound term f succ well-typed.

To summarize, nodes with flexible permissions allow four forms of instance. However, those transformations are semantically irreversible: the inverse transformation is in general unsound.

Nodes at flexible-rigid binding paths We next consider nodes which are rigidly bound. In fact, as we have seen above, we can consider all binding paths of the form $(\geqslant^*=)$. We say that nodes having such a binding path have *rigid permissions*.

Of course, such nodes cannot be weakened: they are already rigidly bound. We have also seen above that making their binding flag flexible would be unsound in general. Can we graft at them? Notice that the type of $\omega_\perp$ corresponds to the F type $(\forall(\alpha)\,\alpha) \to (\forall(\beta)\,\beta)$, and the type system should (hopefully) prevent us from creating a value of type $\forall(\alpha)\,\alpha$. Nevertheless, if we allow grafting at variables with rigid permissions, we can give as argument to $\omega_\perp$ an integer, resulting for example in the application of 1 to itself.

It is harder to give intuitions regarding merging and raising of nodes with rigid permissions, as those two operations are intrinsically linked to decidability of type inference. Both operations are sound, and so are the corresponding inverse operations. In fact, in the implicit version of MLF (where type inference is impossible), types can be considered up to raising/merging of such nodes. This is done on a significant fragment of MLF, where types are be given a semantic in terms of set of F types [8].

As a consequence (or, more precisely, as a design choice), we only allow merging and raising of nodes with rigid permissions in the instance relation. The inverse operations are still available, but only *explicitly*, through the use of type annotations.

Nodes at flexible-rigid-flexible binding paths The last step of binding paths we consider are those of the form $(\geqslant^* = \geqslant^+)$. As we will see shortly, operations on those nodes are in general unsound, so we say they have *locked* permissions.

Consider the type of ω_{id}, and suppose we allow grafting at nodes with locked permissions. Then by grafting int at node $\langle 11\rangle$, we would make the application ω_{id} succ (and thus succ succ) well-typed, which is clearly incorrect. Similarly, if we allow raising this node, it acquires flexible permissions and can be instantiated to int, yielding the same contradiction. Thus both grafting and raising must be forbidden at locked nodes.

As a more intricate example, suppose we allow weakening at locked nodes. If we weaken $\langle 11\rangle$ in the type of ω_{τ_1}, we can give it as argument ω_{id}. Indeed, as we have seen in the two previous paragraphs, the type of ω_{id} can be instantiated to τ_3, which is the type of the argument of ω_{τ_1} after the weakening. Thus (in an untyped view) the application $\omega\,\omega$ would be well-typed.

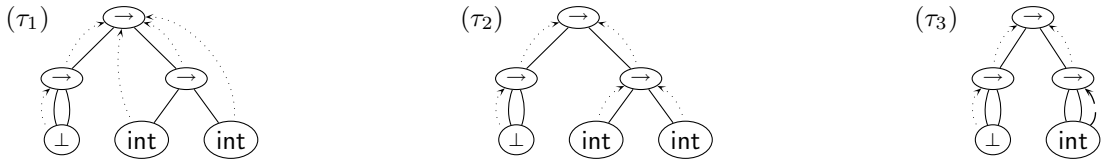


Figure 10: Graphic types with monomorphic nodes

Perm	Name	Allows	Binding Path
F	Flexible	Instance	$\geqslant^*$
R	Rigid	Abstraction	$(\geqslant\|=)^* =$
L	Locked	Nothing	$(\geqslant\|=)^* = \geqslant^+$

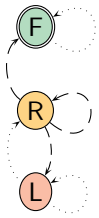


Figure 11: Permissions for the instance relation.

Inert nodes Some nodes hold in fact no instantiable polymorphism, either because there is no type variable under them, or because those variables are *protected* lower in the type by a rigid edge.

Definition 9 We suppose that Σ is partitioned into two sets of *monomorphic* and *polymorphic* type constructors. The symbol $\bot$ is considered polymorphic.

A node n of a type τ is said to be *inert* if $\tau(n)$ is not a monomorphic constructor, and for any node n' labelled by a polymorphic constructor such that $n' \overset{\overline{\diamond}}{\rightarrowtail} n$, there is a rigid flag in the sequence $\overline{\diamond}$. *Monomorphic* nodes are the subset of inert nodes on which no polymorphically labelled node is bound. □

Polymorphic type constructors (other than $\bot$) can for example be used to model polymorphic type abbreviations; otherwise, in this paper only $\bot$ will be polymorphic. Monormophic nodes correspond in essence to the inner nodes of term-graphs.

In all three types of Figure 10, the nodes $\langle 2\rangle$, $\langle 21\rangle$ and $\langle 22\rangle$ are monomorphic. Intuitively, all three graphic types represent the type τ given above, with different—yet unimportant—binding edges and sharing for monomorphic nodes.

More generally, inert nodes are an exception to the system of permissions above. Indeed, since they hold no instantiable polymorphism, it is always sound to transform them. Indeed, all the examples of unsoundness above crucially rely on the fact that a variable can be grafted, or that two variables with flexible bindings can be merged or split (as, in some context, it will be possible to instantiate those variables). As a consequence, inert nodes can be freely raised, merged, weakened, or transformed in the inverse direction of those operations. (Grafting is not possible, as inert nodes cannot be variables.)

5.2 Permissions

Of course, we have not treated all possible cases in the previous paragraphs. As binding paths increase in complexity, examples become much more involved (and more difficult to find!). In this section we attach permissions to the remaining cases. Hopefully the intuitions given by the examples above will carry over.

In the syntactic presentation of ML$^\mathsf{F}$, finding which transformations are allowed at a given position in a type is not readily apparent, and is determined by contextual inference rules and the stratification between abstraction and instance. In graphic types, they are given by the permissions of the nodes, plus the fact that the node is inert.

We abbreviate the three different permissions Flexible, Rigid and Locked by F, R, and L respectively. Notice that we have listed them here in strictly decreasing order—*i.e.* fewer transformations are permitted on L-nodes than on F-nodes.

To each permission intuitively corresponds a class of transformations that will be allowed at nodes having this permission. Instance transformations are permitted at flexible nodes. As we have seen, they are in general semantically not reversible, *i.e.* the inverse transformations would be unsound. Abstraction transformations are a subset of instance transformations that are permitted at rigid nodes. Since we want to use unification-based

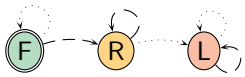
type inference, we disallow all inverse transformations (*i.e.* un-merging, un-raising and un-weakening) at rigid nodes, even though they would be sound. (However we will add them into a larger relation.)

The permission system is given by a function $\mathcal{P}$ from strings of flags to the set $\{\mathsf{F}, \mathsf{R}, \mathsf{L}\}$. Then, the permission of a bound node n of a type τ, which we write $\mathcal{P}_\tau(n)$ or $\mathcal{P}(n)$ when τ is clear from context, may be computed as $\mathcal{P}(\overline{\diamond}_\tau(n))$. We write $\overline{\diamond}_n$ instead of $\overline{\diamond}_\tau(n)$ when τ is clear from context.

The simplest way to define $\mathcal{P}$ is by a finite automaton, given in Figure 11. The states of the automaton are the three permissions, with F being the initial state (*i.e.* also the permission of the root node). Transitions are (inverse) binding edges labeled by their flag. The permission $\mathcal{P}(\overline{\diamond})$ is the state the automaton reaches when given the string $\overline{\diamond}$ as input. It is striking on this definition that flexible nodes form a prefix of the binding tree, followed by an alternation of rigid and locked regions as flags alternate. Notice that rigid permissions are larger than the ones defined in the previous section: all nodes under a rigid edge have rigid permissions.

Permissions are summarized in the table of Figure 11. The binding path column is given as a regular expression that describes the sets of binding paths having the corresponding permission. The colors of the rows of this table are sometimes used in drawings below to remind of the permissions unobstructively. We represent inert nodes in white, as permissions are not significant for them. For example, permissions are explicitly drawn on all types of Figure 12. For instance, node $\langle 11 \rangle$ of type τ_1 is rigid—its binding path is $\geqslant=$.

More restrictive permissions The looser the permissions, the larger the instance relation, the more "inference". Of course, permissions should remain within the limit of type soundness. The permissions we have described above are a slight generalization of the ones that could be reconstructed from a careful reading of the syntactic instance relation, and described by the following automaton:



The difference lies in the set of rigid nodes. In the syntactic permissions, one can only encounter some flexible flags followed by rigid ones; afterwards, all the permissions are locked. With our looser definition, rigid binding edges behaves as a "protection" and reset locked-nodes to rigid ones. We have good reasons to believe that looser permissions preserve type soundness—-a formal verification is ongoing work.

A variant of looser permissions was initially suggested by François Pottier on syntactic types. Unfortunately, as mentioned earlier, a naive integration of looser permissions on syntactic types is unsound, due to annoying interaction with administrative rules used to deal with syntactic artifacts.

Interestingly, the instance relation is implicitly parameterized by the permission system $\mathcal{P}$: all results depending on permissions are obtained through lemmas that abstract over important properties of permissions, and thus apply to all permission systems that satisfy those lemmas. Hence, we may easily fall back to the stricter permissions, if ever need be.

5.3 Instance operations

We now classify the different ways in which two types may be in an instance relation. We isolate atomic instance relation steps that instantiate either the term-graph or the binding tree. The former relations, *grafting* and *merging* are strongly linked to the instance relation $\leqslant$ on term-graphs; they essentially operate on the structure of the type. The latter relations, *raising* and *weakening*, only operate on the binding tree. Moreover, each step is controlled by the permissions of the node it operates on (if the nodes involved are polymorphic).

We examine each of the instance transformations in more detail below. For each operation both a schematic depiction and a formal definition are given—most of the technical details may safely be skipped on a first read. Figure 12, which is used throughout this section, introduces a sequence of types, each of which is in a particular form of instance relation with it successor as shown at the bottom of the figure.

5.3.1 Grafting

The grafting operation (introduced in (§4.4) corresponds to the operation of substituting a polymorphic type variable by a type, as can be done in ML. In term-graphs, it corresponds to the operation of instantiating the underlying skeleton. As such, the corresponding instance transformation is semantically irreversible (as the skeleton is irrevocably changed), and can only occur at flexible nodes.

Definition 10 (*Grafting*) A type τ' is an (instance-)grafting of a type τ if τ' is obtained by replacing a flexible bottom node of τ by a type τ''.

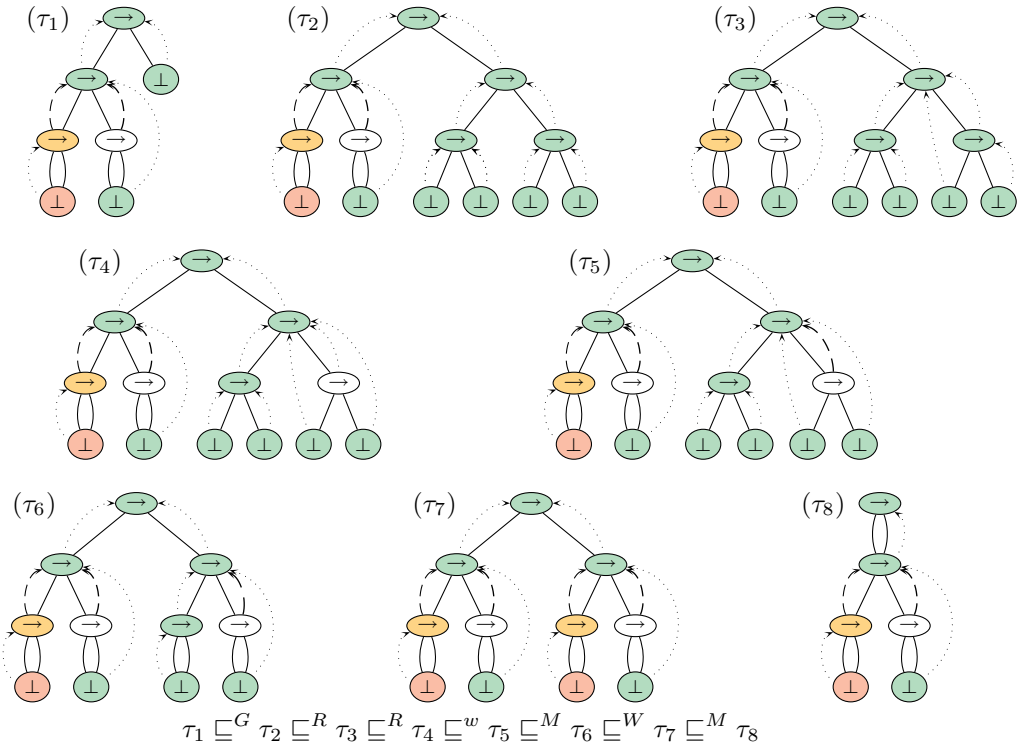


Figure 12: Example of type instance.

