THÈSE

présentée à

l'Université Paris 7 – Denis Diderot

pour obtenir le titre de

Docteur en Informatique

Types et contraintes graphiques : polymorphisme de second ordre et inférence

soutenue par

BORIS YAKOBOWSKI

le 17 Décembre 2008

JURY

Président	Roberto Di Cosmo
Rapporteurs	Stéphanie Weirich
	Hugo Herbelin
Examinateurs	Fritz Henglein
	Alexandre Miquel
Directeur	Didier Rémy

Remerciements

Ce travail n'aurait pas vu le jour sans l'aide, le concours et le soutien de nombreuses personnes, que je tiens à remercier ici.

En premier lieu, merci à Didier pour avoir encadré cette thèse, tout particulièrement pour sa disponibilité sans faille pendant ces 4 années. En plus de ses conseils scientifiques, j'aurai également largement bénéficié de son expertise TEX, me permettant d'atteindre le niveau enviable (?) de «TEX utilisateur frustré mais capable».

Mes deux rapporteurs, Stephanie Weirich et Hugo Herbelin, ont eu la lourde tâche de relire mon imposant manuscrit ; qu'ils soient ici remerciés pour leur intérêt et leur persévérance. Merci en particulier à Stephanie pour ses nombreuses remarques, toujours d'une grande pertinence. Merci également à Fritz Henglein et Alexandre Miquel pour m'avoir fait l'honneur de s'intéresser à mon travail. En plus de présider mon jury, Roberto Di Cosmo m'aura initié à la recherche alors que je n'étais encore qu'un jeune padawan de Licence. Pour tout cela, et bien d'autres choses encore, je lui suis à jamais reconnaissant. Enfin, cette thèse aura également été l'occasion de discussions scientifiques très enrichissantes. Merci donc à Didier Le Botlan, Daan Leijen et Dimitrios Vytiniotis pour toutes leurs remarques et suggestions sur mon travail.

Le projet CRISTAL, qui s'est sublimé en GALLIUM pendant ma thèse, est un environnement de travail d'une richesse scientifique exceptionnelle. Qu'il me soit donc permis de remercier ici tous ceux qui ont contribué à le faire vivre pendant ces 4 années. En particulier, merci à Xavier Leroy pour nous avoir fait bénéficier de son savoir aussi encyclopédique qu'éclectique, et à Sandrine Blazy, Damien Doligez, Alain Frisch et Michel Mauny pour leur bonne humeur contagieuse. Merci également à Jacques Garrigue et François Pottier pour leurs remarques sur mon travail ; la Définition 4.3.3, qui aurait dû être le bien moins élégant Lemme 4.3.4, doit par exemple beaucoup à François. GALLIUM serait incomplet sans notre projet frère MOSCOVA ; merci à eux. Enfin, séparés de nous uniquement par la distance, merci infiniment à Daniel Hirschkoff et Yves Bertot pour m'avoir fait découvrir et aimer les théories de la programmation et de la preuve.

Primus inter pares parmi nos petits condisciples, merci à Yann pour nos interminables discussions sur le typage et tant d'autres sujets, ainsi que pour m'avoir supporté comme cobureau. Ma thèse aurait été fort différente sans lui. De même, merci à Zaynah pour avoir été là toutes ces années. Même si la teneur en typage de nos discussions aura été bien moindre, elles auront été tout aussi enrichissantes. Un autre immense merci à mes compagnons de 4ème année Jade et Benoît M. . Enfin, je me dois de remercier Benoît R. (deuxième cobureau, et premier Benoît chronologiquement !), ainsi que Nico et J.B. pour toutes nos discussions dans mon bureau ou au coin café/thé.

Mon arrivée à PPS aura été l'occasion d'un véritable renouveau. En tout premier lieu, je remercie Vincent qui m'aura donné l'opportunité fantastique de travailler sur Ocsigen. Merci à Sam pour ses blagues nulles, mais aussi pour avoir défriché le labyrinthe des semaines précédant la soutenance. Merci également à Grégoire pour toutes nos discussions, à Gim, aux amis et collègues du LIAFA (dont Claire et Julien) et des autres laboratoires parisiens, en particulier Pierre, Aurélien, Mathieu, Matthieu et David. Plus généralement, merci à tous ceux avec qui j'ai eu la chance d'interagir ces 3 derniers mois.

Merci à ma famille, tout particulièrement mes parents et ma soeur. Même si je les ai peu vus ces dernières années, ils ont toujours été là pour moi, et j'espère qu'ils continueront à l'être longtemps. (Et j'espère avoir la possibilité de les voir plus souvent !)

Enfin, et surtout, merci à tous mes amis, pour m'avoir aidé à sortir de mon λ-monde, et avoir bien voulu me faire partager le leur. Merci donc à Anne-Laure, François, Caro, Gilles, Mathieu[1], Jeremy, Sébastien, Jordan, Luc et Benoît[2]. Plus généralement, merci à tous les MIM01 et MIM02, les nanars-clubiens, les Ulmiens non amateurs de nanars, les Anciens Martins, ainsi que tous ceux que j'oublie mais qui se reconnaîtront. Merci pour tout, et bien plus encore.

[1]Un troisième !
[2]Cf. note 1.

Contents

1

Notations and conventions

1.1 Conventions

In this document, we distinguish four kinds of formal results: lemmas, properties, corollaries, and theorems. A *lemma* states a simple result, and is usually used to show other results. A *corollary* is a direct consequence of the previous results. A *theorem* is a fundamental result of this document. A *property* is a simple—but often used—result that we implicitly use inside proofs. Results and definitions are numbered with respect to the current section.

1.2 Mathematical notations

The symbol $\triangleq$ is used to give the formal definition of an object, and means "is equal by definition to". The symbol $\overset{\rightarrow}{=}$ signifies that the left-hand side rewrites to the right-hand one, but the converse might not be true in general (because some side-conditions are missing on the right-hand side).

We write logical conjunctions and disjunctions on multiple lines as shown below

$$\wedge\begin{cases} A \\ B \end{cases} \quad \triangleq \quad A \wedge B \qquad\qquad \vee\left[\begin{array}{l} A \\ B \end{array}\right. \quad \triangleq \quad A \vee B$$

We write $|A|$ the cardinal of a set A, $A \times B$ the cartesian product of A and B, and $A \# B$ the fact that A and B are disjoint (*i.e.* $A \cap B$ is empty). If a is a meta-variable ranging over some set A, we write $\overline{a}$ for an ordered sequence of elements of A. Given a function f, $\mathsf{dom}(f)$ and $\mathsf{codom}(f)$ are respectively its domain and codomain.

1.3 Relations

In this document, a binary relation $\mathcal{R}$ over a set S is often seen as a set of pairs of S, and we write $x \mathrel{\mathcal{R}} y$ for $(x, y) \in \mathcal{R}$. A function f can be seen as the binary relation $\mathcal{R}_f$ verifying

$$x \mathrel{\mathcal{R}_f} y \iff y = f(x)$$

We often view relations as (potentially non-deterministic) rewriting systems. Consequently, we write $f\,;g$ for the inverse composition $g \circ f$. The semicolon notation emphasizes the order in which the rewritings are done. Similarly, given two relations, we write $\mathcal{R}\,;\mathcal{R}'$ for the composition of relations defined by

$$x\,(\mathcal{R}\,;\mathcal{R}')\,y \iff \exists z,\ x\,\mathcal{R}\,z \wedge z\,\mathcal{R}'\,y$$

Given a relation $\mathcal{R}$, we write $\mathcal{R}^{-1}$ for its symmetric relation, $\mathcal{R}^{+}$ its transitive closure and $\mathcal{R}^{*}$ its reflexive transitive closure. The kernel of $\mathcal{R}$ is the relation $\mathcal{R} \cap \mathcal{R}^{-1}$. We also use $>$ for $<^{-1}$ when $<$ is a relation symbol with a symmetric symbol. Finally, given two relations $\mathcal{R}_1$ and $\mathcal{R}_2$, we write $\mathcal{R}_1 \odot \mathcal{R}_2$ the relation $(\mathcal{R}_1 \cup \mathcal{R}_2)^*$

1.4 Graphs

Let G be an arbitrary directed graph with nodes N and edges E labeled in L, *i.e.* $E \subseteq N \times L \times N$. We write

$$n_1 \xrightarrow{l} n_2 \in G \quad \text{for} \quad (n_1, l, n_2) \in E$$

Often, G may be left implicit and we simply write $n_1 \xrightarrow{l} n_2$. We may also fix a label $l \in L$ and see $\xrightarrow{l}$ as the binary relation $\{(n_1, n_2) \mid (n_1 \xrightarrow{l} n_2)\}$.

Fixing one side of the arrow to a particular set of nodes S, we write $(S \longrightarrow)$ and $(\longrightarrow S')$ for the set of nodes reached from a node in S, and reaching a node in S' respectively:

$$(S \longrightarrow) \ \triangleq\ \{n' \mid \exists n \in S, n \longrightarrow n'\} \qquad (\longrightarrow S') \ \triangleq\ \{n \mid \exists n' \in S', n \longrightarrow n'\}$$

If $\bar{l}$ is a string of labels $l_1 \ldots l_k$, we write $n_1 \xrightarrow{\bar{l}} n_k$ for $n_1 \xrightarrow{l_1} \ldots n_{k-1} \xrightarrow{l_{k-1}} n_k$. We also write $n \xrightarrow{*} n'$ if there exists a string of labels $\bar{l}$ such that $n \xrightarrow{\bar{l}} n'$, and $n \xrightarrow{+} n'$ if this string is non-empty.

1.4.1 Directed acyclic graph

Given a directed graph G over a set N, we say that G is a *directed acyclic graph*, abbreviated as *dag*, if no element n of N is such that $n \xrightarrow{+} n$.

1.4.2 Domination

Given a graph G over a set N, we say that G is *rooted* if there exists an element r of N such that all the elements of N are accessible from r: $\forall n, r \xrightarrow{*} n$.

Given two nodes n and n' of a rooted graph G, we say that n *dominates* n', written $n \longrightarrow\!\!\!\gg\!\!\longrightarrow n'$, if for any sequence $n_0 \longrightarrow n_1 \ldots \longrightarrow n_k$ with $n_0 = r$ and $n_k = n'$, there exists i such that $n_i = n$. Intuitively, all the paths from the root to n' contain n. The domination relation is a partial order over the nodes of N. Moreover, for any three nodes n_1, n_2 and n_3, if $n_1 \longrightarrow\!\!\!\gg\!\!\longrightarrow n_3$ and $n_2 \longrightarrow\!\!\!\gg\!\!\longrightarrow n_3$, either $n_1 \longrightarrow\!\!\!\gg\!\!\longrightarrow n_2$ or $n_2 \longrightarrow\!\!\!\gg\!\!\longrightarrow n_1$.

1.5 Types

We assume the existence on an unspecified algebra of type constructors Σ, containing at least the arrow constructor $\rightarrow$. In the examples we will sometimes use type constructors such as int or list. Each constructor C comes with its *arity*, written $\mathsf{arity}(C)$; the arrow constructor has arity 2. The meta-variable C ranges over Σ.

First-order types, ML type schemes and second-order types are defined by the following grammar:

$$\begin{array}{rcll} t & ::= & \alpha \mid C\,\overline{t} & \text{First-order types} \\ \tau & ::= & t \mid \forall\alpha.\ \tau & \mathsf{ML}\text{ type schemes} \\ \sigma & ::= & \alpha \mid C\,\overline{\sigma} \mid \forall\alpha.\ \sigma & \text{System }\mathsf{F}\text{ types} \end{array}$$

A first-order type is either a type variable α, or the application of a type constructor respecting the arity of the constructor. ML type schemes only allow prenex quantification *i.e.* at the front of the type. System F types are more general and allow type quantification everywhere.

In the following, the metavariables α, β, γ and δ range over a denumerable set of type variables. As usual, $\forall$ binds to the right as far as possible, and the arrow constructor associates to the right. That is,

$$\forall\alpha.\ \forall\beta.\ (\alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \forall\gamma.\ \gamma \rightarrow \gamma \qquad \text{is} \qquad \forall\alpha.\ (\forall\beta.\ ((\alpha \rightarrow \alpha) \rightarrow (\beta \rightarrow \forall\gamma.\ (\gamma \rightarrow \gamma))))$$

The free variables of a type are written ftv.

1.6 Expressions

We reason on the expressions of the λ-calculus enriched with let constructions.

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

The metavariables x and y range over a denumerable set of variables. The expressions $\lambda(x)\ a$ and $\mathsf{let}\ x = a'\ \mathsf{in}\ a$ bind x in a but not in a'. The simultaneous capture-avoiding substitution of a sequence of variable $\overline{x}$ by a sequence of expressions $\overline{e}$ inside an expression e' is written $e'[\overline{e}/\overline{x}]$.

In some cases, we annotate expressions or λ-bound variables with types, resulting in the grammar

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

Type annotations will be defined precisely in §12.3.2.

2 Introduction

2.1 Types in functional languages

Types are a key part of the design of statically typed functional languages such as ML (Milner 1978) or Haskell (Peyton Jones 2003). One of the reasons of the success of these languages is undoubtedly type inference, which relieves the programmer from the burden of writing the types of the variables of the program. This facilitates rapid prototyping and code maintenance.

Both ML and Haskell are based at their core on the Damas-Milner type system (Damas and Milner 1982). In this system, type inference is decidable, and *principal*: a program can be assigned a type that is more general than all its all other possible types. This is a very desirable property, as the compiler never needs to make arbitrary choices during type inference. Moreover, type inference is total: the programmer never needs to write types to make his program typecheck.

Another key part of the success of the Damas-Milner system is the possibility to write *polymorphic* functions, that can be applied to arguments of different types. For example, a function computing the length of a list would receive type

$$\forall\alpha.\ \alpha\,\mathsf{list} \to \mathsf{int}$$

Such a function can be applied to a list of any type. That is, while lists are required to be homogeneous (*i.e.* contain only one type of element), we can compute the length of lists of integers, of binary trees, of functions...

Using the fact that functions are first-class in functional languages, we can also write so-called iterators, such as the ubiquitous map function over lists

$$\forall\alpha.\ \forall\beta.\ (\alpha \to \beta) \to \alpha\,\mathsf{list} \to \beta\,\mathsf{list}$$

This function uses its first argument f to convert each element e in the list to $f\ e$.

The form of polymorphism offered by the Damas-Milner type system is somewhat weak, as type quantification can only appear at the front of the type; the quantification is said to be prenex. For example, we cannot write a function that takes an iterator over lists such as the function map above. Such a function would have type

$$(\forall\alpha.\ \forall\beta.\ (\alpha \to \beta) \to \alpha\,\mathsf{list} \to \beta\,\mathsf{list}) \to \ldots$$

This type is not permitted, as α and β are introduced under an arrow constructor instead of at the beginning of the type.

During the last 25 years, the system proposed by Damas and Milner has been a remarkable point of equilibrium in the design space of programming languages. While more expressive systems have been proposed, they often were too complicated, or had undecidable type inference, or were not a conservative extension of ML... As a result, while Damas-Milner has been enriched by many new constructions such as qualified types (Jones 1994), or generalized algebraic data types (Xi *et al.* 2003; Jones *et al.* 2006; Pottier and Régis-Gianas 2006), it still forms the core of our type systems.

Still, the form of polymorphism it offers is sometimes too limited. Peyton Jones *et al.* (2007, §2) provide a good survey on why second-order polymorphism can be needed. Let us just mention the possibility to write functions taking iterators as arguments (as shown above); generic programming, in which the compiler automatically generates some functions (such as a pretty-printer) for the objects of a certain type; or the ability to encode invariants, by embedding polymorphic arguments inside the datastructure.

From an expressivity point of view, we would like to obtain at least the same power as the second-order polymorphic λ-calculus, also called *System F* (Girard 1972; Reynolds 1974). In System F, polymorphism can appear everywhere, and

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

is a valid type. Unfortunately, System F has undecidable type inference (Wells 1994). Moreover, as shown by the next section, System F has poor properties as a programming language, in particular because it does not have principal types. As a result, over the years a considerable amount of effort has been devoted to finding a type system that combines the expressivity of System F with the convenience of (at least some) ML-style type inference. We give a summary of these works in §16.

2.2 Type inference and System F

We recall that the instance relation $\leqslant_{\mathsf{F}}$ in the implicit presentation of System F is defined by

$$\forall\overline{\alpha}.\ \sigma \ \leqslant_{\mathsf{F}}\ \forall\overline{\beta}.\ \sigma[\overline{\sigma'}/\overline{\alpha}] \qquad \overline{\beta} \mathrel{\#} \mathsf{ftv}(\forall\overline{\alpha}.\ \sigma)$$

This relation allows instantiating the type variables quantified at the head of the type, and generalizing on-the-fly the newly introduced type variables.

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

As an example, consider the expressions $\mathsf{choose}\ \mathsf{id}$, where choose and id are defined by

$$\begin{array}{rcll} \mathsf{id} & \triangleq & \lambda(x)\ x & :\ \forall\alpha.\ \alpha \to \alpha \\ \mathsf{choose} & \triangleq & \lambda(x)\ \lambda(y)\ \mathsf{if}\ \mathit{false}\ \mathsf{then}\ x\ \mathsf{else}\ y & :\ \forall\beta.\ \beta \to \beta \to \beta \end{array}$$

(In the following, we abbreviate $\forall\alpha.\ \alpha \to \alpha$ as σ_{id}.)

In System F, we can give $\mathsf{choose}\ \mathsf{id}$ the following types σ_1 and σ_2

$$\mathsf{choose}\ \mathsf{id} : \left\{ \begin{array}{ll} \forall\gamma.\ (\gamma \to \gamma) \to (\gamma \to \gamma) & (\sigma_1) \\ (\forall\alpha.\ \alpha \to \alpha) \to (\forall\alpha.\ \alpha \to \alpha) & (\sigma_2) \end{array} \right.$$

Those two types are incomparable for $\leqslant_{\mathsf{F}}$, as none is more general than the other. Indeed, the inner polymorphism of σ_2 cannot be recovered by instantiating σ_1. Conversely, up to useless quantification, σ_2 has no other instance by $\leqslant_{\mathsf{F}}$ than itself. The—crucial—information that the two instances of σ_{id} are linked, and that instantiating them together would be sound, has been lost. This shortcoming is inherent to using System F types, which cannot express that kind of dependency—hence the language $\mathsf{ML^F}$, described below.

2.3 $\mathsf{ML^F}$

This section briefly presents the $\mathsf{ML^F}$ language (Le Botlan and Rémy 2003), on which a large part of this work is based. However, we purposefully do not dig into details, as much of the material covered here will be presented using quite different approaches elsewhere in this document.

2.3.1 $\mathsf{ML^F}$

The $\mathsf{ML^F}$ language (Le Botlan and Rémy 2003; Le Botlan 2004) aims at smoothly combining the advantages of ML-style type inference with the expressiveness of System F second-order polymorphism. In $\mathsf{ML^F}$, terms are partially annotated. All functions that use their parameters in a polymorphic way—and only those—need an annotation. In particular, ML terms never require one. In fact, $\mathsf{ML^F}$ is a conservative extension of ML: all ML terms are typable in $\mathsf{ML^F}$. Moreover, the full power of first-class polymorphism is also available, as any System F term can be typed by using type annotations (containing second-order types). Still, as in ML, all typable expressions have principal types.

$\mathsf{ML^F}$ is a language with very good stability properties: the set of well-typed programs is invariant under a wide class of program transformations, including let-expansion, let-reduction, η-expansion of functional expressions, reordering of arguments, currying... Moreover, syntactic application receives no special treatment in typing rules: $a_1\ a_2$ is typable if and only if *apply* $a_1\ a_2$ is (*apply* being $\lambda(f)\ \lambda(x)\ f\ x$). Furthermore, since only lambda-bound arguments that are used polymorphically need an annotation, it is very easy for the user to predict where and which annotations to write. Finally, $\mathsf{ML^F}$ is an impredicative type system, which allows for example embedding polymorphism inside containers. Thus $(\forall\alpha.\ \alpha \to \alpha)\ \mathsf{list}$ is a valid type, quite different from the weaker $\forall\alpha.\ ((\alpha \to \alpha)\ \mathsf{list})$.

$\mathsf{ML^F}$ type inference is decidable. Moreover, it is also principal: every well-typed source program provided with some annotations has a principal type—*i.e.* one of which all other correct types are *instances*. Interestingly, the typing rules of $\mathsf{ML^F}$ are a simple generalization of the ones of ML, and are quite straightforward: the power of $\mathsf{ML^F}$ does not come from its typing rules, but from its types, which are described next.

2.3.2 Enriching the types of System F

$\mathsf{ML^F}$ achieves the results above, and overcomes the lack of principal types in System F, by going beyond System F types. We describe $\mathsf{ML^F}$ types below.

2.3.2.1 Flexible quantification

One solution to the lack of principal types in System F is to enrich the system with a new form of (bounded) quantification, so that choose id receives the type

$$\tau \quad \triangleq \quad \forall\,(\alpha \geqslant \sigma_{\mathsf{id}})\;\alpha \to \alpha$$

Unlike in $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$, the two occurrences of σ_{id} are linked in τ. Thus it is safe to instantiate σ_{id} in the type above, and the variable α is allowed to range over all the possible instances of its bound σ_{id}, as indicated by the sign $\geqslant$. We say it is *flexibly* bound. Of course, the two occurrences of α on both sides of the arrow must simultaneously pick the same instance: the weaker the argument, the weaker the result.

Afterwards, the type τ can be instantiated in the following ways:

1. We can decide that α can no longer be instantiated, and "freeze" its bound. Thus we recover the type $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$.
2. We can introduce a dummy quantification in front of τ, resulting in $\forall\,(\beta)\;\forall\,(\alpha \geqslant \sigma_{\mathsf{id}})$ $\alpha \to \alpha$, and decide that α is instantiated with $\beta \to \beta$—which is indeed an instance of σ_{id}. Then the bound of α can no longer be instantiated, and can safely be inlined. We thus recover the F type $\forall\,(\beta)\;(\beta \to \beta) \to (\beta \to \beta)$ of choose id

More generally, flexible quantification is used to postpone the moment at which the operation of taking an instance is applied. The idea is to keep types as polymorphic as possible, in order to be able to recover later—just by (implicit) instantiation—what they would have been if some part had been instantiated earlier.

2.3.2.2 Rigid quantification

Flexible quantification, while expressive, is not yet sufficient to encode all of System F. For example, consider the function

$$f \quad \triangleq \quad \lambda(x)\;(x\;1,\;x\;\text{'c'})$$

It is not typable in ML, as the variable x is used on two arguments with incompatible types, int and char. In System F, it can be given the type

$$\sigma_{\mathsf{id}} \to \mathsf{int} \times \mathsf{char}$$

However, it would be incorrect to give it the type

$$\forall\,(\alpha \geqslant \sigma_{\mathsf{id}})\;\alpha \to \mathsf{int} * \mathsf{char}$$

Indeed, this type could be instantiated into

$$(\mathsf{int} \to \mathsf{int}) \to \mathsf{int} * \mathsf{char}$$

which would erroneously allow the application of the successor function to a character.

For reasons related to type inference (and partially described in the next paragraph), we do not give to f the System F type above. Instead, $\mathsf{ML^F}$ uses another form of quantification, called hereafter *rigidly*-bounded quantification and written with an "=" sign. Then f is given the type[1]

$$\forall\,(\alpha = \sigma_{\mathsf{id}})\ \alpha \to \mathsf{int} * \mathsf{char}$$

Rigid quantification cannot be (significantly) weakened by instantiation. Hence, it appears when polymorphism is *required*, while flexible quantification is present when polymorphism is available.

Flexible versus rigid Flexible and rigid quantification are two forms of bounded quantification, and share the same syntax. However, there is a deep asymmetry between them:

- flexible quantification is used to obtain more expressive types, in order to have a system with principal types;
- on the contrary, rigid quantification is used to *restrict* the expressivity of types: in a way, the type $\sigma_{\mathsf{id}} \to \mathsf{int} * \mathsf{char}$ is more general than the type $\forall\,(\alpha = \sigma_{\mathsf{id}})\ \alpha \to \mathsf{int} * \mathsf{char}$. A system giving to the term f above the type $\sigma_{\mathsf{id}} \to \mathsf{int} * \mathsf{char}$ is described by Le Botlan and Rémy (2007), and forms the basis of the *implicit* presentation of $\mathsf{ML^F}$, in which type annotations are never needed. However this system is more expressive than System F, and thus cannot be used to perform type inference—hence the introduction of rigid quantification.

This question is detailed further in §5.1.

2.3.3 Syntactic $\mathsf{ML^F}$ types

This section briefly presents the formal definition of $\mathsf{ML^F}$ syntactic types, as it makes it easier to refer to the original syntactic definition later on.

$\mathsf{ML^F}$ types are second-order types, but use the two forms of bounded quantification described in the previous section

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

A syntactic second-order type σ is a first-order type t, a bottom type $\bot$ (which stands for the System F type $\forall\alpha.\ \alpha$), or a quantified type $\forall\,(\alpha \diamond \sigma)\ \sigma'$. Unlike in System F, variables are always given bounds (that are themselves second-order types) to range over. Bounds are called rigid when introduced by the $=$ flag, and flexible when introduced by $\geqslant$.

[1]More precisely, in $\mathsf{ML^F}$ a type annotation $\forall\,(\alpha)\ \alpha \to \alpha$ must be added on x in f in order to obtain this type; otherwise f is untypable.

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

$$\forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha \tag{σ_{id}}$$

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 writing types such as $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$. We instead use an auxiliary variable with a rigid bound and write

$$\forall\,(\beta = \sigma_{\mathsf{id}})\ \beta \to \beta \tag{σ_1}$$

Alternatively, we could have used two different bounds, as in

$$\forall\,(\beta = \sigma_{\mathsf{id}})\ \forall\,(\gamma = \sigma_{\mathsf{id}})\ \beta \to \gamma \tag{σ_1'}$$

From a type-soundness point of view, rigid bounds can always be expanded, and there is no difference between the two types above. However, this is not the case from a type inference point of view. This difference is at the heart of $\mathsf{ML^F}$, and will be explained later.

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

$$\forall\,(\beta \geqslant \sigma_{\mathsf{id}})\ \beta \to \beta \tag{σ_2}$$

This time, σ_2 should 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

$$\forall\,(\beta \geqslant \sigma_{\mathsf{id}})\ \forall\,(\beta' \geqslant \sigma_{\mathsf{id}})\ \beta \to \beta' \tag{σ_3}$$

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}.

Combining both forms of quantification, the type

$$\forall\,(\beta = \sigma_{\mathsf{id}})\ \forall\,(\beta' \geqslant \sigma_{\mathsf{id}})\ \beta \to \beta' \tag{σ_4}$$

may be (roughly) understood as the set of all F-types $\sigma_{\mathsf{id}} \to \sigma$ such that σ is an instance of σ_{id}.

2.4 Improving $\mathsf{ML^F}$

While $\mathsf{ML^F}$ is a very powerful system, it could be improved in several ways:

1. The original syntactic presentation of $\mathsf{ML^F}$ (Le Botlan and Rémy 2003) is quite technical, and most extensions of the system in this form require a large amount of work. Indeed, while type instance and a subrelation called abstraction play a key role in $\mathsf{ML^F}$, they are defined by purely syntactic means, with little intuitive support. For a long time, these relations were mainly justified a posteriori by the properties of $\mathsf{ML^F}$. A more semantic-based definition has been proposed, but only for a significant restriction of the language (Le Botlan and Rémy 2007).

2. From an algorithmic point of view, the type inference algorithm based on syntactic types has obvious sources of inefficiencies. It is likely it would not scale up well to large, possibly automatically generated programs. Devising a more efficient algorithm was a question left open by Le Botlan (2004, page 221).

3. Although ML^F has been proven sound (Le Botlan 2004; Le Botlan and Rémy 2007), this has so far been done by proving the soundness of a system larger than the one in which type inference is performed. Indeed, proving subject reduction of the surface language requires to maintain—and to actually transform—type annotations during reduction. So far, finding an appropriate language to transform the annotations was an open question, precluding the use of ML^F as a typed internal language inside a compiler.

4. The power of ML^F has a price: ML^F types are more general than System F types, making them look unfamiliar to the user. Moreover, the bounded quantification used insides types obfuscates the structure of the type, making them quite difficult to read and to interpret.

A large part of this work aims at solving the issues above. In particular, Part I of this document develops an alternative representation of ML^F types which drastically simplifies the meta-theory of ML^F, and allows for efficient algorithms. Part III introduces an explicitly-typed presentation of ML^F, suitable for use as the core language of a typed compiler. The next section, which details our contributions, develop those points further.

The flavours of ML^F The different versions of ML^F that have been studied so far, including in this document, are summarized in Appendix A (page 291).

2.5 Outline of this document

This section explains our contributions, as well as how this document is structured. For each point, we mention the chapter in which it is developed.

2.5.1 Part I: graphic types and type instance

The first part of this document introduces an alternative representation of ML^F types as graphs, and studies the instance relation and unification on this presentation.

- We recall the graphical representation of first-order terms as term-dags, as well as of the type instance relation on this presentation of types. Term-dags are already used to represent types in efficient ML type inference, and are well-known *(§3)*.

  We generalize this presentation, first to System F, and then to System $\mathcal{F}$, an extension of System F with flexible quantification *(§3)*.

- ML^F types are derived from System $\mathcal{F}$ types by adding rigid quantification. We represent ML^F types by *graphic types (§4)*, which are the superposition of a term-dag (representing the structure of the type) and of a binding tree (which indicates where and how each node of the graphic type is bound), with further properties relating the two. The existence of a graphic presentation for $\mathsf{ML^F}$ types had already been suggested (Le Botlan 2004), but it was not sufficiently well-understood to be used formally.

- We express type instance $\sqsubseteq$ on graphic types by adapting the instance relation of System $\mathcal{F}$ to rigid quantification *(§5)*. Instance is simply the combination of four simple atomic operations on graphic types. Two of those operations are already present on term-dags, and the two others act on the binding tree. Valid instances are controlled through the use of *permissions*, which ensure that the operations permitted on a node are sound.

- Furthermore, we define two équivalece relations $\approx$ and $\boxminus$ *(§5)*. The first relation abstracts over useless binders on monomorphic type constructors such as int; two types equivalent for $\approx$ should not really be distinguished. The second relation is larger, and essentially identifies types that contain the same amount of polymorphism. The relation $(\sqsubseteq \cup \boxminus)^*$ is used to define an implicit version of $\mathsf{ML^F}$, in which type annotations are not needed (but in which type inference is not possible).

- Using permissions avoids the stratification of instance into two relations (abstraction and instance) present in the syntactic versions of $\mathsf{ML^F}$, and permits a simpler study of the properties of instance *(§6)*. The use of permissions also allows for a natural extension of the instance relation (compared to the original syntactic relation), with no technical overhead *(§8)*.[2]

- Unification on graphic $\mathsf{ML^F}$ types finds the smallest instance of two types for the instance relation, and is sound, complete and principal *(§7)*. The algorithm follows the same pattern as the instance relation: first-order unification on the term-dags, 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 unifiers. Of these three steps, only the third is not immediate. Moreover, our unification algorithm has optimal (linear) complexity.

- We show that graphic types are more canonical than syntactic types, as they factor out most of the syntactical artifacts present in the latter *(§8)*. We give algorithms translating to and from syntactic $\mathsf{ML^F}$ types in linear time *(§8)*. We propose a limited form of syntactic sugar to display $\mathsf{ML^F}$ types. Experimentally, using this sugar, most terms have a System F-like type, much more readable than their real $\mathsf{ML^F}$ type *(§8)*.

2.5.2 Part II: type inference with graphic constraints

The second part of this document generalizes the graphic types of the first part to *graphic constraints*. Indeed, we do not adapt the syntactic type inference algorithm by replacing its unification algorithm on syntactic types with the unification algorithm on graphic types: repeatedly translating to and from graphic types would be both inelegant and inefficient, losing the quite compact representation of graphic types. Moreover, we believe that the graphic presentation is better suited for studying the meta-theoretical properties of $\mathsf{ML^F}$. Instead, we propose an entirely graphical presentation of type inference.

[2]The instance relation has also been extended in a new syntactic presentation of $\mathsf{ML^F}$ (Le Botlan and Rémy 2007), but this required a more complex definition of the abstraction relation.

- We propose a small set of *graphic constraints*, featuring unification and instantiation edges, existential nodes, and generalization levels *(§9)*. Interestingly, graphic constraints are only a slight generalization of graphic types; thus the study of the meta-theory of graphic constraints is quite light. Moreover, since our approach to type inference is constraint-based, it is more general than just a particular type inference algorithm; for example, we can define different strategies for solving a constraint.

- Graphic constraints are in fact parameterized by a type system and the operation of taking the instance of a type scheme. We instantiate this framework with the graphic presentations of both ML and $\mathsf{ML^F}$, thus highlighting the strong ties between those two systems and reproving that the former is a subsystem of the latter *(§10)*. In particular, typing problems, which are obtained by translating λ-terms into graphic constraints in a compositional manner, are the same in both ML and $\mathsf{ML^F}$ *(§9)*—but are interpreted with different instance relations.

- Our constraints allow polymorphic recursion, and their solutions are in general undecidable. However, a very natural subset of constraints (called *acyclic*) has decidable solutions. This subset includes in particular all the constraints that are obtained when typing the λ-terms of §1.6, which are not recursive.

  We give a very simple algorithm to solve an acyclic constraint *(§12)*. The algorithm is a simple generalization of the one used for ML type inference, and is based on a conjunction of unification and type generalization.

- We prove that λ-terms that do not contain type annotations are typable in $\mathsf{ML^F}$ if and only if they are typable in ML. Thus the difference between the two systems lies in the type annotations in source terms, which are only available in their full generality in $\mathsf{ML^F}$ *(§12)*.

- We study the theoretical complexity of solving typing constraints *(§12)*. We establish the complexity of $\mathsf{ML^F}$ type inference, and observe that our algorithm has optimal complexity for both ML and $\mathsf{ML^F}$. Moreover, under reasonable assumptions, our algorithm for type inference in $\mathsf{ML^F}$ has linear complexity—as in ML.

- We compare the expressivity of the system we have studied (called $g\mathsf{ML^F}$) with the systems $e\mathsf{ML^F}$ and $i\mathsf{ML^F}$ that are obtained by taking as the type instance relation the relations $\sqsubseteq^{\approx}$ and $\sqsubseteq^{\boxminus}$ respectively *(§13)*. $e\mathsf{ML^F}$ and $g\mathsf{ML^F}$ have the same expressivity. This justifies our use of $\sqsubseteq$, which is simpler from a meta-theoretical standpoint, instead of $\sqsubseteq^{\approx}$. $i\mathsf{ML^F}$ is strictly more expressive, but this extra expressivity can be recovered in $g\mathsf{ML^F}$ through the use of type annotations.

2.5.3 Part III: an explicit language for $\mathsf{ML^F}$

The third part of this document introduces an explicit language for $\mathsf{ML^F}$, suitable as a typed internal language.

- We introduce $x\mathsf{ML^F}$, a Church-style version of $\mathsf{ML^F}$ in which all type information is explicit *(§14)*. Type instantiation in $x\mathsf{ML^F}$ generalizes type instantiation in System F. Moreover, all instantiation steps are made entirely explicit through the use of *type computations*, which serve as witnesses for type instance.

- Reduction in $x\mathsf{ML^F}$ is a combination of the usual β-reduction, and of a set of six reduction rules for type applications *(§14)*. All the rules preserve typing. Moreover, reduction is confluent when performing strong reduction.
- We show that $x\mathsf{ML^F}$ is sound, for both call-by-value and call-by-name semantics *(§14)*. This is the first time that an $\mathsf{ML^F}$-based language is proven sound for call-by-name. For languages with side effects, we show that the value restriction can be used in $x\mathsf{ML^F}$.
- Finally, we exhibit a translation from a solved typing constraint into a well-typed $x\mathsf{ML^F}$ term *(§15)*. This ensures in particular the soundness of our system of graphic constraints. We also discuss the translation of the syntactic presentations of $\mathsf{ML^F}$ into $x\mathsf{ML^F}$ *(§15)*.

2.5.4 Part IV: conclusions

To conclude this document, §16 discusses related works, while §17 summarizes our contributions and present some research perspectives.

2.6 Published works

A preliminary version of Part I of this document has been published in (Rémy and Yakobowski 2007), while Part II has been published in (Rémy and Yakobowski 2008); Part III is currently under submission for publication. The examples used in (Yakobowski 2008) are taken from an $\mathsf{ML^F}$ type-checker we have developed for this work.

Publications

Didier Rémy and Boris Yakobowski. From ML to MLF: Graphic type constraints with efficient type inference. In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08), Victoria, British Columbia, Canada, pages 63–74. ACM Press, September 2008. doi: `10.1145/1411203.1411216`.
`http://www.yakobowski.org/icfp08.html`

Didier Rémy and Boris Yakobowski. A graphical presentation of MLF types with a linear-time unification algorithm. In Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI'07), pages 27–38. ACM Press, Nice, France, January 2007. ISBN 1-59593-393-X. doi: `10.1145/1190315.1190321`.
`http://www.yakobowski.org/tldi07.html`

Boris Yakobowski. Le caractère ‘ à la rescousse - factorisation et réutilisation de code grâce aux variants polymorphes. In JFLA 2008 - Dix-neuvièmes Journées Francophones des Langages Applicatifs, pages 63–77. INRIA, Étretat, January 2008. ISBN 2-7261-1295-11.
`http://www.yakobowski.org/jfla08.html`

Part

I

A graphical presentation of ML^F types and type instance

3

Representing first- and second-order types by graphs

Abstract

We introduce the formalism behind the graphs used in this document. Our first step is to see first-order terms as trees (§3.1), and then as graphs (§3.2). This graph representation is often used in efficient algorithms for first-order unification. It is also very well-suited to ML^{F} types, in which sharing is important.

The next two sections are intendedly more informal. We present a graphical representation for System F types (§3.3), and extend it with flexible quantification (§3.4). The graphical representation of ML^{F} types will build upon this presentation.

In order to remain general, §3.1 and §3.2 are parameterized by an algebra Σ of *term* constructors. First-order *types* can be obtained by taking for Σ the algebra of type constructors of §1.5.

