A Graphical Presentation of ML^F Types with a Linear-Time Unification Algorithm.

Didier Rémy Boris Yakobowski

INRIA Rocquencourt

http://gallium.inria.fr/~remy http://www.yakobowski.org

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 binding 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 dags, 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 the specification.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; E.1 [*Data Structures*]: Graphs

General Terms Algorithms, Design, Languages

Keywords System F, ML^F, Unification, Types, Graphs, Binders

Introduction

The language ML^F [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]. ML^F is a conservative extension of ML that allows to type all System-F terms [7]. ML^F 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.

ML^F 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 ML^F 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.

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

TLDI'07 January 16, 2007, Nice, France.

Copyright © 2007 ACM 1-59593-393-X/07/0001…$5.00

Unfortunately, while type instance and a subrelation called *abstraction* play a key role in ML^F, 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 ML^F. 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 ML^F-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 out most of the syntactical artifacts that can be found in the original definition of ML^F.

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. Equivalence on syntactic types becomes, on graphic types, equality up to the sharing of monomorphic nodes—a much simpler relation. Syntactic types in the original instance relation are mapped to graphic types in instance relation (modulo equivalence), and conversely—both in linear time.

We also present a sound and complete linear-time unification algorithm on graphic types that implements unification on the corresponding syntactic types.

The paper is organized as follows. First, we recall the definition of syntactic ML^F types (§ 1) and introduce their graphic representation (§ 2). Then, we present the instance relation on graphs and some of its properties (§ 3). Finally, we describe the unification algorithm and its correctness proof (§ 4).

Full technical details and proofs can be found in the extended version of the paper [13]. A black-and-white version is also available at `http://gallium.inria.fr/~remy/project/mlf/`.

1. A brief introduction to (syntactic) ML^F

1.1 ML^F syntactic types

ML^F 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; both are defined by the following grammar in BNF form.

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

A first-order type t is defined as usual. A syntactic type σ is a first-order type t, a bottom type $\bot$ (that intuitively stands for the System-F type $\forall\alpha.\alpha$), or a quantified type of the form $\forall\,(\alpha \diamond \sigma)\ \sigma'$. A difference with System-F is that quantification always assign bounds to variables, which 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.

The type $\forall\alpha.\ \alpha \to \alpha$ of System F can be represented in $\mathsf{ML^F}$ as

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

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

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

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

In $\mathsf{ML^F}$, we can also write the type

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

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 $\forall\,(\beta = \sigma)\ \beta \to \beta$ 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

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

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

Combining both forms of quantification, the type

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

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

1.2 Type instance

A peculiarity of $\mathsf{ML^F}$ 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 bounds are often used in covariant contexts and rigid bounds in contravariant ones, quantification in $\mathsf{ML^F}$ 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 $\mathsf{ML^F}$. 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 did not request it. Conversely, it is generally unsafe—and 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—$\mathsf{ML^F}$ 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, $\mathsf{ML^F}$ 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.

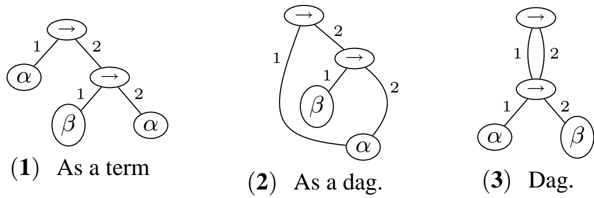
One of our main contributions is to revisit the instance relation (§ 3) using a graph presentation of types (§ 2). This new presentation eliminates most of the syntactic artifacts and so is more direct, allows more support for intuition, and supports extensions of the abstraction relation just mentioned without endangering soundness.

2. Graphic types

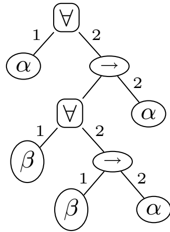
We remind of the graph representations of first-order types, which is often used—behind the scene—in efficient unification algorithms. We then introduce graphical notations on the well-known System-F types, as they offer a good support for intuitions. We finally define the representation of $\mathsf{ML^F}$ types as a refinement of System-F types.

2.1 Representing first-order types

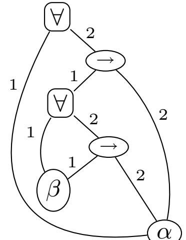
First-order types are usually understood as trees. For example, the tree (1) of Figure 1 represents the type $\alpha \to (\beta \to \alpha)$. However, it is sometimes convenient to identify all variables with the same name as shown in the dag (2). Efficient unifications algorithms often use such a graph representation explicitly, when described as imperative algorithms, or implicitly [5]. In fact, they not only share variables of the same name, but may also share inner nodes with identical subtrees, as illustrated with the representation (3) of $(\alpha \to \beta) \to (\alpha \to \beta)$.



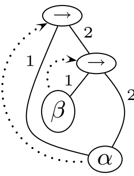
(**1**) As a term (**2**) As a dag. (**3**) Dag.



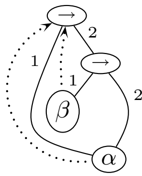
(**4**) Second-order term



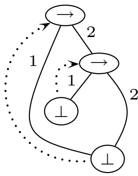
(**5**) Second-order dag



(**6**) binding edges



(**7**)



(**8**) anonymous variables.

Figure 1. Representations of first- and second-order types.

In the rest of this paper, we call *term-graph* a graph representing a first-order term that enforces the use of a single node for all occurrences of the same variable. Conversely, we call *skeleton* the tree expansion of a term-graph.

A more formal presentation of term-graphs can be found in the Appendix (§A).

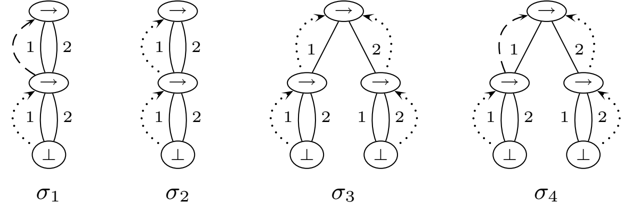
2.2 Representing second-order types

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 (4) of Figure 1. Using dags, we may represent it as in (5).

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 (6). We have oriented the binding edge from the bound variable to its binding node, for convenience.

Notice that this notation looses the order of adjacent binders and useless binders—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 (7).