We write $\mathsf{Graft}(\tau'', n)$ for the function $\tau \mapsto \tau'$ and $\sqsubseteq^G$ for the reflexive transitive closure of the relation $\tau \mathcal{R} \tau'$ defined by $\exists n, \exists \tau'', \tau' = \mathsf{Graft}(\tau'', n)(\tau)$. □

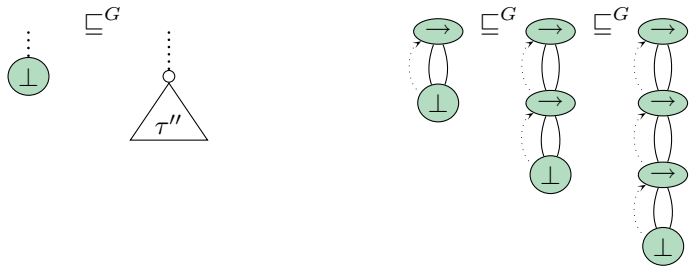


Figure 13: Sketch and example of grafting.

Let us consider some examples. The left part of Figure 13 presents a schematic depiction of grafting. In Figure 12, $\tau_1 \sqsubseteq^G \tau_i$ holds for $2 \leq i \leq 7$, the grafting occurring at node $\langle 2 \rangle$. The right part of Figure 13 shows a derivation of $\sigma_{id} \sqsubseteq^G \forall\,(\alpha \geqslant \sigma_{id})\ \alpha \to \alpha \sqsubseteq^G \forall\,(\alpha \geqslant \forall\,(\beta \geqslant \sigma_{id})\ \beta \to \beta)\ \alpha \to \alpha$. Indeed, let us temporarily call $\tau_{g,1}$, $\tau_{g,2}$ and $\tau_{g,3}$ those three graphs. The three following relations hold:

$$\tau_{g,2} = \mathsf{Graft}(\tau_{g,1}, \langle 1 \rangle)(\tau_{g,1}) \qquad \tau_{g,3} = \mathsf{Graft}(\tau_{g,1}, \langle 11 \rangle)(\tau_{g,2}) \qquad \tau_{g,3} = \mathsf{Graft}(\tau_{g,2}, \langle 1 \rangle)(\tau_{g,1})$$

Hence $\tau_{g,1} \sqsubseteq^G \tau_{g,3}$ can be proved either by transitivity of the grafting relation applied to the two first grafting steps, or by the single atomic last grafting step. Notice that grafting maintains the original binder and permissions, allowing in particular further refinements.

5.3.2 Merging

Merging is a subrelation of the relation induced by the fusion operator (§4.4). It corresponds to increasing the equivalence relation through instance in term-graphs.

The general form of merging is sketched on Figure 14. As required by the fusion operator, the type on the left is such that its subgraphs under the nodes n_1 and n_2 are equal. Some subparts of the subgraphs can already be shared, hence the overlap in the sketch. Moreover, n_1 and n_2 must be bound at the same node, with the same flag. Merging fuses those two nodes; it also requires as an additional condition that they both have the same non-locked permissions (*i.e.* flexible or rigid), which we refer to as the roots of the merging. Merging is allowed provided the roots are not locked, or inert; we represent them in blue to remind of this fact.

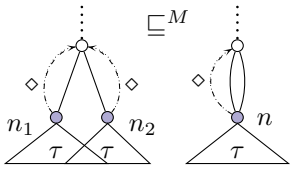


Figure 14: Sketch of merging.

Examples of merging are presented in Figure 12, where it is used thrice. Two pairs of bottom nodes are merged independently in type τ_5, $\langle 211\rangle$ and $\langle 212\rangle$ on the one hand, $\langle 221\rangle$ and $\langle 222\rangle$ on the other hand, leading to type τ_6. In type τ_7, the subgraphs under $\langle 1\rangle$ and $\langle 2\rangle$ are merged, resulting in τ_8.

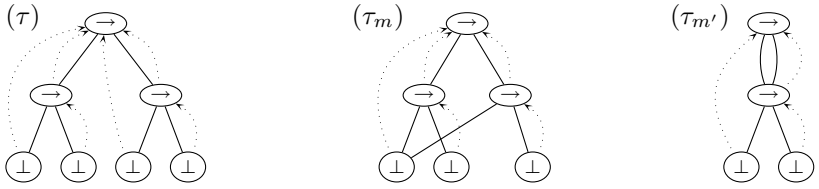


Figure 15: Merging conditions.

Let us call *merged* two nodes which were initially different, but are mapped into the same node by the merge. The formal definition of merging adds an additional condition on binding edges, explained below.

Definition 11 (*Merging)* A type τ' is a merging of a type τ at nodes n_1 and n_2 of τ if the following holds:

(1) τ' is the fusion of n_1 and n_2 in τ

(2) n_1 and n_2 have non-locked permissions, or are inert;

(3) for any two merged bound nodes n'_1 and n'_2 respectively under n_1 and n_2, n'_i must be transitively bound at n_i (*i.e.* $n'_i \succ\!\!\!\rightarrow^* n_i \in \tau$) for i in $\{1, 2\}$.

We write $\mathsf{Merge}(n_1, n_2)$ for the function $\tau \mapsto \tau'$, and $\mathsf{merge}(n_1, n_2)$ for its restriction to the case where n_1 and n_2 are monomorphic[5]. We write $\sqsubseteq^M$ (resp. $\sqsubseteq^m$) for the reflexive transitive closure of the relation $\mathcal{R}$ defined by $\tau\ \mathcal{R}\ \tau' \iff \exists n_1, n_2, \tau' = \mathsf{Merge}(n_1, n_2)(\tau)$ (*resp.* $\tau' = \mathsf{merge}(n_1, n_2)(\tau)$). □

Merging of leaf nodes is exactly as with term-graphs. Otherwise, the interesting condition is 3, which prevents mergings that would recursively merge nodes bound above n_1 and n_2. In those cases, mergings would only be correct under a much more complex control of permissions than condition 2. For example, consider the types τ, τ_m and $\tau_{m'}$ of Figure 15. The type $\tau_{m'}$ is $\tau[n_1 = n_2]$, but not $\mathsf{Merge}(\langle 1\rangle, \langle 2\rangle)(\tau)$: nodes $\langle 11\rangle$ and $\langle 21\rangle$ fail condition 3, since $\langle 11\rangle$ is not bound under $\langle 1\rangle$ in $\tau_{m'}$. In this particular case, the transformation can be decomposed into two atomic merges that both satisfy condition 3:

$$\tau_{m'} = \mathsf{Merge}(\langle 1\rangle, \langle 2\rangle)(\tau_m) \qquad\qquad \tau_m = \mathsf{Merge}(\langle 11\rangle, \langle 21\rangle)(\tau)$$

However, it is not always possible to do such a decomposition, as permissions may prevent merging the nodes that are bound above the nodes to be merged.

[5]Since all merged nodes are bound on either n_1 or n_2, all merged nodes are also monomorphic.

5.3.3 Raising

Raising performs in essence a scope extrusion, similar to coercing the System-F type $\tau' \to (\forall\alpha.\tau)$ into $\forall\alpha.(\tau' \to \tau)$ whenever α does not appear free in τ'. However, sharing of type variables in ML$^\mathsf{F}$ allows raising to soundly occur under left sides of arrows and deeper inside types. Namely, given two successive binding edges $n \rightarrowtail n' \rightarrowtail n''$, the first one can be raised above the second one to yield the edge $n \rightarrowtail n''$ whenever n is not locked. Raising is sketched on the left side of Figure 16. As for raising, raising is possible provided the node to be raised is either unlocked or inert.

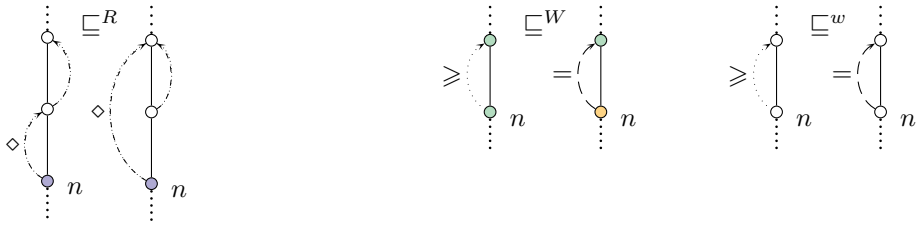


Figure 16: Sketches of raising and weakening.

Definition 12 (*Raising)* A binding tree $\overset{\succ}{\tau}'$ is the raising of a binding tree $\overset{\succ}{\tau}$ at node n if n is not locked or inert in τ and $\overset{\succ}{\tau}$ and $\overset{\succ}{\tau}'$ coincide except for the binding edge of n, which is such that $n \overset{\diamond}{\rightarrowtail} n' \rightarrowtail n'' \in \overset{\succ}{\tau}$ and $n \overset{\diamond}{\rightarrowtail} n'' \in \overset{\succ}{\tau}'$. We write $\mathsf{Raise}(n)$ for the function $\overset{\succ}{\tau} \mapsto \overset{\succ}{\tau}'$ (and raise if n is monomorphic).

Similarly, a type τ' is the raising at node n of a type τ if $\mathring{\tau}$ and $\mathring{\tau}'$ are equal and $\overset{\succ}{\tau}'$ is the raising at node n of $\overset{\succ}{\tau}$. We extend the function Raise and raise to types accordingly.

We write $\sqsubseteq^R$ (*resp.* $\sqsubseteq^r$) for the reflexive transitive closure of the relation defined by $\tau \mathcal{R} \tau' \iff \exists n, \tau' = \mathsf{Raise}(n)(\tau)$ (*resp.* $\tau' = \mathsf{raise}(n)(\tau)$). □

In Figure 12, τ_3 is a raising of node $\langle 221\rangle$ in τ_2 and τ_4 is a raising of node $\langle 222\rangle$ in τ_3.

5.3.4 Weakening

Weakening has two uses. On polymorphic nodes, it is used to forbid irreversible instance operations to occur underneath a node. It turns a flexible or inert binding edge leaving a flexible node into a rigid one, as illustrated on the right side of Figure 16. (Merging rigid edges has no sense, as they already have a rigid flag).

Definition 13 (*Weakening)* A type τ' is a weakening at a flexible or inert node n of a type τ if τ and τ' coincide except for the binding edge $n \overset{\geqslant}{\rightarrowtail} n' \in \tau$, which is replaced by $n \overset{=}{\rightarrowtail} n'$ in τ'.

We write $\mathsf{Weaken}(n)$ for the function $\tau \mapsto \tau'$, or $\mathsf{weaken}(n)$ if n is monomorphic. We write $\sqsubseteq^W$ (*resp.* $\sqsubseteq^w$) for the reflexive transitive closure of the relation defined by $\tau \mathcal{R} \tau' \iff \exists n, \tau' = \mathsf{Weaken}(n)(\tau)$ (*resp.* $\tau' = \mathsf{weaken}(n)(\tau)$). □

In Figure 12, type τ_7 is a (polymorphic) weakening of τ_6 at node $\langle 21\rangle$, while τ_5 is a monomorphic weakening of τ_4 at node $\langle 22\rangle$.