3.1 First-order terms

3.1.1 Definition of first-order terms

We view first-order terms as trees, which we describe using the notion of paths.

Definition 3.1.1 (*Paths*) A *path* π is a sequence of integers. The empty path is written ϵ. The concatenation of π' after π is written $\pi \cdot \pi'$. ■

The metavariable π ranges over paths. We often write $\pi\pi'$ for $\pi \cdot \pi'$ when there is no ambiguity. In fact, since in the examples we never use integers greater than 2, we allow writing 11 for $1 \cdot 1$. 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 can be defined as a partial function from the set of paths to the constructors of the term.

Definition 3.1.2 (*First-order terms*) A (first-order) *term* t over a set of variables $\mathcal{V}$ is a finite non-empty mapping from the set of paths to $\Sigma \cup \mathcal{V}$ that is prefix-closed and respects arities. More precisely, t must verify

$$\forall \pi \in \mathsf{dom}(t),\ \forall k \in \mathbb{N},\ (\pi \cdot k \in \mathsf{dom}(t) \iff 1 \leq k \leq \mathsf{arity}(t(\pi))) \qquad \blacksquare$$

The restriction on the domain of t ensures that there is no gap in the structure of t, and that the constructors have the correct number of arguments.

For first-order types, this definition coincides with the one given by the BNF grammar of §1.5 (page 3), although the approaches are quite different. Thus, we use the metavariable t to range over first-order types, whichever the chosen definition.

▶ **Example** First-order terms can be understood as trees: the tree (a) of Figure 3.2.1 represents the type

$$(\alpha \to \beta) \to (\alpha \to \beta)$$

Equivalently, this type is the function

$$\begin{cases} \epsilon,\ 1,\ 2 & \mapsto & (\to) \\ 11,\ 21 & \mapsto & \alpha \\ 12,\ 22 & \mapsto & \beta \end{cases}$$

This tree representation makes it easy to find the subterm of a term at a given path.

Definition 3.1.3 (*Term projection*) The *projection* of a term t at a path π of $\mathsf{dom}(t)$ is the term t/π that maps any π' such that $\pi \cdot \pi'$ is in $\mathsf{dom}(\tau)$ to $t(\pi \cdot \pi')$. ■

The projection of t at π is also called the subterm of t rooted at π.

▶ **Example** Projecting the type (a) at paths 1 or 2 yields the same type $\alpha \to \beta$.

3.1.2 Instance and unification on first-order terms

Definition 3.1.4 (*Substitution*) A *substitution* is a mapping from variables to terms. They are extended to a mapping from terms to terms by the usual canonical morphism. ■

Definition 3.1.5 (*Term instance*) A term t' is an instance of a term t, which we write $t \leqslant_{\mathsf{T}} t'$, if it is the image of t by some substitution φ. ■

Definition 3.1.6 (*Unification on terms*) Two terms t and t' are unifiable if there exists a substitution φ, called a unifier of t and t', such that $\varphi(t)$ and $\varphi(t')$ are equal. The unifier φ is said to be principal if any other unifier can be written as $\varphi' \circ \varphi$ for some substitution φ'.■

Alternatively, if terms are viewed up to α-renaming of their variables, unification can be defined without explicitly resorting to substitutions. A term t'' is a unifier of the terms t and t' if $t \leqslant_{\mathsf{T}} t'$ and $t \leqslant_{\mathsf{T}} t''$. It is principal if any other unifier is also an instance of t''. This second definition is actually easier to extend to richer types.

Unification on first-order terms is a well-known problem, which admits principal solutions. It can be solved in linear time, as shown by Paterson and Wegman (1978); in this algorithm, terms are represented as dags. Other algorithms (Huet 1976; Martelli and Montanari 1982) use union-find structures and have $n\alpha(n)$ time complexity (where α is the inverse of the Ackermann function). However, they run faster in practice and are simpler to implement.

Interestingly, all three algorithms internally use a graph representation of terms, and reinterpret the resulting graphs as terms. The use of the dag representation may be explicit when algorithms are described imperatively, or left implicit as in Huet's algorithm.

3.2 Term-graphs

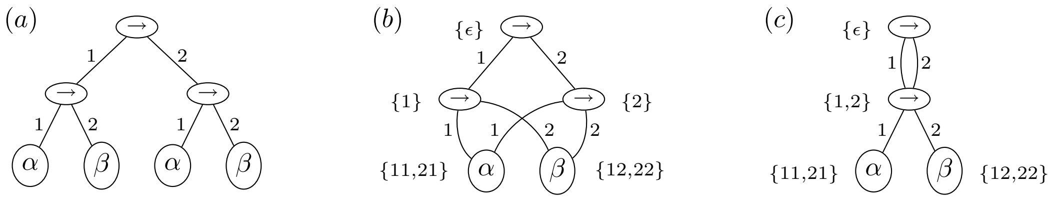


Figure 3.2.1 – Several representations of $(\alpha \to \beta) \to (\alpha \to \beta)$.

3.2.1 Definition

When representing first-order terms, it is sometimes convenient (and often more efficient) to identify all variables with the same name. Following this convention, the representation of the term (a) in Figure 3.2.1 is the dag (b) in the same figure.

Going one step further, inner nodes with identical subtrees can also be shared, as illustrated by graph (c). This enables sharing of common suffixes, hence a more compact—but also richer—representation, where sharing of nodes asserts the equality of nodes for the relation « projects to ».

Definition 3.2.1 (*Term-graphs*) Let t be a term and $\sim$ an equivalence relation on the paths in $\mathsf{dom}(t)$. The relation $\sim$ is said to be:

- *Congruent* if it is closed by suffix, *i.e.*

$$\forall \pi, \pi', \forall k, \quad \pi \sim \pi' \ \wedge \ \{\pi \cdot k,\ \pi' \cdot k\} \subseteq \mathsf{dom}(t) \implies \pi \cdot k \sim \pi' \cdot k$$

- *Consistent* if the image of an equivalence class by t is a singleton.

- *Weakly consistent* if the image of an equivalence class by t contains at most one symbol of Σ (*i.e.* it possibly contains variables, and at most one constructor).

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.[1] Those equivalence classes are called *nodes*. ■

Congruent relations ensure that when two paths are shared, the subtrees under those paths are also shared. Consistent relations guarantee that different constructors are not mixed together. Weakly consistent relations are used to reason about unification, which fuses together variables and nodes with identical constructors.

We use the metavariables g and n to range over term-graphs and nodes. We write $\dot{g}$ and $\tilde{g}$ the underlying term and equivalence relation of g (or $\dot{\overline{g_{foo}}}$ and $\widetilde{g_{foo}}$ for long names).

The relation $\tilde{g}$ partitions the paths of $\mathsf{dom}(\dot{g})$ into disjoint equivalence classes that constitute the nodes of g. We write $\mathsf{dom}(g)$ for this set of nodes. Since $\dot{g}$ is constant on each node, we may extend it to nodes by mapping each node n to the common value of $\dot{t}$ on all paths of n. We simply write $g(n)$ for this value. Similarly, we extend the projection / into a function from nodes to term-graphs.

▶ **Example** The dags (b) and (c) on Figure 3.2.1 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 have the same constructor. Moreover, both nodes are labelled with the arrow symbol; hence, the equivalence relation of (b) could be enriched with $1 \sim 2$ while remaining congruent and consistent. This results in exactly the equivalence relation of dag (c). Intuitively, the subgraphs under $\{1\}$ and $\{2\}$ were identical in (b), and have been merged in (c).

In this simple example, only the nodes $\{1\}$ and $\{2\}$ have been merged; in more complex cases, entire subgraphs may be shared. Notice that the (name of a) node resulting from the merge is the union of the (names of the) nodes that have been merged.

3.2.1.1 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 convenience, we extend this notation to sets of paths. In particular, $\langle n\rangle$ is n itself.

▶ **Example** In picture (c) of Figure 3.2.1, $\langle 12\rangle$ and $\langle 22\rangle$ refer to the same node $\{12, 22\}$.

More generally, given two term-graphs g and g' such that $\tilde{g} \subseteq \tilde{g}'$ (*i.e.* g' shares more nodes than g), a node n of g can be translated unambiguously into a node n' of g', by taking the only node of g' that is a superset of n. We will often use n instead of n' when g' is the result of a transformation applied to g.

▶ **Example** If n is the node $\langle 1\rangle$ of graph (b) in Figure 3.2.1, we will freely use n for the node $\{1, 2\}$ of graph (c).

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

[1]This last invariant is not strictly required, but there is no real advantage to using graphs if it is not enforced (as substitution would then not be noticeably simpler than on first-order terms).

3.2.1.2 Term-graphs as ordinary graphs

A term-graph g may be read as an ordinary graph whose nodes are the set $\mathsf{dom}(g)$, and whose edges are such that

$$n \xrightarrow{k} n' \iff n \in \mathsf{dom}(g) \wedge 1 \le k \le \mathsf{arity}(g(n)) \wedge \langle n \cdot k \rangle = n'$$

In essence, 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 the inner structure of nodes as set of paths could be exponential in the size of the graph. However, the named view is more convenient in the formal development, for referring to nodes and to keep track of them during a sequence of graph transformations.

3.2.2 Instance on term-graphs

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

Definition 3.2.2 (*Instance on term-graphs*) A term-graph g' is an *instance* of a term-graph g, written $g \sqsubseteq_{\mathsf{G}} g'$, if $\dot{g} \leqslant_{\mathsf{T}} \dot{g}'$ and $\tilde{g} \subseteq \tilde{g}'$. It is a *reversible* instance if moreover $\dot{g} = \dot{g}'$.

The *equivalence* relation $\equiv_{\mathsf{G}}$ is the kernel of the instance relation. The *similarity* relation $\approx_{\mathsf{G}}$ is the symmetric reflexive and transitive closure of the reversible instance relation. ■

Instance is an oriented relation, and its kernel is quite "small": two equivalent term-graphs are in fact equal modulo α-conversion.

Reversible instance only changes the representation of terms (by using more inner, hence unimportant, sharing), but not their meaning as first-order terms. In particular, g and g' are similar if and only if $\dot{g} = \dot{g}'$. Similarity is thus used to abstract over the sharing not semantically meaningful that is brought by the use of term-graphs.

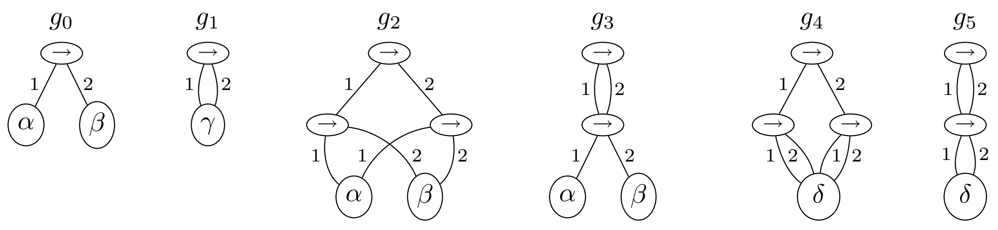


Figure 3.2.2 – Term-graph instance.

▶ **Examples** In Figure 3.2.2, the term-graphs g_3 and g_5 are two instances of g_1, through the substitutions $\gamma \mapsto \alpha \to \beta$ and $\gamma \mapsto \delta \to \delta$ respectively. The graph g_3 is also a reversible instance of g_2, obtained by adding $1 \sim 2$ to $\tilde{g_2}$. Thus those two graphs are also similar. In this simple example, g_3 is an instance of g_2. In more complicated cases, one graph could share more in one branch and less in the other.

The term-graph g_4 is also an instance of g_2, through the substitution $\alpha, \beta \mapsto \delta$. However, even though $\dot{g}_3 \leqslant_{\mathsf{T}} \dot{g}_4$ holds, g_4 is *not* an instance of g_3. Indeed, the nodes $\langle 1 \rangle$ and $\langle 2 \rangle$ are shared in g_3 but not in g_4. Similarly, g_5 is an instance of g_4 but the converse is not true.

$$\begin{tikzcd}
& g_1 \arrow[r] & g_3 \arrow[dr] & \\
g_0 \arrow[ur] \arrow[dr] & & & g_5 \\
& g_2 \arrow[r] \arrow[ur, dashed] & g_4 \arrow[ur, dashed] &
\end{tikzcd}$$

Figure 3.2.3 – Valid instances in the examples of Figure 3.2.2

The various relations that hold in Figure 3.2.2 are summarized in Figure 3.2.3. Plain edges represent $\sqsubseteq_{\mathsf{G}}$, while dashed ones represent its reversible subset; all transitive or reflexive edges have been omitted to simplify the drawing.

Notation We always use $\leqslant$-derived symbols for instance on syntactic terms (such as $\leqslant_{\mathsf{T}}$ on the terms of the previous section, or $\leqslant_{\mathsf{F}}$ for System F types), and $\sqsubseteq$-derived symbols for instance on terms represented by graphs.

3.2.3 Unification on term-graphs

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.

Definition 3.2.3 (*Unification on term-graphs)* A term-graph g' is a unifier of two nodes n and n' of a term-graph g if g' is an instance of g that merges n and n'; *i.e.* $\exists n'' \in g'$, $n'' \supseteq n, n'$. A unifier g' of two nodes is *principal* if any other unifier of those nodes is an instance of g'. ■

▶ **Example** The term-graphs g_1, g_3 and g_5 are unifiers of the nodes $\langle 1 \rangle$ and $\langle 2 \rangle$ in g_0, with g_1 being a principal unifier. Similarly, g_4 and g_5 are unifiers of the nodes $\langle 11 \rangle$ and $\langle 12 \rangle$ in g_2, and g_4 is principal.

The unification of two nodes of g can be computed as the smallest weakly consistent, congruent equivalence that contains $\tilde{g}$ and identifies both nodes (Huet 1976).

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 $\dot{g}'/n$ is a (principal) unifier of $\dot{g}/n_1$ and $\dot{g}/n_2$, n being the node of g' that is a superset of n_1 and n_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.

3.2.4 Anonymous variables

The last condition of Definition 3.2.1 implies that a variable is represented by a single node. If we allow reading term-graphs modulo α-conversion, we may advantageously draw variable nodes anonymously. 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 (*i.e.* it is not an element of Σ) but a new pseudo-symbol that does not clash with other symbols during unification.

We call *anonymous* a term-graph that uses $\bot$ nodes instead of named variables. An example will be given in the next section.

3.2.4.1 Congruent nodes

On anonymous term-graphs, we can identify nodes that are the root of entirely identical subgraphs. (This was not possible on the named presentation, because we would have needed to reason up to α-conversion.)

Definition 3.2.4 (*Congruent nodes)* Given a term-graph g, we say that two nodes n_1 and n_2 are *congruent* in g if they are distinct and verify

$$\wedge\begin{cases} \dot{g}/n_1 = \dot{g}/n_2 \\ \forall\pi, \forall\pi',\ \langle n_1 \cdot \pi\rangle\ \tilde{g}\ \langle n_1 \cdot \pi'\rangle \iff \langle n_2 \cdot \pi\rangle\ \tilde{g}\ \langle n_2 \cdot \pi'\rangle \end{cases} \qquad \blacksquare$$

The first condition ensures that the subtrees under n_1 and n_2 have the same shape, and are labelled by the same constructors. The second condition checks that the amount of sharing is the same below each node.

By construction, two nodes n_1 and n_2 congruent in a term-graph g can be merged, by adding to $\tilde{g}$ the relation $\langle n_1 \cdot \pi\rangle \sim \langle n_2 \cdot \pi\rangle$ for any valid π.

Definition 3.2.5 (*Fusion)* Given two congruent nodes n_1 and n_2 of a term-graph g, we call *fusion* of n_1 and n_2 in g the term-graph $g[n_1 = n_2]$ verifying $\dot{\overline{g[n_1 = n_2]}} = \dot{g}$ and

$$\widetilde{g[n_1 = n_2]} = \tilde{g} \cup \big\{(\pi_1 \cdot \pi, \pi_2 \cdot \pi) \mid \pi_1 \in n_1,\ \pi_2 \in n_2,\ \pi_1 \cdot \pi \in \mathsf{dom}(g)\big\} \qquad \blacksquare$$

Notice that being congruent is a sufficient condition for two nodes to be unifiable, but not a necessary one.

▶ **Example** The nodes $\langle 1\rangle$ and $\langle 2\rangle$ are congruent in the anonymous term-graphs g_1 and g_2 of Figure 3.2.4. In both cases, their fusion is the term-graph g_5. However, even though n_1 and n_2 can be unified in g_3 and g_4, they are not congruent in those two term-graphs. Indeed, in g_2, we have $\langle 11\rangle \notin \mathsf{dom}(g_3)$ but $\langle 21\rangle \in \mathsf{dom}(g_3)$; hence $\dot{g_3}/\langle 1\rangle$ and $\dot{g_3}/\langle 2\rangle$ are distinct. In g_4, the amount of sharing between the two nodes differ: we have $\langle 21\rangle\ \tilde{g_4}\ \langle 22\rangle$, but $\langle 11\rangle\ \tilde{g_4}\ \langle 12\rangle$ does not hold.

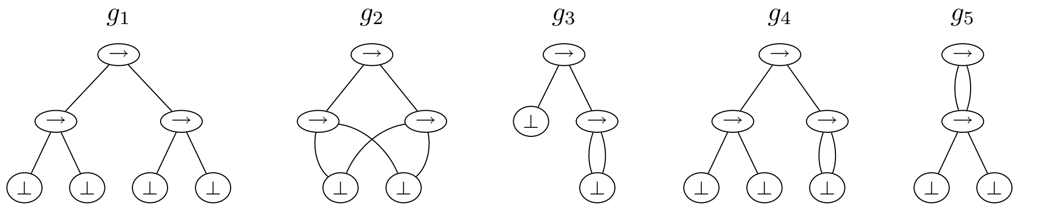


Figure 3.2.4 – Congruent nodes

3.2.4.2 Small-step instance

Definition 3.2.2 is essentially big-step, as it permits instantiating the skeleton and the equivalence relation of a term-graph in a very general way. In the next section we are going to consider second-order types, on which some instance transformations are not permitted (as they would be unsound w.r.t. the semantics of the types). However, with a big-step relation, deciding whether an operation is allowed or not is complicated. Hence we introduce a small-step relation, in which operations are more atomic—thus more easily checkable.

Definition 3.2.6 (*Small-step instance on term-graphs*) The small-step instance relation on anonymous term-graphs is the reflexive transitive closure of the two atomic transformations defined below

- *grafting* a variable node, *i.e.* replacing a $\bot$-labelled node by a term-graph;
- *merging* two congruent nodes.

An instance operation is reversible if it is a merging that does not merge variable nodes. ■

It is straightforward to check that this definition and Definition 3.2.2 coincide: grafting exactly corresponds to the substitution of a variable by a closed type, and merging merely instantiates the equivalence relation of the term-graph in an atomic way.

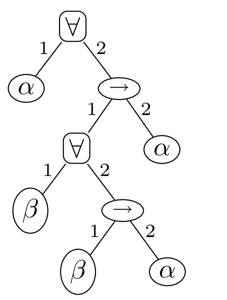
Interestingly, the grafting operation is easier to describe on anonymous term-graphs than on term-graphs with named variables. Indeed, we need not check that we are not grafting a term-graph containing a variable already present in the type (which would result in a term-graph with two nodes for the same variable). Instead, in essence we are always grafting fresh variables, which can be merged a posteriori with another variable if desired.

3.3 Representing second-order types

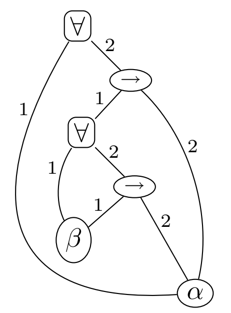
When representing types as trees, binders are traditionally represented with an explicit node labeled with a special symbol $\forall$ of arity two. For example, the System F type σ defined by

$$\sigma = \forall\alpha.\alpha \to (\forall\beta.\beta \to \alpha)$$

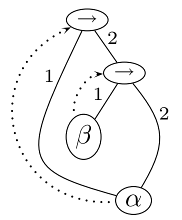
is usually represented as the tree (1) of Figure 3.3.1. Using dags (hence sharing at least the variables), we obtain the representation (2).



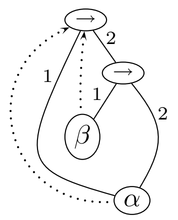
(**1**) Second-order term



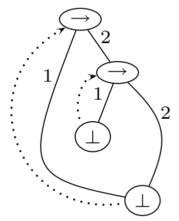
(**2**) Second-order dag



(**3**) Binding edges



(**4**)



(**5**) Anonymous variables

Figure 3.3.1 – Representations of second-order types.

Notice that all graphs are not correct types, as variables must be used within their scope. In graph (2), the node ⟨211⟩ could not have been the second child of node ⟨2⟩, as β has not been introduced yet. However, we do not detail which graphs are well-formed any further in this chapter, as this question will be treated in detail on ML^{F} graphic types in §4.3.

3.3.1 Binding edges

Unfortunately, representing quantifiers as special nodes inserted in the structure—which will need to be modified, for example when a variable is no longer used—hides the underlying common structure of all instances. A better solution is to introduce a *binding edge* between the bound variable and the node at which it is bound[2]. This is illustrated in graph (3), in which there is a binding edge from ⟨1⟩ to the root for α, and from ⟨21⟩ to ⟨2⟩ for β.

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 slightly easier to think about, as each variable node is bound to a single node, but a single node can be a binding position for several variables.

Notice that this representation looses the order of adjacent binders and makes useless binders not representable—two artifacts of the syntactic notations that we are quite happy

[2]Except for nodes representing quantification of the form $\forall\alpha.\ \alpha$, which have no binding edge.

to eliminate. For instance, the representation of all three types

$$\begin{array}{r}\forall\alpha.\ \forall\beta.\ \beta \to (\beta \to \alpha) \\ \forall\beta.\ \forall\alpha.\ \beta \to (\beta \to \alpha) \\ \forall\gamma.\ \forall\alpha.\ \forall\beta.\ \beta \to (\beta \to \alpha)\end{array}$$

will be the same, namely the graph (4) of Figure 3.3.1. Notice also that the graphs (3) and (4), which represent two types differing only by the extrusion of a quantifier, have the same skeleton. By using binding edges, the instances in the skeleton of a type or in its binding structure become more orthogonal.

3.3.2 Anonymous variables

As for term-graphs, if we allow reading second-order types modulo alpha-conversion, we can use anonymous variables; in fact, we do so in the remainder of this document. For the type σ, we obtain the graph (5) of Figure 3.3.1.

3.3.3 Instantiation on graphic System F types

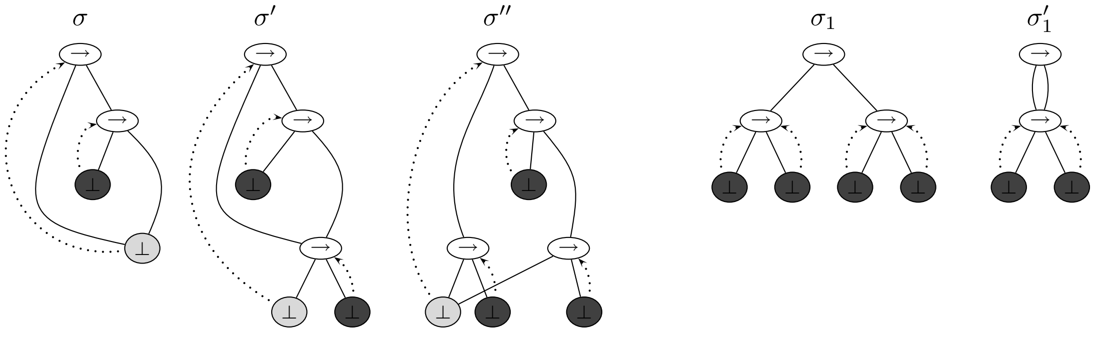


Figure 3.3.2 – Instance on graphic System F types

Let us define the instance relation $\sqsubseteq_{\mathsf{F}}$ on the graphic representation of System F types. As System F types generalize first-order types, we expect $\sqsubseteq_{\mathsf{F}}$ to generalize $\sqsubseteq_{\mathsf{G}}$. However, in System F, not all variables can be instantiated. For example, the variable α in $(\forall\alpha.\ \alpha \to \alpha) \to \tau$ is locked. Naturally, the same distinction exists in the graphical presentation: a bottom node can be instantiated if and only if it is bound to the root.

Colors In drawings, we remind of the fact that a variable can be instantiated by drawing it in green; conversely, we draw a locked variable in red. For System F, this might seem overkill, as the distinction is always easy to make: green variables are those bound on the root, while red ones are bound on an inner node. However we will gradually expand our convention to more complicated systems, in which colors will be a useful visual remainder.

For brevity, in the text we refer to « green » or « red » variables. This also offers some form of abstraction over systems that have related instance relations. However, let us stress that colors are only a visual help, and can always be deduced from the instance relation of the system under consideration.

3.3.3.1 Grafting

Consider the type σ defined at the beginning of this section, which we have drawn with colors in Figure 3.3.2. We can instantiate it into the type σ' of the same figure by replacing the green variable by the type $\forall\beta.\ \alpha \to \beta$, where α is a fresh variable which is introduced at the root of σ'. We obtain the syntactic type

$$\forall\alpha.\ (\forall\beta.\ \alpha \to \beta) \to (\forall\gamma.\ \gamma \to (\forall\beta.\ \alpha \to \beta))$$

Notice in particular that the grafting operation of System F is more complex than the one on term-graphs, as we must take into account the binding structure of the graphs. When grafting a type τ at a node n of a type τ', τ can have bound variables (which are left unchanged by the grafting), but also free variables (which should be bound at the root of τ' after the grafting operation).

▶ **Example** When grafting $\forall\beta.\ \alpha \to \beta$ at $\langle 1\rangle$ in σ, the (free) variable α becomes bound at the root of σ'.

3.3.3.2 Merging inner nodes

Interestingly, the graph σ'' of Figure 3.3.2 is another representation of the syntactic type given above. This time, the two occurrences of $\forall\beta.\ \alpha \to \beta$ are represented by distinct subgraphs. However, since α is quantified higher in the type, there is still only one node corresponding to that variable.

As for term-graphs, using a graphic presentation brings some redundancy into the representation of types, which might differ by the amount of sharing they contain. As for term-graphs, we capture those differences by a *similarity* relation $\approx_{\mathsf{F}}$ that is the equivalence relation induced by the reversible subset of this instance relation.

Here, since σ' and σ'' represent the same syntactic type, they must be equivalent for $\approx_{\mathsf{F}}$. In this simple example, the difference between the two types is very small: the nodes $\langle 12\rangle$ and $\langle 222\rangle$ are shared in σ', but not in σ''. In par with the beginning of the chapter, we choose to make σ' an instance of σ'' (*i.e.* $\sigma'' \sqsubseteq_{\mathsf{F}} \sigma''$) as sharing increases when going from σ'' to σ'.

As discussed above, this instance must be reversible. However, two variables (the nodes $\langle 12\rangle$ and $\langle 222\rangle$) are merged by the operation. This is in stark contrast with first-order types, where reversible instance only involves inner nodes, and never variables.

A misleading—and incorrect—intuition would be to think that red variables can be freely merged. If we consider the type σ_1 of Figure 3.3.2, which is a valid type for the identity function, merging the nodes $\langle 11\rangle$ and $\langle 12\rangle$ would result in the syntactic type

$$(\forall\alpha.\ \alpha \to \alpha) \to (\forall\beta.\ \forall\beta'.\ \beta \to \beta')$$

This is of course unsound with respect to $\leqslant_{\mathsf{F}}$, and such an instance is forbidden. More generally, merging red variables is not permitted; indeed, it is syntactically the same as substituting one variable by the other, and the System F instance relation does not permit instance on red variables.

However, we still have not explained why $\sigma'' \sqsubseteq_{\mathsf{F}} \sigma'$ holds. The idea is that the two variables $\langle 12 \rangle$ and $\langle 222 \rangle$ were not directly merged. Instead two subgraphs containing these two variables *and their binders* were merged; in this particular case the subgraphs under $\langle 1 \rangle$ and $\langle 22 \rangle$. Thus we have in fact merged two graphs that were α-convertible one into the other. From a semantic point of view, this kind of sharing in the type cannot be observed. This is easily seen on σ_1; a function of this type returns something that has exactly type $\forall \alpha.\ \forall \beta.\ \alpha \to \beta$, and it can only receive as its argument an expression that has exactly the same type. Whether the representations of the argument and of the return type are shared or not is unimportant; those two types only contain red variables, and we cannot take a semantically meaningful instance on those nodes. Thus the relation $\sigma_1 \sqsubseteq_{\mathsf{F}} \sigma'_1$ holds, the instance being reversible.

A merging can be reversible only if does not result in the indirect merging of nodes quantified higher in the type. (Indeed, this indirect merging would change the semantics of the type, and could even be unsound.) We characterize nodes that result in such "well-behaved" merging by the following definition.

Definition 3.3.1 (*Locally congruent nodes)* Two nodes n_1 and n_2 of a graph σ are *locally congruent* if

- n_1 and n_2 are congruent in the term-graph g underlying σ;
- the binding edges under n_1 and n_2 are compatible with $g\widetilde{[n_1 = n_2]}$
- for any two distinct nodes n'_1 and n'_2 under n_1 and n_2 respectively, if n'_1 and n'_2 are merged in $g[n_1 = n_2]$, then n'_1 and n'_2 are bound below n_1 and n_2. ■

From an operational point of view, restraining the merging operation to locally congruent nodes makes this operation more local: only binding edges in the subgraphs under n_1 and n_2 can be merged. It makes also easier to verify that red variables are never merged.

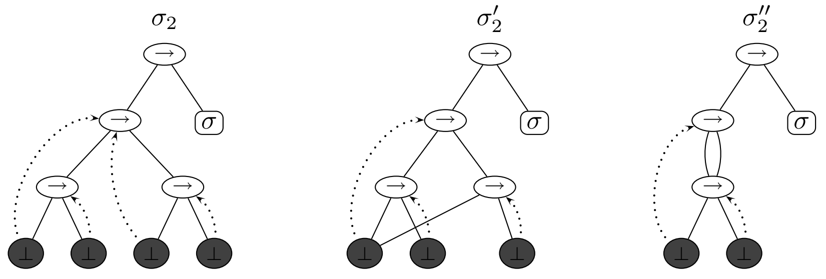


Figure 3.3.3 – Merging locally congruent nodes

▶ **Example** The nodes $\langle 11 \rangle$ and $\langle 12 \rangle$ are locally congruent in the type σ'_2 of Figure 3.3.3. Indeed, while the nodes $\langle 111 \rangle$ and $\langle 121 \rangle$ are bound above $\langle 11 \rangle$ and $\langle 12 \rangle$, they are already equal in σ'_2; meanwhile the nodes $\langle 112 \rangle$ and $\langle 122 \rangle$ are bound under $\langle 11 \rangle$ and $\langle 22 \rangle$. Merging

$\langle 11 \rangle$ and $\langle 12 \rangle$ results in the type σ_2''. Both σ_2' and σ_2'' represent the syntactic type

$$(\forall\alpha.\ (\forall\beta.\ \alpha \to \beta) \to (\forall\beta.\ \alpha \to \beta)) \to \tau$$

Conversely, $\langle 11 \rangle$ and $\langle 12 \rangle$ are not locally congruent in σ_2, as $\langle 111 \rangle$ and $\langle 121 \rangle$ are bound above the former nodes. Indeed, σ_2 represents the syntactic type

$$(\forall\alpha.\ \forall\alpha'.\ (\forall\beta.\ \alpha \to \beta) \to (\forall\beta.\ \alpha' \to \beta)) \to \sigma$$

in which there are two distinct variables (α and α') quantified on the left of the toplevel arrow.

3.3.3.3 Summary: instance in graphic System F

We can now formally define the instance relation for the graphical presentation of System F. Compared to the instance relation on term-graphs, here are the differences:

1. In order to preserve type soundness, grafting is only possible on green variables. Moreover, it now takes into account the binders of the grafted types, as described in §3.3.3.1.
2. Merging is limited to locally congruent nodes, in order to be more atomic (§3.3.3.2). Moreover, red variables cannot be merged, again for soundness related reasons.

Definition 3.3.2 (*Instance on graphic F-types)* The instance relation $\sqsubseteq_{\mathsf{F}}$ on graphic System F types is the reflexive transitive closure of the three following atomic instance operations:

1. grafting a type σ at a green variable node n; the binding edge of n is implicitly removed; the free variables of σ are bound at the root of the type in which the grafting occurs.
2. merging two locally congruent non-variable nodes;
3. merging two green variable nodes.

An instance operation is reversible if and only if it of the second form, and we write $\approx_{\mathsf{F}}$ the symmetric reflexive transitive closure of reversible instance. Finally, we write $\sqsubseteq_{\mathsf{F}}^{\approx}$ the instance modulo reversible instance relation $(\sqsubseteq_{\mathsf{F}} \cup \approx_{\mathsf{F}})^*$. ■

We do not specifically require the merging of variables to be on locally congruent nodes, as it is in fact always the case (given the definition of local congruence).

Notice that $\sqsubseteq_{\mathsf{F}}$ and $\approx_{\mathsf{F}}$ subsume (*i.e.* extend) the relations $\sqsubseteq_{\mathsf{G}}$ and $\approx_{\mathsf{G}}$ of §3.2.2 when the variables in the term-graph are considered to be implicitly bound to the root. Moreover, $\sqsubseteq_{\mathsf{F}}^{\approx}$ is sound and complete w.r.t. to the instance relation $\leqslant_{\mathsf{F}}$ on syntactic System F types: given two syntactic F-types τ_1 and τ_2, and σ_1 and σ_2 two graphic representations of those types, we have $\tau_1 \leqslant_{\mathsf{F}} \tau_2$ if and only if $\sigma_1 \sqsubseteq_{\mathsf{F}}^{\approx} \sigma_2$.

3.4 Adding flexible quantification to second-order graphic types

3.4.1 Beyond system F

System F is poorly suited as a programming language with type inference since, as we mentioned in the introduction, it lacks principal types. Even simple terms such as

$$K' \triangleq \lambda(x)\,\lambda(y)\,y$$

can be typed with incomparable types, *e.g.*

$$\forall\alpha.\ \forall\beta.\ \alpha \to (\beta \to \beta) \qquad (\sigma_1)$$

and

$$\forall\alpha.\ \alpha \to (\forall\beta.\ \beta \to \beta) \qquad (\sigma_2)$$

System Fη One solution to remedy this problem is the system Fη, proposed by Mitchell (1988). Roughly, the instance relation $\leqslant_{\mathsf{F}\eta}$ of Fη allows to soundly instantiate types along $\leqslant_{\mathsf{F}}$ on the right of an arrow (*i.e.* in a covariant position), and along $\geqslant_{\mathsf{F}}$ on the left of an arrow (*i.e.* in contravariant position). In particular, the second type above is more general than the first:

$$\forall\alpha.\ \alpha \to (\forall\beta.\ \beta \to \beta) \ \leqslant_{\mathsf{F}\eta}\ \forall\alpha.\ \forall\beta.\ \alpha \to (\beta \to \beta)$$

Yet, Fη is not quite satisfactory. First, the contravariance of the instance relation is often too powerful (and in fact rarely needed). Moreover, it goes against the idea of unification-based type inference, which is at the heart of ML. Indeed, this would require performing unification on the right of an arrow, but anti-unification on the left.

More problematic, Fη still does not have principal types. Even though $\forall\alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha)$ is a principal type for choose id in Fη (while this expression does not have a principal type in System F), this property does not generalize to more complex examples. For example, the term choose (choose id), can receive in particular the System F types

$$\begin{array}{ll}
\forall\alpha.\ ((\alpha \to \alpha) \to (\alpha \to \alpha)) \to ((\alpha \to \alpha) \to (\alpha \to \alpha)) & (\rho_1)\\
(\forall\beta.\ (\beta \to \beta) \to (\beta \to \beta)) \to (\forall\beta.\ (\beta \to \beta) \to (\beta \to \beta)) & (\rho_2)\\
((\forall\gamma.\ \gamma \to \gamma) \to (\forall\gamma.\ \gamma \to \gamma)) \to ((\forall\gamma.\ \gamma \to \gamma) \to (\forall\gamma.\ \gamma \to \gamma)) & (\rho_3)
\end{array}$$

In System Fη we can derive $\rho_1 \leqslant_{\mathsf{F}\eta} \rho_2$, but not $\rho_1 \leqslant_{\mathsf{F}\eta} \rho_3$ or $\rho_2 \leqslant_{\mathsf{F}\eta} \rho_3$: indeed, we can instantiate covariantly on the right of the toplevel arrow, resulting in

$$\rho_2 \leqslant_{\mathsf{F}\eta} (\forall\beta.\ (\beta \to \beta) \to (\beta \to \beta)) \to ((\forall\gamma.\ \gamma \to \gamma) \to (\forall\gamma.\ \gamma \to \gamma))$$

but we cannot transform $\forall\beta.\ (\beta \to \beta) \to (\beta \to \beta)$ into $(\forall\gamma.\ \gamma \to \gamma) \to (\forall\gamma.\ \gamma \to \gamma)$ on the left of the arrow.

As in System F, the lack of principal types in Fη stems from the fact that we cannot express the correlation between two sub-types (*e.g.* the various instances of σ_{id} in the examples above). We believe that this limitation is in fact inherent to using F types.

Flexible quantification ML$^{\mathsf{F}}$ follows an entirely different solution, and enriches the types of System F with a new construction that indicates what parts of a type can be soundly

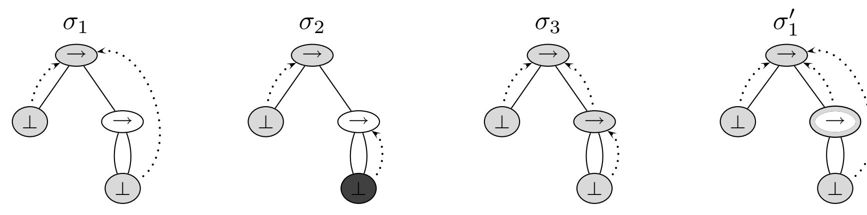


Figure 3.4.1 – $\mathsf{ML^F}$ types for K'

instantiated. In particular, unlike in $\mathsf{F}\eta$, this information is added explicitly, and is not linked to the variance of the arrow constructor.