Finally, as quantified variables are treated modulo α-renaming, we may advantageously draw them anonymously, as in (8). 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 (and not an element of Σ) as it does not clash with other symbols during unification. We intendedly reuse the same notation as the bottom type of syntactic $\mathsf{ML^F}$ types, since the notation "$\forall\,(\alpha \diamond \bot)$" plays the same role as "$\forall\alpha.$" in System-F types.



σ_1 σ_2 σ_3 σ_4

Figure 2. Examples of graphic $\mathsf{ML^F}$ types.

Nodes in graphs Let us introduce our convention for naming nodes in graphs. In the remainder of the paper, we designate a node n by the set of paths along which it can be reached from the root. Paths are obtained by following the arities on the structure edges of the graphic representation of the type. (In the following we often leave arities implicit, as we always write structure edges downwards and from left to right.) Letter π ranges over paths and ϵ stands for the empty path. As a path belongs to at most one node, we may write $\langle\pi\rangle$ for the node n that contains π.

Consider for example, type (3) of Figure 1: node $\{\epsilon\}$ is the root, labelled by $\to$; Both $\langle 1\rangle$ and $\langle 2\rangle$ designate the same node immediately below the root, reachable by the left or right paths. The node labelled α can be reached by the two paths 11 and 21; hence we may designate it by $\langle 11\rangle$, $\langle 21\rangle$, or its full name $\{11, 21\}$.

2.3 Representing $\mathsf{ML^F}$ types

Let us illustrate the graphic representation of $\mathsf{ML^F}$ types on the four types σ_1, σ_2, σ_3, and σ_4 introduced earlier (§ 1.1) and drawn in Figure 2. 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. In addition, we represent the bound of the variable in place of the unique node representing that variable. Hence, non-bottom nodes 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.

Notice that sharing of nodes that are not variables is possible (and significant). In $\mathsf{ML^F}$, 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 reflected in the graphic presentation by the fact that there are two occurrences 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 $\mathsf{ML^F}$ types. We first define pre-types, and then state well-formedness conditions they must satisfy in order to be types.

DEFINITION 1. A *(graphic) pre-type* τ is a pair of:

1. A term-graph $\mathring{\tau}$, whose nodes are labelled by elements of Σ or $\bot$. The bottom nodes must be leaves and the other nodes must respect the arity of their symbol.
2. A binding tree $\check{\tau}$ for $\mathring{\tau}$. That is, a set of binding edges labelled with flags that form an upside-down tree rooted at $\langle\epsilon\rangle$ and whose leaves include all bottom nodes. □

Remark that all bottom nodes need to be bound. Hence, we may only represent *closed* $\mathsf{ML^F}$ types.

Conversely, not all nodes need to be bound. We call *polymorphic* (*resp.* monomorphic) the nodes of τ that are bound (*resp.* unbound). For instance, node $\langle 12\rangle$ in τ on Figure 3 is monomorphic.

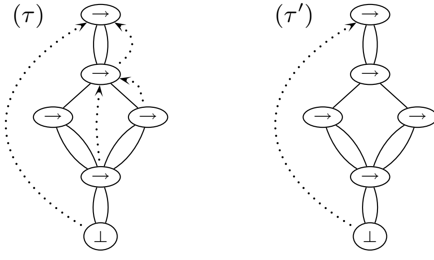


Figure 3. Illustration of gc.

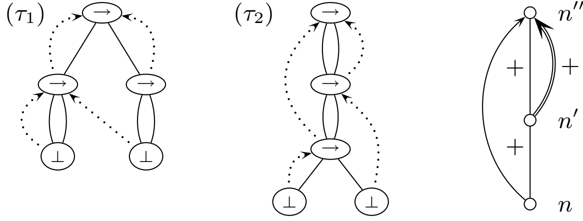


Figure 4. Invalid graphic types and diagram for scope nesting.

Useless binding edges In fact, according to Definition 1, non-bottom nodes with no incoming binding edge need not be bound. A binding edge leaving from such a node may then be deleted. This deletion preserves well-formedness and also the intuitive meaning of types, *i.e.* the two pre-types before and after deletion represent the same syntactic type. Given a pre-type τ, we write $\mathsf{gc}(\tau)$ for the result of recursively deleting all such edges from τ. Computing $\mathsf{gc}(\tau)$ can be done in linear time, by a depth-first traversal of τ.

Let us consider again the pre-type τ of Figure 3: the binding edges leaving from nodes $\langle 111\rangle$ and $\langle 12\rangle$, and, in turn, the one leaving from node $\langle 1\rangle$ can be deleted. The only (mandatory) remaining binding edge is the one leaving from the bottom node. Hence, the resulting pre-type τ' is $\mathsf{gc}(\tau)$.

Notations In the text, we write $n \circ\!\!\longrightarrow n' \in \tau$ (resp. $n \succ\!\!\xrightarrow{\diamond} n' \in \tau$) to mean that there is a structure edge (resp. binding edge with flag $\diamond$) from n to n' in τ. We may drop "$\in \tau$" when τ is implicit from context. We may also drop the flag if it is unimportant. If $n \succ\!\!\xrightarrow{\diamond} n' \in \tau$, we also write $\overset{\diamond}{\tau}(n)$ and $\overset{\smile}{\tau}(n)$ (or simply $\overset{\smile}{n}$) for $\diamond$ and n', respectively. We call n' the *binder* of n and we say that n is bound at n'. Finally, $\tau(n)$ denotes the symbol on the node n. We use $+$ and $*$ on top of arrows to denote the transitive closure and the reflexive transitive closure of the corresponding relation.

2.4 Well-formedness

All pre-types are not types. Types must correspond to syntactic types, hence ensure that variables have lexical scopes and that variables bounds are not mutually recursive. Figure 4 provides two examples of ill-formed binding trees:

- In pre-type τ_1, the node $\langle 21\rangle$ is bound at a node that is not among its parents, which is not permitted: in a syntactic presentation, the anonymous variable α used to represent the binding would be introduced on the left branch and used on the right branch, where it is out of scope.
- In pre-type τ_2, the nodes $\langle 1\rangle$ and $\langle 11\rangle$ are bound at the root. When translating the graph 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. Hence, τ_2 is not a type.

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

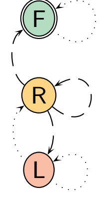


Figure 5. Permissions for the instance relation.

Before we present the well-formedness condition failed by τ_1, we recall the definition of dominators.