5.4 The instance relation

Instance is simply the union of all forms of instance operations.

Definition 14 (*Instance)* The instance relation on types $\sqsubseteq$ is the reflexive transitive closure $(\sqsubseteq^G \cup \sqsubseteq^M \cup \sqsubseteq^R \cup \sqsubseteq^W)^*$ of all forms of instances. □

Coming back to our example, we have seen above that $\tau_1 \sqsubseteq^G \tau_2 \sqsubseteq^R \tau_3 \sqsubseteq^R \tau_4 \sqsubseteq^w \tau_5 \sqsubseteq^M \tau_6 \sqsubseteq^w \tau_7 \sqsubseteq^M \tau_8$ holds. Hence, $\tau_1 \sqsubseteq \tau_8$ holds by definition of $\sqsubseteq$; note that a shortened decomposition of this fact is $\tau_1 \sqsubseteq^G \tau_7 \sqsubseteq^M \tau_8$. Moreover, operations can also be performed in a different order. However, the weakening of node $\langle 21\rangle$ must always be performed after the nodes $\langle 211\rangle$ and $\langle 212\rangle$ have been merged. Indeed, both nodes are locked after the weakening, which prevents any further operation on them.

Interestingly, the instance relation of ML$^\mathsf{F}$ can be seen as a refinement of the instance relation on term-graphs.

Property 3 *Given two types τ and τ' such that the instance $\tau \sqsubseteq \tau'$ holds, then $\mathring{\tau} \leqslant \mathring{\tau}'$ also holds.* □

Links with the syntactic presentation By using the syntactic permission (§5.2) instead of our usual system, we obtain another instance relation on graphs which is more restrictive than the one we used so far (as syntactic permissions are a restriction of the standard ones). This point is discussed further in (§8).

The two following properties abstract over the permission system (and so serve as interface to many proofs that do not then need to directly refer to the definition of permissions). It also indirectly highlights some properties of the instance operations that transform the binding tree.

Property 4 *Both permissions systems $\mathcal{P}$ (graphic and syntactic) satisfie:*

1. *If $\mathcal{P}(\overline{\diamond}_1\diamond_2\diamond_3) \neq \mathsf{L}$, then $\mathcal{P}(\overline{\diamond}_1\diamond_2\diamond_3\overline{\diamond}_4) = \mathcal{P}(\overline{\diamond}_1\diamond_3\overline{\diamond}_4)$.*
2. *If $\mathcal{P}(\overline{\diamond}\geqslant) = \mathsf{F}$, then $\mathcal{P}(\overline{\diamond}=\overline{\diamond}') \leq \mathcal{P}(\overline{\diamond}\geqslant\overline{\diamond}')$ for the order $\mathsf{L} \leq \mathsf{R} \leq \mathsf{F}$.* □

In particular, raising preserves permissions (which follows from 1) and weakening only restricts them (which follows from 2).

Binding trees carry two independent pieces of information: *where* and *how* nodes are bound. Interestingly, the two can almost be treated independently. The *where* is computationally essential and determines the shape of the binding tree while the *how* mostly acts as a filter by blocking certain instances. In particular, when raising is blocked by permission constraints, weakening never helps (Property 4.2). This enables to perform unification by computing the binding edges and their labeling independently (§7).

Notations In the remainder of the article we write $\sqsubseteq_1$ for the subrelation of $\sqsubseteq$ obtained by performing exactly one instance operator application. For $\mathsf{X}, \mathsf{Y} \in \{G, R, W, M, r, m, w\}$, we let $\sqsubseteq^{\mathsf{XY}}$ be $(\sqsubseteq^{\mathsf{X}} \cup \sqsubseteq^{\mathsf{Y}})^*$, and $\sqsupseteq^{\mathsf{X}}$ be the inverse relation of $\sqsubseteq^{\mathsf{X}}$. Thus $\sqsubseteq = \sqsubseteq^{GRMW}$. We also allow any meaningful combinations of those notations.

5.5 Similarity

As for first-order term-graphs, the instance relation is too fine grained (§3.3) and one may wish to read types modulo some inessential details, using a similarity relation which abstracts over them. A first cause of similarity already present in term-graphs is sharing of inner nodes. However, sharing of polymorphic (flexible) nodes is semantically significant in MLF, so we restrict similarity to monomorphic ones. The second source of similarity is the way monormophic nodes are bound[6]—*i.e.* what is their binder and binding flag.

Definition 15 (*Similarity*) We call *reversible instance* the subrelation $\sqsubseteq^{rmw}$. We call *similarity* the equivalence relation $(\sqsubseteq^{rmw} \cup \sqsupseteq^{rmw})^*$, written $\approx$. We call *instance modulo similarity* and write $\sqsubseteq_\approx$ the relation $(\sqsubseteq \cup \approx)^*$, also equal to $(\sqsubseteq \cup \sqsupseteq^{rmw})^*$. □

Thus all three types of Figure 10 are similar. Indeed, $\tau_3 = (\mathsf{merge}(\langle 21\rangle, \langle 22\rangle)\,;\,\mathsf{weaken}(\langle 21\rangle))(\tau_2)$, and $\tau_1 = (\mathsf{raise}(\langle 21\rangle)\,;\,\mathsf{raise}(\langle 22\rangle))(\tau_2)$ hold.

In practice, we work modulo similarity and are interested in $\sqsubseteq_\approx$; however, we often express results for $\sqsubseteq$ alone, as they are stronger than for $\sqsubseteq_\approx$, as well as easier to establish.

Notice that, from a purely semantic standpoint, we could include all transformations on rigid and inert nodes. However the resulting relation would not permit type inference. This is discussed in §5.6.

As for term-graphs, we could define an equivalence relation on graphic types (two types being equivalent if they are instances of the other). However, since graphic types do not require α-conversion, this relation degenerates to equality.

Lemma 2 *The kernel of $\sqsubseteq$ is equality.* □

Reversible instance is the reversible part of the instance modulo similarity relation. That is, all operations not in $\sqsubseteq^{rmw}$ are indeed irreversible.

Lemma 3 *The kernel of $\sqsubseteq_\approx$ is $\approx$.* □

Similarity of two types can be characterized in a quite simple way, by comparing the sharing and binding edges of their polymorphic nodes. This gives an efficient and simple algorithm for checking similarity (Fig. 17). In essence, it verifies that the two types unify without any change in their polymorphic nodes.

Lemma 4 *The algorithm* Similar *is a sound and complete algorithm for testing similarity, in linear time.* □

[6] We could have chosen to represent monormophic nodes without binding edges, but this solution has some technical cost.

Input: Two types τ_1 and τ_2
Output: A boolean indicating whether τ_1 and τ_2 are similar

1. Compute the first-order type-graph unifier of τ_1 and τ_2 (treating $\bot$ as a variable). Return false it does not exist.
2. Return false if an equivalence class holds any of the following:
   (a) A polymorphic and a monomorphic node,
   (b) Two polymorphic nodes of the same graph,
   (c) One polymorphic node of each type, but with different flag on their binding edges, or such that their binders are not in the same equivalence class.

   Otherwise Return true.

Figure 17: Algorithm Similar for testing similarity

5.6 The abstraction relation

(The definitions and results of this section suppose that the permissions are the graphic ones, and not *the ones matching the syntactic permissions.)*

By comparison with syntactic types, the instance relation on graphic types has been defined without referring to abstraction. This section reintroduces the abstraction relation on graphic types. Although technically unnecessary for solving unification, it remains interesting for pedagogical purposes.

The abstraction operations are sketched in Figure 18 and detailed in the definition below.

Definition 16 (*Abstraction)* The graphic abstraction relation on types, written $\Subset$, is the subrelation $(\Subset^M \cup \Subset^R \cup \Subset^W)^*$ of $\sqsubseteq$ where $\Subset^M$ is the subrelation of $\sqsubseteq^M$ such that the roots of the mergings are rigid or inert nodes, $\Subset^R$ is the subrelation of $\sqsubseteq^R$ that only raises rigid or inert nodes, and $\Subset^W$ is the subrelation of $\sqsubseteq^W$ that weakens only inert nodes. □

Notice that $\sqsubseteq^{rmw} \subset \Subset$.

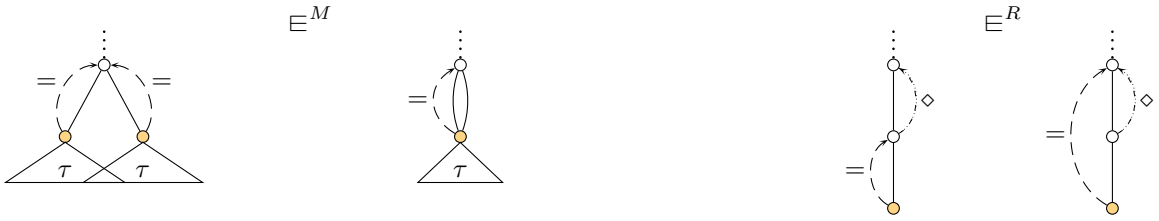
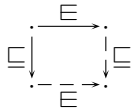


Figure 18: Abstraction operations.

The intuition behind the extended abstraction relation is hard to explain without referring to subject reduction. Roughly, paths of the form $\overline{\diamond}{=}\overline{\diamond'}$, *i.e.* below a rigid flag are *protected*, as they never allow a truly flexible instance (requiring flexible permission). Moreover, this remains true when stripping off any prefix of $\overline{\diamond}$, which simulates the possible deconstruction of the type during type-checking. Hence, performing an abstraction at path $\overline{\diamond}=$ will not have more observationable effect than performing this abstraction (later, during deconstruction) under the flag =, which was already allowed by the syntactic permissions.

The following commutative diagram is one of the key properties for type soundness.



Interestingly, this results follows by a very simple case when reasoning with graphic types, while it was a difficult and technically involved proof when reasoning with syntactic types.

In fact, when type inference is not an issue, *e.g.* in the type soundness proof, we may treat types up to the relation $\boxminus = (\sqsubseteq \cup \ni)^*$. That is, we may replace type instance by the larger relation $\sqsubseteq^{\boxminus} = (\sqsubseteq \cup \boxminus)^*$ [8]. It easily follows from the above commutative diagram (and the fact that $\sqsubseteq$ is a subrelation of $\sqsubseteq$) that $\boxminus$ and $\sqsubseteq^{\boxminus}$ are equal to $\sqsubseteq; \ni$ and $\sqsubseteq; \ni$, respectively. This also implies that the unification problem for $\sqsubseteq^{\boxminus}$ can be reduced to the unification problem for $\sqsubseteq$, which we solve in (§7). However, it is not the case that solving unification for $\sqsubseteq^{\boxminus}$ enables more aggressive type inference: indeed, taking $\sqsubseteq^{\boxminus}$ for type instance interacts with other rules in such a way that type inference can no longer be reduced to unification (and copying) for the type instance relation.

6 Properties of instance

The instance relation $\sqsubseteq$ is quite rich, as it features four different operations: grafting, raising, merging and weakening. However, those operations are mostly orthogonal. Hence it is possible to constrain the instance relation and subrelations, so as to obtain more canonical derivations (resulting in simpler proofs).

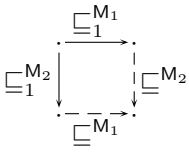
We use three different approaches:

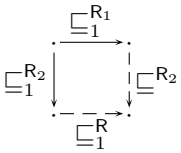
- Instance, similarity and instance modulo similarity can be reorganized so that to follow a certain order.
- We introduce "big-steps" relations that compare the shapes and two types and asserts they are in instance relation, without asking for a decomposition of this instance in term of the instance operators.
- We prove inversion lemmas proving that operations occurring inside a derivation can often be pushed at the beginning of the derivation.