Let us give some examples, using the types of K'. We have represented σ_1 and σ_2 in Figure 3.4.1. However, in $\mathsf{ML^F}$, K' has principal type σ_3, which differs from σ_2 by adding a binding edge from the node $\langle 2 \rangle$ to the root. The node $\langle 2 \rangle$ corresponds to the root of the type $\forall\,(\beta)\ \beta \to \beta$; since the binding edge of $\langle 2 \rangle$ goes to the root, it indicates that $\langle 2 \rangle$ can be instantiated. Then, by transitivity, the node $\langle 21 \rangle$, which is bound at $\langle 2 \rangle$ and corresponds to β, can also be instantiated. Hence both $\langle 2 \rangle$ and $\langle 21 \rangle$ are green. (Since the root allows the nodes bound at it to be instantiated, we also draw it in green. This is essentially a convention.)

Extending System F with binding edges to non-variable nodes can either be seen as a restriction of $\mathsf{ML^F}$, or as a system in its own right. In this document we follow the second approach, as it permits explaining $\mathsf{ML^F}$ in a much simpler way. We call the resulting system *System $\mathcal{F}$*, where the curly $\mathcal{F}$ stands for « flexible F ». However, we intendedly remain informal, as studying System $\mathcal{F}$ is not the goal of this document:

- most of the interesting properties of System $\mathcal{F}$ can be deduced from those of $\mathsf{ML^F}$;
- System $\mathcal{F}$ has been studied in detail by Le Botlan and Rémy (2007), albeit on a restricted version of the system presented here.[3] We will present one important result in §3.4.3.

Instead, we use System $\mathcal{F}$ as an intermediary step to gain better intuitions.

3.4.2 Type instance in System $\mathcal{F}$

Let us review the operations that compose the instance relation $\sqsubseteq_{\mathcal{F}}$ of System $\mathcal{F}$. We will start by examining the types of K' given in Figure 3.4.1. Since σ_3 is the principal type of K' in $\mathsf{ML^F}$, $\sigma_3 \sqsubseteq_{\mathcal{F}} \sigma_1$ and $\sigma_3 \sqsubseteq_{\mathcal{F}} \sigma_2$ must hold.

[3]In Le Botlan and Rémy (2007), System $\mathcal{F}$ is called $i\mathsf{ML^F}$. However, in this document we use $\mathsf{ML^F}$ derived names, including $i\mathsf{ML^F}$, for systems based on graphic types.

3.4.2.1 Weakening

The operation transforming σ_3 into σ_2 is the removal of the binding edge leaving from $\langle 2\rangle$. From a semantic point of view, it relinquishes the right to instantiate $\langle 2\rangle$ (and $\langle 21\rangle$ by transitivity). We call this operation a *weakening*.

Of course, it is sound only because $\langle 2\rangle$ is green.[4] Otherwise, by weakening a red node not bound at the root of the type, we could require a different amount of polymorphism than what was requested, which is unsound. The inverse operation (adding binding edges) is unsound for the same reason.

3.4.2.2 Raising

The operation transforming σ_3 into σ_1 can actually be decomposed into two atomic steps. At first, we instantiate the type $\forall\beta.\ \beta \to \beta$ under $\langle 2\rangle$ into the type $\gamma \to \gamma$, where γ is a new type variable introduced at the root. This results in the type σ_1'.[5] Next, we can remove the binding edge of $\langle 2\rangle$ by a weakening. Thus the following relation holds

$$\sigma_3 \sqsubseteq_{\mathcal{F}} \sigma_1' \sqsubseteq_{\mathcal{F}} \sigma_1$$

From an operational standpoint, σ_1' is obtained by extruding the binding edge of $\langle 21\rangle$ along the binding edge of $\langle 2\rangle$ (*i.e.* the node on which $\langle 21\rangle$ is bound). We call this operation *raising*.

Raising is sound only when n is green. Indeed, from a semantical standpoint, raising a node n loses the ability to quantify variables at the level of the binder of n. Hence it can be permitted only if the node can be instantiated.

3.4.2.3 Grafting

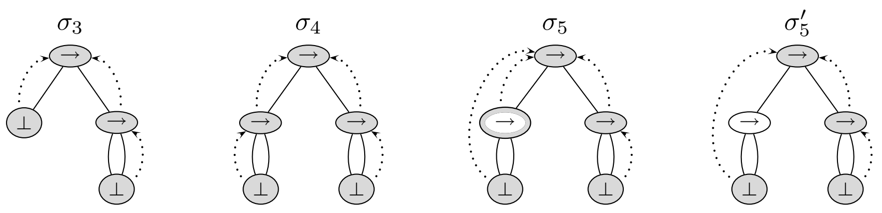


Figure 3.4.2 – Grafting in System $\mathcal{F}$

Interestingly, compared to System F, raising and weakening simplify the operation of replacing a green bottom node by a type. Indeed, instead of (1) adding a new type, (2) binding its free variables to the root and (3) removing the binding edge, it is now possible to replace the bottom node by a closed type. The free variables (if there are any) can be

[4]This is actually an over-simplification. We will come back on this point later.
[5]The reason why $\langle 2\rangle$ is hollow in σ_1' is linked to previous footnote, and will be explained later.

raised in a second time, and the binding edge of the bottom node can finally be removed by weakening, if needed. An example is given in the derivation

$$\sigma_3 \sqsubseteq_{\mathcal{F}} \sigma_4 \sqsubseteq_{\mathcal{F}} \sigma_5 \sqsubseteq_{\mathcal{F}} \sigma'_5$$

of Figure 3.4.2, where we substitute the variable α of σ_3 (*i.e.* the node $\langle 1 \rangle$) by σ_{id}, raise the newly introduced type variable (node $\langle 11 \rangle$), and finally weaken $\langle 1 \rangle$.

Moreover, flexible quantification and raising allow quite a bit more freedom w.r.t. to where to bind nodes. Indeed, after grafting a type τ at a node n of a type τ', the nodes bound on the root of τ can be raised to any of the nodes on which n is transitively bound, instead of only to the root of τ' or to n.

3.4.2.4 Merging

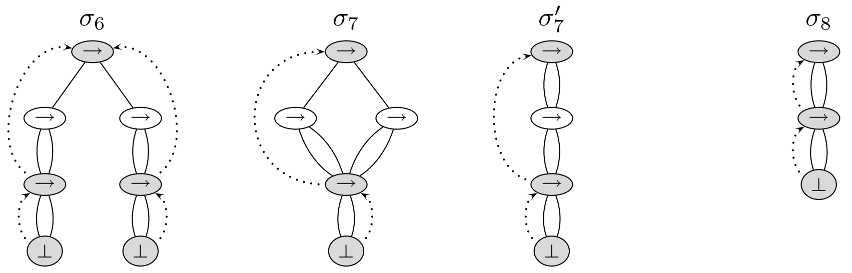


Figure 3.4.3 – Merging in System $\mathcal{F}$

Let us now consider the merging of congruent nodes. As in System F, the nodes must be locally congruent, so as not to merge nodes indirectly—thus potentially losing some polymorphism. Of course, this includes not indirectly merging variables. However, in System $\mathcal{F}$, this is also the case for inner nodes with binding edges. Indeed, by merging two inner nodes, we lose the ability to instantiate the subgraphs under them in incompatible ways.

▶ **Example** It is not possible to directly merge the nodes $\langle 1 \rangle$ and $\langle 2 \rangle$ in the type σ_6 of Figure 3.4.3: those two nodes are not locally congruent, as the nodes $\langle 12 \rangle$ and $\langle 21 \rangle$ are bound above them.

Thus, as in System $\mathcal{F}$, we only allow the merging of locally congruent nodes. Moreover, the merging is reversible if and only the two nodes at the root of the merging are unbound, *i.e.* if we only alter the representation of types, not their meaning from a semantic standpoint. Finally, for the usual soundness-related reasons, the merging of bound nodes is only permitted if they are green.

▶ **Example *(continued)*** Let us consider again the merging of $\langle 1 \rangle$ and $\langle 2 \rangle$ in σ_6. Although a direct merging is impossible, it is possible to start by merging $\langle 11 \rangle$ and $\langle 21 \rangle$, as they are both green and locally congruent. The resulting type is σ_7, which can now be (reversibly) instantiated into σ'_7.

As mentioned above, the step $\sigma_6 \sqsubseteq_{\mathcal{F}} \sigma_7$ is not reversible. Indeed, even though it merges inner nodes (and not variables), those nodes are green, and allow the instantiation of the nodes bound on them. In this case, the difference in the amount of sharing *can* be observed. Let us illustrate this by showing that splitting green nodes would be unsound. For the sake of simplicity, we use a slightly simpler example. Consider the type σ_8 of Figure 3.4.3, which verifies

$$\sigma_3 \sqsubseteq_{\mathcal{F}} \sigma_4 \sqsubseteq_{\mathcal{F}} \sigma_8$$

(σ_8 being the result of merging $\langle 1\rangle$ and $\langle 2\rangle$ in σ_4). Thus σ_8 is a valid type for K', since σ_3 is the principal type of K'.

Moreover, σ_8 is actually the principal type of the term choose id. In this case, it is crucial for the two occurrences of σ_{id} on each side of the arrow to be correlated: choose id could potentially return its argument. Thus, if σ_8 is a valid type for this term, σ_4 is not; indeed

$$\sigma_4 \sqsubseteq_{\mathcal{F}} (\mathsf{int} \to \mathsf{int}) \to (\mathsf{char} \to \mathsf{char})$$

holds, and the latter type is not valid for choose id. Hence $\sigma_8 \sqsubseteq_{\mathcal{F}} \sigma_4$ cannot hold, as it would be unsound.

3.4.2.5 Inert nodes

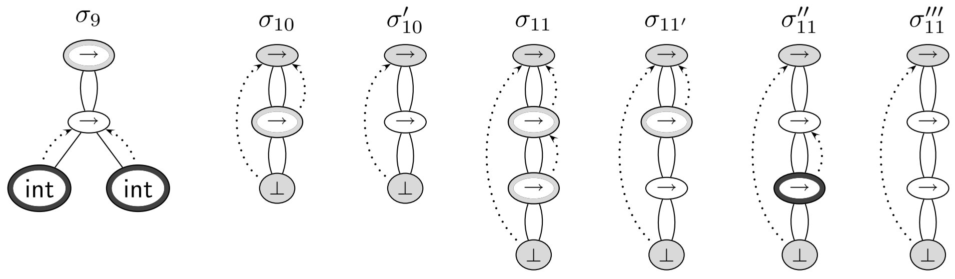


Figure 3.4.4 – Inert nodes

Perhaps surprisingly, not all binding edges are semantically meaningful. As a simple example, consider the nodes $\langle 11\rangle$ and $\langle 12\rangle$ of the type σ_9 of Figure 3.4.4. Binding them brings no additional expressivity to the type, as they are labelled by the monomorphic type int, which contains no polymorphism. Thus, even though they are red, it is safe to weaken them, and we allow this transformation. Moreover, this operation is reversible.

As a slightly more involved example, consider the type σ_{10} of this figure. Even though $\langle 1\rangle$ is bound, no node is bound on it. Thus, the presence or the absence of the binding edge does not allow taking semantically different instances.[6] In particular, given a term a of type σ_{10}, any term a' such that $a\ a'$ is well-typed can also be soundly applied to a term

[6]Of course, this is true only because no instance operation allows introducing a variable bound on $\langle 1\rangle$. However, this property is true in System $\mathcal{F}$.

of type σ'_{10}—and conversely. Moreover, in both cases, the return type is the same: the type of a'.

This means that we should not distinguish these two types for type soundness; therefore, they must be in relation by the reversible part of the instance relation. To do so, we allow the weakening of $\langle 1 \rangle$; furthermore, unlike proper weakenings—that really change the semantics of the type by requiring different amount of polymorphism—this operation is reversible.

As a last example, consider σ_{11}. We can slightly generalize the reasoning above, by considering applications of the form $a\ a'\ a''$. This shows that the binding edge of the node $\langle 11 \rangle$ is not semantically meaningful. In turn, σ'_{11} shows that the one of $\langle 1 \rangle$ is also unimportant. Thus, we allow both the weakenings of $\langle 1 \rangle$ and $\langle 11 \rangle$ in σ_{11} (represented by σ'_{11} and σ''_{11}) as reversible operations. Notice that weakening $\langle 1 \rangle$ transforms $\langle 11 \rangle$ into a red node: nevertheless, when a node does not permit semantically meaningful instantiations, its color is actually unimportant, as it can be soundly weakened.

Finally, generalizing one more step, it is actually safe to add or remove a binding edge on all the nodes that will never permit instantiating a variable, which we call *inert*. This also means that it is safe to raise or merge those nodes.

Definition 3.4.1 (*Inert nodes)* A bound node n of a type σ is *inert* if there is no variable transitively bound on it. ■

In the figures of this section, all hollow-colored nodes were inert.

3.4.2.6 Summary

Let us summarize what operations are in the instance relation of System $\mathcal{F}$, as well as those that are reversible.

Definition 3.4.2 (*Instance in System $\mathcal{F}$)* The instance relation of System $\mathcal{F}$, written $\sqsubseteq_{\mathcal{F}}$, is the transitive reflexive closure of the relation defined by the following atomic instance operations:

- grafting a closed type under a green bottom node;
- raising a green or inert node;
- merging two green, inert, or unbound nodes that are locally congruent;
- weakening a green or inert node.

The operations on unbound and inert nodes are reversible, and we write $\approx_{\mathcal{F}}$ the transitive symmetric reflexive closure of this relation. We write $\sqsubseteq^{\approx}_{\mathcal{F}}$ the instance modulo similarity relation of System $\mathcal{F}$, defined as $(\sqsubseteq_{\mathcal{F}} \cup \approx_{\mathcal{F}})^*$. ■

▶ **Examples** The various instances that hold in the figures of this section are summarized in Figure 3.4.5. Plain edges represent $\sqsubseteq_{\mathcal{F}}$, while dashed ones represent its reversible subset; all transitive or reflexive edges have been omitted to simplify the drawing. For example, $\sigma'_{11} \approx_{\mathsf{F}} \sigma''_{11}$ holds, since $\sigma'_{11}, \sigma''_{11} \sqsubseteq_{\mathcal{F}} \sigma'''_{11}$ hold, and both instances are reversible.

Importantly, $\sqsubseteq^{\approx}_{\mathcal{F}}$ extends $\sqsubseteq^{\approx}_{\mathsf{F}}$:

Lemma 3.4.3 *Consider two System* F *types σ and σ'. If $\sigma \sqsubseteq^{\approx}_{\mathsf{F}} \sigma'$, then $\sigma \sqsubseteq^{\approx}_{\mathcal{F}} \sigma'$.* □

$$\begin{tikzcd}
 & \sigma_2 & & & & & & \\
\sigma_3 \arrow[ur] \arrow[r] \arrow[ddr] & \sigma'_1 \arrow[rr, dashed] & & \sigma_1 \arrow[r] & \sigma_6 \arrow[r] & \sigma_7 \arrow[dr, dashed] & & \\
 & & & & & & \sigma'_7 \arrow[ddr] & \\
 & \sigma_4 \arrow[r] \arrow[ddr] & \sigma_5 \arrow[r, dashed] \arrow[ddr] & \sigma'_5 \arrow[uur] \arrow[r] & \sigma'_{10} \arrow[urr] & & & \\
 & & & & & & \sigma''_{11} \arrow[r, dashed] & \sigma'''_{11} \\
 & & \sigma_8 \arrow[r] & \sigma_{10} \arrow[uur, dashed] \arrow[rr] & & \sigma_{11} \arrow[ur, dashed] \arrow[dr, dashed] & & \\
 & & & & & & \sigma'_{11} \arrow[uur, dashed] &
\end{tikzcd}$$

Figure 3.4.5 – Instances in the examples of §3.4

Proof: All atomic operations of $\sqsubseteq^{\approx}_{\mathsf{F}} \sigma'$ are in $\sqsubseteq^{\approx}_{\mathcal{F}}$, except for grafting which can be simulated by grafting, raising and weakening.

3.4.3 An informal semantics for the types of System $\mathcal{F}$

A syntactic presentation of a restriction of System $\mathcal{F}$, which we call *Shallow-$\mathcal{F}$* has been studied in detail by Le Botlan and Rémy (2007, §3). More precisely, the instance relations of the two systems are the same, but the types of Shallow-$\mathcal{F}$ are a restriction of those of System $\mathcal{F}$ (modulo the translation into syntactic types):

Definition 3.4.4 (*Shallow types)* A System $\mathcal{F}$ type σ is *shallow* if all its red nodes are variables. ■

In other words, shallow types disallow flexible quantification of inner nodes in red positions: projecting a type on an unbound node results in a System F type. This means that types are stratified, with flexible quantification at the top, and System F types at the bottom. By contrast, in System $\mathcal{F}$ alternating unbound and bound non-variables nodes is possible.

▶ **Example** The type σ_{s} of Figure 3.4.6 is shallow, but σ_{ns} is not: the projection of σ_{ns} at the unbound node $\langle 11 \rangle$ is not an F type.

From a theoretical point of view, Shallow-$\mathcal{F}$ is an interesting restriction of System $\mathcal{F}$, as it is possible to give a semantics to Shallow-$\mathcal{F}$ types as a set of System F type. We refer to (Le Botlan and Rémy 2007, §3.2) for the exact definition, and will only give the general idea below (adapting it to graphic types):

Definition 3.4.5 (*Informal semantics of shallow types)* Let σ be a shallow type. The semantics $\{\!\{n\}\!\}$ of a green node n of σ is the set of F-types recursively defined by:

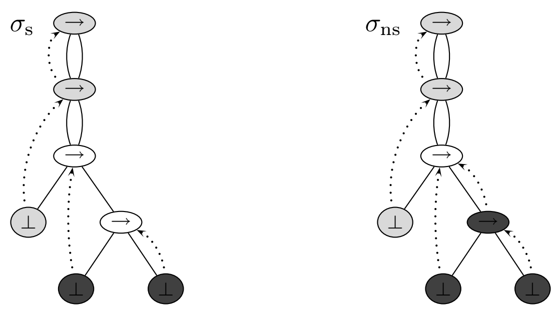


Figure 3.4.6 – Shallow and non-shallow types

- if n is a bottom node, $\{\{n\}\}$ is the set of all System F types;
- if n is a non-variable node n with $n_1, \dots, n_k$ nodes bound at n, $\{\{n\}\}$ is the closure by $\sqsubseteq_{\mathsf{F}}^{\approx}$ of the set of (graphic System F) types obtained by replacing each n_i with an instance of a type in the semantics of n_i.

The semantics of σ is the semantics of its root. Two shallow types σ and σ' are in semantic instance relation if $\{\{\sigma'\}\} \subseteq \{\{\sigma\}\}$ holds. ■

It is proven by Le Botlan and Rémy (2007) that the syntactic $\mathsf{ML^F}$ relation $\sqsubseteq^{\boxminus}$, which corresponds to the relation $\sqsubseteq_{\mathsf{F}}^{\approx}$ in System $\mathcal{F}$, is sound w.r.t. to the semantic instance relation defined above. Completeness is conjectured.

▶ **Examples** Let us write F for the set of all syntactic System F types (including types with free variables). Using the semantics above, it is possible to prove that $\{\{\sigma_3\}\}$ and $\{\{\sigma_8\}\}$ (*i.e.* the principal types for $\lambda(x)\ \lambda(y)\ y$ and choose id) are the types of the form

$$\{\forall\overline{\alpha}.\ \sigma_1 \to (\forall\overline{\beta}.\ \sigma_2 \to \sigma_2) \mid \sigma_1, \sigma_2 \in \mathsf{F}\} \qquad \text{and} \qquad \{\forall\overline{\alpha}.\ (\forall\overline{\beta}.\ \sigma \to \sigma) \to (\forall\overline{\beta}.\ \sigma \to \sigma) \mid \sigma \in \mathsf{F}\}$$

Thus flexible quantification captures the properties System F lacks:

- for $\lambda(x)\ \lambda(y)\ y$, the type variables in the instance of σ_{id} can be bound either at the root of the type, or under the arrow;
- for choose id, the two instances of σ_{id} on both sides of the arrow are the same.

▶ **Example: semantics of raising** As a more complex example, the semantics of the type σ_{s} of Figure 3.4.6 is the set of closed types of the form

$$\forall\overline{\alpha}.\ (\forall\overline{\gamma}.\ \sigma_{11} \to \sigma_{11}) \to (\forall\overline{\gamma}.\ \sigma_{11} \to \sigma_{11})$$

where σ_{11} is in the semantics of $\langle 11\rangle$, *i.e.* in the set

$$\left\{\forall\overline{\beta}.\ \sigma \to (\forall\overline{\delta}.\ \sigma' \to \sigma'') \mid \sigma, \sigma', \sigma'' \in \mathsf{F},\ \mathsf{ftv}(\sigma) \# \overline{\beta},\ \mathsf{ftv}(\sigma') \# \overline{\delta}\right\}$$

Notice the restrictions on the variables that can occur in σ and σ'. This is due to the fact that $\langle 111 \rangle$ is bound above $\langle 11 \rangle$, (hence σ cannot refer to $\overline{\beta}$). Similarly, $\langle 1121 \rangle$ is bound above $\langle 112 \rangle$ and σ' cannot use a variable of $\overline{\delta}$. Of course, the four occurrences of σ_{11} are the same.

Let us now consider the semantics of the type obtained by raising $\langle 111 \rangle$ in σ_{s}. In this case, the free variables of σ in σ_{11} can only be quantified at the root, and the semantics is now of the form

$$\forall \overline{\alpha}.\ (\sigma_{11} \rightarrow \sigma_{11}) \rightarrow (\sigma_{11} \rightarrow \sigma_{11})$$

with σ_{11} ranging over the same set as previously.

The difference between the shallow and non-shallow versions of ML^{F} are discussed further in Appendix A.

4
$\mathsf{ML^F}$ graphic types

Abstract

In order to obtain $\mathsf{ML^F}$ graphic types, we add rigid quantification to the types of System $\mathcal{F}$ (§4.1). We characterize those types as the superposition of a first-order skeleton and of a binding tree (§4.2), and isolate the graphs that are well-scoped (§4.3). Finally we present a few operators to transform $\mathsf{ML^F}$ graphic types (§4.4).

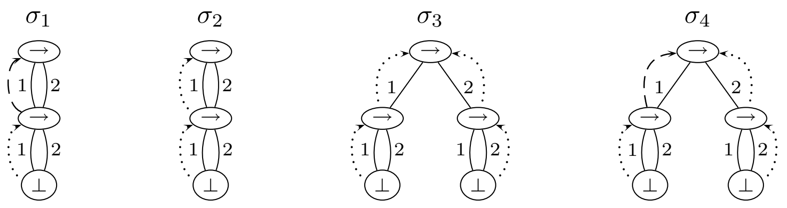
The syntactic presentation of $\mathsf{ML^F}$ types includes the flexible quantification seen in the previous chapter. However, polymorphism is requested through the use of rigid quantification, and not by removing binding edges, as in System $\mathcal{F}$. We follow the same approach in graphic $\mathsf{ML^F}$ types, and use a second type of binding edge. However we will not develop the reasons behind this choice here, and postpone the explanations to §5. This chapter instead focuses on the formal definition of graphic types.

4.1 Representing $\mathsf{ML^F}$ graphic types

From a representational standpoint, the main difference between (graphic) System $\mathcal{F}$ and $\mathsf{ML^F}$ types is the introduction of a new kind of binding edge for rigid quantification; in drawings we use dashed lines.[1]

▶ **Example** Consider the types σ_1, σ_2, σ_3, and σ_4 of Figure 4.1.1. The node $\langle 1 \rangle$ is rigidly quantified in both σ_1 and σ_4, and flexibly quantified in σ_2 and σ_3.

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



$$\sigma_1 = \forall\,(\alpha = \sigma_{\mathsf{id}})\ \alpha \to \alpha \qquad \sigma_3 = \forall\,(\alpha \geqslant \sigma_{\mathsf{id}})\ \forall\,(\beta \geqslant \sigma_{\mathsf{id}})\ \alpha \to \beta$$
$$\sigma_2 = \forall\,(\alpha \geqslant \sigma_{\mathsf{id}})\ \alpha \to \alpha \qquad \sigma_3 = \forall\,(\alpha = \sigma_{\mathsf{id}})\ \forall\,(\beta \geqslant \sigma_{\mathsf{id}})\ \alpha \to \beta$$

Figure 4.1.1 – Examples of graphic $\mathsf{ML^F}$ types.

4.1.1 From syntactic to graphic

The four types above are actually the graphic representation of the types introduced in §2.3.3 and whose definition is recalled in the figure. For readers familiar with the syntactic presentation of $\mathsf{ML^F}$, we describe here how to translate a quantified type $\forall\,(\alpha \diamond \sigma)\ \sigma'$; the full algorithm is given in §8.2.2, but uses a few notations not yet introduced:

1. translate σ' as if α was a variable;
2. translate σ;
3. replace the node corresponding to α in σ' by σ;
4. bind that node to the root of σ' according to $\diamond$.

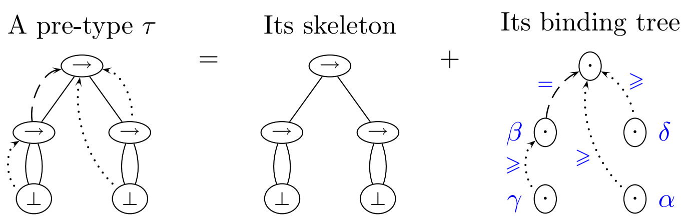
▶ **Example** The graph representing σ_3 contains at the node $\langle 1\rangle$ a subgraph representing the bound σ_{id} of the variable α, and it is bound by a flexible edge.

▶ **Another example** The syntactic definition of the graphic type τ on the left of Figure 4.2.1 is given at the bottom of the figure. The node $\langle 11\rangle$ corresponds to the variable γ. This variable is flexibly quantified at the level of β, which is represented by the node $\langle 1\rangle$; hence there is a flexible binding edge from $\langle 11\rangle$ to $\langle 1\rangle$. Similarly, β is rigidly quantified at the toplevel of τ, hence the rigid edge from $\langle 1\rangle$ to $\langle \epsilon\rangle$.

The structure of the underlying graph of τ can also be read directly in the syntactic $\mathsf{ML^F}$ type. For example, the equation for the root of τ is the rightmost part of the syntactic definition of τ, *i.e.* $\beta \to \delta$. Likewise, the equation for the node corresponding to δ is $\alpha \to \alpha$, as indicated by the bound $\forall\,(\beta \geqslant \alpha \to \alpha)$.

Syntactic and graphic sharing $\mathsf{ML^F}$ syntactic quantification $\forall\,(\alpha \diamond \tau)\ \tau$ is used in particular to denote sharing. In graphs, it is directly captured by the intrinsic sharing of dags—hence our use of this representation. In both $\mathsf{ML^F}$ and System $\mathcal{F}$, the type σ_3 of Figure 4.1.1, in which the two occurrences of σ_{id} may be instantiated separately, is quite different from σ_2, 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 copies of the graph representing σ_{id} in σ_3, but only one in σ_2.

4.2 Pre-types



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

Figure 4.2.1 – Decomposition of an MLF graphic type

The formal definition of MLF graphic types is given in two steps. We start by defining a set of graphs that contains all graphic types. Afterwards, we give a criterion characterizing graphic types as well-scoped graphs (§4.3).

Definition 4.2.1 A *(graphic) pre-type* τ is a triple composed of:

1. A first-order anonymous term-graph $\breve{\tau}$, called the *skeleton* of τ.
2. A set of binding edges $\hat{\tau}$, that forms an upside-down tree of domain $\mathsf{dom}(\breve{\tau})$ rooted at $\langle\epsilon\rangle$.
3. A set of binding flags for all the nodes of $\breve{\tau}$ but the root, *i.e.* a function $\overset{\diamond}{\tau}$ mapping each node in $\mathsf{dom}(\breve{\tau}) \setminus \{\langle\epsilon\rangle\}$ to one of the *binding flags* $\geqslant$ or $=$.[2]

The union of $\hat{\tau}$ and $\overset{\diamond}{\tau}$ is called the *binding tree* of τ. ■

The term-graph $\breve{\tau}$ is the structure of the pre-type. It is first-order: all the information related to binders, in particular where and how each node is bound, is contained in the binding tree.

▶ **Example** The decomposition of the pre-type τ of Figure 4.2.1 is given in the same figure. For the binding tree, we have exceptionally annotated the nodes of the binding tree with the name of the corresponding syntactic type variable, and the binding edges with the binding flags of the nodes they correspond to.

Notations In the text, we write $n \multimap n' \in \tau$ (*resp.* $n \rightarrowtriangle n' \in \tau$) to mean that there is a structure edge (*resp.* a binding edge) from n to n' in τ. We often drop "$\in \tau$" when τ is clear from the context. Notice that $\multimap$ arrows are downwards oriented, while $\rightarrowtriangle$ ones are upwards oriented. If π is a path, we write $n \overset{\pi}{\multimap} n' \in \tau$ to denote the fact there exists a (structure) path π from n to n' in $\breve{\tau}$, *i.e.* that $n' = \langle n \cdot \pi\rangle$. We write $n \overset{b}{\rightarrowtriangle} n'$ if $n \rightarrowtriangle n'$ and $\overset{\diamond}{\tau}(n) = b$. If $n \rightarrowtriangle n' \in \tau$, we also write $\hat{\tau}(n)$ (or simply $\hat{n}$) for n'; we call n' the *binder* of n and we say that n is bound at n'.

[2]While we have chosen to attach the binding flags to the nodes instead of to the binding edges, this is purely a matter of convention, as both views are isomorphic. Our definition allows us to define some operations on pre-types in a simpler way, as $\hat{\tau}$ and $\overset{\diamond}{\tau}$ sometimes change independently.

We write $\dot{\tau}$ and $\tilde{\tau}$ for the term and equivalence defining $\breve{\tau}$, and $\tau(n)$ for $\dot{\tau}(n)$, *i.e.* the symbol on the node n. We use the notations $\overset{\multimap}{\overline{\tau_{foo}}}$, $\overrightarrow{\tau_{foo}}$ and $\overset{\diamond}{\overline{\tau_{foo}}}$ instead of $\breve{\tau}_{foo}$, $\hat{\tau}_{bar}$ and $\overset{\diamond}{\tau}_{foo}$ for wide arguments.

▶ **Examples** Consider the pre-types of Figure 4.3.1. We have $\langle 1\rangle \xrightarrow{=} \{\epsilon\} \in \tau_1$. Hence, the binder of $\langle 1\rangle$ is $\hat{\tau_1}\langle 1\rangle = \{\epsilon\}$, and $\overset{\diamond}{\tau_1}(\langle 1\rangle)$ is $=$. We also have $\langle\epsilon\rangle \multimap \langle 1\rangle \multimap \langle 12\rangle \in \tau_2$ or, leaving τ_2 implicit, $\langle\epsilon\rangle \overset{12}{\multimap} \langle 12\rangle$.

4.2.1 Why binding all nodes

The definition of graphic types implies that all nodes but the root have a binding edge. From a theoretical standpoint, some nodes, such as those labelled with ground types like int, need not be bound. However, this makes reasoning on the meta-theoretical properties of graphic types more complicated. Instead, we require that all nodes be bound, and define the instance relation so that the additional binding edges can be freely transformed (§5.3.4). Then we prove that those supplementary edges are unimportant, as they “commute” with type inference in a certain sense (§13.2). Finally, when translating graphic types into syntactic types, for example for display purposes, those edges are entirely removed (§8.3.3).

4.3 Well-formedness of graphic types

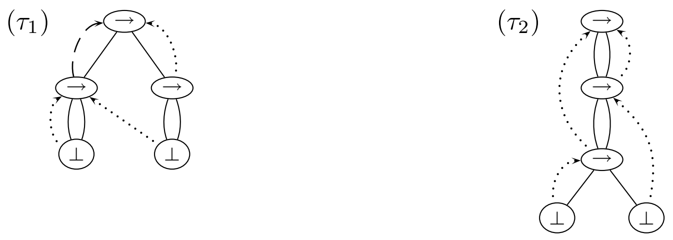


Figure 4.3.1 – Invalid graphic types.

4.3.1 Well-formed pre-types

All pre-types are not well-formed types. Indeed, graphic types must have a binding tree compatible with the lexical scoping of variables. Two ill-formed binding trees are presented in Figure 4.3.1:

1. In the pre-type τ_1, the node $\langle 21\rangle$ is bound at a node that is not among its ancestors. This is not permitted; in a syntactic presentation, the variable would be bound on the left branch and used on the right branch, out of its scope.
2. In the pre-type τ_2, the bounds of the nodes $\langle 1\rangle$ and $\langle 11\rangle$ both depend on the bound of the other node:

- if $\langle 1\rangle$ is bound first: the equation of its bound is $\langle 11\rangle \to \langle 11\rangle$, but $\langle 11\rangle$ has not been bound yet;
- if $\langle 11\rangle$ is bound first: its bound is $\langle 111\rangle \to \langle 112\rangle$, which refers to the node $\langle 112\rangle$. This node is itself bound at $\langle 1\rangle$, which has not been bound yet.

In both cases, a variable is used outside of its scope. The second example is merely a generalization of the first one to graphs with internal quantification and internal sharing of nodes. The invariant that variables are used in their scope is generally captured by the notion of *domination*: a bound must be dominated by its binder. The very same invariant exists for the graphic types of $\mathsf{ML^F}$, up to the fact that we must take into account binding edges.[3]

Definition 4.3.1 (*Mixed paths)* Let $\mathrel{\lhd\!\!\multimap}$ be the relation $(\multimap) \cup (\leftarrowtriangle)$. Given some nodes $n_1, ..., n_k$, we say that the sequence $n_1 \mathrel{\lhd\!\!\multimap} n_2 \ldots \mathrel{\lhd\!\!\multimap} n_{k-1} \mathrel{\lhd\!\!\multimap} n_k$ is a *mixed path* from n_1 to n_k; this path is said to *contain* n if $n = n_i$ for some $1 \le i \le k$. ■

▶ **Example** In the pre-type τ_2 of Figure 4.3.1, the relations $\{\epsilon\} \leftarrowtriangle \langle 11\rangle \overset{2}{\multimap} \langle 112\rangle$ hold, and form a mixed path from $\{\epsilon\}$ to $\langle 112\rangle$.

Definition 4.3.2 (*Domination for* $\mathrel{\lhd\!\!\multimap}$*)* Let τ be a pre-type, n and n' two nodes of τ. We say that n dominates n' and we write $n \mathrel{\lhd\!\!\gg\!\!\multimap} n'$ if every mixed path from the root to n' contains n. ■

▶ **Example *(continued)*** Consider again the pre-type τ_1 of Figure 4.3.1; the mixed paths between $\{\epsilon\}$ and $\langle 11\rangle$ are

$$\begin{array}{ll}
\{\epsilon\} \overset{1}{\multimap} \langle 1\rangle \overset{1}{\multimap} \langle 11\rangle & \qquad \{\epsilon\} \leftarrowtriangle \langle 1\rangle \overset{1}{\multimap} \langle 11\rangle \\
\{\epsilon\} \overset{1}{\multimap} \langle 1\rangle \overset{2}{\multimap} \langle 11\rangle & \qquad \{\epsilon\} \leftarrowtriangle \langle 1\rangle \overset{2}{\multimap} \langle 11\rangle \\
\{\epsilon\} \overset{1}{\multimap} \langle 1\rangle \leftarrowtriangle \langle 11\rangle & \qquad \{\epsilon\} \leftarrowtriangle \langle 1\rangle \leftarrowtriangle \langle 11\rangle
\end{array}$$

All six 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 $\{\epsilon\} \overset{2}{\multimap} \langle 2\rangle \overset{1}{\multimap} \langle 21\rangle$. Similarly, in τ_2, $\langle 1\rangle$ does not dominate $\langle 112\rangle$, since $\{\epsilon\} \leftarrowtriangle \langle 11\rangle \overset{2}{\multimap} \langle 112\rangle$.

Well-formed types are simply the pre-types in which the binder of a node dominates the node for the relation $\mathrel{\lhd\!\!\gg\!\!\multimap}$.

Definition 4.3.3 (*Graphic types)* The binding tree of a pre-type τ is *well-dominated* if every bound node is dominated by its binder, *i.e.* for all n and n' in τ, $n \rightarrowtriangle n'$ implies $n' \mathrel{\lhd\!\!\gg\!\!\multimap} n$. A *(graphic) type* is a pre-type whose binding tree is well-dominated. ■

In the following, we will nearly always consider $\mathsf{ML^F}$ graphic types and, unless specified otherwise, we abbreviate "graphic type" by "type". The metavariable τ is used to range over types.

[3]Or, more generally, for the types of System F and $\mathcal{F}$ in §3.3 and §3.4.

▶ **Example *(continued)*** As seen in the example above, neither τ_1 nor τ_2 are types, as they are not well-dominated. In particular, $\hat{\tau_1}(\langle 21\rangle)$ does not dominate $\langle 21\rangle$ in τ_1 and $\hat{\tau_2}(\langle 112\rangle)$ does not dominate $\langle 112\rangle$ in τ_2. Conversely, one can check that the pre-type τ in Figure 4.2.1 is well-dominated.

4.3.2 Invariants induced by well-formedness

Well-domination is a fairly strong property, and it creates several invariants relating the structure and the binding tree of a type. We give one of them below, which we often use inside proofs.

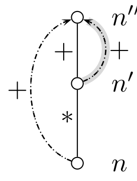


Figure 4.3.2 – Invariant induced by well-domination

Lemma 4.3.4 *For any type τ, if $n \xrightarrow{+}\!\triangleright n'' \xrightarrow{+}\!\circ n' \xrightarrow{*}\!\circ n$, then $n' \xrightarrow{+}\!\triangleright n''$.* □

This lemma is shown graphically in Figure 4.3.2, the conclusion being the highlighted edge. Notice in particular that, by well-domination, n'' dominates n'.

<u>Proof</u>: The proof is by induction on the integer k such that $n\ (\longrightarrow\!\triangleright)^k\ n''$.