DEFINITION 2 (*Dominators).* Let g be a term-graph and n and n' two nodes of g. We say that n dominates n', and we write $n \lhd n'$ if every path from the root to n' goes through n. □

For example, in type σ_2 of Figure 2, $\langle 1\rangle$ dominates $\langle 11\rangle$, as all four paths 11, 12, 21 and 22 to $\langle 11\rangle$ go through $\langle 1\rangle$. On the other hand, in the second type of Figure 1, $\langle 2\rangle$ does *not* dominate $\langle 22\rangle$. Indeed, path 1 does not go through $\langle 2\rangle$. Notice that domination is a partial order on nodes.

DEFINITION 3 (*Domination).* The binding tree of a pre-type τ is *well-dominated* if every bound node is dominated by its binder, *i.e.*, $\forall n, n' \in \tau, n \succ\!\!\longrightarrow n'$ implies $n' \lhd n$. □

Domination rules out type τ_1 of Figure 4, in which node $\langle 1\rangle$ does not dominate node $\langle 21\rangle$.

To rule out type τ_2 we introduce the following condition:

DEFINITION 4 (*Nesting).* The binding tree of a pre-type τ is *well-nested* if for any nodes n, n' and n'' such that $n \succ\!\!\longrightarrow n'' \circ\!\!\xrightarrow{+} n' \circ\!\!\xrightarrow{+} n$, we also have $n' \succ\!\!\xrightarrow{+} n''$. □

This condition is presented as a diagram on the right side of Figure 4. On this diagram, binding edges are drawn with solid lines to mean that they may indifferently be rigid or flexible. The conclusion of the diagram is represented as a double line.

The pre-type τ_2 is not well-nested: the premises are met when taking $\langle 112\rangle$ for n and $\langle 11\rangle$ for n', but $\langle 11\rangle$ is bound at $\langle 1\rangle$, above the binding node $\langle 1\rangle$ of $\langle 112\rangle$.

DEFINITION 5 (*Types).* A *(graphic) type* is a well-dominated well-nested pre-type. □

3. 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. However, these transformations are only allowed in certain contexts determined by the binding structure. We abstract this contextual information as *permissions*.

3.1 Permissions

Permissions may be seen as additional information attached to every bound node by a preprocessing step. There are three different permissions: Flexible, Rigid and Locked, abbreviated by F, R, and L respectively, and listed 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. In general, they are semantically not reversible, *i.e.* the inverse transformations

would be unsound. Abstraction transformations are a subset of instance transformations that are still permitted at rigid nodes. Reversing abstraction transformations is not technically allowed for type inference purposes, although this would be sound.

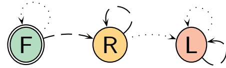
The permission of a node is uniquely determined by the sequence of flags along the path in the binding tree leading to that node. Remember that the binding tree (and hence the binding paths) walks the binding edges in inverse direction, from the root to the node. For example, in Figure 2, the binding path of nodes $\langle 11 \rangle$ and $\langle 2 \rangle$ of type σ_4 are $(=\geqslant)$ and $(\geqslant)$ respectively.

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 may be computed as $\mathcal{P}(\bar{\diamond})$ where $\bar{\diamond}$ if the binding path of n in τ.

The simplest way to define $\mathcal{P}$ is by a finite automaton, given in Figure 5. 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 $\mathcal{P}(\bar{\diamond})$ is the state the automaton reaches when given the string $\bar{\diamond}$ 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. It can also be noticed that the permission of a node determines the flag of the binding edge leaving that node.

Permissions are summarized in the table of Figure 5. 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. For example, permissions are explicitly drawn on all types of Figure 6. 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.

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.

3.2 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. Moreover, each step is controlled by the permissions of the node it operates on. Figure 6, 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.

We identify exactly four atomic transformations on types, *grafting* and *merging* that essentially operates on the underlying term-graph, and *raising* and *weakening* that only operates on the underlying binding tree. Each transformation is restricted by local side conditions on binding edges and permissions.

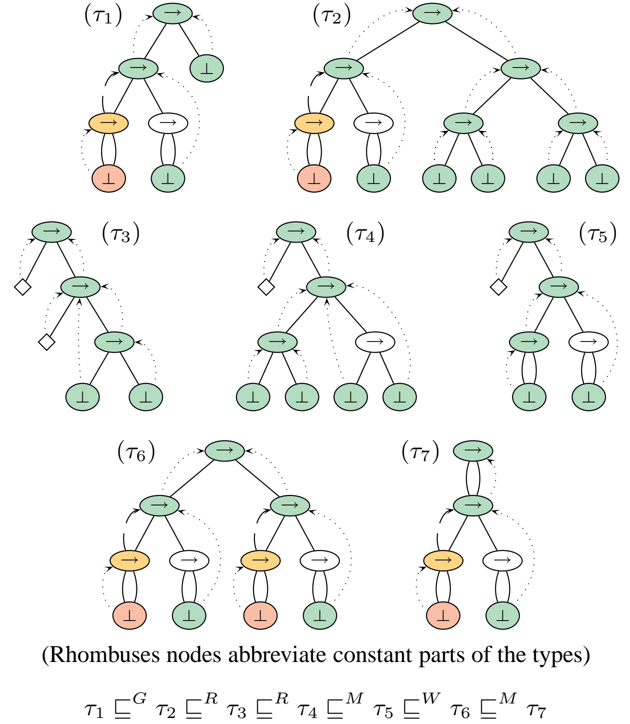


Figure 6. Example of type instance.

We examine each of the four 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.

3.2.1 Grafting

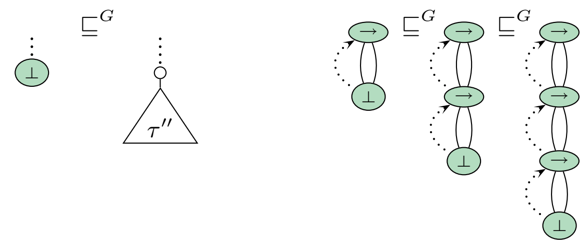


Figure 7. Sketch and example of grafting.

Grafting corresponds to the operation of substituting a polymorphic type variable by a type, as can be done in ML. It is a semantically irreversible transformation as the skeleton of the type changes, and thus it can only occur at flexible nodes.

Definition 6 (*Grafting*). A type τ' is a grafting of a type τ if τ' is obtained by replacing a flexible bottom node of τ by a type τ''.

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 $\exists n, \exists \tau'', \tau' = \mathsf{Graft}(\tau'', n)(\tau)$. □

Notice that if the type τ'' to be grafted at the node n of τ is monomorphic (*i.e.* its root has no incoming binding edge), then the edge $n \rightarrowtail \tilde{n}$ may be be garbage-collected in τ'.