6.1 Reorganizing relations

We start by showing that raising and merging are confluent:

Lemma 5 *The following local confluence diagrams are verified (where* M_i *range over* M *and* m*, and* R_i *range over* R *and* r*):*





□

Notice that similar results do not hold for $\sqsubseteq^G$ and $\sqsubseteq^W$: grafting at the same node two different types results in incompatible types, while weakening two different nodes on the same binding path must be done bottom-up, as the top-down strategy is forbidden by permissions.

The instance relation is such that one may consider ordered sequences of instance operations without loss of generality: graftings can always occur first, followed by raisings, and then mergings and weakenings interleaved. This flexibility is the key to an efficient implementation of unification (§7).

Theorem 1 (Ordered derivations) *The instance relation* $\sqsubseteq$ *is equal to* $\sqsubseteq^G ; \sqsubseteq^R ; \sqsubseteq^{MW}$*.* □

A sequence of elementary instance transformations is called *ordered* when they come in the decomposition order of Theorem 1. This is the case of the proof of $\tau_1 \sqsubseteq \tau_8$ in Figure 12.

Other simple decompositions (such as $\sqsubseteq^R ; \sqsubseteq^{MW} ; \sqsubseteq^G$) are false in the general case. Grafting must occur first, as it introduces new nodes which might need to be raised later. Weakening must occur last, because it restricts permissions. Merging and weakening must be interleaved because the former requires binding flags to be consistent.

Instance modulo similarity can also be decomposed, as all inverse reversible operations can be done at the end of the derivation. Likewise, similarity can be decomposed.

Lemma 6 *The relations* $\sqsubseteq_{\approx}$ *and* $\sqsubseteq ; \sqsupseteq^{rmw}$ *are equal. The relations* $\approx$ *and* $\sqsubseteq^{rmw} ; \sqsupseteq^{rmw}$ *are equal.* □

Note that $(\sqsubseteq_{\approx}) = (\sqsupseteq^{rmw} ; \sqsubseteq)$ does not hold: some of the operations which would need to be done at the beginning of the derivation would be polymorphic, *i.e.* not in $\sqsupseteq^{rmw}$.

6.2 Big step instance subrelations

6.2.1 Big step raising

The relation $\sqsubseteq_1^R$ requires checking that the node is raisable, a slightly complicated operation. In order to prove that $\tau \sqsubseteq^R \tau'$, we must prove that all intermediary types are well-dominated. Alternatively, we can define a relation that compares binding trees between two well-dominated types.

Definition 17 Given two types τ and τ', we say that τ' is big-step raising of τ, written $\tau \sqsubseteq^\uparrow \tau'$, if and only if:

- $\mathring{\tau} = \mathring{\tau}'$
- $\forall n,\ \overset{\diamond}{\tau}(n) = \overset{\diamond}{\tau'}(n)$
- $\forall n,\ n \succ\!\!\rightarrow n' \in \tau' \implies \begin{cases} n \succ\!\!\rightarrow n' \in \tau \\ \text{or} \\ n \succ\!\!\rightarrow \overset{+}{\succ\!\!\rightarrow} n' \in \tau \text{ and } n \text{ is not locked or inert in } \tau' \end{cases}$ □

The relation $\sqsubseteq^\uparrow$ avoids checking well-domination of intermediary types, but is equivalent to $\sqsubseteq^R$: a sequence of correct atomic raisings can always be found through careful ordering. In fact, we prove that all orderings raising lowest nodes first are possible. Of course, other valid raising strategies might also be possible.

Lemma 7 *Consider two types τ and τ' such that $\tau \sqsubseteq^\uparrow \tau'$. Then all derivations $\tau \sqsubseteq^R \tau'$ of the form $\tau' = (\mathsf{Raise}(n_1)\,;\ldots;\mathsf{Raise}(n_k))(\tau)$ where $i > j$ implies $\neg(n_j \overset{+}{\circ\!\!\rightarrow} n_i)$ are valid.* □

Theorem 2 *The relations $\sqsubseteq^R$ and $\sqsubseteq^\uparrow$ are equal.* □

6.2.2 Big step merging and weakening

As we have for $\sqsubseteq^R$, we can characterize a "big-step" merging and weakening operation, and prove that it is derivable using the usual "small-steps" operations.

Definition 18 We say that τ' is a merging-weakening of τ, written $\tau \sqsubseteq^\leqslant \tau'$ if

1. $\hat{\tau} = \hat{\tau}'$
2. $\forall n,\ n \succ\!\!\rightarrow n' \in \tau \implies n \succ\!\!\rightarrow n' \in \tau'$
3. $\tilde{\tau} \subset \tilde{\tau'}$
4. $\forall n, n'$ such that $\overset{\succ}{n} = \overset{\succ}{n'}$, if $n \not\sim n'$ and $n \overset{\sim}{\tau'} n'$, then n and n' are either not locked or inert.
5. $\forall n, \overset{\diamond}{\tau}(n) = \diamond \implies \begin{cases} \overset{\diamond}{\tau'}(n) = \diamond \\ \text{or} \\ \diamond = (\geqslant) \text{ and } n \text{ is flexible or inert in } \tau \end{cases}$ □

The first two points asserts that the underlying tree and binding edges are unchanged. The third point ensures that τ' merges more nodes than τ, while the fourth verifies that permissions are verified for the merging. The last point checks that all weakenings that occurred were allowed.

Theorem 3 *Given two types τ and τ', $\tau \sqsubseteq^\leqslant \tau'$ if and only if $\tau \sqsubseteq^{MW} \tau'$.* □

Notice that the operations are performed bottom-up. A top-down approach is often impossible, because the weakenings could prevent some operations under them.

6.2.3 Grafting unconstrained graphs

Given two types τ and τ' such that $\tau \sqsubseteq \tau'$, it is always possible to find a derivation of this result which starts by using graftings (Theorem 1). However, there are many possibilities as to which type to graft. As we have already mentioned, in Figure 12, the relation $\tau_1 \sqsubseteq^G \tau_i$ holds for $2 \leq i \leq 7$. In τ_2, we have grafted a "big" type (in terms of number of nodes), but with a simple structure: there is no sharing, and all binders are flexible. Conversely, in τ_7 we have directed grafted a complicated type (thus making the derivation $\tau_1 \sqsubseteq \tau_8$ shorter). From a reasoning point of view, working with τ_2 is much easier than with τ_7. In this section we show that this type of grafting is always possible.

Definition 19 (*Expansion type)* Given a term-graph t, we define its *expansion* $t_\triangle$ by:

- $\mathring{t_\triangle}$ is the unique tree-like term-graph which has the skeleton of t, *i.e.* $\hat{t_\triangle}$ is $\hat{t}$ and every node is reduced to a single path in $\tilde{t_\triangle}$.
- $\overset{\succ}{t}_\triangle$ binds flexibly all the nodes to their ancestor, *i.e.* $\overset{\succ}{t}_\triangle = \{n \succ\!\!\!\longrightarrow n' \mid n' \circ\!\!\!\longrightarrow n\}$. □

For example, in Figure 12, the subgraph $\tau_2/\langle 2\rangle$ is the expansion of $\mathring{\tau_2}/\langle 1\rangle$.

It is possible to prove that τ is always an instance of $\mathring{\tau}_\triangle$. The strictly more general result also holds:

Lemma 8 *Let τ' be an instance of a type τ, and n a bottom node of τ that is not a bottom node in τ'. Then $\tau \sqsubseteq^G \tau[(\mathring{\tau'}_\triangle/n)/n] \sqsubseteq \tau'$.* □

As a direct consequence:

Corollary 1 *Let τ' be an instance of a type τ. Let $n_i^{i\in 1..k}$ be the bottom nodes of τ that are not bottom nodes in τ'. We define*[7] *$\tau[\tau'/\bot]$ as $\tau[(\mathring{\tau'}_\triangle/n_1)/n_1]\ldots[(\mathring{\tau'}_\triangle/n_k)/n_k]$. Then the relations $\tau \sqsubseteq^G \tau[\tau'/\bot] \sqsubseteq \tau'$ holds.* □

$\tau[\tau'/\bot]$ is the smallest type τ'' (w.r.t. the ordering induced by the instance relation) such that $\tau \sqsubseteq^G \tau'' \sqsubseteq \tau'$ holds, and $\hat{\tau}'$ and $\hat{\tau}''$ coincide. Indeed, the derivation of $\tau[\tau'/\bot] \sqsubseteq \tau'$ does not use any grafting, as both sides already have the same skeleton.

6.3 Performing an instance operation early

In this section, we consider two types τ and τ' such that $\tau \sqsubseteq \tau'$. Intuitively, we prove that if an instance operation o is performed "sometimes" in the derivation $\tau \sqsubseteq \tau'$, and if it can be applied to τ, then $o(\tau) \sqsubseteq \tau'$ holds (when o is not weakening).

If the underlying tree is instantiated in τ', we can start by performing an atomic grafting.

Lemma 9 *Let n be a node of τ such that $\tau(n) = \bot$ and $\tau'(n) \neq \bot$. Let τ'' be the type such that $\tau''(\{\epsilon\}) = \tau'(n)$ and with $\mathsf{arity}(\tau'(n))$ children bottom nodes flexibly bound at the root. Then $\tau \sqsubseteq_1^G \mathsf{Graft}(\tau'', n)(\tau) \sqsubseteq \tau'$ holds.* □

If a node n is raised in τ' and can be raised in τ, the raising can be performed immediately.

Lemma 10 *Let n be such that it is raisable in τ, and $\overset{\succ}{\tau}(n) \neq \overset{\succ}{\tau'}(n)$. Then $\tau \sqsubseteq_1^R \mathsf{Raise}(n)(\tau) \sqsubseteq \tau'$ holds.* □

Similarly, if two nodes are merged in τ' and mergeable in τ, the merging can be done first.

Lemma 11 *Suppose there exists n_1 and n_2 merged in τ' such that merging n_1 and n_2 in τ is possible. Then $\tau \sqsubseteq_1^M \mathsf{Merge}(n_1, n_2)(\tau) \sqsubseteq \tau'$ holds.* □

For weakenings, the result is not as obvious. Indeed, if we weaken too early, we will present some valid transformation later. However, if a flexibly bound node must be merged with a rigidly bound one, and the nodes bound under them are isomorphic, we can use the second node as a "witness": indeed, transformations which would become impossible under the first node are already impossible under the second.

Lemma 12 *Let n a node that can be weakened in τ such that $\overset{\diamond}{\tau'}(n) = (=)$. Suppose there exists n' merged with n in τ', $\overset{\diamond}{\tau}(n') = (=)$, and the subgraphs for the bound nodes under n and n' are identical. Then $\tau \sqsubseteq_1^W \mathsf{Weaken}(n)(\tau) \sqsubseteq \tau'$ holds.* □

[7] The result of this operation does not depend on the order of $n_i^{i\in 1..k}$ as grafting at nodes $n_1, \ldots, n_k$ commutes.

7 Unification

This section presents the unification problem for ML^F types and an efficient algorithm to solve it.

7.1 Unification problem

The unification problem for ML^F is quite standard: given two types τ_1 and τ_2, find a type τ_u that *unifies* those types, *i.e.* such that $\tau_1 \sqsubseteq \tau_u$ and $\tau_2 \sqsubseteq \tau_u$. It is already known that the ML^F unification problem for the syntactic presentation is *principal*, *i.e.* that all solutions are instances of a more general unifier τ_u [7]. However, we propose a more general definition.

Definition 20 (*Generalized unification)* Given a type τ, we say that a type τ' is a *unifier* of a set of nodes N in τ if τ' is an instance of τ in which all nodes of N are shared. Moreover, τ' is a *principal unifier* if any other unifier of N in τ is an instance of τ'. □

Generalized unification is more general than unifying two types. In fact, the latter class of problems can be encoded into the former one.

Figure 19: Encoding for standard unification.