▷ <u>*Case $k = 1$*</u>: since $n' \xrightarrow{*}\!\circ n$, there exists a mixed path P of the form $\langle\epsilon\rangle \triangleleft\!\xleftarrow{+} n' \xrightarrow{*}\!\circ n$. By well-domination, $\hat{n}$ (which is also n'' in this subcase) is in P. Since by hypothesis n'' is strictly above n', n'' is in the subpath $\langle\epsilon\rangle \triangleleft\!\xleftarrow{+} n'$ (and is not n'). This proves $n' \xrightarrow{+}\!\triangleright n''$, which is the desired result.

▷ <u>*Case $k = k' + 1 > 1$*</u>:
Let n''' be the node such that $n \xrightarrow{+}\!\triangleright n''' \longrightarrow\!\triangleright n''$. Consider a mixed path $\{\epsilon\} \xrightarrow{*}\!\circ n'' \xrightarrow{+}\!\circ n' \xrightarrow{*}\!\circ n$. By iterating the well-domination property, this path must contain n'''. Since $n''' \longrightarrow\!\triangleright n''$, n''' is strictly under n''. We compare the relative positions of n''' and n' in the path $n'' \xrightarrow{+}\!\circ n' \xrightarrow{*}\!\circ n$.

- <u>*If $n''' = n'$*</u>: the result is proven, as $n''' \longrightarrow\!\triangleright n''$ by hypothesis.
- <u>*If n''' is strictly between n'' and n'*</u>: the conclusion is by induction hypothesis applied to $n\ (\longrightarrow\!\triangleright)^{k'}\ n''' \xrightarrow{+}\!\circ n' \xrightarrow{*}\!\circ n$.
- <u>*If n''' is strictly between n' and n*</u>: the conclusion is by induction hypothesis applied to $n'''\ (\longrightarrow\!\triangleright)^1\ n'' \xrightarrow{+}\!\circ n' \xrightarrow{*}\!\circ n'''$.

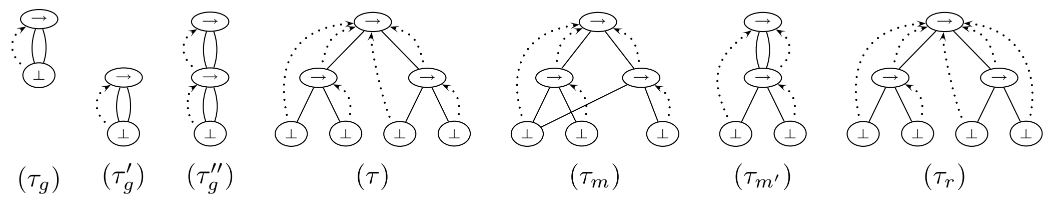


Figure 4.4.1 – Operations on graphs

4.4 Operators for building and transforming types

We conclude this chapter by defining a few operators to transform graphic types. Most of them closely follow the instance operations of System $\mathcal{F}$. However, the definitions of this section do not take into account the fact that the operation is sound, as this point will be treated in §5.

4.4.1 Grafting

We write $\tau[\tau'/n]$ for the replacement of a bottom node n of a type τ by a type τ'; the resulting type is described by τ' for nodes below n and by τ for the other nodes.

▶ **Example** In Figure 4.4.1, grafting the type τ'_g at the node $\langle 1\rangle$ in the type τ_g results in the type τ''_g.

Definition 4.4.1 (*Grafting)* The *grafting* $\tau[\tau'/n]$ of a type τ' at a node n in a type τ is defined by:

- $\overline{\tau[\tau'/n]}$ 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 $n \cdot \tilde{\tau}'$ is the set of pairs $(n \cdot m, n \cdot m')$ for m and m' verifying $m \;\tilde{\tau}'\; m$;
- $\overset{\diamond}{\overline{\tau[\tau'/n]}}$ maps m in $\mathsf{dom}(\overset{\diamond}{\tau})$ to $\overset{\diamond}{\tau}(m)$ and nm to $\overset{\diamond}{\tau}'(m)$ for all $m \in \mathsf{dom}(\overset{\diamond}{\tau}')$;
- $\overrightarrow{\tau[\tau'/n]}$ is $\hat{\tau}$ extended with the edges $nm \longrightarrow nm'$ for all $m \longrightarrow m' \in \tau'$. ■

Property 4.4.2 *Given two types τ and τ', and n a bottom node of τ, $\tau[\tau'/n]$ is a type.* □

Proof: Let us call τ'' the grafting $\tau[\tau'/n]$. The fact that τ'' is a pre-type is immediate, and it remains to prove that τ'' is well-dominated. Let n' be a node of τ''.

▷ If $n' \in \mathsf{dom}(\tau)$, all the mixed paths from $\langle\epsilon\rangle$ to n' are mixed paths in τ. We conclude by well-domination of τ.

▷ Otherwise, n' is of the form $n \cdot n''$. By construction of τ'', all mixed paths from $\langle\epsilon\rangle$ to n'' are the concatenation of a mixed path from $\langle\epsilon\rangle$ to n in τ and of a mixed path from $\langle\epsilon\rangle$ to n'' in τ'. The conclusion is thus by well-domination of τ'.

4.4.2 Projection

Definition 4.4.3 (*Closed nodes)* A node n is *closed* if all the nodes in the subgraph under n are transitively bound at n, *i.e.* if $n \overset{+}{\multimap} n'$ implies $n' \overset{+}{\rightarrowtriangle} n$. ■

Given a closed node n, we write τ/n for the projection of τ at n, obtained by removing all the nodes not under n and all the dangling edges, and renaming nodes accordingly (thus making n the root node of the resulting graph).

▶ **Example** In Figure 4.4.1, projecting at the node $\langle 1\rangle$ in τ''_g yields the type τ'_g. Projecting at $\langle 1\rangle$ 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).

Definition 4.4.4 (*Projection)* The projection τ/n of a type τ at a closed node n of τ is defined by:

- $\overset{\cdot}{\overline{\tau/n}}$ is $\dot{\tau}/n$.
- $\widetilde{\tau/n}$ is such that $\pi\ \widetilde{\tau/n}\ \pi'$ if and only if $n\pi\ \tilde{\tau}\ n\pi'$.
- $\overset{\diamond}{\overline{\tau/n}}$ maps a node m to $\overset{\diamond}{\tau}(nm)$
- $\overrightarrow{\tau/n}$ is defined by $m \rightarrowtriangle m' \in \tau/n$ if and only if $nm \rightarrowtriangle nm' \in \tau$. ■

Property 4.4.5 *Let n be a closed node of a type τ. The projection τ/n is a type.* □

Proof: Let τ' be τ/n. The fact that τ' is a pre-type is immediate. For domination, consider a node n' of τ' and a mixed path P from $\langle\epsilon\rangle$ to n'. Let P' be a mixed path from $\langle\epsilon\rangle$ to n in τ. By well-domination of τ, $\hat{n}'$ is in $P' \cdot P$. Since n is closed, $\hat{n}'$ cannot be contained in P'; hence it is in P. This is the desired result.

4.4.3 Fusion

Fusion is the generalization of the fusion operation on term-graphs to graphs with binding edges. We formalize the fact that two nodes are congruent in such a graph by the following definition.

Definition 4.4.6 (*Binding-congruent nodes)* Consider two nodes n_1 and n_2 of a type τ congruent in $\check{\tau}$. Let $\sim'$ be $\check{\tau}[\widetilde{n_1 = n_2}]$. Then n_1 and n_2 are *binding-congruent* in τ if

$$\forall n, \forall n',\ n \sim' n' \implies \wedge\begin{cases} \hat{\tau}(n) \sim' \hat{\tau}(n') \\ \overset{\diamond}{\tau}(n) = \overset{\diamond}{\tau}(n') \end{cases}$$ ■

The first condition asserts that binding edges can be merged, while the second checks that this is also the case for binding flags. The fusion operation is possible on binding-congruent nodes, and merges them.

▶ **Example** In Figure 4.4.1, 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 $\tau_{m'}$. Notice that the binding edges $\langle 11\rangle \mathrel{\longrightarrow\!\!\triangleright} \{\epsilon\}$ and $\langle 21\rangle \mathrel{\longrightarrow\!\!\triangleright} \{\epsilon\}$ of τ are fused in $\tau_{m'}$, as a side-effect of fusing $\langle 1\rangle$ and $\langle 2\rangle$.

Definition 4.4.7 (*Fusion*) The fusion $\tau[n_1 = n_2]$ of two binding-congruent nodes n_1 and n_2 of τ is defined by

- $\overset{\longrightarrow\!\circ}{\tau[n_1 = n_2]}$ is $\breve{\tau}[n_1 = n_2]$;
- $\overset{-\!\diamond\!-}{\tau[n_1 = n_2]}$ is the quotient of $\overset{\diamond}{\tau}$ by $\tau[\widetilde{n_1 = n_2}]$
- $\overset{\longrightarrow\!\!\triangleright}{\tau[n_1 = n_2]}$ is the quotient of $\hat{\tau}$ by $\tau[\widetilde{n_1 = n_2}]$ ■

The result below (which we afterwards use to prove that fusion returns types) expresses that fusion preserves domination.

Lemma 4.4.8 *Let τ be a type, n_1 and n_2 two binding-congruent nodes of τ. Let τ' be $\tau[n_1 = n_2]$. Let $\mathrel{-\!\!\gg\!\!\rightarrow}$ be the domination relation corresponding to either $\mathrel{\triangleleft\!\!-\!\!-}$, $\mathrel{-\!\!-\!\circ}$ or $\mathrel{\triangleleft\!\!-\!\!-\!\circ}$. For any n and n' of τ if $n \mathrel{-\!\!\gg\!\!\rightarrow} n'$ in τ, then $n \mathrel{-\!\!\gg\!\!\rightarrow} n'$ in τ'.* □

Proof: Consider a mixed path P' between $\{\epsilon\}$ and n' in τ'. We rewrite the nodes in P by removing all the occurrences of the paths under n_2 (for example $\{\epsilon\} \mathrel{\triangleleft\!\!-\!\!-} n_1 \cup n_2 \mathrel{\triangleleft\!\!-\!\!-} (n_1 \cup n_2) \cdot P'$ is rewritten into $\{\epsilon\} \mathrel{\triangleleft\!\!-\!\!-} n_1 \mathrel{\triangleleft\!\!-\!\!-} n_1 \cdot P''$). Let us justify that the resulting path P is valid in τ.

▷ An edge $\mathrel{-\!\!-\!\circ}$ of τ' is rewritten into a valid edge $\mathrel{-\!\!-\!\circ}$ of τ by congruence of n_1 and n_2.

▷ An edge $\mathrel{\triangleleft\!\!-\!\!-}$ of τ' is rewritten into an edge $\mathrel{\triangleleft\!\!-\!\!-}$ of τ by well-domination of τ and binding-congruence of n_1 and n_2.

Thus P is a correct path from $\{\epsilon\}$ to n' in τ. By the hypothesis $n \mathrel{-\!\!\gg\!\!\rightarrow} n' \in \tau$, P contains n. Thus P' also contains n, which is the desired result.

Property 4.4.9 *Let τ be a type, n_1 and n_2 two binding-congruent nodes of τ. The fusion $\tau[n_1 = n_2]$ is a type.* □

Proof: Let τ' be $\tau[n_1 = n_2]$. The well-formedness of $\breve{\tau}'$, $\hat{\tau}'$ and $\overset{\diamond}{\tau}'$ are by definition of binding-congruent nodes. The preservation of well-domination is by Lemma 4.4.8.

4.4.3.1 Local congruence

The definition of locally congruent nodes (Definition 3.3.1) is still correct on graphic ML^{F} types. However, we give an alternative, slightly more formal, definition below.

Definition 4.4.10 (*Locally congruent nodes*) Two nodes n_1 and n_2 of a type τ are locally congruent if they are binding-congruent in τ and

$$\forall \pi \neq \epsilon,\ \langle n_1\pi\rangle\ \tilde{\tau}\ \langle n_2\pi\rangle\ \vee\ \langle n_1\pi\rangle \xrightarrow{+}\!\!\triangleright n_1 \wedge \langle n_2\pi\rangle \xrightarrow{+}\!\!\triangleright n_2$$ ■

4.4.4 Raising

Given a node n of a type τ such that $n \xrightarrow{\diamond} n' \xrightarrow{\diamond'} n''$ holds, the operation of raising n consists in lifting the binding edge $n \longrightarrow n'$ above the edge $n' \longrightarrow n''$, resulting in the edge $n \xrightarrow{\diamond} n''$. The resulting pre-type is called the *raising* of n in τ, and is written $\tau \uparrow n$.

▶ **Example** In Figure 4.4.1, the type τ_r is the raising of the node $\langle 22\rangle$ in τ.

Definition 4.4.11 (*Raising)* The raising $\tau \uparrow n$ of a node n in a type τ is the pre-type defined by

- $\overset{\multimap}{\tau \uparrow n}$ is $\breve{\tau}$;
- $\overset{\diamond}{\tau \uparrow n}$ is $\overset{\diamond}{\tau}(n)$;
- $\overset{\longrightarrow}{\tau \uparrow n}$ is $\hat{\tau}$, except on n where it is $n \longrightarrow \hat{\tau}(\hat{\tau}(n))$. ■

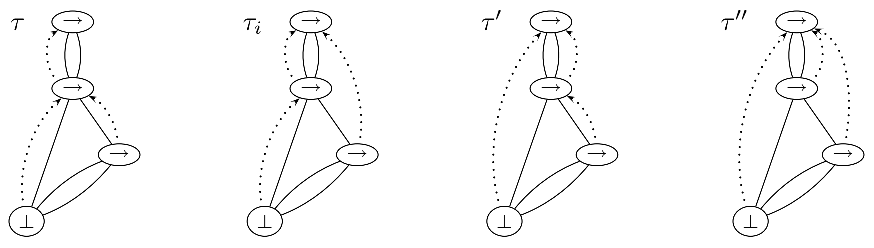


Figure 4.4.2 – Raising and well-domination

Raising and well-domination Raising at arbitrary nodes can result in ill-dominated pre-types. In the type τ of Figure 4.4.2, raising the node $\langle 12\rangle$ results in the ill-dominated type τ_i, as shown by the mixed path

$$\langle \epsilon\rangle \longleftarrow \langle 12\rangle \multimap \langle 11\rangle$$

which does not contain the binder of $\langle 11\rangle$, *i.e.* the node $\langle 1\rangle$.

In the case of τ and τ i, the structure path $\langle 12\rangle \multimap \langle 11\rangle$ prevents the raising of $\langle 12\rangle$. Syntactically, the bound of $\langle 12\rangle$ depends on the bound of $\langle 11\rangle$, and $\langle 12\rangle$ cannot thus be introduced first. This property can be generalized, and exactly characterizes the set of nodes n which can be raised while preserving well-domination.

Definition 4.4.12 (*Raisable node)* Given a type τ and a bound node $n \in \tau$, n is *raisable* in τ if no other node bound on $\hat{n}$ can be reached from n. Formally,

$$\forall n',\ n' \longrightarrow \hat{n} \implies n \overset{+}{\not\multimap} n' \qquad ■$$

▶ **Example** The node $\langle 11\rangle$ is raisable in the type τ of Figure 4.4.2, but not $\langle 12\rangle$. However, $\langle 12\rangle$ is raisable in τ', since $\langle 11\rangle$ and $\langle 12\rangle$ are no longer bound at the same node.

Property 4.4.13 *$\tau \uparrow n$ is a type iff n is raisable in τ.* □

Proof: We let $\tau' = \tau \uparrow n$, $n' = \hat\tau(n)$, $n'' = \hat\tau(n')$ (in particular, $n'' = \hat\tau'(n)$).

- ▷ If n is not raisable: we show that τ' is not well-dominated.
  Let m be a node bound on n' in τ such that $n \overset{+}{\multimap} m$. Thus, in τ' we have mixed paths of the form $\{\epsilon\} \overset{*}{\lhd\!\!\multimap} n'' \longleftarrow n \overset{+}{\multimap} m$. Those mixed path does not contain $\hat\tau'(m) = n'$, as n' is strictly under n'' and strictly above n'. Hence τ' is not well-dominated.
- ▷ If n is raisable: It is immediate to see that $\check\tau'$ and $\overset{\circ}{\hat\tau}{}'$ are correct, as they are unchanged by the raising. The fact that $\hat\tau'$ forms a tree is also immediate. Let us show that τ' is well-dominated. We consider a node m and a mixed path π' from $\{\epsilon\}$ to m in τ'. Let π be the mixed path obtained by replacing the edge $n \longleftarrow n''$ by $n \longleftarrow n' \longleftarrow n''$ (if it appears in π'). By construction, π is a valid mixed path between $\{\epsilon\}$ and m in τ. We must prove that $\hat\tau'(m)$ is in π'.
  - ◦ If $m = n$ and $n \longrightarrow n''$ is in π': by the first hypothesis $\hat\tau'(m)$ is n'', hence the conclusion by the second.
  - ◦ If $m = n$ (**1**) and $n \longrightarrow n''$ is not in π' (**2**): By well-domination of τ and (1), n' and n'' are in π. By (1), n'' is $\hat\tau'(m)$. Hence $\hat\tau'(m)$ is in π', since $\pi = \pi'$ by (2).
  - ◦ If $m \neq n$ and $n \longrightarrow n''$ is not in π': by those hypotheses, $\hat\tau(m) = \hat\tau'(m)$ and $\pi = \pi'$. We conclude by well-domination of τ.
  - ◦ If $m \neq n$, $n \longrightarrow n''$ is in π' and $\hat\tau(m) \neq n'$:
    By well-domination of τ, $m' = \hat\tau(m)$ is in π. Since m' is not n', then m' is in π' too (as this part of π is shared with π'). Since $m \neq n$, m' is $\hat\tau'(m)$, hence the conclusion.
  - ◦ If $m \neq n$ (**3**), $n \longrightarrow n''$ is in π' (**4**) and $\hat\tau(m) = n'$ (**5**): by (4), $n \longrightarrow n' \longrightarrow n''$ is in π. Thus π is in particular of the form $\langle\epsilon\rangle \overset{*}{\lhd\!\!\multimap} n'' \longleftarrow n' \longleftarrow n \overset{*}{\lhd\!\!\multimap} m$. By (3), we moreover have $n \overset{+}{\lhd\!\!\multimap} m$, hence also $n \overset{+}{\multimap} m$. Together with (4), this contradicts the fact that n is raisable in τ.

4.4.4.1 Raising multiple nodes

Instead of raising a single node, we are sometimes interested in raising all the nodes bound on a given node n.[4] We call this operation a *multi-raising.*

Definition 4.4.14 (*Multi-raising*) Given a type τ and a node n of τ different from the root, the *multi-raising* of n in τ is the graph τ' verifying $\check\tau = \check\tau'$, $\overset{\circ}{\hat\tau} = \overset{\circ}{\hat\tau}{}'$ and

$$\forall n',\ \hat\tau'(n') = \vee\left[\begin{array}{ll} \hat\tau(n) & \text{if } n' \longrightarrow n \in \tau \\ \hat\tau(n') & \text{if } n' \not\longrightarrow n \in \tau \end{array}\right.$$ ■

[4]This operation will mostly be useful in Part II, when we compare type inference in ML and in MLF.

▶ **Example** Multi-raising the node $\langle 1 \rangle$ of the type τ of Figure 4.4.2 results in the type τ''. Notice that raising n changes the binding edge of n, while multi-raising it changes the binding edges of the nodes bound on n.

Multi-raising is an interesting operation, because it does not require checking for raisability (or equivalently for well-domination).

Property 4.4.15 *Given a type τ and a node n of τ, the multi-raise of n in τ is a type.* □

Proof: Let S be the set of nodes bound on n in τ. It suffices to order this set by $\multimap_\tau$ (the lowest nodes being first), and to raise the nodes of S in this order.

5
Instance on ML^F graphic types

Abstract

We explain the introduction of rigid quantification in ML^F types (§5.1). The instance relation of ML^F is obtained by adapting the instance relation of System $\mathcal{F}$ to this form of quantification (§5.2). In fact, we obtain two relations, one permitting type inference, but not the other. We formally define the first one in §5.3.3, the second in §5.3.5. A third relation, designed to abstract over inessential details on nodes without polymorphism is also introduced (§5.3.4).

5.1 Why rigid quantification?

The System $\mathcal{F}$ instance modulo reversible instance relation $\sqsubseteq_{\mathcal{F}}^{\approx}$ generalizes the corresponding relation $\sqsubseteq_{\mathsf{F}}^{\approx}$ of System F (Lemma 3.4.3). Moreover, $\sqsubseteq_{\mathsf{F}}^{\approx}$ is sound and complete w.r.t. the syntactic instance relation $\leqslant_{\mathsf{F}}$ of System F. Hence $\sqsubseteq_{\mathcal{F}}^{\approx}$ extends this last relation. In parallel, ML^F is designed to be an extension of ML and the syntactic typing rules of ML^F are those of ML, modulo the richer types and type instance relation—exactly as is the case for System F. Thus, if the instance relation on ML^F graphic types $\sqsubseteq$ allowed all the operations of $\sqsubseteq_{\mathcal{F}}^{\approx}$, type inference in ML^F would likely be undecidable—just as in System F (Wells 1994).

Rigid quantification is introduced in ML^F to find a retriction of $\sqsubseteq_{\mathcal{F}}^{\approx}$ suitable for type inference. Thus, although the two forms of quantification share a similar syntax, there is a profound asymmetry between them:

- flexible quantification is introduced to obtain more expressive types, and more principal type derivations;
- rigid quantification is used to restrict the expressiveness of types, in order make type inference decidable.

From a semantic point of view, and without delving too much into the details yet, rigid edges have the same role as the absence of binding edges in System F graphic types: they forbid the merging or the instantiation of variables. Hence, there is a very simple way to transform an ML$^{\mathsf{F}}$ type into the corresponding System $\mathcal{F}$ type.

Definition 5.1.1 (*Mapping from ML$^{\mathsf{F}}$ to System $\mathcal{F}$*) We write $\Downarrow$ the injection from ML$^{\mathsf{F}}$ types to System $\mathcal{F}$ graphic types that removes all rigid edges from its argument. ■

The ML$^{\mathsf{F}}$ instance relation is obtained by adapting $\sqsubseteq^{\approxeq}_{\mathcal{F}}$ to the richer binding structure of graphic types. In fact, we simultaneously define two relations:

- $\sqsubseteq^{\boxminus}$ is the largest relation of the two, and comprises all the possible transformations: it is designed to be sound and complete w.r.t. $\sqsubseteq^{\approxeq}_{\mathcal{F}}$ (modulo $\Downarrow$).
- $\sqsubseteq$ is a restriction of $\sqsubseteq^{\boxminus}$ that permits type inference.

Those two relations define two different versions of ML$^{\mathsf{F}}$, with type inference being possible only in the smallest of the two. The interest of the first system lies mainly in its expressivity.

The decision to put a transformation in $\sqsubseteq^{\boxminus}$ but not in $\sqsubseteq$ is somewhat arbitrary at this point. Indeed, by definition it cannot be explained by type soundness, and is only justified by the fact that the system based on $\sqsubseteq$ allows type inference. Another design guideline is that $\sqsubseteq$ should be as large as possible, in order to obtain a system as expressive as possible.

5.2 Shaping the instance relation

In this section, we review informally the operations of $\sqsubseteq^{\approxeq}_{\mathcal{F}}$ and adapt them to rigid quantification.

Nodes are partitioned according to the operation they permit. As in System $\mathcal{F}$, this partition includes red, green and inert nodes. There is also a new category, called orange nodes. For type inference purposes we also isolate a subset of inert nodes, called monomorphic. This section also makes precise whether or not an operation is allowed in both $\sqsubseteq^{\boxminus}$ and $\sqsubseteq$, or only in $\sqsubseteq^{\boxminus}$. A formal definition of $\sqsubseteq$ and $\sqsubseteq^{\boxminus}$ is also given in §5.3.3 and §5.3.5.

In the first three subsections below, we suppose that the nodes discussed are not inert. They will be handled separately in §5.2.4.

5.2.1 Green ML$^{\mathsf{F}}$ nodes

In ML$^{\mathsf{F}}$, green nodes are the same as in System $\mathcal{F}$: they are transitively bound to the root:

$$(\xrightarrow{\geqslant})^*\langle\epsilon\rangle$$

Three operations on green variables and green inner nodes (grafting, merging, raising) are directly "inherited" from $\sqsubseteq^{\approxeq}_{\mathcal{F}}$, and are all in $\sqsubseteq$. Moreover, since the symmetric operations are unsound in general (thus not in $\sqsubseteq^{\approxeq}_{\mathcal{F}}$), they are neither in $\sqsubseteq$ nor in $\sqsubseteq^{\boxminus}$.

Weakening a green node (*i.e.* removing its binding edge in $\mathcal{F}$) is also possible, but we must change it slightly, as we require all nodes to be bound. Thus, on graphic ML$^{\mathsf{F}}$ types, we change the (flexible) binding edge into a rigid one; we again call this operation weakening. Notice that it exactly corresponds to the informal semantic of rigid edges.

Weakening is in both $\sqsubseteq$ and $\sqsubseteq^{\boxminus}$. Moreover, as in $\mathcal{F}$, the symmetric operation would be unsound (once polymorphism is requested, it must be given), and it is forbidden.

5.2.2 Red $\mathsf{ML^F}$ nodes

In System $\mathcal{F}$, the red nodes are the bound nodes which are not connected to the root by binding edges. It is straightforward to adapt this definition to graphic types: red nodes are flexibly bound nodes with a rigid edge above them, *i.e.*

$$\xrightarrow{\geqslant} (\xrightarrow{\diamond})^* \xrightarrow{=} (\xrightarrow{\diamond})^* \langle \epsilon \rangle$$

Remember that, in the general case, transforming such a node is unsound, as its polymorphism is requested—in $\mathsf{ML^F}$ by the rigid edge above. Thus no instance operation is allowed on those nodes.

5.2.3 Nodes with a rigid edge

The handling of the nodes that have no binding edge in System $\mathcal{F}$—*i.e.* that have a rigid binding edge in $\mathsf{ML^F}$ graphic types—is more subtle. The easy cases are the transformation of a rigid edge into a flexible one, and the substitution of a rigidly bound variable, which would for example allow to transform $(\forall\,(\alpha)\;\alpha) \to (\forall\,(\alpha)\;\alpha)$ into $\mathsf{int} \to (\forall\,(\alpha)\;\alpha)$. Both operations are clearly unsound, as they would allow requiring less polymorphism than originally requested, and they are forbidden.

Next, merging of locally congruent unbound nodes is possible in System $\mathcal{F}$, and is part of the reversible instance relation. In order to allow the same expressiveness, the relation $\sqsubseteq^{\boxminus}$ allows raising and merging, but also lowering and splitting, the nodes with rigid edges. However our key design choice is to disallow the last two operations in $\sqsubseteq$, *i.e.* to only allow in this relation the raising and merging of nodes with rigid edges. This very restriction is what keeps type inference decidable.

In the following, we draw nodes with rigid edges in orange, thus completing our "traffic lights" metaphor. Green nodes allow true instances (that change the semantics of the type), orange nodes only allow transformations changing the representation of types (but not their semantics, and types are in particular invariant modulo $\Downarrow$) and red nodes disallow instance entirely.

5.2.4 Inert and monomorphic nodes

In order to be slightly more general, we add the possibility for a type constructor to be intrinsically polymorphic.

Definition 5.2.1 (*Polymorphic symbols*) The set of type constructors Σ is supposed to be partitioned into two sets of *regular* and *polymorphic* type constructors, the $\to$ constructor being regular. The symbol $\bot$ is considered to be polymorphic.[1] We write Poly the set of polymorphic symbols. ■

This definition allows to easily model type constructors representing polymorphic type abbreviations, such as $\mathsf{type}\ \mathsf{t} = \forall\alpha.\ \alpha \to \alpha$. However, in the remainder of this section we will not use this possibility in the examples, and $\bot$ can safely be thought as the only polymorphic symbol.

[1] We recall that $\bot$ is not part of Σ

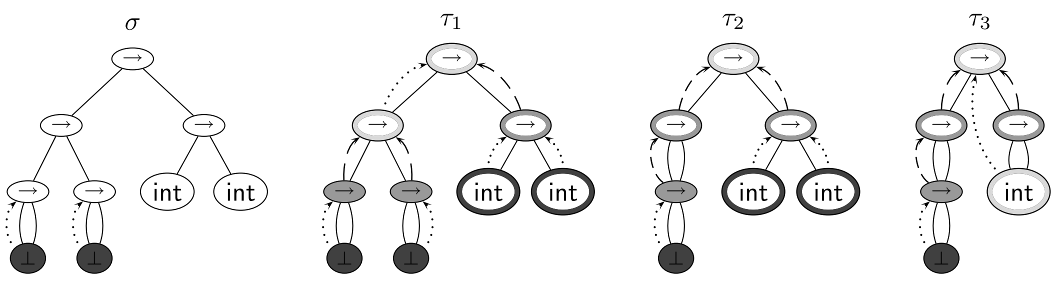


Figure 5.2.1 – Inert and monomorphic nodes

Terminology In the following, we say that n is an *intrinsically polymorphic* node in τ if $\tau(n)$ is a polymorphic symbol.

5.2.4.1 Inert nodes

Let us adapt the definition of inert nodes to rigid quantification and polymorphic symbols. In System $\mathcal{F}$, a node is inert if there is no variable transitively (flexibly) bound on it. In $\mathsf{ML^F}$ graphic types, a node is inert if all the polymorphic symbols below the node are protected by at least one rigid edge.

Definition 5.2.2 (*Inert nodes*) Let τ be a graphic type. A node n of τ is inert if it is not intrinsically polymorphic, and if there is a rigid edge between n and any other intrinsically polymorphic node n' below n. Formally,

$$\forall n',\ n' \xrightarrow{*} n \ \wedge\ \tau(n') \in \mathsf{Poly} \implies n' \xrightarrow{*} \xrightarrow{=} \xrightarrow{*} n \qquad \blacksquare$$

▶ **Example** Figure 5.2.1 shows the System $\mathcal{F}$ type

$$\sigma \quad \triangleq \quad (\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}) \to (\mathsf{int} \to \mathsf{int})$$

as well as three possible $\mathsf{ML^F}$ graphic representations of this type τ_1, τ_2 and τ_3. Indeed, the relation $\Downarrow(\tau_i) \approx_{\mathcal{F}} \sigma$ holds for i in $1 \ldots 3$. Moreover, all the hollow-colored nodes in the $\mathsf{ML^F}$ types are inert.

As defined by the previous sections, $\sqsubseteq^{\boxminus}$ is too restrictive. Indeed, we have not permitted transforming inert nodes, which is permitted by $\sqsubseteq_{\mathcal{F}}$. Indeed, $\sqsubseteq^{\approx}_{\mathcal{F}}$ also allows to raise, lower, merge, split, weaken and strengthen (the inverse operation of weaken) those nodes. Thus $\sqsubseteq^{\boxminus}$ also allows all those transformations, and $\tau_1 \sqsubseteq^{\boxminus} \tau_2 \sqsubseteq^{\boxminus} \tau_3 \sqsubseteq^{\boxminus} \tau_1$ holds.

Conversely, in order to preserve the decidability of type inference, $\sqsubseteq$ only allows merging, raising and weakening inert nodes. (However, this means that $\tau_1 \sqsubseteq \tau_2 \sqsubseteq \tau_3$ still holds.)

5.2.4.2 Monomorphic nodes

The fact that inert nodes cannot be *e.g.* unmerged or lowered by $\sqsubseteq$ may seem unfortunate. Indeed, it means that the binding edge of a node labelled by a ground type constructor such as int is significant. Thus, we should distinguish the types τ_2 and τ_3 of Figure 5.2.1.

Happily, this is in fact not the case. On a subset of inert types—those with no variables under them—we can allow unmerging, unraising and unweakening without losing decidability of type inference.

Definition 5.2.3 (*Monomorphic nodes)* Let τ be a graphic type. A node n is *monomorphic* if n and all the nodes bound under it have non-polymorphic symbols. Formally,

$$\forall n',\ n' \xrightarrow{*} n \implies \tau(n') \notin \mathsf{Poly} \qquad \blacksquare$$

Of course, as for inert nodes, this definition only involves the binding tree. In particular, there can be polymorphic symbols under n for $\multimap$. However, since polymorphism is only requested through the binding tree, we need not make such a distinction.

Perhaps surprisingly, we nevertheless do not add unmerging, unraising and unweakening on monomorphic nodes in $\sqsubseteq$. Indeed, it in fact holds (§13.2) that this would not add expressiveness to the type system. Hence we keep $\sqsubseteq$ as simple as possible. Moreover, as it is oriented towards "more sharing", we can use efficient first-order unification algorithms that implement unification exactly for $\sqsubseteq$.

Representing inert and monomorphic nodes In the following, we represent monomorphic nodes in white. Inert nodes, which allow the same transformations as orange nodes, are represented as hollow orange nodes.

5.3 Formal definition of the instance relations

5.3.1 Permissions

In the syntactic presentation of ML^F, finding what transformations are allowed at a given position in a type is not readily apparent, at it is determined by contextual inference rules and the stratification between the abstraction and instance relations. In graphic types, the allowed transformations are obtained by looking at the color of the node. However, as we stressed in §3, colors are only a visual help. Indeed, they are solely determined by the shape of the binding tree above the node (for green, orange and red) or below the node (for white, *i.e.* inert nodes). We introduce the notion of *flag path* to determine the non-white colors.

Definition 5.3.1 (*Flag path)* Let τ be a type, n a node of τ. The *flag path* of n in τ, written $\overline{\diamond}_\tau(n)$ is the sequence of binding flags $\diamond$ such that $\langle\epsilon\rangle \xleftarrow{\overline{\diamond}} n$. $\blacksquare$

We write $\overline{\diamond}(n)$ when τ is implicit from context. Notice that the flag path is read by following binding edges in the *inverse* direction of the one in τ.

▶ **Example** In the type τ_1 of Figure 5.2.1, $\overline{\diamond}_{\tau_1}(\langle 11\rangle) = (\geqslant =)$

The definition below gives the exact definition of colors as we have used them so far. Since colors define which operations are allowed on a given node, we also use the term of *permission.* Permissions are also summarized in Figure 5.3.1.

Permission	Name	Flag Path
G	Green	$\geqslant^*$
O	Orange	$\diamond^* =$
R	Red	$\diamond^* = \geqslant^+$
I	Inert	Definition 5.2.2
M	Monomorphic	Definition 5.2.3

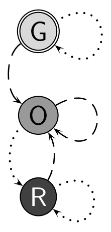


Figure 5.3.1 – Permissions

Definition 5.3.2 (*Permissions*) A node n of a type τ is said to have *permission* p if

$p = \mathsf{M}$:	n is monomorphic in τ
$p = \mathsf{I}$:	n is inert in τ
$p = \mathsf{G}$:	n is not inert in τ and $\overline{\diamond}_\tau(n)$ is of the form $\geqslant^*$
$p = \mathsf{O}$:	n is not inert in τ and $\overline{\diamond}_\tau(n)$ is of the form $(\geqslant|=)^*=$
$p = \mathsf{R}$:	n is not inert in τ and $\overline{\diamond}_\tau(n)$ is of the form $(\geqslant|=)^* = \geqslant^+$

The permissions of n in τ are written $\mathcal{P}_\tau(n)$. If n has permission p, it is said to be a *p node*.■

Notice that G, O, R and I are disjoint sets of nodes (and that each node has exactly one of those permissions), as they usually require distinct treatment in proofs. However I nodes can be M nodes. Those two choices are only a matter of presentation; we could have instead said that all nodes at $\geqslant^*$ flag path have G permission, or that I and M nodes are distinct.

Disregarding the fact it can be inert, a simple way to find whether a node is G, O or R is to follow its flag path in the automation given at the right of Figure 5.3.1. The state G is the initial state, *i.e.* the permission of the root.

▶ **Example** Consider the node $\langle 11 \rangle$ of type τ_1 in Figure 5.2.1, whose flag path is $\geqslant=$. We first follow the flexible edge from G to itself, and then the rigid edge from G to O; $\langle 11 \rangle$ is indeed orange, as it is not inert,

5.3.2 Atomic instance operations

Instance is composed of the four operations grafting, merging, raising and weakening introduced in the previous sections and formally defined below. All four transformations are displayed schematically in Figure 5.3.2, with the permissions permitting the transformation. Nodes in blue have either green, orange or inert permission. Smaller nodes, in light grey, have permissions either irrelevant or unconstrained by the other nodes and edges in the drawing.

▶ **A concrete example** Figure 5.3.3 introduces a sequence of types, each of which is in a particular form of instance relation with it successor. The type τ_1 is actually a valid type (albeit not the principal one) for the term K' defined by

$$\lambda(x)\ \lambda(y : \forall\,(\alpha)\ \alpha \to \alpha)\ y \qquad (K')$$

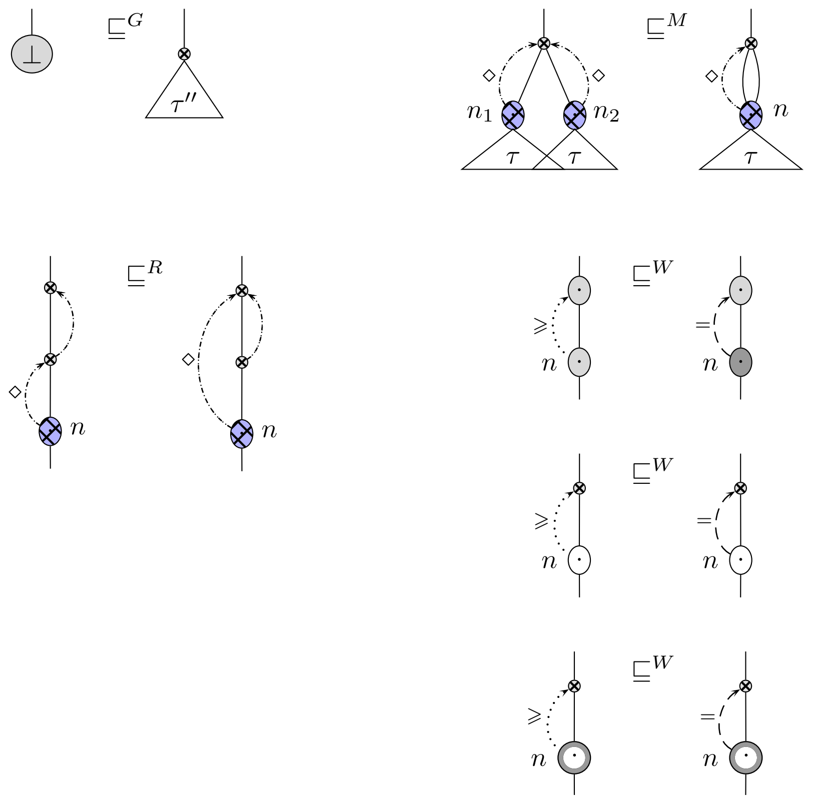


Figure 5.3.2 – Schematic depiction of the atomic instance operations

By construction of instance, all the types $\tau_2 \dots \tau_8$ are instances of τ_1, hence valid types for K'. In τ_8 the nodes $\langle 1 \rangle$ and $\langle 2 \rangle$ are merged. Thus, schematically, the instance steps of Figure 5.3.3 prove that K' and the identity function can be put inside an homogeneous container such as a list. Indeed, τ_8 is a common instance of τ_1 and of the principal type $\forall\,(\beta)\;\beta \to \beta$ of the identity function.