Let us consider some examples. The left part of Figure 7 presents a schematic depiction of grafting. In Figure 6, $\tau_1 \sqsubseteq^G \tau_i$ holds for $2 \leq i \leq 6$, the grafting occurring at node $\langle 2 \rangle$. The right part of Figure 7 shows a derivation of $\sigma_{id} \sqsubseteq^G \forall (\alpha \geqslant \sigma_{id})\ \alpha \rightarrow \alpha \sqsubseteq^G \forall (\alpha \geqslant \forall (\beta \geqslant \sigma_{id})\ \beta \rightarrow \beta)\ \alpha \rightarrow \alpha$. Indeed, let us temporarily call $\tau_{g,1}$, $\tau_{g,2}$ and $\tau_{g,3}$ those three graphs. The three following relations hold:

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

Hence $\tau_{g,1} \sqsubseteq^G \tau_{g,3}$ can be proved either by transitivity of the instance relation applied to the two first grafting steps, or by the single atomic last grafting step. The fact that instantiation maintains the original binder and permissions so as to allow further refinements is particularly striking on graphic types.

3.2.2 Merging

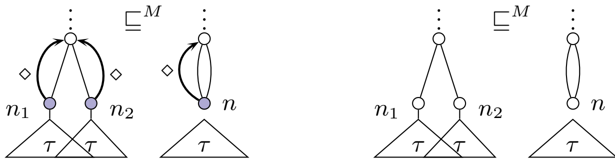


Figure 8. Sketch of merging.

Merging can be explained in two steps. Ignoring binding edges at first, merging two bottom nodes is akin to saying that $\alpha \rightarrow \alpha$ is an instance of $\alpha \rightarrow \beta$. Non-bottom nodes may also be merged, which increases sharing but leaves the underlying skeleton of the type unchanged. Now, taking back the binding structure into account, we also require that merged nodes agree on their binding edges, *i.e.* that their binding edges have the same flag and lead to the same (or merged) parents nodes.

The general form of merging is sketched on the left side of Figure 8. 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. They can be both flexible or both rigid, but not locked (we represent them in blue to remind of this fact). Merging n_1 and n_2 in the type on the left yields the one on the right, in which the identical subgraphs have been fused.

Simple examples of merging are presented in Figure 6, where it is used thrice. Two pairs of bottom nodes are merged independently in type τ_4, $\langle 211 \rangle$ and $\langle 212 \rangle$ on the one hand, $\langle 221 \rangle$ and $\langle 222 \rangle$ on the other hand, leading to type τ_5. In type τ_6, the subgraphs under $\langle 1 \rangle$ and $\langle 2 \rangle$ are merged, resulting in τ_7.

A particular case of merging is when all the merged nodes are monomorphic, which is depicted on the right side of Figure 8.

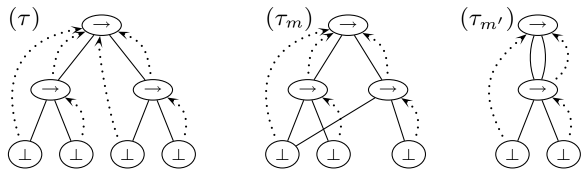


Figure 9. Merging conditions.

We 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.

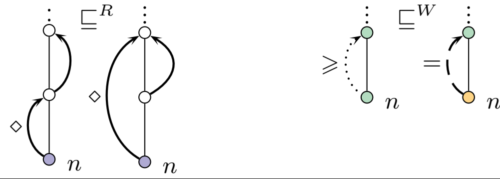


Figure 10. Sketches of raising and weakening.

DEFINITION 7 (*Merging*). A type τ' is a merging of a type τ at nodes n_1 and n_2 of τ if the following conditions hold:

(1) the subgraphs of τ under n_1 and n_2 are equal;
(2) τ' is the result of merging those two subgraphs in τ;

In addition, either all merged nodes are monomorphic or the two following extra conditions hold:

(3) n_1 and n_2 are bound on the same node, with the same flag, and have non-locked permissions;
(4) 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.* $n'_i \rightarrowtail^* n_i \in \tau$) for i in $\{1, 2\}$.

We write $\mathsf{Merge}(n_1, n_2)$ for the function $\tau \mapsto \tau'$. □

Merging of monomorphic nodes is as with first-order terms. Otherwise, the interesting condition is 4, which prevents merging 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 3. For example, consider the types τ, τ_m and $\tau_{m'}$ of Figure 9. The type $\tau_{m'}$ is not $\mathsf{Merge}(\langle 1 \rangle, \langle 2 \rangle)(\tau)$: nodes $\langle 11 \rangle$ and $\langle 21 \rangle$ fail condition 4, 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 4:

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

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.

We write $\sqsubseteq^M$ for the reflexive transitive closure of the relation $\mathcal{R}$ defined by $\tau\ \mathcal{R}\ \tau' \iff \exists n_1, n_2, \tau' = \mathsf{Merge}(n_1, n_2)(\tau)$. We write $\sqsubseteq^m$ the restriction of $\sqsubseteq^M$ to merging of monomorphic nodes (*i.e.* all pairs (n_1, n_2) in the definition of $\sqsubseteq^M$ are required to be monomorphic.

Merging has no direct equivalent in the syntactic presentation of $\mathsf{ML^F}$, and can only be obtained by a combination of several rules. For instance, one could prove that $\forall (\alpha \geqslant \sigma)\ \forall (\beta \geqslant \sigma)\ \alpha \rightarrow \beta$ can be syntactically instantiated into $\forall (\alpha \geqslant \sigma)\ \forall (\beta \geqslant \alpha)\ \alpha \rightarrow \beta$, which in turn is equivalent to $\forall (\alpha \geqslant \sigma)\ \alpha \rightarrow \alpha$. 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. The graphic proof is direct and simpler.

3.2.3 Raising

Raising performs in essence a scope extrusion, similar to coercing the System-F type $\tau' \rightarrow (\forall \alpha. \tau)$ into $\forall \alpha. (\tau' \rightarrow \tau)$ whenever α does not appear free in τ'. However, sharing of type variables in $\mathsf{ML^F}$ 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 10.