Property 5 *Two types τ_1 and τ_2 unify into a type τ_u if and only if the nodes $\langle 1\rangle$ and $\langle 2\rangle$ unify in the type τ of Figure 19 into a type having the shape of τ'_u (*i.e. *$\langle 1\rangle$ is closed and $\tau'_u/1 = \tau_u$).* □

7.2 Admissible problems

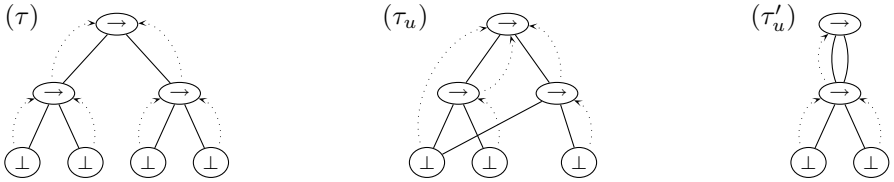


Figure 20: A problem without a principal solution.

Generalized unification significantly expands syntactic unification, and is in fact *too* powerful: some problems (not expressible in a syntactic setting) can have a non-principal set of solutions.

Consider unifying the nodes $\langle 11\rangle$ and $\langle 21\rangle$ in the type τ of Figure 20. A first unifier is τ_u: the two nodes have been raised once, and then merged. However, other unifiers exist, including τ'_u which is obtained by merging the nodes $\langle 1\rangle$ and $\langle 2\rangle$, which indirectly merges $\langle 11\rangle$ and $\langle 21\rangle$. There does not exist a unifier more general than those two ones, as there is an incompatible choice to be made between raising the edges (and merging the leaves), which irreversibly instantiates the binding structure, or merging the upper nodes, which irreversibly instantiates the upper nodes of the underlying term-graph.

We nevertheless use generalized unification, as it is possible to characterize an important set of problems that admit principal solutions. This set includes unification under the root of the type, as used to encode unification of two different types (Figure 19), but also other interesting cases to be used in type inference. We call *admissible* those problems.

The remainder of this section, which is a little technical, can easily be skipped on a first read.

Definition 21 Given a type τ and a set of nodes N, we say that (τ, N) is an *admissible* problem (or that N is admissible for τ) if the set of nodes $\{n'' \in \tau \mid \exists n \in N, \exists n' \in \tau, n' \succ\!\!\rightarrow n'' \circ\!\!\xrightarrow{+} n \circ\!\!\xrightarrow{*} n'\}$ is totally ordered by the domination relation $\circ\!\!\triangleright\!\!\rightarrow$ induced by $\circ\!\!\rightarrow$. □

Input: A type τ and a set of nodes N.
Output: A type τ_u that unifies N, or Failure.

1. Let t_u be the first-order unifier of the nodes N in the term-graph $\mathring{\tau}$, treating $\bot$ as a variable. Fail if t_u does not exist, or if it is cyclic.
2. Let $\overset{\succ}{\tau}_u$ be $\mathsf{Rebind}(t_u, \overset{\succ}{\tau})$. Fail if Rebind fails.
3. Return $\tau_u = (t_u, \overset{\succ}{\tau}_u)$.

Figure 21: Unif_N algorithm.

It is difficult to give an intuition of this definition without actually proving that it ensures principality of unification problems. Very roughly, non principality of unification always originates from a merging/raising competition (as illustrated on the example of Figure 20). Admissible problems will ensure that such potential conflicts will always occur between nodes in domination relation and thus can only be solved by raising, as merging would create cycles in the structure.

In the example of Figure 20, the set $N = \{\langle 11\rangle, \langle 21\rangle\}$ is not admissible for τ or τ'. Indeed, $\langle 1\rangle$ and $\langle 2\rangle$ (which are the binders of $\langle 11\rangle$ and $\langle 21\rangle$, and verify the condition above) are not comparable for $\circ\!\!\rightarrowtail\!\!\rightarrow$ in $\mathring{\tau}$ or $\mathring{\tau}'$.

We characterize a few set of nodes that are guaranteed to be admissible. In particular, they subsume the problems encoding unification under the root.

Property 6 *Consider a type τ and a node n of τ:*

- *Any subset N of $\{n' \mid n \circ\!\!\longrightarrow n'\}$ is admissible for τ.*
- *Any subset N of $\{n' \mid n' \succ\!\!\longrightarrow n\}$ is admissible for τ.*
- *Any set $N = \{n, n'\}$ where $n' \overset{+}{\succ\!\!\longrightarrow} \overset{\succ}{n}$ is admissible for τ.*
  (This last case is very useful in type inference.) □

Admissible problems are also stable by instance.

Property 7 *Consider a type τ and a node n of τ. If N is admissible for τ, for any type τ' such that $\tau \sqsubseteq \tau'$, N is admissible for τ'.* □

This result does not hold with instance modulo similarity $\sqsubseteq_\approx$, as unmerging or lowering nodes can make a problem non admissible.

7.3 Unification algorithm

We present our unification algorithm Unif_N in Figure 21. The algorithm takes a type τ as input and outputs a type τ_u that unifies N, or fails. The algorithm is in two steps:

1. The first step unifies the nodes of N in $\mathring{\tau}$ using first-order unification; the result of this phase will be the structure of the unifier.
2. The second phase uses an auxiliary algorithm Rebind (presented in Figure 22) to build the binding tree of the unifier. Given a type τ and a term-graph t_u instance of $\mathring{\tau}$, it returns a binding-tree $\overset{\succ}{\tau}_u$ such that $(t_u, \overset{\succ}{\tau}_u)$ is an instance of τ, or fails.

We write $\mathsf{LCA}_G(n_1, ..., n_k)$ for the least common ancestor of the nodes $n_1, ..., n_k$ in a rooted graph G. In the following, nodes of τ are called m while those of τ_u are called n, with the following exception: for any node m of τ, we write $\tilde{m}$ the corresponding node of τ_u (*i.e.* the unique node of τ_u whose name extends the name of m). We say that a node n is *partially grafted* if there exists a bottom node m such that $\tilde{m} \overset{+}{\circ\!\!\longrightarrow} n$.

The algorithm Rebind proceeds in two steps:

1. *Building the binding tree.*

   The first phase binds the nodes of $\mathring{\tau}_u$. Given a node n, we call M_n the nodes of τ that are merged into n. The binding edges of those nodes (whose ending nodes are B_1^n) must be raised until they are all bound at

Input: A type τ and a term-graph t_u instance of $\mathring{\tau}$
Output: A binding tree $\overset{\succ}{\tau}_u$ for t_u, or Failure

1. **Building the binding tree**

   For each node n of t_u (visited in a top-down ordering), do:

   (a) Let M_n be $\{m \in \tau \mid \tilde{m} = n\}$.

   (b) Let $\diamond_n$ be $(=)$ if $(=)$ is in $\overset{\diamond}{\tau}(M_n)$, or $(\geqslant)$ otherwise.

   (c) Let n_B be $\mathsf{LCA}_{\overset{\succ}{\tau}_u}(B_1^n \cup B_2^n)$, with

   $$B_1^n = \left\{\overset{\succ}{\tau}(m) \mid m \in M_n\right\}$$
   $$B_2^n = \begin{cases} \{n' \mid n' \circ\!\!\longrightarrow n\} & \text{if } n \text{ is partially grafted} \\ \emptyset & \text{otherwise} \end{cases}$$

   (d) Let $\overset{\succ}{\tau}_u$ be $\overset{\succ}{\tau}_u + n \overset{\diamond_n}{\succ\!\!\longrightarrow} n_B$.

2. **Correction of the instance steps**

   (a) *Grafting:* Fail if there exists a non flexible bottom node m in τ such that $\tilde{m}$ is not a bottom node in t_u.

   (b) *Raising and weakening:* Fail if there exists a node n of τ_u non inert in τ_u, and a node m of M_n such that:

   > $\overset{\succ}{\tau}(m)$ is not n_B and m is locked in τ
   > *or*
   > $\diamond_n$ is $(=)$, $\overset{\diamond}{\tau}(m)$ is $(\geqslant)$ and m is not flexible in τ

   (c) *Merging:*

   i. Build the graph $\tau_\uparrow$ verifying $\mathring{\tau}_\uparrow = \mathring{\tau}$ and
   $m \overset{\star}{\succ\!\!\longrightarrow} m' \in \tau \wedge \tilde{m} \succ\!\!\longrightarrow \tilde{m}' \in \overset{\succ}{\tau}_u \implies m \succ\!\!\longrightarrow m' \in \tau_\uparrow$.

   ii. Fail if there exists m and m' distinct such that one of them is locked and non inert, $\tilde{m} = \tilde{m}'$, and $\overset{\succ}{\tau}_\uparrow(m) = \overset{\succ}{\tau}_\uparrow(m')$.

3. **Return** $\overset{\succ}{\tau}_u$.

Figure 22: Rebind algorithm.

the same node (step 1c)[8]. In parallel, a new flag $\diamond_n$ is computed for n; it is the best flag common to the nodes of M_n (step 1b).

The computation of $\overset{\succcurlyeq}{\tau}_u$ is incremental and is done in a top-down fashion: results found for the nodes that have already been considered are reused for computing the binders of the nodes underneath.

2. *Correction of the instance steps.*

The second phase verifies that the instance operations performed are correct w.r.t. permissions. Step 2a checks that the graftings performed to obtain the skeleton of τ_u from the skeleton of τ are allowed. Step 2b verifies that the weakenings and raisings that have been performed are correct. Step 2c revisits the polymorphic merging transforming $\overset{\circ}{\tau}$ into $\overset{\circ}{\tau}_u$.

This last step is difficult, as it needs to find where exactly the mergings originated. Consider the type τ_6 in Figure 12. In τ_7, the nodes $\langle 1\rangle$ and $\langle 2\rangle$ were merged, and we must verify that their permissions were correct. However, $\langle 11\rangle$ and $\langle 21\rangle$ were also indirectly merged. Yet, for them no check must be done. We use the following criterion: when two nodes are merged and their binders are equal, they are the root of a polymorphic merge.

Notations For pedagogical purposes, we introduce two intermediate graphs τ_g and τ_r that correspond to the steps of an ordered derivation of $\tau \sqsubseteq \tau_u$. Although they are never actually built[9] by the algorithm, they are useful to reason on it.

- The graph τ_g is τ in which all the graftings have been performed, *i.e.* exactly $\tau[\tau_u/\bot]$.
- The graph τ_r is τ_g in which all the raisings have been performed. It has the same term-graph as τ_g and its binding tree is defined by: $m \overset{\diamond}{\succ\!\!\longrightarrow} m' \in \tau_r$ if and only if $\tilde{m} \succ\!\!\longrightarrow \tilde{m}' \in \tau_u$ and $m \overset{\diamond\overline{\diamond}}{\succ\!\!\longrightarrow} m' \in \tau_g$.

7.4 Example of unification

Our running example will be Figure 12, in which we unify the nodes $\langle 1\rangle$ and $\langle 2\rangle$ of τ_1. Of course, τ_7 is one suitable unifier; in fact, τ_7 is $\mathsf{Unif}_{\{1,2\}}(\tau_1)$, while τ_2 and τ_4 are τ_g and τ_r respectively. In τ_2, the proper expansion type is grafted under the node $\langle 2\rangle$, which gives τ_g. For τ_r, the only nodes that must be raised in τ_g are $\langle 221\rangle$ and $\langle 221\rangle$, which are exactly the ones raised between τ_2 and τ_4.

Let us examine the actions of Rebind on our example:

Step 1 We suppose that Rebind tries to bind the node $n = \langle 121\rangle$. The only node of τ_1 merged into n in τ_u is $\langle 121\rangle$, thus M_n is $\{\langle 121\rangle\}$. However, there are three such nodes in τ_g (*i.e.* τ_2), namely $\langle 121\rangle$, $\langle 221\rangle$, and $\langle 222\rangle$. Let this set be M'_n.

The computation of $\diamond_n$ is easy, as M_n is a singleton. Consequently, $\overset{\diamond}{\tau}_u(n)$ is $\overset{\diamond}{\tau}_1(\langle 121\rangle)$, *i.e.* $(\geqslant)$.