5.3.2.1 Grafting

Definition 5.3.3 (*Grafting*) A type τ' is the (instance-)grafting of τ'' at n in τ if n is a bottom node with green permission in τ and τ' is $\tau[\tau''/n]$. We write $\mathsf{Graft}(\tau'', n)$ for the function $\tau \mapsto \tau'$. ■

Notice that a node can become inert or monomorphic by grafting, if the root of τ'' is itself inert or monomorphic.

▶ **Example** In Figure 5.3.3, $\tau_i = \mathsf{Graft}(\tau_i/\langle 1 \rangle, \langle 1 \rangle)(\tau_1)$ holds for $2 \leq i \leq 7$.

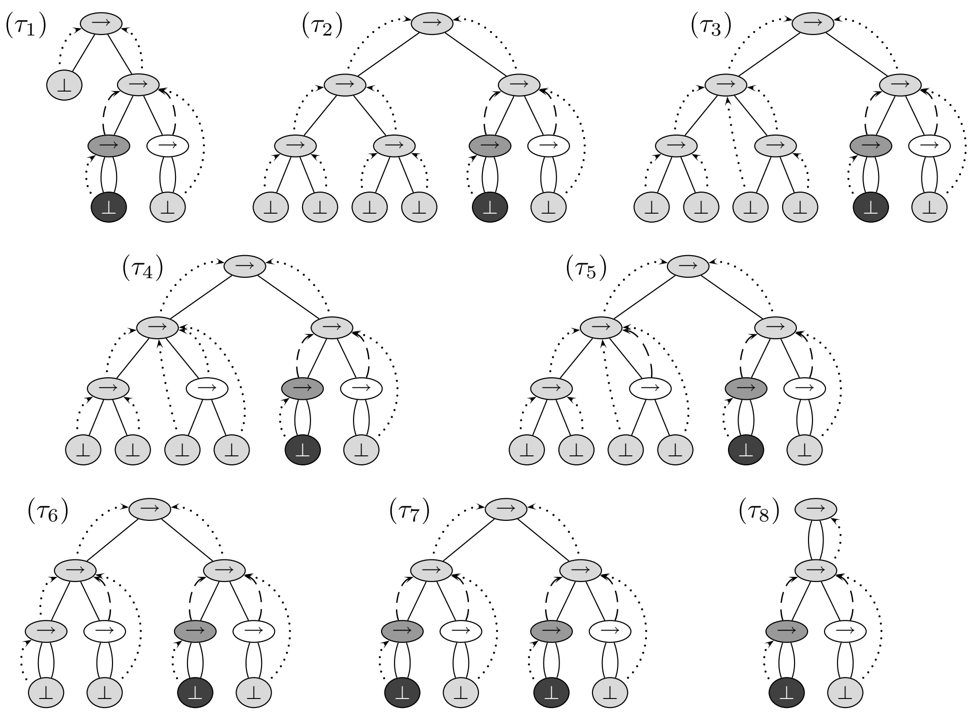


Figure 5.3.3 – Example of type instance.

5.3.2.2 Merging

As in Systems F and $\mathcal{F}$, merging is only possible on locally congruent nodes—otherwise the control of permissions would much more complicated.

Definition 5.3.4 (*Merging)* A type τ' is the (instance-)merging of the nodes n_1 and n_2 in τ if:

1. τ' is $\tau[n_1 = n_2]$;
2. n_1 and n_2 have non-red permissions[2];
3. n_1 and n_2 are locally congruent.

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

As usual, some subparts of the subgraphs under n_1 and n_2 can already be shared, hence the overlap in the sketch of Figure 5.3.2.

[2]By definition, n_1 and n_2 have the same permissions: their flag path and the binding tree under them are identical.

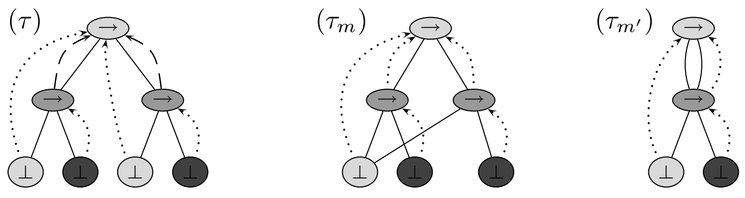


Figure 5.3.4 – Merging and local congruence.

▶ **Examples** In Figure 5.3.3, the following two relations are verified:

$$\tau_6 = (\mathsf{Merge}(\langle 111\rangle, \langle 112\rangle) \,;\, \mathsf{Merge}(\langle 121\rangle, \langle 122\rangle))(\tau_5) \qquad \tau_8 = \mathsf{Merge}(\langle 1\rangle, \langle 2\rangle)(\tau_7)$$

Consider next the types τ, τ_m and $\tau_{m'}$ of Figure 5.3.4. The type $\tau_{m'}$ is $\tau[\langle 1\rangle = \langle 2\rangle]$ but not $\mathsf{Merge}(\langle 1\rangle, \langle 2\rangle)(\tau)$. Indeed the nodes $\langle 11\rangle$ and $\langle 21\rangle$ fail condition 3, since they are not bound under $\langle 1\rangle$ and $\langle 2\rangle$ in τ.

In this particular case, the transformation can be decomposed into two atomic mergings that both satisfy condition 3:

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

However, such a decomposition does not always exist. In our example, had $\langle 11\rangle$ and $\langle 21\rangle$ been red, the first merging would not have been possible.

5.3.2.3 Raising

Definition 5.3.5 (*Raising*) A type τ' is the (instance-)raising of n in τ if n has non-red permissions in τ, and $\tau' = \tau \uparrow n$. We write $\mathsf{Raise}(n)$ for the function $\hat{\tau} \mapsto \hat{\tau}'$. ■

▶ **Example** In Figure 5.3.3, τ_3 is the raising of $\langle 121\rangle$ in τ_2, while τ_4 is the raising of $\langle 122\rangle$ in τ_3.

5.3.2.4 Weakening

Definition 5.3.6 (*Weakening*) A type τ' is the (instance-)weakening of n in τ if n has green or inert permissions in τ, and τ and τ' coincide except for the binding edge $n \xrightarrow{\geqslant} n'$ in τ, which is replaced by $n \xrightarrow{=} n'$ in τ'. We write $\mathsf{Weaken}(n)$ for the function $\tau \mapsto \tau'$. ■

▶ **Example** In Figure 5.3.3, τ_7 is the weakening of $\langle 11\rangle$ in τ_6, while τ_5 is the weakening of the inert node $\langle 12\rangle$ in τ_4.

In the following, we order binding flags by $(\geqslant) < (=)$, following the transformation induced by a weakening.

Definition 5.3.7 (*Order on binding flags*) We write $\overset{\diamond}{<}$ the order defined by $(\geqslant) \overset{\diamond}{<} (=)$, $(\geqslant) \overset{\diamond}{<} (\geqslant)$ and $(=) \overset{\diamond}{<} (=)$. ■

5.3.3 The instance relation

Definition 5.3.8 (*Instance subrelations)* We write $\sqsubseteq^G$, $\sqsubseteq^M$, $\sqsubseteq^R$ and $\sqsubseteq^W$ for the reflexive transitive closures of the relations respectively defined by

$$\begin{array}{lcl} \tau \sqsubseteq^G_1 \tau' & \triangleq & \exists n, \exists \tau'',\ \tau' = \mathsf{Graft}(\tau'', n)(\tau) \\ \tau \sqsubseteq^M_1 \tau' & \triangleq & \exists n_1, n_2,\ \tau' = \mathsf{Merge}(n_1, n_2)(\tau) \\ \tau \sqsubseteq^R_1 \tau' & \triangleq & \exists n,\ \tau' = \mathsf{Raise}(n)(\tau) \\ \tau \sqsubseteq^W_1 \tau' & \triangleq & \exists n,\ \tau' = \mathsf{Weaken}(n)(\tau) \end{array}$$ ■

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

Definition 5.3.9 (*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. ■

▶ **Example** Coming back to Figure 5.3.3, we have seen in the previous section 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 $\sqsubseteq$. A shortened decomposition of this fact is

$$\tau_1 \sqsubseteq^G \tau_7 \sqsubseteq^M \tau_8$$

Moreover, some operations could 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 red after the weakening, which prevents any further operation on them.

Notations Consider a $\sqsubset$-like relation symbol such as $\sqsubseteq$. In the remainder of this document, for any valid X and Y, we let $\sqsubset^{\mathsf{XY}}$ be $\sqsubset^{\mathsf{X}} \odot \sqsubset^{\mathsf{Y}}$. Thus $\sqsubseteq$ is just $\sqsubseteq^{GRMW}$. We also let $\sqsupset^{\mathsf{X}}$ be the symmetric relation of $\sqsubset^{\mathsf{X}}$, and allow any meaningful combinations of those notations, as well as with the relations $\sqsubset^{\mathsf{X}}_1$.

By construction, the instance relation of $\mathsf{ML^F}$ is a refinement of the instance relation on term-graphs.

Property 5.3.10 *Given two types τ and τ', $\tau \sqsubseteq \tau'$ implies $\check{\tau} \sqsubseteq_{\mathsf{G}} \check{\tau}'$.* □

Proof: By induction on a derivation of $\tau \sqsubseteq \tau'$. Once the binding tree and the permissions checks are removed, $\sqsubseteq^G$ and $\sqsubseteq^M$ become subrelations of $\sqsubseteq_{\mathsf{G}}$, while $\sqsubseteq^R$ and $\sqsubseteq^W$ do not change the underlying term-graph at all.

Instance is an oriented relation. Since graphic types have anonymous variables, α-conversion is directly captured by the representation of types, and non-reflexive instance steps permanently change the type.

Lemma 5.3.11 *The kernel of $\sqsubseteq$ is equality.* □

Proof: Non-reflexive instance strictly decreases lexicographically the measure $(-N_1(\tau), N_2(\tau))$, where

$N_1(\tau)$ is the number of paths π such that $\tau(\pi)$ is not $\bot$.
Raise, Merge and Weaken leave N_1 unchanged. For Graft, suppose the operation is $\mathsf{Graft}(\tau, n)$:

- ▷ if τ is reduced to $\bot$, the instance is reflexive;
- ▷ if τ is not $\bot$, the number of non-bottom paths strictly increases, as all paths π such that $\pi \subseteq n$ are no longer labelled by $\bot$.

$N_2(\tau)$ is the sum of the binding heights of the nodes of τ plus the number of flexibly bound nodes of τ.

Raising strictly decreases at least one binding height, and does not change the number of flexible nodes. Merging strictly decreases the number of nodes, hence the sum of the binding heights (and possibly the number of flexible nodes). Weakening keeps the binding height unchanged and strictly decreases the number of flexible nodes.

Notice that this measure is not well-founded, as $N_1(\tau)$ can grow arbitrarily.

5.3.4 Instance modulo similarity

As in the systems seen in §3, the instance relation is too fine-grained: we wish to read types modulo some inessential details, using a similarity relation that abstracts over them.

By design, $\mathsf{ML^F}$ similarity only captures "simple" semantic equivalences. In particular, reasoning modulo similarity preserves decidability of type inference. Precisely, similarity captures the differences in both the sharing and the binding of monomorphic nodes.

Definition 5.3.12 (*Similarity)* We write $\sqsubseteq^m$, $\sqsubseteq^r$ and $\sqsubseteq^w$ for the subrelations of $\sqsubseteq^M$, $\sqsubseteq^R$ and $\sqsubseteq^W$ that only merge, raise or weaken monomorphic nodes, *i.e.* the transitive closures of the relations

$$\begin{array}{lcll} \tau \sqsubseteq^m_1 \tau' & \triangleq & \exists n_1, n_2 \text{ monomorphic in } \tau, & \tau' = \mathsf{Merge}(n_1, n_2)(\tau), \\ \tau \sqsubseteq^r_1 \tau' & \triangleq & \exists n \text{ monomorphic in } \tau, & \tau' = \mathsf{Raise}(n)(\tau) \\ \tau \sqsubseteq^w_1 \tau' & \triangleq & \exists n \text{ monomorphic in } \tau, & \tau' = \mathsf{Weaken}(n)(\tau) \end{array}$$

We call *reversible instance* the subrelation $\sqsubseteq^{rmw}$ of $\sqsubseteq$. We call *similarity*, written $\approx$, the equivalence relation $(\sqsubseteq^{rmw} \cup \sqsupseteq^{rmw})^*$ and *instance modulo similarity*, written $\sqsubseteq^{\approx}$, the relation $(\sqsubseteq \cup \approx)^*$. ■

Types are meant to be understood modulo similarity; in fact, when we display them in syntactic form, we entirely remove the binding edges and the sharing on monomorphic nodes. However, we often express results for $\sqsubseteq$ alone, as they are both stronger than for $\sqsubseteq^{\approx}$, and easier to establish.

▶ **Example** Consider again Figure 5.2.1. The relation

$$\tau_3 = (\mathsf{Raise}(\langle 21 \rangle) \,;\, \mathsf{Raise}(\langle 22 \rangle) \,;\, \mathsf{Merge}(\langle 21 \rangle, \langle 22 \rangle))(\tau_2)$$

holds, and all the nodes involved are monomorphic. Thus τ_3 is a reversible instance of τ_2, and both types are similar. Likewise, in Figure 5.3.3, the types τ_4 and τ_5 are similar, since $\tau_5 = \mathsf{Weaken}(\langle 12\rangle)(\tau_4)$ holds, and this weakening is monomorphic. Of course, in more complicated cases one type is not necessarily a reversible instance of the other.

Unsurprisingly, $\approx$ is exactly the reversible part of $\sqsubseteq^{\approx}$.

Lemma 5.3.13 *The kernel of $\sqsubseteq^{\approx}$ is $\approx$.* □

Proof: Consider two types τ and τ' such that $\tau \sqsubseteq^{\approx} \tau' \sqsubseteq^{\approx} \tau$, and a derivation I of this result. We show that an operation of $\sqsubseteq \setminus \sqsubseteq^{rmw}$ cannot be undone, and hence cannot appear in I.

- ▷ I cannot contain a non-reflexive grafting (**1**), as $\dot{\tau}$ and $\dot{\tau}'$ would be different.
- ▷ the binding heights of non-monomorphic nodes (and the fact that the nodes are non-monomorphic) are preserved by $\sqsubseteq^M$, $\sqsupseteq^m$, $\sqsubseteq^W$, $\sqsupseteq^w$ and strictly decrease for at least one node by $\sqsubseteq^R \setminus \sqsubseteq^r$. Moreover, they are preserved by $\sqsubseteq^r$ and $\sqsupseteq^r$: since non-monomorphic paths are upwards-closed for $\longrightarrow$, those two operations cannot be used to lower or raise a node above a non-monomorphic node to change the binding height of the latter. Thus, together with (1), I cannot contain a non-monomorphic raising (**2**).
- ▷ the number of non-monomorphic nodes is preserved by $\approx$ and $\sqsubseteq^W$, and strictly decreases by $\sqsubseteq^M \setminus \sqsubseteq^m$. Thus, by (1) and (2), I cannot contain a non-monomorphic merging.
- ▷ the number of paths π such that $\mathring{\tau}(\pi) = (\geqslant)$ and $\langle\pi\rangle$ is non-monomorphic is stable by $\approx$ and $\sqsubseteq^M$, and strictly decreases by $\sqsubseteq^W \setminus \sqsubseteq^w$. Thus, by (1) and (2), I cannot contain a non-monomorphic weakening.

Thus, by the four points above, I only contains similarity steps.

5.3.5 Instance modulo abstraction

The (instance modulo) abstraction relation is used to abstract over all the notational details brought by the introduction of rigid quantification to System $\mathcal{F}$ types. In particular, it allows freely transforming nodes with rigid edges, as well as all inert nodes.

Definition 5.3.14 (*Abstraction*) We write $\sqsubseteq\!\!\!\!\cdot^M$, $\sqsubseteq\!\!\!\!\cdot^R$ and $\sqsubseteq\!\!\!\!\cdot^W$ for the subrelations of $\sqsubseteq^M$, $\sqsubseteq^R$ and $\sqsubseteq^W$ that only merge, raise and weaken inert or orange nodes, *i.e.* the transitive closures of the relations

$$
\begin{array}{rcll}
\tau \mathrel{\boxminus}_1^M \tau' & \triangleq & \exists n_1, n_2 \text{ inert or orange in } \tau, & \tau' = \mathsf{Merge}(n_1, n_2)(\tau), \\
\tau \mathrel{\boxminus}_1^R \tau' & \triangleq & \exists n \text{ inert or orange in } \tau, & \tau' = \mathsf{Raise}(n)(\tau) \\
\tau \mathrel{\boxminus}_1^W \tau' & \triangleq & \exists n \text{ inert in } \tau, & \tau' = \mathsf{Weaken}(n)(\tau)
\end{array}
$$

We call *abstraction* the subrelation $\sqsubseteq\!\!\!\!\cdot$ of $\sqsubseteq$ defined as $\sqsubseteq\!\!\!\!\cdot^{MRW}$. We write $\boxminus$ the equivalence relation defined by $(\sqsubseteq\!\!\!\!\cdot \cup \sqsupseteq\!\!\!\!\cdot)^*$. We call *instance modulo abstraction*, written $\sqsubseteq^{\boxminus}$, the relation $(\sqsubseteq \cup \boxminus)^*$. ■

▶ **Example** In Figure 5.2.1,

$$\tau_2 = (\mathsf{Weaken}(\langle 1\rangle)\,;\mathsf{Merge}(\langle 11\rangle, \langle 12\rangle))(\tau_1)$$

Since $\langle 1\rangle$ is inert and both $\langle 11\rangle$ and $\langle 12\rangle$ are orange, this operations is an abstraction. Thus both $\tau_1 \sqsubseteq\!\!\!\!\cdot\ \tau_2$ and $\tau_1 \boxminus \tau_2$ hold.

Notice that the following inclusions hold by construction:

$$(\approx) \subset (\boxminus) \qquad\qquad (\sqsubseteq^{rmw}) \subset (\sqsubseteq) \subset (\sqsubseteq) \subset (\sqsubseteq^{\approx}) \subset (\sqsubseteq^{\boxminus})$$

The relation $\sqsubseteq^{\boxminus}$ defines the *implicit* version of $\mathsf{ML^F}$, where type annotations are not needed in source terms. Conversely, type inference is no longer decidable in this system.

We conclude this section by proving that $\boxminus$ is exactly the reversible part of $\sqsubseteq^{\boxminus}$.

Lemma 5.3.15 *The kernel of $\sqsubseteq^{\boxminus}$ is $\boxminus$.* □

Proof: The proof is the same as for Lemma 5.3.13, replacing "non-monomorphic" by "green" and the subrelations of $\approx$ by the corresponding relations in $\boxminus$. The only difference is the fact that green paths and nodes are not stable, but instead decrease, by $\sqsubseteq^{W}$.

5.4 Instance and permissions

In this last section, we characterize instance through an entirely operational point of view—as opposed to the semantic one used when defining instance in the previous sections. We also characterize how permissions evolve through instance.

5.4.1 Change in permissions

As a small (but important) technical result, we identify all the atomic instance operations that change the permissions of a node. This result is particularly useful inside proofs, when we need to assert that some permissions do not change. (However we will also use a more abstract presentation, given in §5.4.3.)

Lemma 5.4.1 (Change in permissions) *Let τ be a type, and n a type of τ such that an atomic instance operation o is applied to n. Let τ' be $o(\tau)$. If there exists a node n' of τ' with different permissions in τ and τ', then necessarily one of the following holds*

- $o = \mathsf{Graft}(\tau'', n)$ *for some type τ'', n and n' are green in τ, $n \xrightarrow{*} n'$, and n' is monomorphic or inert in τ'.*
- $o = \mathsf{Weaken}(n)$*, n and n' are green in τ and one of the following holds:*
  - *$n' = n$ and n' is orange in τ'*
  - *$n' \xrightarrow{+} n$ and n' is red in τ*
  - *$n \xrightarrow{+} n$' and n' is inert in τ*
- $o = \mathsf{Raise}(n)$*, $n' = \hat{\tau}(n)$ and either*
  - *n is orange or inert in τ, n' is inert in τ and monomorphic in τ', or*
  - *n and n' are green in τ, and n' is monomorphic or inert in τ'* □

Proof: Since the permissions of a node n'' are entirely determined by the binding edges and the binding flags of the nodes above and below n'' (for $\longrightarrow\!\!\triangleright$), we have necessarily either $n \xrightarrow{*}\!\!\triangleright n'$ or $n' \xrightarrow{*}\!\!\triangleright n$. The proof is by case disjunction on o.

▷ If o is a merging: the binding tree and the binding flags are entirely unchanged in τ', hence n' has exactly the same permissions in τ and τ'.

▷ If o is a grafting: by definition of grafting, n has green permissions. The nodes above n (which are necessarily also green) can become inert or monomorphic depending on the binding tree of the type grafted, or remain green. There is no node under n, since it is a variable.

▷ If o is a weakening: we proceed by case disjunction on the permissions of n in τ.

- If n is green: by definition of green, there exists an intrinsically polymorphic node n'' such that $n''(\xrightarrow{\geqslant}\!\!\triangleright)^*n$ in τ. This path still exists in τ', thus n is orange in τ'.

  Next, consider a node n' such that $n \xrightarrow{+}\!\!\triangleright n'$. Necessarily, it is green. Hence, it can only remain green, or become inert if n is in the only flag path to an intrinsically polymorphic node.
  Finally, consider a node n' such that $n' \xrightarrow{+}\!\!\triangleright n$ in τ. If n' is inert or monomorphic in τ, it is also inert or monomorphic in τ': the binding tree under this node is unchanged in τ'. Otherwise, there exists an intrinsically polymorphic node n'' such that $n''(\xrightarrow{\geqslant}\!\!\triangleright)^*n'$, in both τ and τ' (**1**). If n' is red, since n is green, we have $n' \xrightarrow{\geqslant\overline{\diamond}}\!\!\triangleright n \in \tau$, with $(=)$ somewhere in $\overline{\diamond}$. This binding path is the same in τ'; together with (1), this shows that n' is red in τ'. If n' is orange, we have $n' \xrightarrow{=\overline{\diamond}}\!\!\triangleright$ instead, but the reasoning is the same. If n' is green, it becomes red in τ'.
- If n is red: the weakening is forbidden.
- If n is orange: the weakening is impossible, as n is already rigidly bound.
- If n is monomorphic: the permissions of n and of the nodes below are unchanged, since monomorphic nodes are only concerned with binding edges, not with binding flags. For the nodes above, consider n' such that $n \xrightarrow{+}\!\!\triangleright n'$.
  - If n' is not inert: the flag path witnessing this fact cannot go through n. Hence this flag path still exists in τ', and since the flag path above n' is unchanged, n' has the same permissions in τ'.
  - If n' is monomorphic: the binding edges under it are unchanged (only the binding flag of n is changed), and it is still monomorphic in τ'.
  - If n' is inert: there are already rigid edges on all the flag paths under n'; adding a new rigid edge under n' does not change this fact.
- If n is inert: the permissions of n and of the nodes below are unchanged, since the binding trees under all those nodes are unchanged. The conclusion for the nodes above n is the same as in the previous case.

▷ If o is a raising: The automaton of Figure 5.3.1 does not "count" the number of binding flags it sees, but merely their alternation. Thus it returns the same color for n in τ and τ' (**2**): the only problematic case would be if n was flexibly bound, and if it was bound above a rigid edge (making it possibly green after the raising), but this case is forbidden by permissions. Since the binding tree and the binding edges below n are unchanged, by (2), n and the nodes bound on it have the same permissions in τ and τ'.

The permissions of the nodes n' above $\hat{n}$ are unchanged:

- *if* n' is monomorphic in τ: the result is immediate, as no polymorphic node can be transitively bound on n' in τ', since none exists in τ

- *if n'* is non-inert in τ: there exists a flexible flag path to an intrinsically polymorphic node n'' in τ. In τ' this flag path might have changed through the raising of n, but n'' is still accessible from n'. Since the flag path above n' is unchanged, the permissions of n' are unchanged.
- *if n'* is inert but not monomorphic in τ': there is a flag path to an intrinsically polymorphic node n'' in τ', this flag path containing an edge =. After the raising this path might have changed, but at least one edge = still exists, as permissions disallow raising a flexible edge above a rigid one. Hence n' is still inert in τ'.

For $\hat{n}$ itself, since the flag path above $\hat{n}$ is unchanged in τ', the color returned by the automation for n' is the same in τ and τ'. Hence, n can only become inert or monomorphic (**3**), which is possible since n is no longer bound on it. We proceed by case analysis on the permissions of n in τ.

- *If n* is green: $\hat{n}$ must also be green, which is the desired result together with (3).
- *If n* is orange: $\hat{n}$ can be anything but monomorphic (monomorphic nodes are downwards closed). If it is inert, it can become monomorphic, or remain inert. It it is not inert, the flag path witnessing this fact cannot involve n (which is rigidly bound), and $\hat{n}$ has the same permissions in τ'.
- *If n* is inert and not monomorphic: as above $\hat{n}$ can be anything but monomorphic. If it is inert, the result is also as above. If it is not inert, the flag path still cannot involve n, as n is inert. The conclusion is as above.
- *If n* is red: this case is impossible by permissions.
- *If n* is monomorphic: raising n does not change the permissions of $\hat{n}$ at all. Indeed, the flag path above n' is unchanged, and so are all the flag paths to a polymorphic node under n'.

Notice a very important corollary of this result: red nodes never disappear through an instance operation (and they only appear through weakening). This result is not overly surprising, as it already holds in System $\mathcal{F}$, and the interpretations of red nodes in ML^F and this system are supposed to be the same.

Property 5.4.2 *Nodes with red permissions are preserved by $\sqsubseteq^{\boxminus}$.* □

Proof: By Lemma 5.4.1, it is immediate that instance preserve red nodes, as the only nodes for which permissions change are green or inert ones. It remains to prove the result for $\sqsupseteq$. The only problematic case is for weakenings, which can introduce red nodes. However, this is only the case for the weakening of a green node, which is not part of $\sqsubseteq$; hence, the inverse operation is not part of $\sqsupseteq$ either.

5.4.2 Ordering permissions

An interesting way to characterize permissions is to order them according to the operations they allow.

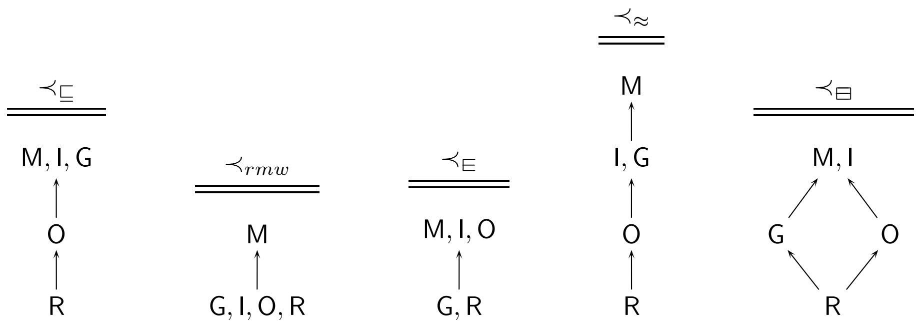
Definition 5.4.3 (*Order on permissions*) Let $\sqsubset$ be a subrelation of $\sqsubseteq^{\boxminus}$. We say that $<$ is an *order on permissions for* $\sqsubset$ if $\mathsf{P} < \mathsf{P}'$ implies that any operation of $\sqsubset$ possible on a node with permission P is also possible on a node with permission P'. ■

In particular, two different permissions can be equal for $\sqsubset$ if they allow the same transformations.

▶ **Example** We have $\mathsf{R} < \mathsf{O} < \mathsf{G}$ for $\sqsubseteq$. Moreover, I and M are equal for this relation.

Unsurprisingly, the ordering between permissions change depending on whether we consider $\sqsubseteq$, $\sqsubseteq^{rmw}$, $⋿$, $\sqsubseteq^{\approx}$ or $\sqsubseteq^{\boxminus}$. We give five suitable orders below.

Lemma 5.4.4 *The transitive and reflexive closure of the orders* $\prec_{\sqsubseteq}$, $\prec_{rmw}$, $\prec_{⋿}$, $\prec_{\approx}$ *and* $\prec_{\boxminus}$ *of Figure 5.4.1 are correct permission orders for the relations* $\sqsubseteq$, $\sqsubseteq^{rmw}$, $⋿$, $\sqsubseteq^{\approx}$ *and* $\sqsubseteq^{\boxminus}$ *respectively.* □


$\prec_{\sqsubseteq}$

$$\begin{tikzcd} \mathsf{M},\mathsf{I},\mathsf{G} \\ \mathsf{O} \arrow[u] \\ \mathsf{R} \arrow[u] \end{tikzcd}$$

$\prec_{rmw}$

$$\begin{tikzcd} \mathsf{M} \\ \mathsf{G},\mathsf{I},\mathsf{O},\mathsf{R} \arrow[u] \end{tikzcd}$$

$\prec_{⋿}$

$$\begin{tikzcd} \mathsf{M},\mathsf{I},\mathsf{O} \\ \mathsf{G},\mathsf{R} \arrow[u] \end{tikzcd}$$

$\prec_{\approx}$

$$\begin{tikzcd} \mathsf{M} \\ \mathsf{I},\mathsf{G} \arrow[u] \\ \mathsf{O} \arrow[u] \\ \mathsf{R} \arrow[u] \end{tikzcd}$$

$\prec_{\boxminus}$

$$\begin{tikzcd} & \mathsf{M},\mathsf{I} & \\ \mathsf{G} \arrow[ur] & & \mathsf{O} \arrow[ul] \\ & \mathsf{R} \arrow[ul] \arrow[ur] & \end{tikzcd}$$

An arrow from P *to* $\mathsf{P'}$ *means that* $\mathsf{P} < \mathsf{P'}$

Figure 5.4.1 – Order on permissions

The proof of this result is immediate by examining the five relations; we however give some details below.

In each case, R is the lowest permission, as it never permits any operation. The order $\prec_{rmw}$ is the simplest: $\sqsubseteq^{rmw}$ only allows transformations on monomorphic nodes. Similarly, $⋿$ only allows raising, merging and weakening, and only on monomorphic, inert or orange nodes.

For $\prec_{\sqsubseteq}$, as expected we have $\mathsf{G} > \mathsf{O} > \mathsf{R}$. At first, it might seem strange to also have G equal to I or M, as the first permission allows grafting but not the last two. But $\mathsf{G} \prec_{\sqsubseteq} \mathsf{I}, \mathsf{M}$ also holds by reasoning as follows: grafting can be vacuously said to be allowed on monomorphic or inert nodes, as it is only possible on variables, which cannot be momorphic or inert. Moreover, our convention of making G, I and M equal is much more convenient to work with. The order $\prec_{\approx}$ is essentially similar to $\prec_{\sqsubseteq}$, except that it distinguishes M from I and G. Indeed, M nodes can be freely transformed by $\sqsupseteq^{rmw}$ in $\sqsubseteq^{\approx}$, which is not the case for I and G nodes.

The ordering for $\prec_{⋿}$ is quite different. First, we cannot merge G with I and M: the last two permissions allow unsharing along $∍$, but not G. In parallel, we cannot merge O with I or M, as O does not allow grafting. As a result, G and O are incomparable, and both less general than M and I, resulting in the order $\prec_{⋿}$. (Another possibility would be to merge

M, I and O, as we did for $\prec_{\sqsubseteq}$, and to make G incomparable with those three permissions, as the first three allow unsharing and G allows grafting. However this order would be less practical to work with for the use we present in the next section.)

5.4.3 Evolution of permissions through instance

Armed with those orders, we can study how permissions change when a type is transformed by an instance operation. The results are mostly as one would expect. In particular, all the transformations but weakening preserve permissions, or slightly increase them (because some nodes can become inert or monomorphic). Weakening decreases permissions for $\prec_{\sqsubseteq}$, $\prec_{rmw}$ and $\prec_{\approx}$; we discuss weakening for $\prec_{\boxminus}$ below.

Property 5.4.5 *Let τ be a type, τ' another type obtained by transforming τ. Let π be a path of τ. Let P (*resp. *P') be the permissions of the node at π in τ (*resp. *τ').*

- *if $\tau \sqsubseteq^M \tau'$ then P' = P for all five orders*
- *if $\tau \sqsubseteq^G \tau'$ then P' > P for all five orders.*
- *if $\tau \sqsubseteq^R \tau'$ then P' > P for all five orders*
- *if $\tau \sqsubseteq^W \tau'$ then P' < P for $\prec_{\sqsubseteq}$, $\prec_{rmw}$ and $\prec_{\approx}$*
- *if $\tau \approx \tau'$, then P' = P for all the orders*
- *if $\tau \boxminus \tau'$, then P' = P for $\prec_{\boxminus}$.* □

Proof: All the results are direct consequences of Lemma 5.4.1; we however give some details below.

Merging does not change permissions at all. Grafting can change green nodes into monomorphic or inert nodes, which increases or preserves permissions for all the orders. Raising changes green or inert nodes into inert or monomorphic ones, *i.e.* it increases permissions for all the orders. Weakening changes green nodes into red, orange or inert ones, which indeed decreases permissions for the given orders.

For $\approx$: grafting is no longer possible; weakening and raising do not change permissions, as they only do so when applied on green, orange or inert nodes, which cannot be changed by $\sqsubseteq^{rmw}$. Thus $\sqsubseteq^{rmw}$ does not change permissions at all, and thus $\sqsupseteq^{rmw}$ does not either.

For $\boxminus$: by the same reasoning as above, only the raising of an orange node can change permissions. This operation can only change an inert node into a monomorphic one, but we have I = M for $\prec_{\boxminus}$, hence the result.

This result is quite useful in proofs, as it is a good layer of abstraction on top of Lemma 5.4.1.

Notice that weakening is not monotonic for $\prec_{\boxminus}$: green nodes can become orange, red or inert. This is a beginning of explanation on why type inference is not possible when $\sqsubseteq^{\boxminus}$ is used as the instance relation. Since weakening increases the permissions of some nodes, using a weakening during type inference (on a node on which this weakening is not strictly required as this step) might yield a better type. However this could also make type inference fail later, since weakening also decreases permissions. Thus, in order to be complete, type inference would at least need to backtrack sometimes during inference.

Convention In the following, we almost always reason on $\sqsubseteq$. Hence, when we write that permissions increase, decrease, or remain stable, this must be understood with respect to $\prec_{\sqsubseteq}$. There are a few exceptions, explicitly mentioned in the text.

6 Properties of the instance relations

Abstract

In this chapter, we study the various instance (sub)relations. Since instance is not noetherian, we isolate some subrelations of instance that have this property (§6.1). We show that $\sqsubseteq$ can be reorganized so that instance derivations always follow a certain order (§6.2). We characterize "big-step" versions of $\sqsubseteq^R$ and $\sqsubseteq^{MW}$, thus removing the need for decomposing an instance derivation into atomic operations (§6.3). For grafting (which is already a big-step operation), we instead show that we can proceed by small, atomic steps (§6.4). We show that, under certain conditions, an instance operation inside an instance derivation can be brought at the beginning of the derivation (§6.6). Finally, we show that most of the subrelations of instance are confluent, and that $\sqsubseteq^{\approx}$ and $\sqsubseteq^{\boxminus}$ can be reorganized so that all instance operations are performed first (§6.7).

The results of this chapter are mostly technical, and used mainly inside proofs. However the definitions of §6.3 and §6.4 are used when discussing the unification algorithm in §7.

Proving instance-related results Many proofs of this document have similar structure, as they proceed by induction on a given instance derivation. However, very often, we need the derivation to be constrained. For example, some operations need to appear before some others. The results of this chapter are in particular used to obtain those constrained derivations.

6.1 Reasoning on restricted instance

Grafting can increase the size of the skeleton of a type in an arbitrary way, and $\sqsubseteq$ is not a noetherian relation. However, all other instance subrelations are noetherian. This provides a powerful reasoning mechanism whenever the structure of the types is guaranteed not to grow arbitrarily.

Definition 6.1.1 (*Structure definedness)* Consider two types τ and τ'. We say that τ is *structurally less-defined* than τ' if

$$\forall \pi \in \mathsf{dom}(\tau),\ \wedge \begin{cases} \pi \in \mathsf{dom}(\tau') \\ \vee \begin{bmatrix} \tau(\pi) = \bot \\ \tau(\pi) = \tau'(\pi) \end{bmatrix} \end{cases} \qquad \blacksquare$$

It is immediate that only grafting changes structure-definedness.

Property 6.1.2 *Let τ and τ' be two types. If $\tau \sqsubseteq^G \tau'$, then τ is less-defined than τ'. If $\tau\ (\sqsubseteq^{RMW} \cup \sqsupseteq^{RMW})\ \tau'$, τ and τ' have the same structure definedness.* $\square$

In particular, structure-definedness is completely invariant by $\approx$ and $\boxminus$.

Definition 6.1.3 (*Restricted instance)* Consider a type τ. We write $\sqsubseteq|_\tau$ the restriction of the instance relation to graphic types with structure less-defined than the one of τ, *i.e.*

$$\tau_1 \sqsubseteq|_\tau \tau_2 \quad \triangleq \quad \tau_1 \sqsubseteq \tau_2 \ \wedge\ \tau_2 \text{ is structurally less-defined than } \tau \qquad \blacksquare$$

Of course, this implies that τ_1 is also less-defined than τ.