DEFINITION 8 (*Raising*). A type τ' is the raising of a flexible or rigid node n of a type τ if τ and τ' coincide except for the binding edges of n, which are such that $n \overset{\diamond}{\rightarrowtail} n' \rightarrowtail n'' \in \overset{\curlyvee}{\tau}$ and $n \overset{\diamond}{\rightarrowtail} n'' \in \overset{\curlyvee}{\tau}'$.

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

In Figure 6, 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.

After raising, the binding edge $n' \rightarrowtail n''$ may sometimes be deleted, namely when n was the only node bound at n'. This is the case for the edge $\langle 22 \rangle \rightarrowtail \langle 2 \rangle$ of τ_3, which we removed.

3.2.4 Weakening

Weakening is used to forbid irreversible instance operations to occur underneath a node. It turns a flexible binding edge leaving a flexible node into a rigid one, as illustrated on the right side of Figure 10.

DEFINITION 9 (*Weakening*). A type τ' is a weakening at a flexible 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 $\tau \mapsto \tau'$ and $\sqsubseteq^W$ for the reflexive transitive closure of the relation defined by $\tau \mathrel{\mathcal{R}} \tau' \iff \exists n, \tau' = \mathsf{Weaken}(n)(\tau)$. □

In Figure 6, type τ_6 is a weakening of τ_5 at node $\langle 21 \rangle$.

3.3 The instance relation

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

DEFINITION 10 (*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 $\tau_1 \sqsubseteq^G \tau_2 \sqsubseteq^R \tau_3 \sqsubseteq^R \tau_4 \sqsubseteq^M \tau_5 \sqsubseteq^W \tau_6 \sqsubseteq^M \tau_7$ holds. Hence, $\tau_1 \sqsubseteq \tau_7$ holds by definition of $\sqsubseteq$; note that a shortened decomposition of this fact is $\tau_1 \sqsubseteq^G \tau_6 \sqsubseteq^M \tau_7$. 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.

Notice that the graphic instance relation needs not to be defined under prefixes, as it uses permission to deeply operate inside terms instead. We believe that this makes the definition significantly simpler than its syntactic counterpart.

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).

PROPERTY 1. *The permission system $\mathcal{P}$ satisfies:*

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

Both properties are also verified by the variant of the permission system that matches syntactic type instance.

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 1.2). This enables to perform unification by computing the binding edges and their labeling independently (§ 4).

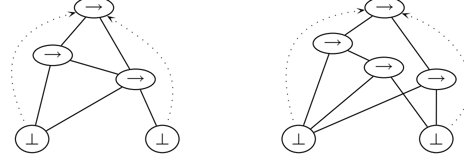


Figure 11. Two similar types.

3.4 Similarity

As defined, the instance relation is slightly too fine grained. While sharing of polymorphic nodes is significant, sharing of monomorphic nodes is semantically meaningless. Therefore, we define a similarity relation that abstracts over those details, and quotient the instance relation accordingly.

DEFINITION 11 (*Similarity*). We call *similarity* the relation $(\sqsubseteq^m \cup \sqsupseteq^m)^*$ on graphs, which is written $\approx$. We call *instance modulo similarity* the relation $(\sqsubseteq \cup \approx)^*$, written $\sqsubseteq_\approx$. □

An example of two similar types is given in Figure 11. The type on the left is obtained by merging the unbound nodes $\langle 12 \rangle$ and $\langle 2 \rangle$ of the one on the right. Hence, both types are similar. In the end, we work up to similarity and are interested in $\sqsubseteq_\approx$. However, we often express results for $\sqsubseteq$ alone, as they are stronger than the corresponding results for $\sqsubseteq_\approx$.

The similarity of two types can be checked efficiently, in linear time in the size of the graphs. The algorithm, presented in Figure 12, merely verifies that the structures of the two types unify without any change to their binding structure.

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 term-graph unifier of τ_1 and τ_2 (treating $\bot$ as a variable). Return false if it does not exist.
2. Return false if an equivalence class holds any of the following:
   (a) Two polymorphic nodes of the same graph,
   (b) A polymorphic node and a monomorphic node,
   (c) A bottom and a non-bottom node.
3. Return false if two polymorphic nodes in the same equivalence class do not have their respective binders in the same equivalence class, or if they do not carry the same flag on their binders.
4. Return true

Figure 12. Algorithm for similarity.

The similarity relation $\approx$ corresponds exactly to the equivalence relation $\equiv$ in the syntactic presentation. More precisely, equivalent syntactic types translate to similar graphic types, and, conversely, similar graphic types translate to equivalent syntactic types. However, graphic types are more canonical than syntactic ones. Hence, similarity is a simpler relation than equivalence. Indeed, similarity does not require prefixes or context rules. Moreover, removal of useless binders and commutation of binders are directly captured in the graphic representation: syntactic types that differ only by applications of those rules are mapped to *equal* graphic types.

In fact, similarity is the residual of equivalence on first-order term graphs: two types τ and τ' are similar if and only if their

underlying term-graphs are equivalent (*i.e.* the skeletons of their terms graphs are equal) and their binding trees are equal.

3.5 Commutation lemmas

In this section, we write $\mathcal{R} \,;\, \mathcal{R}'$ for $\mathcal{R}' \circ \mathcal{R}$, *i.e.* the composition of relations defined by $x\ (\mathcal{R} \,;\, \mathcal{R}')\ y \iff \exists z, x\ \mathcal{R}\ z \wedge z\ \mathcal{R}'\ y$.

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. This flexibility is the key to an efficient implementation of unification (§ 4). It also greatly simplifies reasoning and proofs on instance derivations.

THEOREM 1 (Ordered derivations). *The instance relation* $\sqsubseteq$ *is equal to* $\sqsubseteq^G \,;\, \sqsubseteq^R \,;\, \sqsubseteq^{MW}$*, with* $\sqsubseteq^{MW} = (\sqsubseteq^M \cup \sqsubseteq^W)^\star$. □

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 $\tau_1 \sqsubseteq \tau_7$ in Figure 6.

The similarity and instance modulo similarity relations can also be decomposed. In particular, all usages of monomorphic unsharings can be pushed to the end of a derivation. This is used to prove that unification and similarity commute.

LEMMA 1. *The similarity relation* $\approx$ *is equal to* $\sqsubseteq^m \,;\, \sqsupseteq^m$*. The instance modulo similarity relation* $\sqsubseteq_\approx$ *is equal to* $\sqsubseteq \,;\, \sqsupseteq^m$. □

3.6 The abstraction relation

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. Intuitively, abstraction allows to perform only instance operations on rigid nodes.