Finding the new binder is slightly more subtle. In order to find n_B, the algorithm must raise all the nodes of M'_n until they are all bound at the same level. It starts by computing the binders of the nodes of M'_n:

- B_1^n contains the binders of the nodes present in τ.
- B_2^n contains the binders of the nodes that have been grafted between τ and τ_g. By construction of the expansion graphs, the binding edges of those nodes are the inverse of structure edges.

In our case, $B_1^n = \{\overset{\succcurlyeq}{\tau}_1(n)\} = \{\langle 1\rangle\}$. Meanwhile, $B_2^n = \{\langle 22\rangle\}$, *i.e.* the (common) binder in τ_g of the nodes $\langle 221\rangle$ and $\langle 221\rangle$ of $M'_n \setminus M_n$. The set $B_1^n \cup B_2^n$ is thus equal to $\{\langle 1\rangle, \langle 22\rangle\}$. At this stage of the algorithm, the node $\langle 22\rangle$, which is above n in $\overset{\circ}{\tau}_u$, is already (flexibly) bound to $\langle 2\rangle$. This last node is equal to $\langle 1\rangle$ in τ_u, hence $\mathsf{LCA}_{\overset{\succcurlyeq}{\tau}_u}(B_1^n \cup B_2^n)$ is equal to $\langle 1\rangle$.

Step 2a The only bottom node of τ no longer a bottom node in τ_u is $\langle 2\rangle$. It is flexible in τ, hence the step succeeds. This ensures in particular that $\langle 2\rangle$ can be grafted in τ_1.

Step 2b We again consider node $n = \langle 121\rangle$. The only node in M_n is $\langle 121\rangle$ itself, which is neither raised nor weakened, hence no permissions check are necessary. Note however that in the derivation $\tau_1 \sqsubseteq \tau_7$, the nodes $\langle 221\rangle$ and $\langle 222\rangle$ are raised and weakened, However, since they are the result of the grafting of an expansion graph, they have flexible permissions.

[8] We defer the discussion on B_2^n to (§7.4).

[9] The size of τ_g can be quadratic in the size of τ. Hence, building it would make impossible to have a linear complexity.

Step 2c In our simple example, $\tau_\uparrow$ is in fact equal to τ (*i.e.* τ_1). The only pair of nodes satisfying the conditions of step 2(c)ii other than being locked is $(\langle 1\rangle, \langle 2\rangle)$. However, since neither node is locked, the test succeeds. Note that while the nodes $\langle 211\rangle$ and $\langle 212\rangle$ were merged in our derivation of Figure 12, the algorithm does not check for them, as again it knows that they are flexible (as for $\langle 221\rangle$ and $\langle 222\rangle$).

7.5 Correctness of the algorithms

This section introduces the correctness results of the algorithm. All the results also apply to the stricter permission system. Hence, our algorithm can be reused unchanged to perform unification in exactly the syntactic version of ML^F.

We implicitly quantify over a type τ, a set of nodes N, and a first-order instance t_u of $\mathring{\tau}$. Unless mentioned otherwise, we do *not* assume that (τ, N) is admissible or that t_u is the principal first-order unifier of N in $\mathring{\tau}$.

7.5.1 Auxiliary results

We first give some auxiliary results for the proofs. We write $m \in \tau \subseteq n$ as a shorthand for $m \in \tau \wedge m \subseteq n$.

Lemma 13 *Let $n \succ\!\!\longrightarrow n' \in \tau_u$, and let m be a polymorphic node of τ_g merged into n. Then there exists a node m' merged in n' such that $m \succ\!\!\xrightarrow{+} m' \in \tau_g$. Moreover, m' is unique.* □

Rebind chooses the lowest possible binder for a node:

Lemma 14 *Let n be a node of τ_u. Let n' be a node of τ_u such that for every node m of τ_g merged into n there exists a node m' of τ_g merged into n' verifying $m \succ\!\!\xrightarrow{+} m' \in \tau_g$. Then, $n \succ\!\!\xrightarrow{+} n' \in \tau_u$.* □

Lemma 15 *The type $\tau_u = (t_u, \overset{\succ}{\tau}_u)$ is well-formed.* □

A direct consequence of the principality of first-order unification is that the structure of τ_v is a first-order instance of the structure of τ_u. In particular, this allows using nodes of τ_u as nodes of τ_v.

Lemma 16 *The first-order instance $\mathring{\tau}_u \leqslant \mathring{\tau}_v$ holds.* □

Rebind preserves existing permissions:

Lemma 17 *Let n be a node of τ_u such that every node m of τ that has been merged in n has at least the permission P. Then, n also has the permission P in τ_u.* □

A related result is that Rebind does not "invent" flags: any binding path found in τ_u was already present in τ_r. This result can be used to justify that some raisings and weakenings commute.

Lemma 18 *For any node n in τ_u, there exists $m \in \tau_r$ merged into n such that $\overline{\diamond}_{\tau_u}(n) = \overline{\diamond}_{\tau_r}(m)$.* □

Finally, for admissible problems only, the binding tree of the unifier returned by the algorithm is always more general than that of any other unifier.

Lemma 19 *Suppose that (τ, N) is admissible, and let τ_v be one unifier. For any node n of τ, $n \succ\!\!\xrightarrow{+} \overset{\succ}{\tau}_v(n) \in \tau_u$ holds.* □

Note that this result does *not* hold for some non-admissible problems, as evidenced by Figure 20.

7.5.2 Main correctness results

Theorem 4 *If $\mathsf{Rebind}(\tau, t_u)$ returns $\overset{\succ}{\tau}_u$, the instance relations $\tau \sqsubseteq^G \tau_g \sqsubseteq^R \tau_r \sqsubseteq^{MW} (t_u, \overset{\succ}{\tau}_u)$ hold.* □

In particular Unif is sound even on non admissible problems:

Corollary 2 (Soundness) *The algorithm* Unif *is sound.* □

Unif is also complete *on admissible problems*, and return a principal identifier.

Theorem 5 (Completeness and principality) *Suppose that N is admissible for τ. If there exists an unifier τ_v of (τ_u, N), $\mathsf{Unif}_N(\tau)$ returns a type τ_u; moreover, this type is more general than τ_v,* i.e. *$\tau_u \sqsubseteq \tau_v$.* □

Figure 23: Generalized unification problems and admissibility

Note that the proof of the completeness theorem also shows principality of Rebind on a given term-graph, even if the problem is not admissible.

Corollary 3 *Suppose there exists an unifier τ_v of (τ_u, N) such that $\mathring{\tau_v} = \mathring{\tau_u}$. Then* $\mathsf{Rebind}(\tau, \mathring{\tau_u})$ *returns $\succ\!\tau_u$ such that the type τ_u equal to $(\mathring{\tau_u}, \succ\!\tau_u)$ is more general than τ_v.* □

Finally, the following lemma justifies the fact that we do not need to study principality up to similarity. Indeed, it essentially "commutes" with unification.

Lemma 20 *Let τ_1 and τ_2 be two types, and N a set of nodes admissible for both types. Assume* $\mathsf{Unif}_N(\tau_1)$ *exists and $\tau_1 \approx \tau_2$. Then* $\mathsf{Unif}_N(\tau_2)$ *exists and* $\mathsf{Unif}_N(\tau_1) \approx \mathsf{Unif}_N(\tau_2)$. □

7.6 Complexity

For the sake of the complexity analysis, we assume that each of the following elementary operations takes constant time:

- finding the binder of a node;
- going from $m \in \tau$ to the corresponding node $\tilde{m} \in \tau_u$;
- finding the list of nodes of τ that are merged into a node of τ_u.

This can easily be achieved by using constant-time access structures for storing graphs and by keeping track of merges during unification. For the computation of least common ancestors, we use a dynamic algorithm that computes LCA queries in worst-case constant time, and in which adding new leaves takes constant-time [1].

Theorem 6 Unif *is linear in the size of its argument.* □

This linear-time bound relies on a linear-time unification algorithm for term-graphs. We can also use a union-find based first-order unification algorithm [5] instead, in which case we obtain a $n\alpha(n)$ complexity.

While the complexity bound of the algorithm used in the original syntactic presentation of MLF is not known, it has to perform many duplications and α-conversions. We think that it would not scale to larger inference problems that can appear *e.g.* in automatically generated code, encodings, or extensive use of polymorphic records and variant types.

7.7 Generalized unification problems

The definition of unification problems may be generalized to express simultaneous unification problems on the same type.

Definition 22 A *generalized unification problem* $(\tau, \sim)$ is a pair of a type τ and an equivalence relation $\sim$ on $\mathsf{dom}(\tau)$. A *solution* of $(\tau, \sim)$ is an instance τ' of τ such that $(\sim) \subseteq (\tilde{\tau}')$. □

The equivalence relation $\sim$ of a unification constraint $(\tau, \sim)$ may be represented on τ by unification edges between the nodes to unify. In practice, we only draw a subrelation of $\sim$ whose transitive closure is $\sim$.

Admissibility It is of course possible to generalize admissibility in the obvious way, by requiring all equivalence classes of τ to be admissible problems. However, this definition is too weak, as illustrated by Figure 23. Although it is clear that τ and τ' have the same solutions, τ is admissible according to this criterion, while τ' is not. Accordingly, we generalize the criterion by considering only relevant equivalence classes in τ.

Definition 23 A generalized unification problems $(\tau/\sim)$ is admissible if $\sim$ is equivalent to an equivalence relation $\sim'$ on τ such that the subproblem (τ/N) are admissible for each equivalence class N of $\sim'$. $\sim$ and $\sim'$ are equivalent if they have the same first-order solutions on $\mathring{\tau}$. □

With this criterion, τ' is admissible, as the equivalences relations on τ and τ' are equivalent. More generally, admissibility is preserved by congruence closures.

Generalized unification algorithm The Unif algorithm can be generalized in a straightforward manner to generalized unification problems, thanks to the clean separation between the computation of $\mathring{\tau_u}$ and $\overset{\succ}{\tau_u}$. In the first phase of Unif_N, it suffices to find the principal first-order unifier according to $\sim$ instead of N. This strategy is more efficient than unifying the equivalence classes of $\sim$ one after the other using Unif_N, which would require calling Rebind up to k times, where k is the number of equivalence classes in $\sim$.

This generalized unification algorithm is complete on admissible problems.

Lemma 21 *Given an admissible generalized unification problem $(\tau, \sim)$, if there exists a solution to the problem the generalized unification algorithm returns a type τ_u more general than any other solution.* □

7.8 Unification in restrictions of ML$^\mathsf{F}$

In the terminology of [8], the system we presented here is Full ML$^\mathsf{F}$, the most expressive one. Let us call *trivial* a flexible binding $\forall\,(\alpha \geqslant \sigma)\ \sigma'$ when $\sigma = \bot$. Two natural restrictions of Full ML$^\mathsf{F}$ exist:

Plain ML$^\mathsf{F}$ [10] is obtained by restricting types so that any flexible binding occurring after a rigid binding must be trivial. That is, all binding paths ending by a flexible binding must be of the form $\{\epsilon\} \xleftarrow{\geqslant^+\prec} \bot$ or $\{\epsilon\} \xleftarrow{\geqslant^*\prec}\xleftarrow{=^+\prec}\xleftarrow{\geqslant\prec} \bot$ (while in Full ML$^\mathsf{F}$, all binding paths $\{\epsilon\} \xleftarrow{\{\geqslant,=\}^*\prec}\xleftarrow{\geqslant\prec} \bot$ are allowed).

It can be shown that Plain ML$^\mathsf{F}$ has exactly the expressivity of System F with let bindings. It is also the system obtained by restricting type annotations in source terms to types of System F.