Property 6.1.4 *Let τ be a type. The restriction of $\sqsubseteq|_\tau$ to non-reflexive instance steps is noetherian.* $\square$

Proof: The result is immediate by the proof of Lemma 5.3.11, as the lexicographic order of this proof becomes well-founded: for any type τ' and τ'' such that $\tau' \sqsubseteq|_\tau \tau''$, $-N_1(\tau')$ and -$N_1(\tau'')$ are greater than $-N_1(\tau)$.

6.2 Ordering the instance operations

The subrelations of the instance relation are almost entirely orthogonal: grafting only involves $\breve{\tau}$ (and mainly changes $\dot{\tau}$), while merging, raising and weakening only alter $\tilde{\tau}$, $\hat{\tau}$ and $\mathring{\hat{\tau}}$ respectively. This orthogonality is quite convenient when studying the properties of $\sqsubseteq$, as it makes commutations between the different operations quite simple. In fact, it is possible to strongly constrain the instance relations and subrelations so as to obtain more canonical derivations (resulting in much simpler proofs): graftings can always occur first, followed by raisings, and then mergings and weakenings interleaved. This flexibility is actually one of the keys to an efficient implementation of unification (§7).

Lemma 6.2.1 *The instance relation $\sqsubseteq$ is equal to the relation $\sqsubseteq^G ; \sqsubseteq^R ; \sqsubseteq^{MW}$.* $\square$

Proof: The inclusion $\sqsubseteq^G ; \sqsubseteq^R ; \sqsubseteq^{MW} \subseteq \sqsubseteq$ is by definition of $\sqsubseteq$. For the other inclusion, Figure 6.2.1 shows that $\sqsubseteq_1 ; \sqsubseteq_1$ is included in $\sqsubseteq^G ; \sqsubseteq^R ; \sqsubseteq^{MW}$ (**1**): provided the left-hand side of the equations is defined, the equalities presented in this figure hold. All cases use Property 5.4.5 to justify that there are enough permissions to do the rewriting. In all the

- $\mathsf{Raise}(n)\ ;\ \mathsf{Graft}(\tau', n') \overset{\rightarrow}{\equiv} \mathsf{Graft}(\tau', n')\ ;\ \mathsf{Raise}(n)$
  Raising does not create new green nodes, hence the grafting can be done first.

- $\mathsf{Weaken}(n)\ ;\ \mathsf{Graft}(\tau', n') \overset{\rightarrow}{\equiv} \mathsf{Graft}(\tau', n')\ ;\ \mathsf{Weaken}(n)$

- $$\mathsf{Merge}(n_1, n_2)\ ;\ \mathsf{Graft}(\tau', n') \overset{\rightarrow}{\equiv} \begin{cases} \mathsf{Graft}(\tau', n')\ ;\ \mathsf{Merge}(n_1, n_2) \\ \qquad \text{if } n' \not\xrightarrow{+}\!\!\triangleright \langle n_1 \cup n_2 \rangle \text{ after merging} \\ \mathsf{Graft}(\tau', n_1\pi)\ ;\ \mathsf{Graft}(\tau', n_2\pi)\ ;\ \mathsf{Merge}(n_1, n_2) \\ \qquad \text{if after merging } n' \xrightarrow{+}\!\!\triangleright \langle n_1 \cup n_2 \rangle \\ \qquad \text{and } \langle n_1 \cup n_2 \rangle \xrightarrow{\pi}\!\!\circ\ n' \end{cases}$$

- $\mathsf{Weaken}(n)\ ;\ \mathsf{Raise}(n') \overset{\rightarrow}{\equiv} \mathsf{Raise}(n')\ ;\ \mathsf{Weaken}(n)$

- $$\mathsf{Merge}(n_1, n_2)\ ;\ \mathsf{Raise}(n') \overset{\rightarrow}{\equiv} \begin{cases} \mathsf{Raise}(n')\ ;\ \mathsf{Merge}(n_1, n_2) \\ \qquad \text{if after merging } n' \not\xrightarrow{*}\!\!\triangleright \langle n_1 \cup n_2 \rangle \\ \mathsf{Raise}(n_1)\ ;\ \mathsf{Raise}(n_2)\ ;\ \mathsf{Merge}(n_1, n_2) \\ \qquad \text{if after merging } n' = \langle n_1 \cup n_2 \rangle \\ \mathsf{Raise}(n_1\pi)\ ;\ \mathsf{Raise}(n_2\pi)\ ;\ \mathsf{Merge}(n_1, n_2) \\ \qquad \text{if after merging } \hat{n}' \xrightarrow{+}\!\!\triangleright \langle n_1 \cup n_2 \rangle \xrightarrow{\pi}\!\!\circ\ n' \\ \mathsf{Raise}(n_1\pi)\ ;\ \mathsf{Raise}(n_2\pi)\ ;\ \mathsf{Merge}(n_1\pi, n_2\pi)\ ;\ \mathsf{Merge}(n_1, n_2) \\ \qquad \text{if after merging } \hat{n}' = \langle n_1 \cup n_2 \rangle \xrightarrow{\pi}\!\!\circ\ n' \end{cases}$$

Figure 6.2.1 – Reordering instance

cases but the first (which is justified in the figure), we move an operation restricting or preserving permissions behind an operation increasing or preserving permissions.

Next, consider two types τ and τ' such that $\tau \sqsubseteq \tau'$, and a derivation I of this result. We must show that rewriting I according to the rules of Figure 6.2.1 terminates. Notice that these rules either preserve the number of atomic instance steps or strictly increase this number, and that no reflexive step is introduced. By (1), it is immediate that $(\sqsubseteq_1|_{\tau'}\ ;\ \sqsubseteq_1|_{\tau'}) \subseteq (\sqsubseteq^G|_{\tau'}\ ;\ \sqsubseteq^R|_{\tau'}\ ;\ \sqsubseteq^{MW}|_{\tau'})$. Since $\sqsubseteq|_{\tau'}$ is noetherian, the rewriting rules that strictly increase the number of atomic instance steps can only occur finitely many times. Hence it suffices to show that the rules that preserve the number of instance operations can also be applied only finitely many times; we call these rules R. Let us write I as $o_1\ ;\ o_2\ ;\ \ldots\ ;\ o_k$, *i.e.* as the sequence as atomic instance steps that transform τ into τ'. The rules of R are such that they rewrite $o_p\ ;\ o_q$ into $o_q\ ;\ o_p$. It is immediate that each application of a rule strictly decreases the number of inversions between the instance operations w.r.t. the correct order, *i.e.* the well-founded measure s defined by

$$s(o_1\ ;\ o_2\ ;\ \ldots\ ;\ o_k) = |\{(p, q) \mid o_p\ ;\ \ldots\ ;\ o_q \text{ is not of the form } \mathsf{Graft}(\cdot,\cdot)^*\ ;\ \mathsf{Raise}(\cdot)^*\ ;\ (\mathsf{Merge}(\cdot,\cdot) \cup \mathsf{Weaken}(\cdot))^*\}|$$

Other simple decompositions (*e.g.* $\sqsubseteq^R\ ;\ \sqsubseteq^{MW}\ ;\ \sqsubseteq^G$) are not possible 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 the binding flags to be congruent—hence the need to weaken some nodes.

Definition 6.2.2 (*Ordered instance derivations)* A sequence of elementary instance transformations is called *ordered* when it respects the ordering of Lemma 6.2.1. ■

▶ **Example** The proof of $\tau_1 \sqsubseteq \tau_8$ in Figure 5.3.3 is ordered.

6.3 Big-step instance subrelations

Proving that two types are in instance relation *a priori* requires to exhibit a derivation of this result in term of atomic instance steps. Since this can become quite tedious, we introduce "big-steps" relations that compare the shapes of two types and asserts they are instance of one another.

6.3.1 Big-step raising

Raising can only be applied to raisable nodes. In order to prove that $\tau \sqsubseteq^R \tau'$, we should thus prove that all nodes raised in the derivation are raisable (or alternatively that all intermediary types are well-dominated); this makes proofs quite complicated. An alternative is to define a relation that compares the binding trees of two well-dominated types, and asserts that one is the result of performing multiple raising in the other.

Definition 6.3.1 (*Big-step raising)* Given two types τ and τ', we say that τ' is a *big-step raising of* τ, written $\tau \sqsubseteq^{R\natural} \tau'$, if and only if

$$\wedge\begin{cases} \textbf{(1)} & \breve{\tau} = \breve{\tau}' \\ \textbf{(2)} & \overset{\diamond}{\tau} = \overset{\diamond}{\tau}' \\ \textbf{(3)} & \hat{\tau}' \subseteq (\hat{\tau})^+ \\ \textbf{(4)} & \forall n,\ \hat{\tau}'(n) \neq \hat{\tau}(n) \implies \mathcal{P}_\tau(n) \neq \mathsf{R} \end{cases}$$ ■

The first two points assert that the underlying term-graphs and all the binding flags are equal in τ and τ'. The third point verifies that a binding edge of τ' is in the transitive closure of the binding edges of τ. The fourth point ensures that a raised node has enough permissions.

Next, we characterize raisings in which the nodes lowest in the type are raised first.

Definition 6.3.2 (*Bottom-up raising)* A sequence $\mathsf{Raise}(n_1)\,;\ldots;\mathsf{Raise}(n_k)$ is said to be a *bottom-up raising* if

$$i > j \implies \neg(n_j \xrightarrow{+}\!\!\circ\ n_i)$$ ■

If two types verify $\tau \sqsubseteq^{R\natural} \tau'$, it is easy to obtain a bottom-up raising for $\tau \sqsubseteq^R \tau'$: we can raise any node amongst the lowest ones, and iterate this step until no node remains to be raised. This is not the case if *e.g.* we choose a top-down ordering, as some nodes might not be raisable at first.

Lemma 6.3.3 *Consider two types such that $\tau \sqsubseteq^{R\natural} \tau'$. Then there exists a bottom-up derivation $\mathsf{Raise}(n_1)\,;\ldots;\mathsf{Raise}(n_k)$ such that $\tau' = (\mathsf{Raise}(n_1)\,;\ldots;\mathsf{Raise}(n_k))(\tau)$ and which proves $\tau \sqsubseteq^{R} \tau'$.* □

<u>Proof</u>: Let $m(\tau, \tau')$ be the measure defined by

$$m(\tau,\tau') = \sum_{n \in \mathsf{dom}(\tau)} k \mid n \xrightarrow{\diamond_1 \cdots \diamond_k}\!\!\!\!\triangleright\ \hat{\tau}'(n) \in \tau$$

The proof is by induction on $m(\tau, \tau')$. If this number is 0, all the nodes of τ are bound at the same node in τ and τ'. Since $\tau \sqsubseteq^{R\natural} \tau'$ holds, $\tau = \tau'$ and the empty sequence proves the result. Otherwise, let n be a node lowest for $\multimap$ among those such that $\hat{\tau}(n) \neq \hat{\tau}'(n)$. Let us first prove that n can be raised in τ.

▷ <u>n is raisable in τ</u>: we proceed by contradiction, and assume there exists n' bound at $\hat{\tau}(n)$ in τ such that $n \overset{+}{\multimap} n'$. The mixed path $\langle\epsilon\rangle \overset{*}{\multimap} \hat{\tau}'(n) \leftarrowtriangle n \overset{+}{\multimap} n'$ is valid in τ'. Moreover this path does not contain $\hat{\tau}'(n')$: this node is equal to $\hat{\tau}(n)$ (as otherwise n' would be raised before n), and $\hat{\tau}(n)$ is strictly above n and strictly below $\hat{\tau}'(n)$ (indeed, $\hat{\tau}(n) \neq \hat{\tau}'(n)$ by hypothesis, and point 3 of the definition of $\sqsubseteq^{R\natural}$ implies that $n \overset{+}{\rightarrowtriangle} \hat{\tau}'(n) \in \tau$). Thus τ' would not be well-dominated: contradiction.

▷ <u>n is not red in τ</u>: by point 4 of the definition of $\sqsubseteq^{R\natural}$.

Let τ'' be $\mathsf{Raise}(n)(\tau)$. We have proven $\tau \sqsubseteq^{R} \tau''$. Let us next prove $\tau'' \sqsubseteq^{R\natural} \tau'$. All points but the third in the definition of $\sqsubseteq^{R\natural}$ are immediate since $\tau \sqsubseteq^{R\natural} \tau'$ holds. For point 3, consider a binding edge $n' \rightarrowtriangle \hat{\tau}'(n')$ of τ'. By hypothesis, $n' \overset{+}{\rightarrowtriangle} \hat{\tau}'(n') \in \tau$ and we must prove that $n' \overset{+}{\rightarrowtriangle} \hat{\tau}'(n') \in \tau''$. In τ, if $n' \overset{+}{\rightarrowtriangle} \hat{\tau}'(n')$ is not of the form $n' \overset{*}{\rightarrowtriangle} n \rightarrowtriangle \hat{\tau}(n)$, the result is immediate. Suppose then that we have $n' \overset{*}{\rightarrowtriangle} n \rightarrowtriangle \hat{\tau}'(n') \in \tau$. The node n' cannot be n: we have $\hat{\tau}(n) \neq \hat{\tau}'(n)$ by hypothesis on n. Thus n' is strictly below n, which contradicts the choice of n as lowest node. Thus $\tau'' \sqsubseteq^{R\natural} \tau'$ holds.

Finally, we have $m(\tau'', \tau') = m(\tau, \tau') - 1$ by definition of τ'' and m. Since $\tau'' \sqsubseteq^{R\natural} \tau'$ holds, by induction hypothesis there exists a bottom-up raising for $\tau'' \sqsubseteq^{R} \tau'$. Together with $\mathsf{Raise}(n)(\tau)$, this forms a bottom-up derivation for $\tau \sqsubseteq^{R} \tau'$, which is the desired result.

As $\sqsubseteq^{R}$ is also included in $\sqsubseteq^{R\natural}$, we can prove the equality of these two relations.

Lemma 6.3.4 *The relations $\sqsubseteq^{R}$ and $\sqsubseteq^{R\natural}$ are equal.* □

<u>Proof</u>: The direction $\tau \sqsubseteq^{R\natural} \tau' \implies \tau \sqsubseteq^{R} \tau'$ is by Lemma 6.3.3.

The direction $\tau \sqsubseteq^{R} \tau' \implies \tau \sqsubseteq^{R\natural} \tau'$ is by induction on a derivation of $\tau \sqsubseteq^{R} \tau'$. If $\tau = \tau'$, the result is immediate. Otherwise, let τ'' be such that $\tau \sqsubseteq^{R}_{1} \tau'' \sqsubseteq^{R} \tau'$. By induction hypothesis, $\tau'' \sqsubseteq^{R\natural} \tau'$ holds (**1**). By definition of raising, it is immediate that $\tau \sqsubseteq^{R\natural} \tau''$. Thus it suffices to show that $\sqsubseteq^{R\natural}$ is transitive. For the first two points of the definition, this is by transitivity of equality. For the fourth point, it is a consequence of the fact that raising does not remove or introduce red nodes. For the third point, we have $\widehat{\tau''} \subseteq (\hat{\tau})^{+}$ and $\hat{\tau}' \subseteq (\widehat{\tau''})^{+}$ by $\tau \sqsubseteq^{R\natural} \tau''$ and $\tau'' \sqsubseteq^{R\natural} \tau'$. Then $(\widehat{\tau''})^{+} \subseteq ((\hat{\tau})^{+})^{+}$ by the first inclusion, this last expression being equal to $(\hat{\tau})^{+}$. By transitivity of inclusion we have $\hat{\tau}' \subseteq (\hat{\tau})^{+}$, which is the desired result.

6.3.2 Big-step merging and weakening

As we have done for $\sqsubseteq^R$, we can characterize a big-step merging and weakening operation, and prove that it is derivable using the usual atomic operations. We also isolate derivations that occur bottom-up. We moreover require that weakenings in those derivations occur before mergings.

Definition 6.3.5 (*Big-step merging-weakening)* We say that τ' is a *big-step merging-weakening* of τ, written $\tau \sqsubseteq^{MW\natural} \tau'$ if

$$\wedge\begin{cases} \textbf{(1)} & \dot{\tau} = \dot{\tau}' \\ \textbf{(2)} & \hat{\tau} = \hat{\tau}' \\ \textbf{(3)} & \tilde{\tau} \subseteq \tilde{\tau}' \\ \textbf{(4)} & \forall n, \forall n',\ \wedge\begin{cases} n \mathrel{\tilde{\not\tau}} n' \\ n \mathrel{\tilde{\tau}'} n' \\ \hat{\tau}(n) = \hat{\tau}(n') \end{cases} \qquad \Longrightarrow \mathcal{P}_\tau(n) \neq \mathsf{R} \\ \textbf{(5)} & \forall n,\ \overset{\diamond}{\tau}(n) \overset{\diamond}{<} \overset{\diamond}{\tau}'(n) \\ \textbf{(6)} & \forall n,\ \overset{\diamond}{\tau}'(n) \neq \overset{\diamond}{\tau}(n) \implies \mathcal{P}_\tau(n) \neq \mathsf{R} \end{cases}$$

■

The first two points ensure that the underlying tree and the 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 condition $\hat{\tau}(n) = \hat{\tau}(n')$ makes sure that we do not check permissions for nodes that are indirectly merged. The fifth point checks that binding flags are either unchanged, or that flexible edges are transformed into rigid ones. The last point checks that all the weakenings are allowed.

Definition 6.3.6 (*Bottom-up merging-weakening)* A sequence $i_1\,;\ldots;i_m$ is a *bottom-up merging-weakening* if it verifies

- for any p, o_p is either $\mathsf{Merge}(n_1, n_2)$ or $\mathsf{Weaken}(n)$
- if n_p is transformed by o_p and n_q is transformed by o_q (*e.g.* $n_p = \mathsf{Merge}(n_p, n)$ or $\mathsf{Merge}(n, n_p)$ or $\mathsf{Weaken}(n_p)$), and if $n_p \overset{+}{\multimap} n_q$, then $p > q$
- if o_p is $\mathsf{Merge}(n, n')$ or $\mathsf{Merge}(n', n)$ and o_q is $\mathsf{Weaken}(n)$, then $q < p$ ■

Given a derivation $\tau \sqsubseteq^{MW\natural} \tau'$, we can find a derivation of this result in terms of a bottom-up derivation $\tau \sqsubseteq^{MW} \tau'$. It is simply obtained by finding the lowest node to transform, and performing the required operation.

Lemma 6.3.7 *Consider two type τ and τ' such $\tau \sqsubseteq^{MW\natural} \tau'$ holds. Then there exists a bottom-up merging-weakening derivation $o_1\,;\ldots;o_k$ such that $\tau' = (o_1\,;\ldots;o_k)(\tau)$ and which witnesses $\tau \sqsubseteq^{MW} \tau'$.* □

Proof: Let m be the measure defined by

$$m(\tau, \tau') = \left|\{(n, n') \mid n \mathrel{\tilde{\not\tau}} n' \wedge n \mathrel{\tilde{\tau}'} n'\}\right| + \left|\{n \mid \overset{\diamond}{\tau}(n) = (\geqslant) \wedge \overset{\diamond}{\tau}'(n) = (=)\}\right|$$

The proof is by induction on $m(\tau, \tau')$. If this number if 0, we have $\tau = \tau'$ since $\tau \sqsubseteq^{MW\natural} \tau'$, and the empty derivation proves the result. Otherwise, let n be a node lowest in τ for $\multimap$ among the nodes n such that either

1. $\overset{\diamond}{\tau}(n) \neq \overset{\diamond}{\tau}'(n)$
2. there exists n' distinct from τ such that $n \; \tilde{\tau}' \; n'$ and $\hat{\tau}(n) = \hat{\tau}(n')$.

Let us justify that n exists: if $\left|\{n \mid \overset{\diamond}{\tau}(n) = (\geqslant) \wedge \overset{\diamond}{\tau}'(n) = (=)\}\right|$ is not 0, at least one node verifies point 1. Otherwise, $|\{(n, n') \mid n \; \not\tilde{\tau} \; n' \wedge n \; \tilde{\tau}' \; n'\}|$ is not 0. Let n_1 and n_2 be two nodes merged in τ but not in τ'. If they are not bound to the same node in τ, we consider their binders. Necessarily, they must be merged in τ', as otherwise $\hat{\tau}'$ would not be a tree. Moreover, by hypothesis, $\hat{\tau}(n_1)$ and $\hat{\tau}(n_2)$ are distinct. We can thus iterate this step until we find two binding ancestors n_1' and n_2' of n_1 and n_2 bound on the same node, and n_1' and n_2' verify point 2.

There are now two cases:

- ▷ <u>n verifies point 1</u>: we let τ'' be $\mathsf{Weaken}(n)(\tau)$. The relation $\tau \sqsubseteq^{W} \tau''$ holds by point 6 of the definition of $\sqsubseteq^{MW\natural}$. Notice also that we have $m(\tau'', \tau') = m(\tau, \tau') - 1$
  Let us show that $\tau'' \sqsubseteq^{MW\natural} \tau'$. All points of Definition 6.3.5 but the permissions related ones are immediate, since $\tau \sqsubseteq^{MW\natural} \tau''$ and $\overset{\diamond}{\tau}'(n) = (=)$. By Lemma 5.4.1, a node n' of τ becomes red in $\mathsf{Weaken}(n)(\tau)$ only if $n' \xrightarrow{+} n$. But n' cannot verify the hypotheses of points 4 and 6 of the definition of $\sqsubseteq^{MW\natural}$, as otherwise we would have chosen it before n. Thus points 4 and 6 still hold in τ'', and $\tau'' \sqsubseteq^{MW\natural} \tau'$ holds.
- ▷ <u>n verifies point 2 but not point 1</u>: we let τ'' be $\mathsf{Merge}(n, n')(\tau)$. The nodes n and n' are binding-congruent, as the subgraphs under them is the same as the subgraph under the node in which they are merged in τ'. They are also locally congruent, as otherwise we could have chosen a node strictly under n to merge. Moreover n and n' are not red by point 4 of the definition of $\sqsubseteq^{MW\natural}$. Thus $\tau \sqsubseteq^{M} \tau''$ holds. We also have $m(\tau'', \tau') < m(\tau, \tau')$, since at least two more nodes are merged. Finally, $\tau'' \sqsubseteq^{MW\natural} \tau'$ holds: all points are immediate since $\tau \sqsubseteq^{MW\natural} \tau''$, n and n' are merged in τ', and merging preserves permissions.

In both cases we have proven $\tau \sqsubseteq^{MW}_1 \tau''$, $\tau'' \sqsubseteq^{MW\natural} \tau'$ and $m(\tau'', \tau') < m(\tau, \tau')$. By induction hypothesis we obtain a bottom-up derivation of $\tau'' \sqsubseteq^{MW} \tau'$. Together with the operation transforming τ into τ'', this forms a bottom-up derivation of $\tau \sqsubseteq^{MW} \tau'$, which is the desired result.

Notice that a top-down approach woul be impossible in general, as weakening a node higher in the type might prevent further merging or weakening some lower nodes.

Lemma 6.3.8 *Given two types τ and τ', $\tau \sqsubseteq^{MW\natural} \tau'$ if and only if $\tau \sqsubseteq^{MW} \tau'$.* □

<u>Proof</u>: The direction $\tau \sqsubseteq^{MW\natural} \tau' \implies \tau \sqsubseteq^{MW} \tau'$ is by Lemma 6.3.7.

For the direction $\tau \sqsubseteq^{MW} \tau' \implies \tau \sqsubseteq^{MW\natural} \tau'$, we proceed by induction on $\tau \sqsubseteq^{MW} \tau'$. If $\tau = \tau'$ the result is immediate. Otherwise, let τ'' be such that $\tau \sqsubseteq^{MW} \tau'' \sqsubseteq^{MW} \tau'$. By induction hypothesis, we have $\tau'' \sqsubseteq^{MW\natural} \tau'$, and we must prove that $\tau \sqsubseteq^{MW\natural} \tau'$ holds. All the cases of Definition 6.3.5 except cases 4 and 6 are immediate by definition of $\sqsubseteq^{MW}$. For those points: any node not red in τ'' is not red in τ either, as merging and weakening restrict permissions. Moreover, the nodes transformed between τ and τ'' are not red either, by definition of $\sqsubseteq^{MW}$. This is sufficient to conclude.

6.4 Grafting atomic types

6.4.1 Widening

Lemma 6.2.1 implies that instance derivations may always start by performing all the grafting. However there are many possibilities as to which type to graft. As already mentioned, in Figure 5.3.3, 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 directly grafted a complicated type. Even though this makes 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. This section shows that this form of "simple" graftings is always possible.

Definition 6.4.1 (*Widening*) Given a first-order term with anonymous variables t, we define its *widening* $\triangle(t)$ by:

- $\overset{\longrightarrow\!\circ}{\triangle(t)}$ is the unique tree-like term-graph whose skeleton is t, and such that every node is reduced to a single path in $\widetilde{\triangle(t)}$.
- $\overset{\longrightarrow\!\triangleright}{\triangle(t)}$ binds all the nodes to their ancestor, *i.e.* $\overset{\longrightarrow\!\triangleright}{\triangle(t)} = \{n \longrightarrow\!\triangleright\, n' \mid n' \longrightarrow\!\circ\, n\}$;
- $\overset{-\diamond-}{\triangle(t)}$ binds all the nodes with a flexible flag. ■

▶ **Example** In Figure 5.3.3, the subgraph $\tau_2/\langle 2\rangle$ is exactly the widening of $\dot{\tau}_2/\langle 2\rangle$.

Lemma 6.4.2 *Let τ be a type. The relation $\triangle(\dot{\tau}) \sqsubseteq \tau$ holds.* □

Proof: Let τ' be $\triangle(\dot{\tau})$. We are going to exhibit an ordered derivation of $\tau' \sqsubseteq \tau$. Let τ_r be the pre-type that has the structure of τ', but in which the nodes have been raised as in τ. Formally, $\breve{\tau}_r = \breve{\tau}'$, $\overset{\diamond}{\tau}_r(n) = (\geqslant)$ for any n, and $n \longrightarrow\!\triangleright\, n' \in \hat{\tau}_r \iff n' \overset{+}{\longrightarrow\!\circ}\, n \in \tau_r \wedge n \longrightarrow\!\triangleright\, n' \in \hat{\tau}$. It is immediate that $\hat{\tau}_r$ is well-formed, as it binds any node which is not the root to one of its ancestors.

Let us first prove that τ_r is well-dominated. Consider a node n and a mixed path P from $\{\epsilon\}$ to n in τ_r. Since τ shares more nodes than τ_r, P is also a valid path in τ. By well-domination of τ, $\hat{\tau}(n)$ is contained in P. Since there is no sharing in τ_r, the node of P which extends to $\hat{\tau}(n)$ is restricted to a single path, and is also $\hat{\tau}_r(n)$ by definition of the binding tree of τ_r. Thus $\hat{\tau}_r(n)$ is in P, which is the desired result.

Let us now show that $\tau' \sqsubseteq^R \tau_r \sqsubseteq^{MW} \tau$ holds. All the nodes of τ' are bound to their immediate ancestor. Hence we have $\hat{\tau}_r \subseteq (\hat{\tau}')^+$. Moreover, all the nodes of τ' have a flexible flag, hence non-red permissions. Thus the instance $\tau' \sqsubseteq^{R\natural} \tau_r$ holds, and $\tau' \sqsubseteq \tau_r$ holds by Lemma 6.3.4.

We also have $\tilde{\tau}_r \subseteq \tilde{\tau}$, since $\tilde{\tau}_r = \tilde{\tau}'$ and no node is shared in τ', and $\overset{\diamond}{\tau}_r(n) \overset{\diamond}{<} \overset{\diamond}{\tau}(n)$ for any n, as all nodes of τ_r are flexibly bound. Finally, no node of τ_r is red, since they all are flexibly bound. Thus the relation $\tau_r \sqsubseteq^{MW\natural} \tau$ holds, and $\tau_r \sqsubseteq^{MW} \tau$ holds by Lemma 6.3.8.

More generally, given a node n grafted inside an instance derivation, we can isolate the tree under n and graft its widening.

Lemma 6.4.3 *Let τ' be an instance of a type τ, and n a bottom node of τ that is not a bottom node in τ'. Let τ_n be $\triangle(\dot{\tau}'/n)$. Then $\tau \sqsubseteq^G \tau[\tau_n/n] \sqsubseteq \tau'$.* $\square$

<u>Proof</u>: For $\tau \sqsubseteq^G \tau[\tau_n/n]$: necessarily n must be green in τ, as it is grafted between τ and τ' and no instance operation transforms a non-green node into a green one. Thus n can be grafted in τ.

For $\tau[\tau_n/n] \sqsubseteq \tau'$: consider an ordered derivation (Lemma 6.2.1) of $\tau \sqsubseteq \tau'$, and let τ_g be the type such that $\tau \sqsubseteq^G \tau_g \sqsubseteq^R ; \sqsubseteq^{MW} \tau'$. Without loss of generality, we can suppose that all the graftings under n occur first, as grafting under two distinct nodes commute. Let τ'_g be the type after those grafting. It suffices to prove that $\tau[\tau_n/n] \sqsubseteq \tau'_g$ to obtain the result.

By definition of grafting, n is closed in τ'_g, and we can project τ'_g at this node. Then it suffices to prove that $\triangle(\dot{\tau}'/n) \sqsubseteq \tau'_g/n$ (which we do below). Indeed, we can transform such a derivation I into a derivation I' of $\tau[\tau_n/n] \sqsubseteq \tau'_g$ by changing any operation o of I on a node n' in an operation on $n \cdot n'$.

By Lemma 6.4.2, we have $\triangle(\dot{\overline{\tau'_g/n}}) \sqsubseteq \tau'_g/n$. Since no grafting occurs under n between τ'_g and τ', we have $\dot{\tau}'/n = \dot{\tau}_g{}'/n$. Since moreover $\dot{\overline{\tau'_g/n}}$ is equal to $\dot{\tau}_g{}'/n$, we have $\triangle(\dot{\tau}'/n) = \triangle(\dot{\tau}_g{}'/n) = \triangle(\dot{\overline{\tau'_g/n}})$. This proves $\triangle(\dot{\tau}'/n) \sqsubseteq \tau'_g/n$, which is the desired result.

As a direct consequence:

Definition 6.4.4 (*Minimal grafting)* Let τ and τ' be two types such that $\tau \sqsubseteq \tau'$. Let $n_i^{i\in 1..k}$ be the bottom nodes of τ that are not bottom nodes in τ'. The *minimal grafting of τ w.r.t. τ'*, written $\tau[\tau'/\bot]$, is defined by[1]

$$\tau[\tau'/\bot] \quad \triangleq \quad \tau[\tau_{n_1}/n_1]\ldots[\tau_{n_k}/n_k] \quad \text{where } \tau_{n_i} = \triangle(\dot{\tau}'/n_i) \qquad \blacksquare$$

Corollary 6.4.5 *Let τ' be an instance of a type τ. The relation $\tau \sqsubseteq^G \tau[\tau'/\bot] \sqsubseteq \tau'$ holds.*$\square$

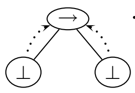
Notice that $\tau[\tau'/\bot]$ is the smallest type τ'' (w.r.t. the ordering induced by the instance relation) such that $\tau \sqsubseteq^G \tau'' \sqsubseteq \tau'$ holds and $\dot{\tau}'$ and $\dot{\tau}''$ coincide. Indeed, the derivation of $\tau[\tau'/\bot] \sqsubseteq \tau'$ does not use any grafting,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

6.4.2 Constructor type

In order to be even more small-step, we further decompose the grafting operation: instead of grafting the entire widening of a term, we create the widening node by node.

Definition 6.4.6 (*Constructor type)* Let C be a type constructor. The *constructor type* for C is the type whose root is labelled by C, and whose children are all distinct, flexibly bound, and labelled by $\bot$. $\blacksquare$

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

▶ **Example** The constructor type for the type int is the type reduced to a single node labelled by int. The constructor type for the arrow constructor is the type  .

Lemma 6.4.7 *Given an instance τ' of a type τ, there exists instance derivation of $\tau \sqsubseteq \tau'$ of the form $\tau \sqsubseteq^G \tau_g \sqsubseteq^{RMW} \tau'$, with all operations in $\tau \sqsubseteq^G \tau_g$ grafting constructor types.* □

Proof: By Lemma 6.2.1 we can consider an ordered derivation $\tau \sqsubseteq^G \tau_g \sqsubseteq^R ; \sqsubseteq^{MW} \tau$ of $\tau \sqsubseteq \tau'$. It is then immediate that $\tau \sqsubseteq^G \tau_g$ can have the required shape, by Corollary 6.4.5, the definition of widening and the definition of constructor types.

6.5 Canonical derivations

As a summary of §6.2, §6.3 and §6.4, we introduce the notion of canonical derivations. (The name is slightly improper, as there usually exists more than one canonical derivation. Derivations can be made fully canonical by adding *e.g.* a left-to-right bias.)

Definition 6.5.1 (*Canonical instance derivation*) An instance derivation $\tau \sqsubseteq \tau'$ is *canonical* if it is of the form $\tau \sqsubseteq^G \tau_g \sqsubseteq^R \tau_r \sqsubseteq^{MW} \tau'$, and if

- all types grafted in $\tau \sqsubseteq^G \tau_g$ are constructor types;
- the raisings in $\tau \sqsubseteq^R \tau_r$ are done bottom-up, as per Definition 6.3.2;
- the operations in $\tau_r \sqsubseteq^{MW} \tau'$ are done bottom-up, as per Definition 6.3.6. ■

As an immediate corollary of our previous results, we can always assume without loss of generality that an instance derivation is canonical.

Property 6.5.2 *Given an instance τ' of a type τ, there exists a canonical derivation of $\tau \sqsubseteq \tau'$.* □

6.6 Performing an instance operation early

Given a derivation $\tau \sqsubseteq \tau'$ containing an operation o, it is sometimes necessary to move this operation at the beginning of the derivation. This section shows that, when this operation can be applied to τ, the relation $\tau \sqsubseteq o(\tau) \sqsubseteq \tau'$ holds, with some restrictions only if o is a weakening.

Lemma 6.6.1 *Let τ and τ' be such that $\tau \sqsubseteq \tau'$. Let n be a node of τ such that $\tau(n) = \bot$ and $\tau'(n) \neq \bot$. Let τ_n be the constructor type for $\tau'(n)$. The relation $\tau \sqsubseteq^G_1 \mathsf{Graft}(\tau_n, n)(\tau) \sqsubseteq \tau'$ holds.* □

Proof: Necessarily n has green permissions in τ, as it must be grafted in the derivation $\tau \sqsubseteq \tau'$. Hence $\tau \sqsubseteq^G_1 \mathsf{Graft}(\tau_n, n)(\tau)$ holds. For the second part of the conclusion, consider a canonical derivation $\tau \sqsubseteq^G \tau'_g \sqsubseteq^R ; \sqsubseteq^{MW} \tau'$. The type grafted under n in this derivation is τ_n by construction of canonical derivations. We can move this grafting first, by commuting it with the other graftings. Thus, $\mathsf{Graft}(\tau_n, n)(\tau) \sqsubseteq \tau'_g$ holds, which implies the result.

Lemma 6.6.2 *Let τ and τ' be such that $\tau \sqsubseteq \tau'$. Let n be a node such that $\tau \sqsubseteq^R_1 \mathsf{Raise}(n)(\tau)$ holds, and $\hat{\tau}(n) \neq \hat{\tau}'(n)$. Then $\mathsf{Raise}(n)(\tau) \sqsubseteq \tau'$ holds.* □

Proof: Let us call $\tau'' = \mathsf{Raise}(n)(\tau)$. Consider a canonical derivation $\tau \sqsubseteq^G \tau_g \sqsubseteq^R \tau_r \sqsubseteq^{MW} \tau'$ of $\tau \sqsubseteq \tau'$. It is sufficient to prove $\tau'' \sqsubseteq^G ; \sqsubseteq^R \tau_r$.

The grafting operations I_G transforming τ into τ_g can be applied unchanged to τ', as grafting and raising commute. Thus, let τ'_g be $I_G(\tau'')$, which is also equal to $\mathsf{Raise}(n)(\tau_g)$ (**1**). The relation $\tau'' \sqsubseteq^G \tau'_g$ holds: raising increases permissions. Hence it suffices to prove $\tau'_g \sqsubseteq^R \tau_r$, or equivalently $\tau'_g \sqsubseteq^{R\natural} \tau_r$ (Lemma 6.3.4).

The points 1 and 2 of Definition 6.3.1 are immediate. For point 4, let us prove that all the nodes bound differently in τ'_g and τ_r are also bound differently in τ_g and τ_r, which implies the result as $\tau_g \sqsubseteq^{R\natural} \tau_r$. By (1), this result is immediate for all the nodes but n. For n, by hypothesis $\hat{\tau}(n) \neq \hat{\tau}'(n)$, and moreover $\hat{\tau}(n) = \hat{\tau_g}(n)$ and $\hat{\tau}'(n) = \hat{\tau_r}(n)$, by definition of canonical derivations. Thus $\hat{\tau_g}(n) \neq \hat{\tau_r}(n)$ (**2**).

Thus it remains to prove that $\hat{\tau_r} \subseteq (\hat{\tau_g}')^+$ (for point 3). Since $\tau_g \sqsubseteq^R \tau_r$, we also have $\tau_g \sqsubseteq^{R\natural} \tau_r$ and $\hat{\tau_r} \subseteq (\hat{\tau_g})^+$. The binding trees of τ_g and τ'_g differ only by the binding edge on n. Suppose $\hat{\tau_r} \subseteq (\hat{\tau_g}')^+$ does not hold. Then, there necessarily exists a node n' of τ_g such that $n' \xrightarrow{*} n \longrightarrow \hat{\tau_g}(n) \in \tau_g$ (**3**) and $n' \longrightarrow \hat{\tau_g}(n) \in \tau_r$ (**4**). By (2) we have $n' \neq n$. Hence $n' \xrightarrow{+} n \longrightarrow \hat{\tau_g}(n) \in \tau_g$. In τ_r, consider a mixed path P of the form $\langle\epsilon\rangle \xleftarrow{*} \hat{\tau_r}(n) \longleftarrow n \xrightarrow{+} n'$. By (4), $\hat{\tau_r}(n') = \hat{\tau_g}(n)$, and this node is strictly below $\hat{\tau_r}(n)$ (by (3) and because n is raised at least once between τ_g and τ_r), and strictly above n. Hence P does not contain $\hat{\tau_r}(n')$: this contradicts the well-formedness of τ_r.