The abstraction operations are sketched in Figure 13 and detailed in the definition below.

DEFINITION 12 (*Abstraction*). The graphic abstraction relation on types, written $\sqsubseteq$, is the subrelation $(\sqsubseteq^M \cup \sqsubseteq^R)^*$ of $\sqsubseteq$ where $\sqsubseteq^M$ is the subrelation of $\sqsubseteq^M$ that only merges rigid nodes and $\sqsubseteq^R$ is the subrelation of $\sqsubseteq^R$ that only raises nodes bound on rigid nodes. □

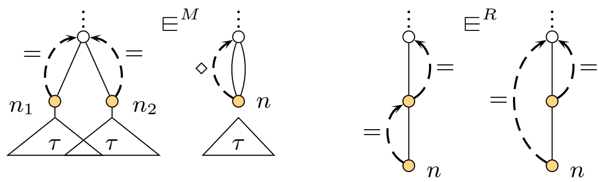


Figure 13. Abstraction operations.

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.

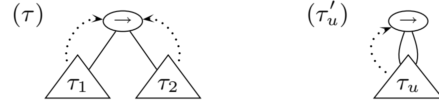
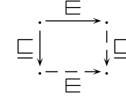


Figure 14. Encoding for standard unification.

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 $(\sqsubseteq \cup \sqsupseteq)^*$, which we write $\boxminus$. That is, we may replace type instance by the larger relation $(\sqsubseteq \cup \boxminus)^*$ [8], which we write $\sqsubseteq^{\boxminus}$. 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 $\sqsubseteq; \sqsupseteq$ and $\sqsubseteq; \sqsupseteq$, respectively. This also implies that the unification problem for $\sqsubseteq^{\boxminus}$ can be reduced to the unification problem for $\sqsubseteq$, which we solve in the next section[1].

4. Unification

This section presents the unification problem for $\mathsf{ML^F}$ types and an efficient algorithm to solve it.

4.1 Unification problem

The unification problem for $\mathsf{ML^F}$ 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 $\mathsf{ML^F}$ 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 13 (*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. Indeed, two types τ_1 and τ_2 unify if the nodes $\langle 1 \rangle$ and $\langle 2 \rangle$ unify in the type τ of Figure 14; if this is the case, an unifier of τ_1 and τ_2 is the subgraph of τ'_u starting at node $\langle 1 \rangle$. The reciprocal implication also holds.

Unfortunately, generalized unification is in fact *too* powerful, as some problems can have a non-principal set of solutions—and are in fact not useful in practice. Consider unifying the nodes $\langle 11 \rangle$ and $\langle 21 \rangle$ in the type τ of Figure 15. 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 also 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

[1] However, it would be misleading to think that solving unification for $\sqsubseteq^{\boxminus}$ enables more aggressive type inference: on the opposite, 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.

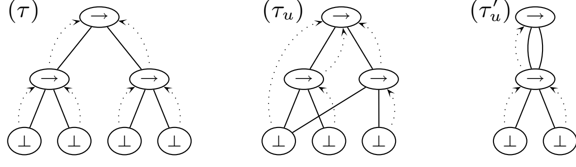


Figure 15. A problem without a principal solution.

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.

4.2 Admissible problems

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 14), 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 14. 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 $\{n'' \in \tau \mid \exists n \in N, n' \in \tau, n' \rightarrowtail n'' \overset{+}{\circ\!\!\longrightarrow} n \overset{*}{\circ\!\!\longrightarrow} n'\}$ is totally ordered by the domination relation $\lhd$ on $\mathring{\tau}$. □

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 15). 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 15, 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 $\lhd$ in $\mathring{\tau}$ or $\mathring{\tau}'$.

As mentioned above, admissible problems subsume unification under the root. In fact, any set of nodes "under" a given node for $\circ\!\!\longrightarrow$ or $\rightarrowtail$ is admissible. Moreover, those problems have useful stability properties.

PROPERTY 2. *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 $\{ n' \mid n' \rightarrowtail n \}$ is admissible for τ.*

Suppose N is admissible for τ. Then for any type τ' such that $\tau \sqsubseteq_{\approx} \tau'$, N is admissible for τ'. □

4.3 Unification algorithm

We present our unification algorithm Unif_N in Figure 16. 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.

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. The second phase uses an auxiliary algorithm Rebind (presented in Figure 17) to build the binding tree of the unifier. Given a type τ and a term-graph $\mathring{\tau}_u$ instance of $\mathring{\tau}$ (defined in Appendix A), it returns a binding-tree $\overset{\succeq}{\tau}_u$ such that $(\mathring{\tau}_u, \overset{\succeq}{\tau}_u)$ is an instance of τ, or fails.

Let us introduce some notations. 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 three steps.

1. Correction of the grafting steps. The first step checks that the graftings performed to obtain the skeleton of τ_u from the skeleton of τ are allowed w.r.t. permissions.

2. Building the binding tree. The second phase binds the nodes of $\mathring{\tau}_u$. Given a node n, it first finds the set M_n of the bound 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 the same node (step 2(d)iii)[2]. In parallel, a new flag $\diamond_n$ is computed for n; it is the best flag common to the nodes of M_n (step 2(d)i). Steps 2(d)ii and 2(d)iv verify that the weakenings and raisings that have been performed respect the permissions of τ.

The computation of $\overset{\succeq}{\tau}_u$ is incremental and is done in a top-down fashion: results found for the nodes that have already been considered are reused for the nodes underneath. The algorithm is conservative and may compute binders for nodes whose binding edges will eventually be deleted by $\mathsf{gc}(\tau_u)$.

3. Correction of the Merge steps. The third phase revisits the mergings performed between $\mathring{\tau}$ and $\mathring{\tau}_u$. Some of them were polymorphic, according to the binding tree found in phase 2. All of these are verified for permissions (during step 3b).

The difficulty of this step lies in finding where exactly the mergings originated. Consider the type τ_6 in Figure 6. 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 fact: when two nodes are merged and their binders are equal, they are the root of a polymorphic merge. Step 3b finds the nodes of τ_u that verify this condition.

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[3] by the algorithm, they are useful to reason on it.

- The graph τ_g is τ in which all the graftings have been performed. Let S_G be the set of bottom nodes of τ that are no longer bottom nodes in τ_u. Then τ_g is obtained by simultaneously grafting under every node m of S_G the expansion of the subgraph of τ_u under $\tilde{m}$ (defined below).