Simple ML$^\mathsf{F}$ is even more restricted, as any non-trivial flexible binding is forbidden. Simple ML$^\mathsf{F}$ is exactly as expressive to System F. In fact, if we quotient types by the abstraction relation (*i.e.* we consider $\sqsubseteq^{\boxminus}$ instead of $\sqsubseteq$), all types are equivalent to a System F type.

In both Plain ML$^\mathsf{F}$ and Simple ML$^\mathsf{F}$, the type instance relation is the restriction of of $\sqsubseteq$ to the types allowed in both systems. Since the restrictions on allowed types are preserved by unification, Unif can be used unchanged to perform unification in Plain ML$^\mathsf{F}$ and Simple ML$^\mathsf{F}$. Furthermore, the restriction of $\sqsubseteq$ to types of Simple ML$^\mathsf{F}$ exactly corresponds to the instance relation in the implicit version of System F:

$$\forall\,(\overline{\alpha})\ \sigma' \sqsubseteq_\mathsf{F} \forall\,(\overline{\beta})\ \sigma'[\overline{\sigma}/\overline{\alpha}] \qquad\qquad \overline{\beta} \notin \mathsf{ftv}(\forall\,(\overline{\alpha})\ \sigma')$$

Thus, independently of ML$^\mathsf{F}$, our unification algorithm can be used to perform unification on System F types.

8 Relating the syntactic and graphic versions of ML$^\mathsf{F}$

8.1 An informal comparison of syntactic and graphic instance

The instance relation of graphic ML$^\mathsf{F}$ is noticeably simpler than its syntactic counterpart. Indeed, it does not need to be defined under prefix, as it instead uses permission to deeply operate inside types. Likewise, context rules such as I-Context-R or I-Context-L are superfluous.

The atomic instance operation can be put in correspondence with some of rules of the syntactic presentation:

- grafting corresponds to the rule I-Bot;

[10] Also called Shallow ML$^\mathsf{F}$ in the original presentations of ML$^\mathsf{F}$ [6, 7].

- weakening corresponds to the rule I-RIGID
- raising corresponds to the derived rules I-UP and A-UP;
- merging has no direct equivalent in the syntactic presentation, and can only be obtained by a combination of several rules. In order to prove $\forall\,(\alpha \geqslant \sigma)\ \forall\,(\beta \geqslant \sigma)\ \alpha \to \beta \sqsubseteq \forall\,(\alpha \geqslant \sigma)\ \alpha \to \alpha$, one needs to syntactically instantiate the first type into $\forall\,(\alpha \geqslant \sigma)\ \forall\,(\beta \geqslant \alpha)\ \alpha \to \beta$, using I-HYP and context rules. This type is in turn equivalent to $\forall\,(\alpha \geqslant \sigma)\ \alpha \to \alpha$. This syntactic derivation requires to abstract the second occurrence of σ behind the name α, and to replace β by α everywhere using the equivalence relation. Comparatively, the graphic proof is direct and simpler.

Our similarity relation is also simpler, thanks to the graphical representation itself. Rules EQ-COMM (which deals with commutation of binder), EQ-FREE, (which is used to remove unused quantification) and EQ-VAR become useless. Additionally, rule EQ-MONO, which inlines monotypes, has no direct equivalent: indeed we require all nodes to be bound. However, as a counterpart to binding all nodes, we must deal with binding edges and sharing of monomorphic nodes. (In the syntactic presentation, this was done indirectly by using EQ-MONO in one direction, then in the other.) Yet, raising, merging and weakening of monomorphic nodes are subrelations of the more general instance relation, and we only need to establish properties once.

8.2 Translating graphic types to and from syntactic types

8.2.1 From graphic to syntactic types

A graphic type τ may be read as a syntactic type $\mathcal{S}_\tau(\{\epsilon\})$ using the algorithm of Figure 24. To each node n which is not the root we associate a variable α_n. On a given node, we first translate all the nodes bound on it, in the order imposed by $<_\mathcal{B}$. By construction, all the children of n are then already bound.

$$\begin{array}{rcl} \mathcal{S}_\tau(n) & = & \forall\,(\alpha_{n_1} \stackrel{\diamond}{\tau}(n_1)\ \mathcal{S}_\tau(n_1)) \ldots \forall\,(\alpha_{n_i} \stackrel{\diamond}{\tau}(n_i)\ \mathcal{S}_\tau(n_i))\ \ \tau(n)(\alpha_{n\cdot 1}, \ldots, \alpha_{n\cdot j}) \\ & & \quad \text{where } n_1, \ldots\ n_i \text{ is one possible ordering of } (\succ\!\!\longrightarrow n) \text{ for } <_\mathcal{B} \\ & & \quad \text{and } j = \mathsf{arity}(\tau(n)) \end{array}$$

Figure 24: Translation from types to syntactic types

Lemma 22 *Given a type τ, $\mathcal{S}_\tau(\{\epsilon\})$ is a well-scoped syntactic type.* □

The algorithm is not deterministic, as $<_\mathcal{B}$ is only a partial order. However this non-determinism is unimportant, as the differences are captured by the equivalence relation on syntactic types.

Lemma 23 *Given a type τ, if σ_1 and σ_2 are two translations of τ for different orderings of bound nodes, then σ_1 and σ_2 are equal up to some permutation of binders (hence syntactically equivalent for $\equiv$).* □

Lemma 24 *$\mathcal{S}_\tau(\{\epsilon\})$ can be computed in linear time in the size of τ.* □

8.2.2 From syntactic to graphic types

We limit ourselves to syntactic types generated by the grammar

$$\begin{array}{rcl} t & ::= & \alpha \mid C(\alpha, \ldots, \alpha) \\ \sigma & ::= & t \mid \bot \mid \forall\,(\alpha \diamond \sigma)\ \sigma \end{array}$$

That is, we disallow monotypes of the form $C(C(\ldots), \ldots)$. Given a syntactic type, it is always possible to transform it into an equivalent one (for $\equiv$) that follows this restriction by introducing new bounds for the monomorphic subtypes.

The translation from such a syntactic type σ into a graphic type $\mathcal{G}(\sigma)$ is defined inductively and uses an auxiliary environment ρ mapping variables to graphs. It returns a standard graph. New nodes are taken all distinct from one another in a global pool of nodes (which we leave implicit) using the notation "$\mathcal{V}\,n$." to mean the allocation of such a fresh node n. We write $r(\tau)$ for the root node of τ and use $+$ to aggregate elements (nodes, structure edges or binding edges) composing a standard graph. The algorithm $\mathcal{G}_\rho(\cdot)$ is described in Figure 25.

$$
\begin{array}{rcl}
\mathcal{G}_\rho(\bot) & = & \mathcal{V}\, n.(n \mapsto \bot) \\
\mathcal{G}_\rho(\alpha) & = & \rho(\alpha) \\
\mathcal{G}_\rho(C(\alpha_i^{i\in I})) & = & \mathcal{V}\, n.(n \mapsto C) + (n \circ\!\!\longrightarrow r(\rho(\alpha_i)))^{i\in I} \\
\mathcal{G}_\rho(\forall\,(\alpha \diamond \sigma)\ \sigma') & = & \mathsf{let}\ \tau = \mathcal{G}_\rho(\sigma)\ \mathsf{in}\ \mathsf{let}\ \tau' = \mathcal{G}_{\rho,\alpha\mapsto\mathcal{V}\, n_\alpha.(n_\alpha\mapsto\bot)}(\sigma')\ \mathsf{in} \\
 & & \mathsf{if}\ r(\tau') = n_\alpha\ \mathsf{then}\ \tau\ \mathsf{else} \\
 & & \mathsf{if}\ n_\alpha \notin \mathsf{dom}(\sigma')\ \mathsf{then}\ \tau'\ \mathsf{else} \\
 & & \tau'[\tau/n_\alpha] + r(\tau) \succ\!\!\!\stackrel{\diamond}{\longrightarrow} r(\tau')
\end{array}
$$

Figure 25: Translation from syntactic types to types

Translating a bottom yields a new graph containing only a bottom node as the root, while translating a variable retrieves the graph associated to this variable in the environment (by well-scopedness of syntactic types, the variable is always already present in ρ). Translating a monotype labelled by C is straightforward.

The difficult case is $\forall\,(\alpha \diamond \sigma)\ \sigma'$. We start by translating the bound σ into τ, and creating a new node n_α which will serve as a placeholder for α in the translation τ' of σ'. Then we graft n_α by τ in τ'. There are two special cases. When σ' is equivalent to α, only one node must be created for α and the translation is exactly τ. When α does not appear free in σ', translating σ is not useful. However, the condition "does not appear free" it too weak: σ must not be translated if α does not appear free in $\mathsf{nf}(\sigma')$ (where nf is the syntactic normal form). We detect this case by checking if n_α appears in τ'.

The translation returns a correct type:

Lemma 25 *Given a syntactic type σ, $\mathcal{G}(\sigma)$ is a well-formed type.* □

Moreover, the translation can be performed in linear time.

Lemma 26 *Given a syntactic type σ, $\mathcal{G}(\sigma)$ can be computed in linear time in the size of σ.* □

9 Conclusion

We have given a formal meaning to the informal graphic types used in the original presentation of ML^F [6]. We proposed a definition of type instance based on several independent operations on types: merging and grafting are well-known operations on first-order term-graphs; raising is a simple operation on the binding tree that reduces polymorphism; weakening and permissions are new and both work together to ensure that requested polymorphism is not reduced during instantiation.

We found that unification for ML^F-types can be performed in linear time. Unsurprisingly, the critical step seems to be the computation of the binding structure.

The most immediate application of our work is a simpler and efficient unification algorithm for ML^F. The language ML^F has already been used in the Morrow compiler [9]—an extension of core Haskell with second-order types—using the syntactic presentation. We believe its performance on large problems would be significantly improved by using graphic types and our algorithm.

Another immediate benefit is a simplification of ML^F presentation and meta-theory. Our understanding of the design space is also much improved, especially in the definition of the instance relation. We have proposed a slightly more permissive definition of permissions—but the soundness of ML^F for our enhanced permissions system remains to be verified.

Our experience with graphic types is that once the definitions and the main lemmas are settled, results are rather intuitive and easy. This contrasts with the previous approach based on syntactic types. Our better understanding may allow us to review other useful features common to ML-like languages, such as recursive types, generalized algebraic data types, subtyping, *etc.*

Future works A continuation of this work is to revisit type inference for ML^F using our graph presentation; we are in the process of formalizing a constraint-based approach. Primary results are encouraging, and draw close parallels with type inference algorithms for ML, known to be quite efficient in practice. In the meantime, we are implementing a graph-based prototype of ML^F, to verify that type inference remains indeed tractable, just as in ML.

By simplifying and increasing our understanding of ML^F types, the graphic presentation also enables exploring several extensions, such as recursive types, higher-order types, or primitive existential types.

The combination of recursive types and second-order polymorphism alone is already tricky [3]. We thus have only considered acyclic types here. Allowing cyclic term-graphs should be possible (even though we did not do so). The difficulty rather lies in the treatment of recursion in the binding structure. While our framework should extend to "monomorphic recursions" that do not interact with the binding structure, the general case should be more challenging.

Probably harder, but also quite useful would be to extend the mechanism of MLF to higher-order types. The interaction of β-reduction at the level of types with a first-order type inference *à la* MLF seems non-trivial.

While the encoding of existential types into universal types behaves rather well in MLF, as unpacking of existential types does not require type information but only the position of unpackings, it is tempting to believe that using primitive existential types instead of encodings would remove the need for unpacking positions as well. Unfortunately, this seems to be against the natural flow of type inference in MLF and hopeless at first glance.

Acknowledgments

We would like to thank anonymous referees for numerous helpful comments, Yann Régis-Gianas for close readings of early versions of this paper, François Pottier for suggesting a simpler charecterization of well-formed types, and Didier Le Botlan for providing us with an initial large collection of examples and counter-examples of graphic types, sharing with us his expertise on MLF, and for many insightful discussions all along this work.

References

[1] Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 235–244, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[2] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the Ninth ACM Conference on Principles of Programming Langages*, pages 207–212, 1982.