Lemma 6.6.3 *Let τ and τ' be such that $\tau \sqsubseteq \tau'$. Suppose there exists n_1 and n_2 merged in τ' such that $\tau \sqsubseteq^M_1 \mathsf{Merge}(n_1, n_2)(\tau)$ holds. Then $\mathsf{Merge}(n_1, n_2)(\tau) \sqsubseteq \tau'$ holds.* □

Proof: Let τ'' be $\mathsf{Merge}(n_1, n_2)(\tau)$. Consider a canonical derivation $\tau \sqsubseteq^G \tau_g \sqsubseteq^R \tau_r \sqsubseteq^{MW} \tau'$ of $\tau \sqsubseteq \tau'$. We are going to prove that $\tau'' \sqsubseteq^G \tau'_g \sqsubseteq^R \tau'_r \sqsubseteq^{MW\natural} \tau'$, where $\tau'_g = \tau_g[n_1 = n_2]$ and $\tau'_r = \tau_r[n_1 = n_2]$. (It is immediate that n_1 and n_2 are binding-congruent in τ_g and τ_r, as they can be merged in τ, and are merged in τ'.) We call *twin nodes* two distinct nodes n'_1 and n'_2 such that there exists π such that $n'_1 = \langle n_1 \cdot \pi\rangle$ and $n'_2 = \langle n_2 \cdot \pi\rangle$. Notice that n'_1 cannot be above n'_2, as τ' would be cyclic; symmetrically, n'_2 cannot be above n'_1.

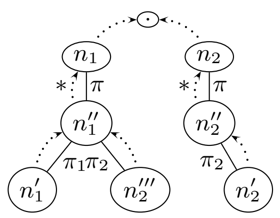
Let us consider a derivation I_g of $\tau \sqsubseteq^G \tau_g$. We can commute the grafting in I_g such that two twin nodes are grafted immediately one after the other, as the graftings of two nodes where neither one is above the other commute. Then, the derivation I'_g defined by removing from I_g the occurrences of $\mathsf{Graft}(\tau'_2, n'_2)$ when I_g contains $\mathsf{Graft}(\tau'_1, n'_1) ; \mathsf{Graft}(\tau'_2, n'_2)$, and n'_1 and n'_2 are twin nodes is a witness of $\tau'' \sqsubseteq^G \tau'_g$.

Next, consider a derivation I_r of $\tau_r \sqsubseteq^R \tau_r$. Again, we can commute the different raisings so that two twin nodes are raised one after the other, as raising two nodes not one above the other commute. As above, we can transform I_r into I'_r by removing the raising of the second twin node, and I'_r is a witness of $\tau'_g \sqsubseteq^R \tau'_r$.

It remains to prove that $\tau_r' \sqsubseteq^{MW\natural} \tau'$. By Lemma 6.3.8, we already have $\tau_r \sqsubseteq^{MW\natural} \tau'$ (**1**). The points 1, 2, 5 and 6 of Definition 6.3.5 are immediate. For point 3, the relation $\tilde{\tau_r}' \subseteq \tau'$ is a consequence of (1) and of the fact that n_1 and n_2 are merged in τ'. Finally, consider two nodes n_1' and n_2' verifying the hypotheses of point 4 in τ_r', *i.e.* $n_1' \not\tilde{\tau}_r' n_2'$ (**2**), $n_1' \tilde{\tau}' n_2'$ (**3**) and $\hat{\tau_r}'(n_1') = \hat{\tau_r}'(n_2')$ (**4**). We need to prove that n_1' and n_2' are not red in τ_r'. Without loss of generality, we assume that n_1' and n_2' are expressed as nodes of τ_r.

By (2) and the definition of τ_r', we have $n_1' \not\tilde{\tau}_r n_2'$ (**5**). If $\hat{\tau}_r(n_1') = \hat{\tau}_r(n_2')$, by (1), (5) and (3), n_1' and n_2'' are not red in τ_r (**6**). Otherwise, if $\hat{\tau}_r(n_1') \neq \hat{\tau}_r(n_2')$, by (4) and the definition of τ_r', the nodes n_1', n_2' are under n_1 and n_2 respectively. Moreover, since n_1 and n_2 are binding congruent in τ_r, the binders in τ_r of n_1' and n_2' must also be under and n_1 and n_2 (otherwise, we would have $\hat{\tau}_r(n_1') = \hat{\tau}_r(n_2')$). In this case, n_1' and n_2' are indirectly merged, and we cannot use (6). However there exists two nodes symmetrical to n_1' and n_2' which are merged directly. We detail this case below.

Let π, π_1 and π_2 be such that $\hat{\tau}_r(n_i') = \langle n_i \cdot \pi \rangle$ for $1 \leq i \leq 2$, $n_1' = \langle n_1'' \cdot \pi_1 \rangle$ and $n_2' = \langle n_2'' \cdot \pi_2 \rangle$. Let also n_2''' be $\langle n_1'' \cdot \pi_2 \rangle$. Since n_1 and n_2 are merged in τ', and by (3), we have $n_1' \tilde{\tau}' n_2'''$ (**7**). Moreover, n_1' and n_2''' are distinct in τ_r (**8**), as otherwise (2) would not hold. Finally, $\hat{\tau}_r(n_1') = \hat{\tau}_r(n_2''')$ (**9**), by definition of π, π_1 and π_2 and τ_r. Thus, by (1), (8), (7) and (9), n_1' and n_2''' are not red in τ_r (and by symmetry n_2' is not red in τ_r either) (**10**).



In both cases ((6) and (10)), n_1' and n_2' are not red in τ_r. Fusion preserves permissions as it does not change the binding tree. Hence n_1' and n_2' are not red in τ_r' either, which is the desired result.

For weakening, the result does not hold in the general case: by weakening too early, we might prevent some valid transformations later in the derivation. However, if the node n to be weakened must be merged with a rigidly bound node n', and both nodes are congruent (up to the binding edge of n), we can use n' as a "witness": indeed, the transformations which would become impossible under n are already impossible under n'. Alternatively, if n is inert, weakening n does not change the permissions of the other nodes, and the result also holds.

Lemma 6.6.4 *Let τ and τ' be such that $\tau \sqsubseteq \tau'$. Let n be a node flexibly bound in τ and rigidly bound in τ'. Suppose that $\tau \sqsubseteq_1^W \mathsf{Weaken}(n)(\tau)$ holds. Suppose also that either*

1. *n is inert;*
2. *there exists a node n' rigidly bound in τ, merged with n in τ', such that the subgraphs consisting of the nodes transitively bound on n or n' are the same in τ.*

Then $\mathsf{Weaken}(n)(\tau) \sqsubseteq \tau'$ holds. □

Proof: In this proof proof, we use «under» for «transitively bound on».

Let τ'' be $\mathsf{Weaken}(n)(\tau)$. Consider a canonical derivation $\tau \sqsubseteq^G \tau_g \sqsubseteq^R \tau_r \sqsubseteq^{MW} \tau'$ of $\tau \sqsubseteq \tau'$. Let τ_g' be $\mathsf{Weaken}(n)(\tau_g)$ and $\tau_r' = \mathsf{Weaken}(n)(\tau_r)$. Those types exist, as n is flexibly bound in both τ_g and τ_r, as it is flexibly bound in τ. We are going to prove that $\tau'' \sqsubseteq^G \tau_g' \sqsubseteq^R \tau_r' \sqsubseteq^{MW\natural} \tau'$.

Consider three instance derivations I_g, I_r and I_m transforming τ into τ_g, τ_g into τ_r and τ_r into τ' respectively. We first prove that none of these derivations can contain an operation on a green node under n (**1**).

▷ <u>*Subcase* 1 (n is inert)</u>: immediate, as there is no green node under n.

▷ <u>*Subcase* 2</u>: Let $G_\tau(n)$ be the subgraph of the nodes under n in τ. Since $G_\tau(n) = G_\tau(n')$ and n and n' are merged in τ', $G_{\tau'_g}(n) = G_{\tau'_g}(n')$ and $G_{\tau'_r}(n) = G_{\tau'_r}(n')$ necessarily hold. (**2**). Thus, the symmetrical node under n' needs to be transformed in the same way. However, since n' is rigidly bound in τ and $G_\tau(n) = G_\tau(n')$, this symmetrical node is red in τ (and in all its instances, since instance preserve red nodes).

Next, by Lemma 5.4.1, permissions decrease for $\prec_{\sqsubseteq}$ in τ'_g and τ'_r (compared to τ_g and τ_r) only for n and the green nodes under n (**3**). Moreover, by the same lemma, n itself is either orange (if it was green in τ), or has the same permissions in τ and τ'' (in all the other cases). In particular, n is not red (**4**).

Let us now prove our main result. By (1), no grafting takes place under n; hence I_g witnesses $\tau'' \sqsubseteq^G \tau'_g$. Moreover, by (1), (3) and (4), all the nodes under n and raised between τ_g and τ_r can still be raised in τ'_r, as they are not red. Hence I'_r witnesses $\tau'_g \sqsubseteq^R \tau'_r$.

It remains to prove that $\tau'_r \sqsubseteq^{MW} \tau'$. We cannot use I_m, as n is already red in τ'_r. However, by Lemma 6.3.8, we have $\tau_r \sqsubseteq^{MW\natural} \tau'$ (**5**), and it suffices to prove $\tau'_r \sqsubseteq^{MW\natural} \tau'$. The points 1, 2, 3 and 5 of Definition 6.3.5 are immediate by (5) and the definition of τ'_r (for point 5). For points 4 and 6, the nodes to transform between τ'_r and τ' are the same as those that must be transformed between τ_r and τ', up to n. All those nodes but n still have non-red permissions in τ'_r by (1) and (3), while n is not red by (4). Thus $\tau'_r \sqsubseteq^{MW\natural} \tau'$ holds, which is the desired result.

6.7 Reorganizing the instance modulo relations

In this section, we study the relationships between $\sqsubseteq$, $\sqsubseteq^{\approx}$ and $\sqsubseteq^{\boxminus}$, and show that the last two relations can be reorganized so that all the instance steps can occur first. In particular, this is the first step in proving that using $\sqsubseteq^{\approx}$ as the instance relation does not significantly increase the expressiveness of MLF.

6.7.1 Confluence of the instance relations

As a preliminary result, this section studies the confluence of all the subrelations of instance, including the subrelations of $\boxminus$ and $\sqsubseteq^{rmw}$. Since we reason simultaneously on $\sqsubseteq$, $\boxminus$ and $\sqsubseteq^{rmw}$, we must consider all three orders $\prec_{\sqsubseteq}$, $\prec_{\boxminus}$ and $\prec_{rmw}$. By Property 5.4.5, we thus have the following results for those orders:

- grafting and raising increase or preserve permissions;
- merging preserves permissions;
- weakening is not monotonic w.r.t. those three orders.

In the following, we consider two relations $\dot{\sqsubset}$ and $\dot{\boxminus}$ which range independently over $\sqsubseteq$, $\boxminus$ and $\sqsubseteq^{rmw}$. As usual, we write *e.g.* $\dot{\sqsubset}_1$ the restriction of $\dot{\sqsubset}$ to one-step instance, and $\dot{\sqsubset}^X$ for $X \in \{G, R, M, W\}$ the relation $\dot{\sqsubset} \cap \sqsubseteq^X$. Our goal is to show that, if $\dot{\sqsubset}$ and $\dot{\boxminus}$ are not simultaneously $\sqsubseteq$, they are locally confluent. The remainder of the section proceeds by case disjunction on the subrelations of $\dot\sqsubset$ and $\ddot\sqsubset$. However, before doing so, we introduce a small technical result which rules out some impossible cases.

Lemma 6.7.1 *Consider a type τ, and two nodes n and n' such that $n \xrightarrow{*} n'$. Let o and o' be two operations of $\dot\sqsubset$ and $\ddot\sqsubset$ respectively, such that $\tau \dot\sqsubset o(n)(\tau)$ and $\tau \ddot\sqsubset o'(n')(\tau)$. Then we also have $\tau \ddot\sqsubset o(n)(\tau)$.* □

Proof: We proceed by case disjunction on $\ddot\sqsubset$ and $\dot\sqsubset$.

- ▷ *Case* $\ddot\sqsubset = \sqsubseteq^{rmw}$: then n' is a monomorphic node. So is n, as monomorphic nodes are downwards-closed. Thus $o(n)$ is also in $\ddot\sqsubset$.
- ▷ *Case* $\ddot\sqsubset = \sqsubseteqq$ and $\dot\sqsubset = \sqsubseteq$: then n' is orange or inert. Green nodes are upwards-closed, hence n is not green. This shows that $o(n)$ is also in $\ddot\sqsubset$.
- ▷ In all the other cases: we have $\dot\sqsubset \subseteq \ddot\sqsubset$, hence the result.

Lemma 6.7.2 *The following diagram is verified*

$$\begin{tikzcd}
\tau \arrow[r, "\dot\sqsubset_1^M"] \arrow[d, "\ddot\sqsubset_1^M"'] & \tau_{12} \arrow[d, dashed, "\ddot\sqsubset^M"] \\
\tau_{34} \arrow[r, dashed, "\dot\sqsubset^M"'] & \cdot
\end{tikzcd}$$

□

Proof: Let n_1, n_2, n_3 and n_4 be the four nodes such that $\tau_{12} = \mathsf{Merge}(n_1, n_2)(\tau)$ and $\tau_{34} = \mathsf{Merge}(n_3, n_4)(\tau)$ (in particular, $n_1 \neq n_2$ and $n_3 \neq n_4$). The degenerate case where $\{n_1, n_2\} = \{n_3, n_4\}$ is immediate, as $\tau_{12} = \tau_{34}$. We proceed by case disjunction.

- ▷ If $n_1 \xrightarrow{+} n_3$ holds (or one of the symmetrical cases):

  By definition of merging, $n_2 \xrightarrow{+} n_3$ also holds. The type

  $$\tau' = \mathsf{Merge}(n_1, n_2)(\tau_{34}) = (\mathsf{Merge}(n_1', n_2') ; \mathsf{Merge}(n_3, n_4))(\tau_{12})$$

  (where n_1' and n_2' are the nodes corresponding to n_1 and n_2 under n_4) closes the diagram. Indeed, by definition of τ', $\tau_{34} \dot\sqsubset^M \tau'$ and τ_{12} $(\dot\sqsubset^M ; \ddot\sqsubset^M)$ τ' hold, since merging preserves permissions. Lemma 6.7.1 shows that $\tau_{12} \ddot\sqsubset^M \tau'$ holds.
- ▷ In all the other cases: we show that the two operations commute, *i.e.* that the type

  $$\tau' = \mathsf{Merge}(n_1, n_2)(\tau_{34}) = \mathsf{Merge}(n_3, n_4)(\tau_{12})$$

  closes the diagram. Since merging preserves permissions, it suffices to prove that one of the merging does not prevent the other by changing the term-graph under only one of the two nodes.

  Without loss of generality, suppose that merging n_1 and n_2 changes the term-graph under n_3 or n_4; we can also suppose that $n_3 \xrightarrow{+}\!\circ\ n_1$. Let π be such that $n_3 \xrightarrow{\pi}\!\circ\ n_1$. Since $n_1 \not\xrightarrow{+} n_3$ (otherwise we would be in the first case), by well-domination we have $\hat{n_1} \xrightarrow{+}\!\circ\ n_3$. By Lemma 4.3.4, we have $n_3 \xrightarrow{+} \hat{n_1}$. By local congruence of n_3 and n_4, since n_1 is not transitively bound on n_3, we have $\langle n_3\pi\rangle = \langle n_4\pi\rangle$. Since $\langle n_3\pi\rangle$ is n_1, the subgraph under n_1 is shared between n_3 and n_4, which is the desired result.

Lemma 6.7.3 *The following diagram is verified*

$$\begin{tikzcd}
\tau \arrow[r, "\sqsubseteq_1^R"] \arrow[d, "\sqsubseteq_1^R"'] & \tau_1 \arrow[d, dashed, "\sqsubseteq^R"] \\
\tau_2 \arrow[r, dashed, "\sqsubseteq^R"'] & \cdot
\end{tikzcd}$$

□

Proof: Let n_1 and n_2 be the two nodes such that $\tau_1 = \mathsf{Raise}(n_1)(\tau)$ and $\tau_2 = \mathsf{Raise}(n_2)(\tau)$. The degenerate case $n_1 = n_2$ is immediate. We distinguish between two cases:

▷ If $n_1 \longrightarrow n_2 \in \tau$ (or the symmetric case): we show that τ' below closes the diagram

$$\tau' = \mathsf{Raise}(n_1)(\tau_2) = (\mathsf{Raise}(n_1) \,;\, \mathsf{Raise}(n_2))(\tau_1)$$

This type is well-dominated. Indeed, n_1 is raisable in τ_2 as n_1 is raisable in τ, $\breve{\tau} = \breve{\tau_2}$, and the nodes bound on $\hat{\tau}(n_1)$ (which is also n_2 and $\hat{\tau_2}(n_1)$) are the same in τ and τ_2. Since raising increases permissions, we have so far proven $\tau_2 \sqsubseteq^R \tau'$ and τ_1 $(\sqsubseteq^R \,;\, \sqsubseteq^R)$ τ'. Lemma 6.7.1 shows that $\tau_1 \sqsubseteq^R \tau'$ holds.

▷ In all the other cases: we show that the two operations commute, and that the diagram is closed by

$$\tau' = \mathsf{Raise}(n_1)(\tau_2) = \mathsf{Raise}(n_2)(\tau_1)$$

Since raising increases permissions, it suffices to justify that τ' is well-dominated. We do this by showing that n_2 is raisable in τ_1. Consider $n_3 = \hat{\tau_1}(n_2)$. This node is also $\hat{\tau}(n_2)$, as n_2 has not been raised in τ_1. Let n_2' be a node bound on n_3 in τ_1 other than n_2. We need to show that $n_2 \overset{+}{\multimap} n_2' \in \tau_1$ does not hold.

∘ If n_2' is n_1: by case hypothesis, we know that n_1 was not bound on n_2 in τ. Thus, let n_2'' be the node such that $n_1 \longrightarrow n_2'' \longrightarrow n_3 \in \tau$. By contradiction, suppose that $n_2 \overset{+}{\multimap} n_1$ holds in τ_1. This relation also holds in τ. Consider a mixed path $\langle\epsilon\rangle \mathrel{\triangleleft\!\overset{*}{-}\!\circ} n_3 \longleftarrow n_2 \overset{+}{\multimap} n_1$ in τ. By well-domination, n_2'' is contained in this path. Necessarily, it is in the path $n_2 \overset{+}{\multimap} n_1$ (as it is below n_3). Thus, since $n_2 \neq n_2'$, we have $n_2 \overset{+}{\multimap} n_2''$ in τ, both nodes being bound on n_3. This contradicts the fact that n_2 is raisable in τ, and $n_2 \overset{+}{\multimap} n_1$ does not hold.

∘ If n_2' is not n_1: then n_2' is also bound on n_3 in τ. Since n_2 is raisable in τ, we have $n_2 \not\multimap n_2'$. This relation still holds in τ_1.

Lemma 6.7.4 *The following diagram is verified*

$$\begin{tikzcd}
\tau \arrow[r, "\sqsubseteq_1^M"] \arrow[d, "\sqsubseteq_1^R"'] & \tau_m \arrow[d, dashed, "\sqsubseteq"] \\
\tau_r \arrow[r, dashed, "\sqsubseteq"'] & \cdot
\end{tikzcd}$$

□

Proof: Let n_1, n_2 and n be such that $\tau_m = \mathsf{Merge}(n_1, n_2)(\tau)$ and $\tau_r = \mathsf{Raise}(n)(\tau)$. We proceed by case disjunction on the position of n w.r.t. n_1 and n_2. In the following we do not detail permissions, as raising and merging increase them.

▷ If $n \xrightarrow{*\triangleright} n_1$ (or the symmetric case): we show that the type below closes the diagram

$$\tau' = \mathsf{Raise}(n)(\tau_m)$$

Let us first justify that n is raisable in τ_m. By contradiction, suppose there exists n' bound on $\overrightarrow{\tau_m}(n)$ such that $n \xrightarrow{+\circ} n' \in \tau_m$. Let π be such that $n' = \langle n \cdot \pi \rangle$. By definition of congruent nodes, the node $\langle n \cdot \pi \rangle$ is also in τ. Moreover, by definition of binding congruent-nodes, we have $\hat{\tau}(n) = \hat{\tau}(n')$. This contradicts the fact that n is raisable in τ. Hence n is indeed raisable in τ_m and $\tau_m \sqsubseteq^R \tau'$ holds.
Next, we prove that $\tau_r \sqsubseteq \tau'$. We proceed by case disjunction on $n \xrightarrow{*\triangleright} n_1$.

- *Case $n = n_1$*: then $\tau' = (\mathsf{Raise}(n_2)\ ;\ \mathsf{Merge}(n_1, n_2))(\tau_r)$
  The fact that n_2 is raisable in τ_r is by symmetry with n_1 in τ, as the subgraph under n_2 in τ_r is the same as the subgraph under n_1 in τ. Moreover, n_1 and n_2 are locally congruent in $\mathsf{Raise}(n_2)(\tau_r)$, as the subgraphs under them are unchanged from τ. Thus $\tau_r \sqsubseteq^R ; \sqsubseteq^M \tau'$ holds. Lemma 6.7.1 shows that $\tau_r \sqsubseteq \tau'$ also holds.
- *Case $n \xrightarrow{\triangleright} n_1$*: then $\tau' = (\mathsf{Raise}(n^s)\ ;\ \mathsf{Merge}(n, n^s)\ ;\ \mathsf{Merge}(n_1, n_2))(\tau_r)$, n^s being the node symmetric to n under n_2.
  n^s can be raised in τ_r by symmetry with n in τ. Let τ_r' be $\mathsf{Raise}(n^s)(\tau_r)$ and $\tau_r'' = \mathsf{Merge}(n, n^s)(\tau_r')$. The nodes n and n^s are binding-congruent in τ_r', since n_1 and n_2 are binding-congruent in this type, and n and n^s are under n_1 and n_2. Similarly, n_1 and n_2 are binding-congruent in τ_r''. Let us prove that both sets of nodes are locally congruent.
  - Merging n and n^s in τ_r': Consider a non-empty path π such that $\langle n\pi \rangle$ and $\langle n^s\pi \rangle$ are distinct in τ_r'. Let us call n_1' and n_2' those two nodes; we must prove that they are bound on n and n^s respectively.
    By definition of τ_r', for $1 \leq i \leq 2$, we have $\hat{\tau_r}'(n_i') = \hat{\tau}(n_i')$. By local congruence of n_1 and n_2 in τ, we have $n_i' \xrightarrow{+\triangleright} n_i$. The case $n_i' \xrightarrow{\triangleright} n_i$ is impossible, as n and n^s would not have been raisable in τ. Thus $n_i' \xrightarrow{+\triangleright}\xrightarrow{\triangleright} n_i$. By well-domination, necessarily the nodes under n_i in those paths are n and n^s, *i.e.* $n_1' \xrightarrow{+\triangleright} n$ and $n_2' \xrightarrow{+\triangleright} n^s$. This is the desired result.
  - Merging n_1 and n_2 in τ_r'': Consider a non-empty path π such that $\langle n_1\pi \rangle$ and $\langle n_2\pi \rangle$ are distinct in τ_r''. Necessarily, $\langle n_1\pi \rangle$ and $\langle n_2\pi \rangle$ are not n, n^s as those nodes are merged in τ_r'' (**1**). By local congruence of n_1 and n_2 in τ, for $1 \leq i \leq 2$. $\langle n_i\pi \rangle \xrightarrow{+\triangleright} n_i \in \tau$. Since $\hat{\tau}(n_i\pi) = \widehat{\tau_r''}(n_i\pi)$ (by (1)), we have $\langle n_i\pi \rangle \xrightarrow{+\triangleright} n_i \in \tau_r''$. This is the desired result.

  So far, we have proven $\tau_r\ (\sqsubseteq^R ; \sqsubseteq^M ; \sqsubseteq^M)\ \tau'$. By Lemma 6.7.1, we have in fact $\tau_r \sqsubseteq \tau'$, which is the desired result.
- *Case $n \xrightarrow{+\triangleright}\xrightarrow{\triangleright} n_1$*: then $\tau' = (\mathsf{Raise}(n^s) ; \mathsf{Merge}(n, n^s))(\tau)$, n^s being again the node symmetric to n under n_2. This case is similar to the two above.

▷ In all the other cases: we show that the two operations commute, and that the diagram is closed by

$$\tau' = \mathsf{Merge}(n_1, n_2)(\tau_r) = \mathsf{Raise}(n)(\tau_m)$$

Raising n does not change the fact that n_1 and n_2 are locally congruent, as n is either strictly above n_1 and n_2, or on disjoint binding paths (and local congruence is only concerned with the nodes below n_1 and n_2). This shows that τ' is a correct type, which implies that n is raisable in τ_m by Property 4.4.13. Thus we have $\tau_m \sqsubseteq^R \tau'$ and $\tau_r \sqsubseteq^M \tau'$, which is the desired result.

The three previous results have shown that raising and merging are locally confluent. The confluence itself is immediate by Newman's Lemma, as raising and merging are noetherian.

Similar results do not hold for $\sqsubseteq^G$ and $\sqsubseteq^W$: grafting two different types at the same node usually results in incompatible types, while weakening two different nodes one above the other must be done bottom-up, as the top-down strategy is often forbidden by permissions. However, if at least one of the operations is an abstraction step, the various relations are confluent. The remainder of this section considers these subcases.

Lemma 6.7.5 *The relation $\sqsubseteq_1^G$ commutes with $\mathrel{\sqsubset\!\!\!-}$ and $\sqsubseteq^{rmw}$.* □

Proof: Immediate: the grafting takes place on a green bottom node, and this part of the type cannot be transformed by $\mathrel{\sqsubset\!\!\!-}$ or $\sqsubseteq^{rmw}$.

(More generally, $\sqsubseteq_1^G$ together with $\sqsubseteq_1^M$ or $\sqsubseteq_1^R$ are locally confluent. Since this result is not useful to us, we do not prove it here. However the proof is very similar to all the other proofs in this section.)

Lemma 6.7.6 *If $(\dot\sqsubset, \ddot\sqsubset) \neq (\sqsubseteq, \sqsubseteq)$, the following diagram is verified*

$$\begin{tikzcd}
\tau \arrow[r, "\dot\sqsubset_1^R"] \arrow[d, "\ddot\sqsubset_1^W"'] & \tau_r \arrow[d, dashed, "\ddot\sqsubset_1^W"] \\
\tau_w \arrow[r, dashed, "\dot\sqsubset_1^R"'] & \cdot
\end{tikzcd}$$

□

Proof: Let n and n' be such that $\tau \mathrel{\dot\sqsubset_1^R} \tau_r$ and $\tau \mathrel{\ddot\sqsubset_1^W} \tau_w$. The type

$$\tau' = \mathsf{Raise}(n)(\tau_w) = \mathsf{Weaken}(n')(\tau_r)$$

closes the diagram. Indeed:

- ▷ n' has more permissions in τ_r than in τ (raising increases permissions), and it is still flexibly bound. Hence it can be weakened by $\ddot\sqsubset^W$ in τ_r.
- ▷ If $\ddot\sqsubset^W$ is $\mathrel{\sqsubset\!\!\!-}^W$ or $\sqsubseteq^w$, n has the same permissions in τ_w and in τ, as only the weakening of a green node changes permissions (Lemma 5.4.1). Moreover n is still raisable in τ_w, as both $\check\tau = \check{\tau_w}$ and $\hat\tau = \hat{\tau_w}$.
  If $\ddot\sqsubset^W$ is the weakening of a green node, $\dot\sqsubset^R$ is either $\sqsubseteq^r$ or $\mathrel{\sqsubset\!\!\!-}^R$, and the permissions of orange, inert and monomorphic nodes are unchanged by $\ddot\sqsubset^W$ (Lemma 5.4.1).
  In both cases n can be raised by $\dot\sqsubset^R$ in τ_w.

Lemma 6.7.7 *If $(\dot\sqsubset, \ddot\sqsubset) \neq (\sqsubseteq, \sqsubseteq)$, the following diagram is verified*

$$\begin{tikzcd}
\cdot \arrow[r, "\dot\sqsubset_1^W"] \arrow[d, "\ddot\sqsubset_1^W"'] & \cdot \arrow[d, dashed, "\ddot\sqsubset_1^W"] \\
\cdot \arrow[r, dashed, "\dot\sqsubset_1^W"'] & \cdot
\end{tikzcd}$$

□

Proof: As for Lemma 6.7.6, the two operations commute. The reasoning on the permissions after the weakening is also the same.

Lemma 6.7.8 *If* $(\dot\sqsubset, \ddot\sqsubset) \neq (\sqsubseteq, \sqsubseteq)$, *the following diagram is verified*

$$\begin{tikzcd}
\tau \arrow[r, "\dot\sqsubset_1^M"] \arrow[d, "\ddot\sqsubset_1^W"'] & \tau_m \arrow[d, dashed, "\ddot\sqsubset"] \\
\tau_w \arrow[r, dashed, "\overline{\dot\sqsubset}"'] & \cdot
\end{tikzcd}$$

□

Proof: Let n, n_1 and n_2 of τ be such that $\tau_r = \mathsf{Merge}(n_1, n_2)(\tau)$ and $\tau_w = \mathsf{Weaken}(n)(\tau)$. We distinguish whether n is under or above n_1 or n_2:

▷ <u>If $n \xrightarrow{*} n_1$ (up to permutation of n_1 and n_2):</u> We prove that the graph

$$\tau' = \mathsf{Weaken}(n)(\tau_m) = (\mathsf{Weaken}(n^s)\ ;\ \mathsf{Merge}(n_1, n_2))(\tau_w)$$

where n^s is the node symmetric to n under n_2 closes the diagram. It is immediate that $\tau_m \ddot\sqsubset^W \tau'$, as merging preserves permissions.
For $\tau_w \dot\sqsubset \tau'$, we first have $\tau_w \ddot\sqsubset^W \mathsf{Weaken}(n^s)(\tau_w)$ (we call this last type τ''): n^s has the same permissions as n in τ by symmetry, and it also has the same permissions in τ and τ_w, as the weakening of n does not change the permissions of n^s, which is not above or below n. By Lemma 6.7.1, $\tau_w \dot\sqsubset^W \tau'$ also holds.
It remains to prove that $\tau'' \dot\sqsubset^M \tau'$. The local congruence of n_1 and n_2 in τ'' is an immediate consequence of the local confluence of those nodes in τ, as $\check{\tau''} = \check{\tau}$ and $\hat{\tau''} = \hat{\tau}$. For permissions, by Lemma 5.4.1, weakening only changes the permissions of green nodes, and only if the weakening occurs on a green node. If $\ddot\sqsubset^W$ is $\mathrel{-\!\!\!\sqsubset}^W$ or $\sqsubseteq^w$, the weakening is not on a green node. If $\ddot\sqsubset^W$ is $\sqsubseteq^W$, $\dot\sqsubset^M$ is not $\sqsubseteq^M$, and n_1 and n_2 are not green. Thus the permissions of n_1 and n_2 are the same in τ, τ_w and τ', and $\tau'' \dot\sqsubset^M \tau'$ holds.

▷ <u>If $n_1 \xrightarrow{+} n$ (up to permutation of n_1 and n_2):</u> the two operations commute, and the type τ' defined below closes the diagram.

$$\tau' = \mathsf{Weaken}(n)(\tau_m) = \mathsf{Merge}(n_1, n_2)(\tau_w)$$

It is immediate that $\tau_m \ddot\sqsubset^W \tau'$ as merging preserves permissions. For $\tau_w \dot\sqsubset^M \tau'$, the reasoning is exactly the same as for proving $\tau'' \dot\sqsubset^M \tau'$ in the previous case.

▷ <u>In all the other cases</u>: the two transformations commute, as they occur on disjoint binding paths.

6.7.2 Reorganizing the instance modulo relations

The results of the previous section show that any combination of $\sqsubseteq$, $\sqsubseteq^{rmw}$ or $\mathrel{-\!\!\!\sqsubset}$ with either $\mathrel{-\!\!\!\sqsubset}$ or $\sqsubseteq^{rmw}$ is locally confluent. This result generalizes to confluence.

Lemma 6.7.9 *If $\ddot\sqsubset$ is not $\sqsubseteq$, the following diagram is verified*

$$\begin{tikzcd}
\cdot \arrow[r, "\dot\sqsubset"] \arrow[d, "\ddot\sqsubset"'] & \cdot \arrow[d, dashed, "\ddot\sqsubset"] \\
\cdot \arrow[r, dashed, "\dot\sqsubseteq"'] & \cdot
\end{tikzcd}$$

□

Proof: Let τ, τ_1 and τ_2 be such that $\tau \dot\sqsubset \tau_1$ and $\tau \ddot\sqsubset \tau_2$. In particular we have $\tau \dot\sqsubset|_{\tau_1} \tau_1$, where $\dot\sqsubset|_{\tau_1}$ is defined by restricting $\dot\sqsubset$ as we did for $\sqsubseteq$ in Definition 6.1.3. Since both $\dot\sqsubset|_{\tau_1}$ and $\ddot\sqsubset$ are noetherian, by Newman's lemma it suffices to show that $\dot\sqsubset|_{\tau_1}$ and $\ddot\sqsubset$ are locally confluent.

Consider three types τ', τ'_1 and τ'_2 such that $\tau' \dot\sqsubset|_{\tau_1} \tau'_1$ and $\tau' \ddot\sqsubset \tau'_2$. By Lemmas 6.7.2, 6.7.3, 6.7.4, 6.7.5, 6.7.6, 6.7.7 and 6.7.8, $\dot\sqsubset$ and $\ddot\sqsubset$ are locally confluent, *i.e.* there exists τ'' such that $\tau'_1 \ddot\sqsubset \tau''$ and $\tau'_2 \dot\sqsubset \tau''$. By Property 6.1.2 applied to $\tau'_1 \ddot\sqsubset \tau''$, τ'' is as structure-defined as τ'. Thus $\tau'_2 \dot\sqsubset|_{\tau_1} \tau''$ also holds, which is the desired result.

As a direct consequence, alternating $\sqsubseteq$ and $\exists$ steps inside $\sqsubseteq^{\boxminus}$ does not augment its expressivity. Instead, all $\exists$ steps can be done in one step and be pushed at the end of the derivation. Similar results hold by considering $\boxminus$, $\sqsubseteq^{\approx}$ and $\approx$ instead of $\sqsubseteq^{\boxminus}$.

Lemma 6.7.10 *The following equalities between relations are verified:*

$$(\boxminus) = (\mathrel{\mathsf{E}} \,;\, \exists) \qquad (\approx) = (\sqsubseteq^{rmw} \,;\, \sqsupseteq^{rmw}) \qquad (\sqsubseteq^{\boxminus}) = (\sqsubseteq \,;\, \exists) \qquad (\sqsubseteq^{\approx}) = (\sqsubseteq \,;\, \sqsupseteq^{rmw})$$ □

Proof: In each case, one inclusion is by definition. The other inclusion is by induction on the number of inversions of the form $\ddot\exists \,;\, \dot\sqsubset$, which rewrite to $\dot\sqsubset \,;\, \ddot\exists$ by Lemma 6.7.9.

Importantly, $\sqsubseteq^{\approx}$ and $\sqsubseteq^{\boxminus}$ are however *not* equal to $(\sqsupseteq^{rmw} \,;\, \sqsubseteq)$ and $(\exists \,;\, \sqsubseteq)$: some of the operations which would need to be done at the beginning of the derivation would be on green or non-monomorphic nodes, *i.e.* not in $\exists$ or $\sqsupseteq^{rmw}$.

One important consequence of Lemma 6.7.10 is the fact that unification is not significantly more complex for $\sqsubseteq^{\approx}$ and $\sqsubseteq^{\boxminus}$ than for $\sqsubseteq$. For example, the $\sqsubseteq^{\boxminus}$-solutions of a unification problem P are all the instances by $\exists$ of the $\sqsubseteq$-solutions of P. Thus, the undecidability of type inference in the system using $\sqsubseteq^{\boxminus}$ as its instance relation is not due to unification (but to the interaction between instance and generalization, as we will show in §13.3).

As a last consequence, let us study the structure induced by $\mathrel{\mathsf{E}}$ and $\sqsubseteq^{rmw}$ on types.

Property 6.7.11 *Let τ be a type, and let $\sqsubset$ be either $\mathrel{\mathsf{E}}$ or $\sqsubseteq^{rmw}$. Let S_τ be the set of types equal to τ for the equivalence relation induced by $\sqsubset$,* i.e.

$$S_\tau = \{\tau' \mid \tau \ (\sqsubset \cup \sqsupset)^* \ \tau'\}$$

Then $(S_\tau, \sqsubset)$ is a finite join semi-lattice. □

Proof: S_τ is finite, as $\{\tau' \mid \tau\ (\sqsubset\ ;\sqsupset)^*\ \tau'\}$ is equal to $\{\tau' \mid \tau\ (\sqsubset \cup \sqsupset)\ \tau'\}$ (by Lemma 6.7.10), both $\sqsubset$ and $\sqsupset$ are noetherian, and there are finitely many types τ' such that $\tau \sqsubset \tau'$ or $\tau \sqsupset \tau'$. The existence of a join is by Lemma 6.7.9.

This result implies in particular the existence of a normal form for $\boxminus$ and $\sqsubseteq^{rmw}$, which we call *maximally instantiated*. We conjecture that the lattice above is also a meet semi-lattice.

7 Unification

Abstract

We study the unification problem for $\mathsf{ML^F}$ types. We show that unifying two nodes inside a type is more general than unifying two different types (§7.1). We isolate a very general subset of unification problems on which unification is complete and principal (§7.2). We present the unification algorithm (§7.3), and proves its correctness (§7.4). We also show that the algorithm has linear complexity (§7.5). We introduce a slightly more general form of unification problem, which we also solve (§7.6). Finally, we discuss unification in variants of $\mathsf{ML^F}$ (§7.7).