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

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

Figure 16. Unif_N algorithm.

[2] We defer the discussion on B_2^n to (§ 4.4).

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

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

1. **Correction of the graft steps**

   Fail if there exists a non flexible bottom node m in τ such that $\tilde{m}$ is not a bottom node in $\mathring{\tau}_u$.

2. **Building the binding tree**

   For each node n of $\mathring{\tau}_u$ (visited in a top-down ordering), do:

   (a) Let M_n be $\{m \in \tau \mid \tilde{m} = n\} \cap \mathsf{dom}(\overset{\succ}{\tau})$.

   (b) Let B_1^n be $\left\{ \overset{\succ}{\tau}(m) \,\middle|\, m \in M_n \right\}$.

   (c) Let B_2^n be $\begin{cases} \{ n' \mid n' \circ\!\!\longrightarrow n \} & \text{if } n \text{ is partially grafted} \\ \emptyset & \text{otherwise} \end{cases}$

   (d) If either B_1^n or B_2^n is not empty:

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

      ii. Fail if $\diamond_n$ is $(=)$ and there exists a non flexible node m in M_n such that $\overset{\diamond}{\tau}(m)$ is $(\geqslant)$.

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

      iv. Fail if there exists m in M_n locked in τ and such that $\tilde{m}$ is not n_B.

      v. Let $\overset{\succ}{\tau}_u$ be $\overset{\succ}{\tau}_u + n \xrightarrow{\diamond_n} n_B$.

   Let $(_, \overset{\succ}{\tau}_u)$ be $\mathsf{gc}(\mathring{\tau}_u, \overset{\succ}{\tau}_u)$.

3. **Correction of the Merge steps**

   (a) Build the graph $\tau_\uparrow$ such that $\mathring{\tau}_\uparrow$ equals $\mathring{\tau}$ and verifying $m \rightarrowtail m' \in \tau_\uparrow \iff m \overset{\star}{\rightarrowtail} m' \in \tau \wedge \tilde{m} \rightarrowtail \tilde{m}' \in \overset{\succ}{\tau}_u$.

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

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

Figure 17. Rebind algorithm.

The expansion of a type τ is the only type τ' whose both term-graph and binding tree are equal to the skeleton of τ and whose nodes are all flexibly bound. For example, the expansion of the subgraph at node $\langle 1\rangle$ of τ_2 in Figure 6 is the subgraph at $\langle 2\rangle$.

- 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: $m \overset{\diamond}{\rightarrowtail} m' \in \tau_r$ if and only if $\tilde{m} \rightarrowtail \tilde{m}' \in \tau_u$ and $m \overset{\diamond\overline{\diamond}}{\rightarrowtail} m' \in \tau_g$.

4.4 Example of unification

Our running example will be Figure 6, 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. Indeed, in our case $S_G = \{2\}$, and τ_2 grafts the proper expansion subtype at $\langle 2\rangle$. 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.

We now examine each of the steps of Rebind in turn.

Step 1 We check that $\langle 2\rangle$ (the only node of S_G) can be grafted. This is the case here, as it is flexibly bound to the root in τ_1.

Step 2 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.* $(\geqslant)$. Here, no weakening takes place, hence no verification is done. Note that it is not necessary to take into account the flags on the nodes of $M'_n \setminus M_n$, as we know they are flexibly bound.

The computation of 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 start by computing the binders of the nodes of M'_n:

- B_1^n contains the binders of the nodes present in τ (*i.e.* τ_1).
- 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, $B_1^n = \{\overset{\succ}{\tau}_1(n)\} = \{\langle 1\rangle\}$. Meanwhile, $B_2^n = \{\langle 22\rangle\}$, which is exactly the (common) binder in τ_g of the nodes $\langle 221\rangle$ and $\langle 221\rangle$ of $M'_n \setminus M_n$ (that are grafted in τ_g and merged in 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 $\mathring{\tau}_u$, is already (flexibly) bound to $\langle 2\rangle$. This last node is equal to $\langle 1\rangle$ in τ_u, hence $\mathsf{LCA}_{\overset{\succ}{\tau}_u}(B_1^n \cup B_2^n)$ is equal to $\langle 1\rangle$. The nodes that need to be raised are $\langle 221\rangle$ and $\langle 222\rangle$, the grafted ones. Again, since they are flexible, no permissions check is needed.

Step 3 In our simple example, $\tau_\uparrow$ is in fact equal to τ (*i.e.* τ_1). The only pair of nodes satisfying condition 3b is $(\langle 1\rangle, \langle 2\rangle)$, for which the permission check succeeds. Note that while the nodes $\langle 211\rangle$ and $\langle 212\rangle$ were merged in our derivation of Figure 6, the algorithm does not check for them, as again it knows that they are flexible (the same for $\langle 221\rangle$ and $\langle 222\rangle$).

4.5 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 $\mathsf{ML^F}$. The first three lemmas are important auxiliary results for the proofs.

Rebind must temporarily bind nodes that cannot be polymorphic in the final result (none of the instance rules allow transforming a monomorphic node into a polymorphic one), in order to be efficient and incremental when building $\overset{\succ}{\tau}_u$. Nevertheless, it does not "invent" polymorphism:

LEMMA 2. *Given a polymorphic node n of τ_u, any node m that is merged into n was polymorphic in τ, τ_g and τ_r.* □

Rebind chooses the lowest possible binder for a node:

LEMMA 3. *Let n be a polymorphic 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 $m \overset{+}{\rightarrowtail} m' \in \tau_g$. Then, $n \overset{+}{\rightarrowtail} n' \in \tau_u$.* □

Rebind preserves existing permissions:

LEMMA 4. *Let n be a polymorphic 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.* □

For the remainder of this section, we implicitly quantify over a type τ, a set of nodes N, and a first-order instance $\mathring{\tau}_u$ of $\mathring{\tau}$. Unless mentioned otherwise, we do *not* assume that (τ, N) is admissible. We do not assume that $\mathring{\tau}_u$ is the principal first-order unifier of N in $\mathring{\tau}$ either. The results are given first for Rebind, then for Unif.

We start by stating the soundness result.

THEOREM 2. *If* $\mathsf{Rebind}(\tau, \mathring{\tau}_u)$ *returns* $\overset{\succ}{\tau}_u$*, the instance relations* $\tau \sqsubseteq^G \tau_g \sqsubseteq^R \tau_r \sqsubseteq^{MW} (\mathring{\tau}_u, \overset{\succ}{\tau}_u)$ *hold.* □

Thus Unif is sound even on non admissible problems.