[3] Nadji Gauthier and François Pottier. Numbering matters: First-order canonical forms for second-order recursive types. In *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pages 150–161, September 2004.

[4] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, University of Paris VII, 1972.

[5] Gérard Huet. *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots, \omega$. Thèse de doctorat d'état, Université Paris 7, 1976.

[6] Didier Le Botlan. *MLF: Une extension de ML avec polymorphisme de second ordre et instanciation implicite*. PhD thesis, University of Paris 7, June 2004. (english version).

[7] 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.

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

[9] 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.

[10] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[11] Michael S. Paterson and Mark N. Wegman. Linear unification. *Journal of Computer and System*, 16(2):158–167, 1978.

[12] D. Plump. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 1 (Term graph rewriting), pages 3–61. World scientific, 1999.

$$\textsc{Eq-Refl}\quad (Q)\ \sigma \equiv \sigma$$

$$\textsc{Eq-Trans}\quad \frac{(Q)\ \sigma_1 \equiv \sigma_2 \qquad (Q)\ \sigma_2 \equiv \sigma_3}{(Q)\ \sigma_1 \equiv \sigma_3}$$

$$\textsc{Eq-Context-R}\quad \frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \equiv \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \equiv \forall\,(\alpha \diamond \sigma)\ \sigma_2}$$

$$\textsc{Eq-Context-L}\quad \frac{(Q)\ \sigma_1 \equiv \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma_1)\ \sigma \equiv \forall\,(\alpha \diamond \sigma_2)\ \sigma}$$

$$\textsc{Eq-Free}\quad \frac{\alpha \notin \mathsf{ftv}(\sigma_1)}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \equiv \sigma_1}$$

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

$$\textsc{Eq-Var}\quad (Q)\ \forall\,(\alpha \diamond \sigma)\ \alpha \equiv \sigma$$

$$\textsc{Eq-Mono}\quad \frac{(\alpha \diamond \sigma_0) \in Q \qquad (Q)\ \sigma_0 \equiv \tau_0}{(Q)\ \tau \equiv \tau[\tau_0/\alpha]}$$

Figure 26: Type Equivalence

$$\textsc{A-Equiv}\quad \frac{(Q)\ \sigma_1 \equiv \sigma_2}{(Q)\ \sigma_1 \sqsubseteq \sigma_2}$$

$$\textsc{A-Trans}\quad \frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2 \qquad (Q)\ \sigma_2 \sqsubseteq \sigma_3}{(Q)\ \sigma_1 \sqsubseteq \sigma_3}$$

$$\textsc{A-Context-R}\quad \frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \forall\,(\alpha \diamond \sigma)\ \sigma_2}$$

$$\textsc{A-Hyp}\quad \frac{(\alpha_1 = \sigma_1) \in Q}{(Q)\ \sigma_1 \sqsubseteq \alpha_1}$$

$$\textsc{A-Context-L}\quad \frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall\,(\alpha = \sigma_1)\ \sigma \sqsubseteq \forall\,(\alpha = \sigma_2)\ \sigma}$$

Figure 27: Type Abstraction

$$\textsc{I-Abstract}\quad \frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \sigma_1 \sqsubseteq \sigma_2}$$

$$\textsc{I-Trans}\quad \frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2 \qquad (Q)\ \sigma_2 \sqsubseteq \sigma_3}{(Q)\ \sigma_1 \sqsubseteq \sigma_3}$$

$$\textsc{I-Context-R}\quad \frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \forall\,(\alpha \diamond \sigma)\ \sigma_2}$$

$$\textsc{I-Hyp}\quad \frac{(\alpha_1 \geqslant \sigma_1) \in Q}{(Q)\ \sigma_1 \sqsubseteq \alpha_1}$$

$$\textsc{I-Context-L}\quad \frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall\,(\alpha \geqslant \sigma_1)\ \sigma \sqsubseteq \forall\,(\alpha \geqslant \sigma_2)\ \sigma}$$

$$\textsc{I-Bot}\quad (Q)\ \bot \sqsubseteq \sigma$$

$$\textsc{I-Rigid}\quad \frac{}{(Q)\ \forall\,(\alpha \geqslant \sigma_1)\ \sigma \sqsubseteq \forall\,(\alpha = \sigma_1)\ \sigma}$$

Figure 28: Type Instance

[13] 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.

[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.

Appendix

A Syntactic ML^F relations

The equivalence, abstraction and instance relations of the syntactic presentation of ML^F are presented in Fig 26, 27 and 28 respectively. Fig 29 presents two important derived rules.

A-Up
$$\frac{\alpha' \notin \mathsf{ftv}(\sigma_0)}{(Q)\ \forall\,(\alpha = \forall\,(\alpha' = \sigma')\ \sigma)\ \sigma_0 \sqsubseteq \forall\,(\alpha' = \sigma')\ \forall\,(\alpha = \sigma)\ \sigma_0}$$

I-Up
$$\frac{\alpha_2 \notin \mathsf{ftv}(\sigma)}{(Q)\ \forall\,(\alpha \geqslant \forall\,(\alpha' \diamond \sigma')\ \sigma)\ \sigma_0 \sqsubseteq \forall\,(\alpha' \diamond \sigma')\ \forall\,(\alpha \geqslant \sigma)\ \sigma_0}$$

Figure 29: Derived rules

B Table of notations

Notation	Definition	Location
	Metavariables	
π, Π	Paths and sets of paths	Sec. 3.1, p. 7
n	Nodes (*i.e.* sets of paths)	Sec. 3.2, p. 8
t	Term	Sec. 3.1, p. 7
g	Term-graph	Sec. 3.2, p. 8
τ	MLF type	Sec. 4.1, p. 11
$\diamond$, $\overline{\diamond}$	Flags ($\geqslant$ or $=$) and sequences of flags	Sec. 2.1, p. 6
	Paths and nodes	
ϵ	Empty path	Sec. 3.1, p. 7
$\{\epsilon\}$, $\langle\epsilon\rangle$	Root of a term-graph	
$\langle\pi\rangle$, $\langle n\rangle$	Node containing at least π or n	Sec. 3.2, p. 9
$\pi\pi'$, $n\Pi'$, Πn	Concatenation on paths and sets of paths	Sec. 3.1, p. 7
	Components of types	
$\mathring{\tau}$, $\overset{\circ}{\longrightarrow}_{\tau_{\mathrm{foo}}}$	Term-graph associated to τ or τ_{foo}	Sec. 4.1, p. 11
$\overset{\succ}{\tau}$, $\overset{\succ}{\longrightarrow}_{\tau_{\mathrm{bar}}}$	Binding tree of τ or τ_{bar}	Sec. 4.1, p. 11
$\hat{g}$, $\hat{\tau}$	Tree associated to g or $\mathring{\tau}$	Sec. 3.2 and 4.1, p. 8 and 11
$\tilde{g}$, $\tilde{\tau}$	Equivalence relation associated to g or $\mathring{\tau}$	Sec. 3.2 and 4.1, p. 8 and 11
$\overset{\succ}{n}$, $\overset{\succ}{\tau}(n)$	Binder of n in τ	Sec. 4.1, p. 11
$n \overset{k}{\circ\!\!\longrightarrow} n'$	Structure edge from n to n' labelled by arity k	Sec. 4.1, p. 11
$n \overset{\diamond}{\succ\!\!\longrightarrow} n'$	Binding edge from n to n' labelled by $\diamond$	Sec. 4.1, p. 11
$\mathcal{B}(n)$, $\mathcal{B}_\tau(n)$	Bubble of n in τ	Sec. 4.3, p. 12
$\overset{\diamond}{\tau}(n)$	Flag on the outgoing binding edge of n in τ	Sec. 4.1, p. 11
$\overline{\diamond}_n$, $\overline{\diamond}_\tau(n)$	Flags on the binding path between n and $\{\epsilon\}$ in τ	Sec. 5.2, p. 19
$\mathcal{P}(\overline{\diamond})$, $\mathcal{P}(n)$	Permissions associated to $\overline{\diamond}$ or $\overline{\diamond}_n$	Sec. 5.2, p. 18
$t(\pi), g(n), \tau(n)$	Symbol at path π or at node n in t, g or τ	Sec. 3.1, 3.2 and 4.1, p. 7, 8 and 11
	Miscellaneous relations	
$\circ\!\!\rightarrow\!\!\prec$	Domination order induced by $\circ\!\!\longrightarrow \cup \longleftarrow\!\!\prec$	Sec. 4.2, p. 12
$\circ\!\!\rightarrow\!\!\rightarrow$	Domination order induced by $\circ\!\!\longrightarrow$	
$<_{\mathcal{B}}$	Order on bubbles	Sec. 4.3, p. 13
	Operators on types	
τ/n	Projection at n in τ	Sec. 4.4, p. 13
$\tau[\tau'/n]$	Grafting of τ' at n in τ	Sec. 4.4, p. 13
$\tau[n_1 = n_2]$	Fusion of n_1 and n_2 in τ	Sec. 4.4, p. 13
$\tau_\triangle$	Expansion of τ	Sec. 6.2.3, p. 27
$\tau[\tau'/\bot]$	Grafting of all the bottom nodes of τ by the structure of τ'	Sec. 6.2.3, p. 27
	Instance operations on types	
$\mathsf{Graft}(\tau', n)(\tau)$	Grafting of τ' at position n in τ	Sec. 5.3.1, p. 19
$\mathsf{Raise}(n)(\tau)$	Raising of the edge leaving n in τ	Sec. 5.3.3, p. 22
$\mathsf{raise}(n)(\tau)$	Restriction of Raise to monomorphic nodes	Sec. 5.3.3, p. 22

(continued next page)

Notation	Definition	Location
$\mathsf{Merge}(n_1, n_2)(\tau)$	Merging of the subgraphs under n_1 and n_2 in τ	Sec. 5.3.2, p. 21
$\mathsf{merge}(n_1, n_2)(\tau)$	Restriction of Merge to monomorphic subgraphs	Sec. 5.3.2, p. 21
$\mathsf{Weaken}(n)(\tau)$	Weakening of the binder flag at n in τ	Sec. 5.3.4, p. 22
$\mathsf{weaken}(n)(\tau)$	Restriction of Weaken to monomorphic nodes	Sec. 5.3.4, p. 22
Instance related relations		
$\sqsubseteq$	Instance	Sec. 5.4, p. 22
$\sqsubseteq^G$	Instance by grafting	Sec. 5.3.1, p. 19
$\sqsubseteq^R$, $\sqsubseteq^r$	Instance by raising and monomorphic raising	Sec. 5.3.3, p. 22
$\sqsubseteq^\uparrow$	Instance by unchecked raising	Sec. 6.2.1, p. 26
$\sqsubseteq^M$, $\sqsubseteq^m$	Instance by general and monomorphic merging	Sec. 5.3.2, p. 21
$\sqsubseteq^W$, $\sqsubseteq^w$	Instance by weakening and monomorphic weakening	Sec. 5.3.4, p. 22
$\approx$	Similarity	Sec. 5.5, p. 23
$\sqsubseteq_\approx$	Instance modulo similarity	Sec. 5.5, p. 23
$\leqslant$	Instance on term graphs	Sec. 3.3, p. 9
Derived instance relations		
$\sqsubseteq^{XY}$	Reflexive transitive closure of $\sqsubseteq^X \cup \sqsubseteq^Y$	
$\sqsupseteq^X$	Symmetric relation of $\sqsubseteq^X$	
$\sqsubseteq^X_1$	Exactly one step of the instance operator corresponding to X	
Mathematical notations		
$\mathcal{R}\,;\mathcal{R}'$	Inverse composition $(x, z) \mapsto \exists y, x\ \mathcal{R}\ y \wedge y\ \mathcal{R}'\ z$	
$(N \longrightarrow)$	$\{\, n \,\vert\, \exists n' \in N, n' \longrightarrow n \,\}$	

Table 1: Notations

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399