COROLLARY 1 (Soundness). *The algorithm* Unif *is sound.* □

Rebind is also complete.

THEOREM 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* $\overset{\succsim}{\tau}_u$ *such that the type* τ_u *equal to* $(\mathring{\tau}_u, \overset{\succsim}{\tau}_u)$ *is more general than* τ_v, i.e. $\tau_u \sqsubseteq \tau_v$. □

For the existence result we show that, for any permissions check done by the algorithm, any derivations of $\tau \sqsubseteq \tau_v$ uses a transformation requiring at least those permissions. For the principality result, we consider an ordered derivation $\tau \sqsubseteq^G \tau_g' \sqsubseteq^R \tau_r'' \sqsubseteq^{MW} \tau_v$ of $\tau \sqsubseteq \tau_v$, and show (using a commutative diagram) that $\tau_g \sqsubseteq \tau_g'$, $\tau_r \sqsubseteq \tau_r''$ and $\tau_u \sqsubseteq \tau_v$.

This result is not sufficient to prove completeness with a different structure graph. However, completeness holds on admissible problems.

THEOREM 4 (Completeness). *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. □

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

LEMMA 5. *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* $\mathsf{Unif}_N(\tau_1) \approx \mathsf{Unif}_N(\tau_2)$. □

4.6 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 5. 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.

The algorithm Rebind can be improved so that it does not need to visit the whole type during unification, but only the nodes that are visited during the first-order unification phase. This way, it can be used incrementally with a good complexity. However, those improvements are quite technical [13].

While the complexity bound of the algorithm used in the original syntactic presentation of $\mathsf{ML^F}$ 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.

Conclusion

We have given a formal meaning to the informal graphic types used in the original presentation of $\mathsf{ML^F}$ [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 $\mathsf{ML^F}$-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 $\mathsf{ML^F}$ types. The language $\mathsf{ML^F}$ 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 $\mathsf{ML^F}$ 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 $\mathsf{ML^F}$ 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.

Future works A continuation of this work is to revisit type inference for $\mathsf{ML^F}$ 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 $\mathsf{ML^F}$, to verify that type inference remains indeed tractable, just as in ML.

By simplifying and increasing our understanding of $\mathsf{ML^F}$ types, the graphic presentation also permits exploring several extensions. This includes generalized algebraic data types, subtyping, primitive existential types, recursive types, or higher-order 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 $\mathsf{ML^F}$ to higher-order types. The interaction of β-reduction at the level of types with a first-order type inference *a la* $\mathsf{ML^F}$ seems non-trivial.

Acknowledgments

We would like to thank the anonymous referees for numerous helpful comments, Yann Régis-Gianas for close readings of early versions of this paper, 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 $\mathsf{ML^F}$, 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] 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.

[13] Didier Rémy and Boris Yakobowski. A graphical presentation of ML^F types with a linear-time incremental unification algorithm. Extended version, of [12], January 2008.

A. An introduction to term-graphs

Term graphs are a more compact representation of first-order terms, often used in unification algorithms.

A.1 First-order terms

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 $1 \leq k \leq \mathsf{arity}(t(\pi))$.

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 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, and reinterpret the resulting graphs as terms.

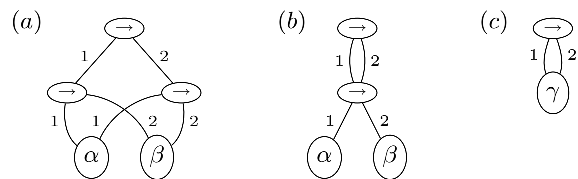


Figure 18. Instance on term-graphs.

A.2 Term-graphs

Term-graphs formalize the dag-based representation of first-order terms.

An equivalence relation $\sim$ on the paths of a term t is *consistent* if t is constant on every equivalence class (each path in a class maps to the same symbol or variable). It is *weakly consistent* if there is at most one symbol of Σ in every equivalence class (each class can contain multiple variables, but at most one symbol). An equivalence relation $\sim$ on 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 $\pi k \sim \pi' k$. Congruences identify identical subterms.

A *term-graph* g is a pair of a term $\hat{g}$ and a consistent congruence $\tilde{g}$ on $\mathsf{dom}(\hat{g})$ such that every variable appears in at most one equivalence class.

In Figure 18, the term of both graphs (a) and (b) is $(\alpha \to \beta) \to (\alpha \to \beta)$. However, their equivalence classes differ. For graph (a), it is $\tilde{a} = \{\langle 11\rangle, \langle 21\rangle\}, \{\langle 12\rangle, \langle 22\rangle\}$, while it is $\tilde{a} \cup \{\langle 1\rangle, \langle 2\rangle\}$ for graph (b).

A.3 Instance and unification on term-graphs

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 $\tilde{g} \subseteq \tilde{g}'$. Two term-graphs are *similar* if they represent the same tree. For instance, graphs (a) and (b) of Figure 18 are similar (as both represent $(\alpha \to \beta) \to (\alpha \to \beta)$) while (b) and (c) are not. On the other hand, (b) is a standard instance of (c).

Instance on term-graphs implies instance of the underlying terms. As an example, coming back to (b) and (c), $(\alpha \to \beta) \to (\alpha \to \beta)$ is indeed an instance of $\gamma \to \gamma$ through the substitution $\gamma \mapsto (\alpha \to \beta)$.

Unification can be *internalized* on term-graphs, that is, defined by giving two nodes of a same graph instead of two graphs to be merged. We say that a term-graph g' is a unifier of nodes n_1 and n_2 of a term-graph g if it is an instance of g that identifies nodes n_1 and n_2 (*i.e.* there exists a node n of g' that is a superset of both n_1 and n_2). For example, in Figure 18, the term-graph (b) is a unifier of the nodes $\{1\}$ and $\{2\}$ in the term-graph (a). A unifier g' of nodes n and n' is *principal* if any other unifier is an instance of g'. Unification of two nodes n and n' of g can be computed as the smallest weakly consistent, congruent equivalence that contains $\tilde{g}$ and merges n and n' [5].

In fact, 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 the nodes $\langle 1\rangle$ and $\langle 2\rangle$ of a term-graph g, then $\hat{g}'/1$ (the subterm of $\hat{g}'$ at occurrence 1), also equal to $\hat{g}'/2$, is a (principal) unifier of $\hat{g}/1$ and $\hat{g}/2$. 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.