7.1 $\mathsf{ML^F}$ unification problem

A first possibility for defining unification in $\mathsf{ML^F}$ is to use the usual approach of unifying two types.

Definition 7.1.1 (*Type unification*) A type τ' is a type unifier of the types τ_1 and τ_2 if $\tau_1 \sqsubseteq \tau'$ and $\tau_2 \sqsubseteq \tau'$. ■

However, we prefer to internalize unification, as we did for term-graphs (§3.2.3).

Definition 7.1.2 (*Node unification*) Given a type τ, a type τ' is a *node unifier* of a set of nodes N of τ if τ' is an instance of τ in which all the nodes of N are merged. Moreover, τ' is a *principal unifier* if any other unifier of N in τ is an instance of τ'. ■

Node unification is more general than type unification. In fact, the latter class of problems can be encoded into the former, as we prove below.

Lemma 7.1.3 *A type τ_u is a unifier of two types τ_1 and τ_2 if and only if the type τ'_u is a unifier of the nodes $\langle 1\rangle$ and $\langle 2\rangle$ in the type τ of Figure 7.1.1.* □

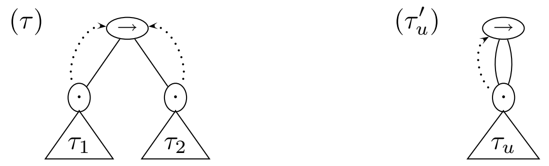


Figure 7.1.1 – Encoding type unification into node unification.

<u>Proof</u>: Assume there exists a type τ_u such that $\tau_1, \tau_2 \sqsubseteq \tau_u$. Let I_1 and I_2 be two derivations witnessing $\tau_1 \sqsubseteq \tau_u$ and $\tau_2 \sqsubseteq \tau_u$ respectively. Let I'_1 (*resp.* I'_2) be the derivations obtained from I_1 (*resp.* I_2) by replacing all the nodes n by $\langle 1 \cdot n \rangle$ (*resp.* $\langle 2 \cdot n \rangle$). Then I'_1 and I'_2 can be applied to τ (in any order, as they operate on distinct parts of the type), and yield a type τ'. Then, by construction of I'_1 and I'_2, $\mathsf{Merge}(\langle 1 \rangle, \langle 2 \rangle)$ can be applied to τ'. Hence τ'_u exists.

For the converse direction, assume a unifier τ'_u of the required form exists. Let I be a canonical derivation witnessing this last result. By definition of bottom-up merging-weakening, the last operation of I is $\mathsf{Merge}(\langle 1 \rangle, \langle 2 \rangle)$. From there, it is easy to extract from I two derivations I_1 and I_2 proving $\tau_1 \sqsubseteq \tau_u$ and $\tau_2 \sqsubseteq \tau_u$.

Consequently, in the following, « unification » will always mean « node unification ».

7.2 Admissible problems

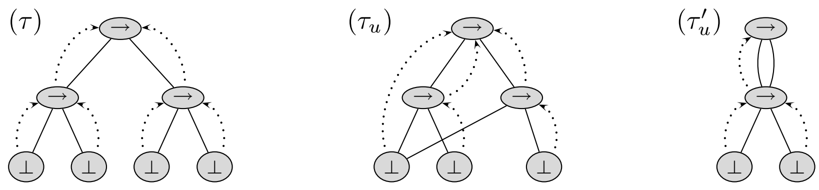


Figure 7.2.1 – A problem without a principal solution.

Node unification is in fact *too* liberal: some problems can have a non-principal set of solutions, as we describe below.[1] Consider the problem of unifying the nodes $\langle 11 \rangle$ and $\langle 21 \rangle$ in the type τ of Figure 7.2.1. A first unifier is τ_u: the two nodes have been raised once, and then merged. However, the type τ'_u obtained by merging the nodes $\langle 1 \rangle$ and $\langle 2 \rangle$—which

[1]This was not the case in the syntactic presentation of $\mathsf{ML^F}$, as unification under prefixes (Le Botlan 2004)—which is used for unifying syntactic types—is more expressive than type unification, but less than node unification.

indirectly merges $\langle 11\rangle$ and $\langle 21\rangle$—is another unifier. 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.

Fortunately, it is possible to characterize an important set of problems that admit principal solutions; we call *admissible* those problems. They include in particular unification under the root of the type, as used to encode unification of two different types.

Definition 7.2.1 (*Admissible problems)* Given a type τ and a set of nodes N of τ, we say that (τ, N) is an *admissible* problem (or that N is admissible for τ) if the set of nodes

$$\{\hat{n}' \mid \exists n \in N,\ \exists n' \in \tau,\ \hat{n}' \overset{+}{\multimap} n \overset{*}{\multimap} n'\}$$

is totally ordered by the domination relation $\mathrel{-\!\!\gg\!\!\multimap}$ induced by $\multimap$. We call *admissibility ancestors* this set. ■

It is difficult to give an intuition of this definition without actually proving that it ensures principality of unification problems. Very roughly, non principality cases always originate from a merging/raising competition, as illustrated on the example of Figure 7.2.1. In admissible problems, such potential conflicts always occur between nodes whose binders are in domination relation. This ensures that the conflict can only be solved by raising, as merging would create cycles in the structure.

► **Example** In Figure 7.2.1, the set $N = \{\langle 11\rangle, \langle 21\rangle\}$ is not admissible for τ or τ'. Indeed, the admissibility ancestors are $\langle 1\rangle$ and $\langle 2\rangle$, and they are not comparable for $\mathrel{-\!\!\gg\!\!\multimap}$ in τ or τ'.

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 7.2.2 *Consider a type τ and a node m of τ:*

1. *Any subset of $(m \multimap)$ is admissible for τ.*

2. *Any subset of $(m \mathrel{\lhd\!\!-})$ is admissible for τ.*

3. *Any set $\{m', m''\}$ where $m' \overset{+}{\mathrel{-\!\!\rhd}} m$ and $m'' \mathrel{-\!\!\rhd} m$ is admissible for τ.*[2] □

Proof: We consider a set N of nodes, and prove that N is admissible for τ.

1. If N is a subset of $(m \multimap)$: it suffices to show that all the admissibility ancestors dominate m for $\mathrel{-\!\!\gg\!\!\multimap}$, as the dominators of a node are totally ordered by the domination relation. Let n' be a node such that $\hat{n}' \overset{+}{\multimap} n \overset{*}{\multimap} n'$ with $n \in N$. We show that $\hat{n}'$ (which is an admissibility ancestor) dominates m.
   By definition of n' and n, there exists mixed paths of the form $\langle\epsilon\rangle \overset{*}{\multimap} m \multimap n \overset{*}{\multimap} n'$. By well-domination, this path contains $\hat{n}'$ above n. If $\hat{n}'$ is m, the result holds, as domination is reflexive. Otherwise $\hat{n}' \overset{+}{\multimap} m \multimap n \overset{*}{\multimap} n'$. By Lemma 4.3.4, $m \overset{+}{\mathrel{-\!\!\rhd}} \hat{n}'$ holds, which shows that $\hat{n}'$ dominates m for $\mathrel{\lhd\!\!\multimap}$, hence also for $\multimap$.

[2]This last case corresponds to the unification under prefix used in the syntactic presentations of $\mathsf{ML^F}$.

2. <u>If N is a subset of $(m \triangleleft\!\!\longleftarrow)$</u>: we also show that all the admissibility ancestors dominate m. The proof is the same as above, except that we have $m \triangleleft\!\!\longleftarrow n$ instead of $m \multimap n$.
3. <u>If $N = \{m', m''\}$ with $m' \overset{+}{\longrightarrow\!\triangleright} m$ and $m'' \longrightarrow\!\triangleright m$</u>: this time we show that all the admissibility ancestors dominate $\hat{m}'$. Let n' be a node such that $\hat{n}' \overset{+}{\multimap} n \overset{*}{\multimap} n'$ with $n \in N$. We proceed by case disjunction on n.
   - ▷ <u>*Case $n = m'$*</u>: by the reasoning used in subcase 2, n' is either $\hat{m}'$ or dominates it. This is the desired result.
   - ▷ <u>*Case $n = m''$*</u>: again by the reasoning of subcase 2, $\hat{n}'$ is either $\overset{\longrightarrow\triangleright}{m''}$ (*i.e.* m) or dominates it. Moreover, m dominates $\hat{m}'$, as $\hat{m}' \overset{*}{\longrightarrow\!\triangleright} m$. We conclude by transitivity of domination.

Importantly, admissible problems are stable by instance.

Property 7.2.3 *Consider an admissible unification problem (τ, N). For any type τ' such that $\tau \sqsubseteq \tau'$, (τ', N) is admissible.* □

<u>Proof</u>: Let A be the admissibility ancestors of τ, A' those of τ'. We need to prove that A' is totally ordered by $\gg\!\!\multimap$ in τ'. It suffices to prove the result for one atomic instance step. Thus we let o be such that $\tau \sqsubseteq_1 \tau'$ with $\tau' = o(\tau)$, and proceed by case disjunction on o.

- ▷ <u>*Case $o = \mathsf{Weaken}(n)$*</u>: A and A' are equal, and $\gg\!\!\multimap_\tau = \gg\!\!\multimap_{\tau'}$ as τ', as $\check{\tau} = \check{\tau}'$ and $\hat{\tau} = \hat{\tau}'$. The conclusion is by admissibility of (τ, N).
- ▷ <u>*Case $o = \mathsf{Graft}(\tau'', n)$*</u>: no node grafted between τ and τ' is an admissibility ancestor, as their binding edges are strictly under n, hence strictly under N. Thus, since the nodes of τ are unchanged in τ', $A = A'$. Also because the nodes of τ are unchanged, the restriction of the domination relation to the nodes of A is equal to the one in τ. Thus the conclusion is again by admissibility of (τ, N).
- ▷ <u>*Case $o = \mathsf{Merge}(n_1, n_2)$*</u>: Let n' be such that $\hat{n}' \in A'$, *i.e.* $\hat{n}' \overset{+}{\multimap} n \overset{*}{\multimap} n'$ with $n \in N$. Let m be a node of τ such that m extends into n in τ' and $m \in N$. Let π be such that $n \overset{\pi}{\multimap} n'$, and let m' be $\langle m \cdot \pi \rangle$. By definition of merging and well-domination, we have $\hat{m}' \overset{+}{\multimap} m \overset{*}{\multimap} m'$. This shows that m is in A.
  Consider now two admissibility ancestors n_1 and n_2 in τ'. By the reasoning above, let m_1 and m_2 be two admissibility ancestors of A that extend into n_1 and n_2. By admissibility of N in τ, m_1 and m_2 are ordered by $\gg\!\!\multimap_\tau$. Lemma 4.4.8 shows that they are still ordered by $\gg\!\!\multimap_{\tau'}$, which is the desired result.
- ▷ <u>*Case $o = \mathsf{Raise}(n')$*</u>: notice that $\gg\!\!\multimap$ is unchanged in τ' (**1**), as $\check{\tau} = \check{\tau}'$. There are two cases, depending on whether $\{n \in N \mid \hat{\tau}(n') \overset{+}{\multimap} n \overset{*}{\multimap} n'\}$ is empty or not, *i.e.* whether there are new admissibility ancestors or not.
  - ○ <u>*Case $\exists n \in N,\ \hat{\tau}(n') \overset{+}{\multimap} n \overset{*}{\multimap} n'$*</u>: then $\hat{\tau}'(n')$ is an admissibility ancestor in τ'. Since the only new binding edge in τ' is $n' \longrightarrow\!\triangleright \hat{\tau}'(n')$, we have $A' \subseteq A \cup \{\hat{\tau}'(n')\}$ (they may not be equal, as $\hat{\tau}(n')$ may or may not be in A'). By (1), it suffices to show that $\hat{\tau}'(n')$ is totally ordered with all the nodes of A. Let n'' be one such node. Notice that we have $\hat{\tau}(n') \in A$; thus by admissibility of (τ, N), we have either $\hat{\tau}(n')$ dominates n'', or n'' dominates $\hat{\tau}(n')$ for $\gg\!\!\multimap$ in τ.

  - ∘ *Case $\hat{\tau}(n') \mathrel{-\!\!\gg\!\!-\!\circ} n''$*: we have $\hat{\tau}'(n') \mathrel{\leftarrow\!\!\gg\!\!-\!\circ} \hat{\tau}(n')$ by well-domination, hence $\hat{\tau}'(n') \mathrel{-\!\!\gg\!\!-\!\circ} \hat{\tau}(n')$. Thus $\hat{\tau}'(n')$ dominates n'' by transitivity of domination.
  - ∘ *Case $n'' \mathrel{-\!\!\gg\!\!-\!\circ} \hat{\tau}(n')$*: as above, $\hat{\tau}'(n')$ dominates $\hat{\tau}(n')$. Hence both $\hat{\tau}'(n')$ and n'' dominate $\hat{\tau}(n')$. Two dominators of a same node are ordered by domination, hence the conclusion.
- ∘ Otherwise: we have $A = A'$. The conclusion is thus by (1) and admissibility of (τ, N).

7.3 Unification algorithm

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

1. Let g_u be the first-order principal unifier of the nodes N in the term-graph $\breve{\tau}$.
   Fail if g_u does not exist, or if it is cyclic.
2. Let τ_u be $\mathsf{Rebind}(\tau, g_u)$. Fail if Rebind fails.
3. Return τ_u.

Figure 7.3.1 – Unif_N algorithm.

We present our unification algorithm Unif_N in Figure 7.3.1. 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 $\breve{\tau}$ using first-order unification; the result of this phase will be the skeleton of the unifier.
2. The second phase uses an auxiliary algorithm Rebind (detailed below) to build the binding tree of the unifier.

Convention In order to ease readability, we use the following convention in the remainder of this chapter: the nodes of a type τ into which we unify some nodes are named using the metavariable "m", while those of an unifier are named using "n". There is a single exception: for a node m of τ, we write $\langle\!\langle m \rangle\!\rangle$ the corresponding node of τ_u (*i.e.* the unique node of τ_u whose paths extend the paths of m).

Given a type τ and a term-graph g instance of $\breve{\tau}$, the algorithm Rebind (presented in Figure 7.3.2) returns an instance τ' of τ whose skeleton is g, or fails. We say that a node n of τ is *partially grafted* if there exists a bottom node m of τ such that $\langle\!\langle m \rangle\!\rangle \overset{+}{-\!\circ} n$ in g. We write $\mathsf{LCA}_T(n_1, ..., n_k)$ for the least common ancestor of the nodes $n_1, ..., n_k$ in a tree T. The two phases of Rebind are detailed below.

1. *Building the binding tree.*

Input: A type τ and a term-graph g instance of $\breve{\tau}$
Output: A type τ' such that $\breve{\tau}' = g$

Let $\hat{\tau}'$ and $\overset{\diamond}{\tau}'$ be $\emptyset$.

1. **Building the binding tree**

   Visit the nodes of $\mathsf{dom}(g) \setminus \{\langle\epsilon\rangle\}$ in a top-down order for $\multimap_g$.
   On each node n, do:

   a) Let M_n be $\{m \in \tau \mid m \subseteq n\}$.

   b) Let $\diamond_n$ be $\max\{\overset{\diamond}{\tau}(M_n)\}$ for $\overset{\diamond}{<}$.

   c) Let n_B be $\mathsf{LCA}_{\hat{\tau}'}\,(B_1^n \cup B_2^n)$, where
   $B_1^n = \hat{\tau}(M_n)$
   $$B_2^n = \begin{cases} (\multimap_g n) & \text{if } n \text{ is partially grafted} \\ \emptyset & \text{otherwise} \end{cases}$$

   d) Let $\overset{\diamond}{\tau}'$ become $\overset{\diamond}{\tau}' + n \mapsto \diamond_n$ and $\hat{\tau}'$ become $\hat{\tau}' + n \longrightarrow n_B$.

2. **Correction of the instance steps**

   Let $\tau_\uparrow$ be the graph verifying $\breve{\tau}_\uparrow = \breve{\tau}$, $\overset{\diamond}{\tau}_\uparrow = \overset{\diamond}{\tau}$ and
   $$m \longrightarrow m' \in \tau_\uparrow \iff \wedge\begin{cases} \langle\!\langle m \rangle\!\rangle \longrightarrow \langle\!\langle m' \rangle\!\rangle \in \hat{\tau}' \\ m \xrightarrow{+} m' \in \tau \end{cases}$$

   Fail if either one the following condition holds:

   a) there exists a non-green bottom node m of τ such that $g(m) \neq \bot$

   b) there exists m red in τ such that $\hat{\tau}(m)$ is different from $\hat{\tau}'(m)$

   c) there exists m red in $\tau_\uparrow$ such that $\overset{\diamond}{\tau}_\uparrow(m)$ is different from $\overset{\diamond}{\tau}'(m)$

   d) there exists m_1 and m_2 distinct in τ and merged in g such that one of them is red in $\tau_\uparrow$ and $\hat{\tau}_\uparrow(m_1) = \hat{\tau}_\uparrow(m_2)$.

3. **Return** $\tau' \triangleq (g, (\hat{\tau}', \overset{\diamond}{\tau}'))$.

Figure 7.3.2 – Rebind algorithm.

The first phase of Rebind finds the binding tree of τ'. To do this, the nodes of τ merged together must be raised until they are all bound to the same node, and some of them must be weakened. Phase 1 of Rebind computes the result of those operations. More precisely, given a node n of τ', we call M_n the nodes of τ that are merged into n in g. Step 1c computes the lowest common binder to the nodes of M_n.[3] In parallel, a new flag $\diamond_n$ is computed for n, by choosing the most restrictive flag among those of the nodes of M_n (step 1b).

Importantly (as it is one of the keys to obtaining a good complexity for the algorithm), the computation of $\hat{\tau}'$ reuses the results found for the nodes that have already been considered when computing the binders of the nodes underneath.

2. *Correction of the instance steps.*

   The second phase verifies that the instance operations that must be performed to transform the binding tree of τ into the one of τ' are allowed: steps 2a, 2b, 2c and 2d check that the permissions are correct for grafting, raising, weakening and merging respectively. The checks for the last two operations are done on a pre-type[4] $\tau_{\uparrow}$ that superimposes the binding edges (but not the binding flags) of τ' over the structure of τ. The most involved checks are for merging, as we must make sure not to perform checks for pairs of nodes that are indirectly merged.

In the following we often reason about the graph built by Rebind, whether phase 2 fails or succeeds. Thus we write rebind(τ, g) the graph obtained by calling Rebind on τ and g without performing phase 2 at all. Hence, rebind (in lowercase) always succeeds.

7.3.1 Two intermediate graphs

Consider a type τ' returned by calling Rebind(τ, g). We are ultimately going to prove that $\tau \sqsubseteq \tau'$ holds. We introduce two intermediate graphs τ_g and τ_r that correspond to the steps of an ordered derivation of $\tau \sqsubseteq \tau'$.

- The graph τ_g is τ in which all the graftings have been performed, *i.e.* exactly $\tau[\tau'/\bot]$.
- The graph τ_r is τ_g in which all the raisings performed when transforming τ into τ' have been performed, *i.e.* $\check{\tau}_r = \check{\tau}_g$, $\overset{\diamond}{\tau}_r = \overset{\diamond}{\tau}_g$, and $\hat{\tau}_r$ is defined by:

$$m \longrightarrow m' \in \tau_r \iff \wedge\begin{cases} \langle\!\langle m \rangle\!\rangle \longrightarrow \langle\!\langle m' \rangle\!\rangle \in \tau' \\ m \xrightarrow{+} m' \in \tau_g \end{cases}$$

Unif and Rebind cannot build τ_g and τ_r, because their size can be exponential in the size of τ, which would make the complexity for Unif also at least exponential. However, those graphs are very useful from a reasoning standpoint.

Notice that $\tau_{\uparrow}$ and τ_r are (intendedly) very similar: the only difference is that τ_r is defined on the structure of τ_g, which is bigger than the one of τ. However, the difference between τ_r and $\tau_{\uparrow}$ is unimportant permissions-wise, as proven by Lemma 7.4.10.

[3] We defer the discussion on B_2^n to the example of §7.3.1.

[4] In fact, $\tau_{\uparrow}$ is a well-dominated type, as shown by Property 7.4.6. However, this is unimportant here.

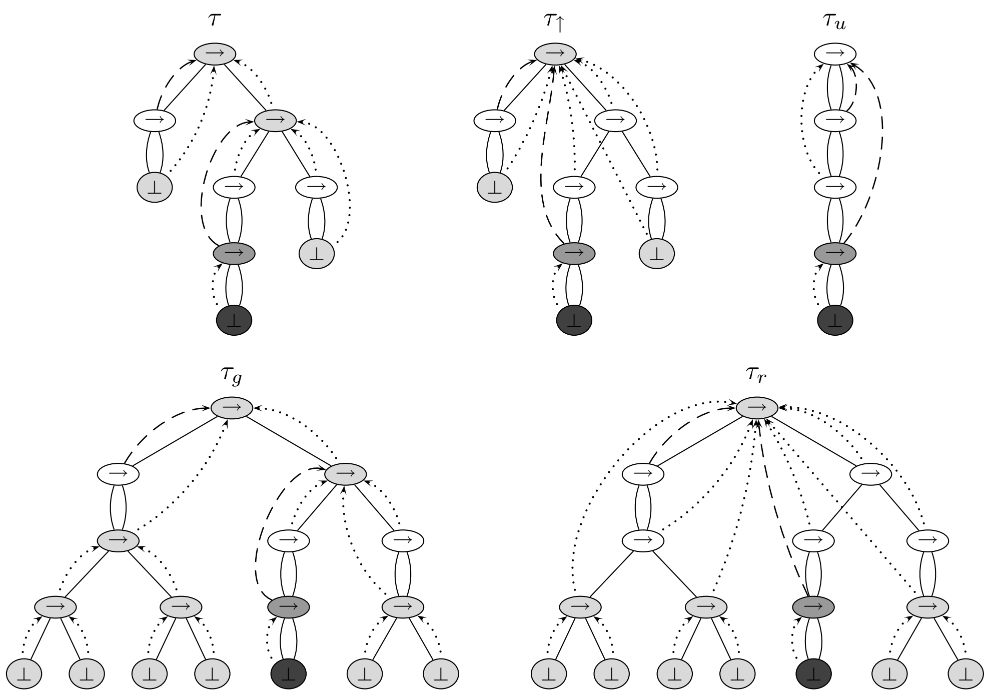


Figure 7.3.3 – Example of unification

► **Example** Consider Figure 7.3.3. Our goal is to unify the nodes $\langle 1\rangle$ and $\langle 2\rangle$ in the type τ. The result of calling Unif on this problem is the type τ_u of the same figure; we have also drawn τ_g, τ_r and $\tau_\uparrow$. Let us examine some actions of Rebind on our example:

Step 1 Assume that we want to bind $n = \langle 11\rangle$. By construction, M_n contains all the nodes of τ merged into n, hence $\langle 11\rangle$, $\langle 21\rangle$ and $\langle 22\rangle$. Since all three nodes have flexible binders, $\diamond_n = (\geqslant)$. Simultaneously, again by construction, B_1^n contains the binders in τ of all the nodes in M_n, *i.e.* $\langle 1\rangle$ and $\langle 2\rangle$. Since n is not partially grafted, we have $B_2^n = \emptyset$. Thus $\mathsf{LCA}_{\hat\tau'}(B_1^n \cup B_2^n)$ is $\mathsf{LCA}_{\hat\tau'}(\{\langle\epsilon\rangle, \langle 2\rangle)\}$. Thus $n_B = \langle 1\rangle$.

As a more involved example, suppose now that we want to bind $n = \langle 111\rangle$. This time we have $M_n = \{\langle 211\rangle, \langle 221\rangle\}$, hence $B_1^n = \{\langle 2\rangle\}$. Thus, if n_B was computed by taking only into account B_1^n, we would have $\hat{\tau_u}(n) = \langle 2\rangle$. This is however incorrect, as we need to take into account the binders of the nodes grafted under $\langle 11\rangle$. Consider indeed τ_g. There are two new nodes which are merged with n in τ_u, namely $\langle 111\rangle$ and $\langle 112\rangle$. Since they are bound on $\langle 11\rangle$, itself bound on the root, the only possible binder for n is $\langle\epsilon\rangle$.

Thus the set B_2^n is used to *simulate* these missing binding edges, *i.e.* those of the nodes grafted between τ and τ_g. Here we have $B_2^n = \{11\}$ (as n is partially grafted).

More generally, B_2^n is such that $n' \in B_2^n$ iff there exists m in τ_g but not τ such that $m \longrightarrow\!\!\triangleright\, m' \in \tau_g$ and $\langle\!\langle m' \rangle\!\rangle = n'$.

In our example, $B_1^n \cup B_2^n = \{\langle 2\rangle, \langle 11\rangle\}$. Since Rebind has bound $\langle 11\rangle$ to $\langle\epsilon\rangle$ in τ_u, we have $\mathsf{LCA}_{\hat{\tau}'}(B_1^n \cup B_2^n) = \langle\epsilon\rangle$, which is the correct result.

Notice that the computation of $\diamond_n$ does not take into account B_2^n. Indeed, all the nodes in this set are flexibly bound in τ_g.

Step 2a There are two bottom nodes grafted between τ and τ_g, $\langle 11\rangle$ and $\langle 221\rangle$. Since both of them are green in τ, the check succeeds.

Step 2b The nodes m of τ such that $\hat{\tau_u}(m)$ is not $\hat{\tau}(m)$ are $\langle 21\rangle$, $\langle 22\rangle$, $\langle 211\rangle$ and $\langle 221\rangle$. All of them have either green permission ($\langle 221\rangle$), orange ($\langle 211\rangle$), or inert ($\langle 21\rangle$ and $\langle 22\rangle$) in τ. Thus the check succeeds.

Notice that we do not make permissions checks for the nodes under $\langle 11\rangle$, although they are indeed raised between τ_g and τ_r. Indeed, as they are grafted between τ and τ_g, we are assured that they have green or monomorphic permissions.

Step 2c The nodes of τ weakened are $\langle 2\rangle$ and $\langle 221\rangle$. They are respectively inert and flexible in $\tau_\uparrow$ or τ_g, hence the check succeeds. Again, we do not check for the nodes only in τ_g and τ_r (but not in $\tau_\uparrow$), as they all have green or monomorphic permissions.

Step 2d The following nodes verify all the conditions of the checks done in this step, except "being red":

- $\langle 1\rangle$ and $\langle 2\rangle$
- $\langle 11\rangle$, $\langle 21\rangle$ and $\langle 22\rangle$
- $\langle 211\rangle$ and $\langle 221\rangle$

Since none of those nodes is red, the checks succeed.

Importantly, we do not check for a merging for $\langle 2111\rangle$: while it is red and merged with other nodes in τ_u (for example $\langle 1111\rangle$), the mergings are all indirect.

7.4 Correctness of the algorithm

This section shows that the algorithms Unif and Rebind are correct. In all the section we implicitly quantify over a type τ, a set of nodes N and a first-order unifier g of N in $\breve{\tau}$. We define the graph τ_u as $\mathsf{rebind}(\tau, g)$; the graphs τ_g and τ_r are defined as in §7.3.1. We do *not* assume that (τ, N) is admissible or that g is the principal first-order unifier of N in $\breve{\tau}$. For the completeness and principality result, we write τ_U the principal unifier of N in $\breve{\tau}$, and τ_U the graph $\mathsf{rebind}(\tau, g_U)$.

The next subsections are structured as follows:

▷ §7.4.1 discusses some properties of the graphs returned by Unif.

▷ §7.4.2 shows that Unif is sound.

▷ §7.4.3 introduces a criterion more general than admissibility, that ensures that Unif is complete and principal.

- ▷ §7.4.4 shows that Unif is complete, *i.e.* that it does not fail when an unifier exists.
- ▷ §7.4.5 shows that Unif is principal, *i.e.* that the type it returns is more general than the other unifiers.

(In fact, we always start by showing that the results hold for Rebind.)

As a high-level result, the theorem below summarizes the results we will show for Unif.

Theorem 7.4.1 (Correctness of Unif) *Let τ be a type, N a set of nodes of τ.*

- ▷ *If the computation of* $\mathsf{Unif}_N(\tau)$ *does not fail, then* $\tau \sqsubseteq \mathsf{Unif}_N(\tau)$ (soundness).
- ▷ *If N is admissible for τ and there exists an unifier τ_v of N in τ, then the computation of* $\mathsf{Unif}_N(\tau)$ *does not fail* (completeness)*, and* $\mathsf{Unif}_N(\tau) \sqsubseteq \tau_v$ (principality). □

7.4.1 Properties of the unifier

In this subsection we never consider the permission checks of phase 2 of Rebind*, and only reason on* $\tau_u = \mathsf{rebind}(\tau, g)$.

The use of a least common ancestor algorithm in Rebind implies that a binding edge of τ_u is in the transitive closure of the binding edges of τ.

Lemma 7.4.2 *Let $n \longrightarrow n' \in \tau_u$, and let m be a node of τ_g merged in n in τ_u. Then there exists a unique node m' merged in n' in τ_u such that $m \xrightarrow{+} m' \in \tau_g$.* □

Proof: The proof for the existence is by induction on the order chosen to bind the nodes of τ_u (step 1 of Rebind). Let m'' be $\hat{\tau_g}(m)$. If m is in τ, we have $m'' \in B_1^n$. Otherwise, m is bound to its structural ancestor and $m'' \in B_2^n$. Thus $m'' \in B_1^n \cup B_2^n$. By definition of Rebind, we have $n' = \mathsf{LCA}_{\check{\tau_u}}(B_1^n \cup B_2^n)$. By definition of a least common ancestor, we have $\langle\langle m'' \rangle\rangle \xrightarrow{*} n' \in \tau_u$. By induction hypothesis applied to each of the nodes in this path (which have all been bound before n), we obtain the existence of a node m' merged in n' such that $m'' \xrightarrow{*} m' \in \tau_g$. Thus $m \longrightarrow m'' \xrightarrow{*} m' \in \tau_g$, and m' is the desired node.

For the unicity: if two such nodes existed, as they are merged with n' in τ_u and are structurally one above the other in τ (as they are on the binding path from n to the root), $\check{\tau_u}$ would be cyclic: this is a contradiction.

As a first important consequence, this means that the binding trees of τ_r, $\tau_\uparrow$ and τ_u are correct, which implies that those graphs are pre-types.[5]

Lemma 7.4.3 *The graphs τ_r, $\tau_\uparrow$ and τ_u are pre-types.* □

Proof: For τ_u :

- ▷ $\check{\tau_u}$ is a well-formed term-dag by soundness of first-order unification. Moreover, it is acyclic by the check done in step 1 of Unif.

[5] We will show later that they are also types. It is also immediate that τ_g is a type, as it is obtained by grafting some nodes in τ.

▷ $\overset{\diamond}{\tau}_u$ is correct, as each node but the root receives a flag.

▷ $\hat{\tau}_u$ binds each node but the root to another node. As a consequence of Lemma 7.4.2, each node n of τ_u is bound to a node strictly above n in $\breve{\tau}_u$. This ensures that $\hat{\tau}_u$ is a tree. Thus $\hat{\tau}_u$ is correct.

For τ_r and $\tau_\uparrow$:

▷ The correctness of $\breve{\tau}_r$, $\breve{\tau}_\uparrow$, $\overset{\diamond}{\tau}_r$ and $\overset{\diamond}{\tau}_\uparrow$ is by correctness of $\breve{\tau}_g$, $\breve{\tau}$, $\overset{\diamond}{\tau}_g$ and $\overset{\diamond}{\tau}$ respectively.

▷ For the correctness of the binding edges, we only consider $\hat{\tau}_r$ as the reasoning is exactly the same for $\hat{\tau}_\uparrow$.
Consider a node m of τ_r that is not the root. Given the definition of $\hat{\tau}_r$, it is immediate that m has at least one binding edge, and that all the potential binders are strictly above m. Thus it suffices to prove the unicity of the binder to show that $\hat{\tau}_r$ is a tree. Let m' and m'' be two nodes such that $m \mathrel{-\!\!\triangleright} m' \in \tau_r$ and $m \mathrel{-\!\!\triangleright} m'' \in \tau_r$. By definition, we have $\langle\!\langle m \rangle\!\rangle \mathrel{-\!\!\triangleright} \langle\!\langle m' \rangle\!\rangle \in \tau_u$ and $\langle\!\langle m \rangle\!\rangle \mathrel{-\!\!\triangleright} \langle\!\langle m'' \rangle\!\rangle \in \tau_u$. By Lemma 7.4.2, $m \overset{+}{\mathrel{-\!\!\triangleright}} m' \in \tau_g$ and $m \overset{+}{\mathrel{-\!\!\triangleright}} m'' \in \tau_g$. Since τ_g is a well-formed type, $\hat{\tau}_g$ is a tree, and $\breve{\tau}_g$ is acyclic. This implies that $m' = m''$, which is the desired result.

The next result essentially expresses that Rebind chooses the lowest possible binder for a node.

Lemma 7.4.4 *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 $m \overset{+}{\mathrel{-\!\!\triangleright}} m' \in \tau_g$. Then, $n \overset{+}{\mathrel{-\!\!\triangleright}} n' \in \tau_u$.* □

Proof: We write $m \in \tau \subseteq n$ as a shorthand for $m \in \tau \wedge m \subseteq n$. Given two nodes n and n' of τ_u, let $P_n(n')$ be the property $\forall m \in \tau_g \subseteq n,\ \exists m' \in \tau_g \subseteq n', m \overset{+}{\mathrel{-\!\!\triangleright}} m' \in \tau_g$ (which is the premise of the result). We first prove three intermediary results:

(1) The set $\mathcal{P}_n$ of nodes n' verifying $P_n(n')$ is totally ordered by $\multimap_{\tau_u}$:
Given $m \in \tau_g \subseteq n$, $\multimap_{\tau_g}$ is a total order on the set $\mathcal{P}_{n,m}$ of nodes m' such that $m \overset{+}{\mathrel{-\!\!\triangleright}} m' \in \tau_g$. Hence so is $\multimap_{\tau_u}$ (up to coercion from nodes of τ into nodes of τ_u), as $(\multimap_{\tau_g}) \subseteq (\multimap_{\tau_u})$. The set $\mathcal{P}_n$ is exactly

$$\bigcap_{m \subseteq n} \{\langle\!\langle m' \rangle\!\rangle \mid m' \in \mathcal{P}_{n,m}\}$$

hence the result.

(2) The smallest node of $\mathcal{P}_n$ for $\multimap_{\tau_u}$ is n_B:
By induction on the building of $\hat{\tau}_u$. If all the $m \in \tau_g \subseteq n$ are bound on the same node in τ_g, n_B is this node and the result holds. Otherwise, the conclusion is by applying the induction hypothesis to the binders $\hat{\tau}_g(m)$ for $m \in \tau_g \subseteq n$, and by the definition of a LCA (more precisely the "least" part).

(3) $\forall n' \in \mathcal{P}_n,\ n' = n_B \vee n' \in M_{n_B}$:
Consider n' in $\mathcal{P}_n$ that is not n_B, and a node $m_B \in \tau_g \subseteq n_B$. It suffices to prove that there exists $m' \in \tau_g \subseteq n'$ such that $m_B \overset{+}{\mathrel{-\!\!\triangleright}} m' \in \tau_g$.
Let π be such that $n_B \overset{\pi}{\multimap} n \in \tau_u$. By construction of τ_g, the node m equal to $\langle m_B \cdot \pi \rangle$ is in τ_g, and is merged with n in τ_u. By Lemma 7.4.2, there exists m'_B merged with n_B such that $m \overset{+}{\mathrel{-\!\!\triangleright}} m'_B$ **(4)**. Moreover m'_B must be m_B: otherwise, since m_B and m'_B are both merged with n_B, $\breve{\tau}_u$ would be cyclic. Since $n' \in \mathcal{P}_n$, there

exists $m' \in \tau_g \subseteq n'$ such that $m \xrightarrow{+} m' \in \tau_g$ (**5**). By (4), (5) and unicity of binding paths, we have either $m_B \xrightarrow{+} m'$ or $m' \xrightarrow{*} m_B$. By (1) and (2), n' is higher than n_B for $\multimap_{\tau_u}$. This implies that $m_B \xrightarrow{+} m'$, which is the desired result.

We return to the main property. We prove that for any node n of τ_u, for any n' such that $P_n(n')$ holds, $n \xrightarrow{+} n' \in \tau_u$. The proof is by induction on $\overset{+}{\multimap}_{\tau_u}$.

- ▷ <u>If n is the root</u>: we have $M_{\{\epsilon\}} = \emptyset$. Hence the quantification is over an empty set and the result holds.
- ▷ <u>Otherwise</u>: let n be a node, and n' a node verifying P_n. If n' is n_B, we have $n \longrightarrow n' \in \tau_u$ by definition of τ_u, and the result holds. Otherwise, by (3), $n' \in M_{n_B}$. The node n' is strictly above n, (as it is n_B or above n_B by (1) and (2)). Hence, by induction hypothesis, $n_B \xrightarrow{+} n' \in \tau_u$. We conclude by the fact that $n \longrightarrow n_B \in \tau_u$.

This result can be used to show that τ_u is a type, which in turn implies that $\tau_\uparrow$ and τ_r are types.

Property 7.4.5 *The graph τ_u returned by* Unif *is a type.* □

<u>Proof</u>: Lemma 7.4.3 shows that τ_u is a pre-type. For well-domination, let n be a node of τ_u, P_u a mixed path from $\{\epsilon\}$ to n in τ_u. We need to prove that $\hat{\tau_u}(n)$ is in P_u.

We first rewrite P_u into a path P_r by restricting the domains of the nodes in P_u so that it becomes a valid path in τ_r. (There isn't unicity of such a rewriting, but this is unimportant.) Then we rewrite P_r into a mixed path P_g of τ_g by transforming all edges $\hat{\tau_r}(m) \longleftarrow m$ of τ_r into $\hat{\tau_r}(m) \longleftarrow \cdots \longleftarrow \hat{\tau_g}(m) \longleftarrow m$ (Lemma 7.4.2).

Let m_1 be the node of P_g that extends into n in P_u. Let m_3 be the node of τ_g subset of $\hat{\tau_u}(m_1)$ such that $m_1 \xrightarrow{+} m_3$ (by Lemma 7.4.2). By iterating the well-domination property in τ_g, m_3 is in P_g. To prove our result, it suffices to show that $\langle\!\langle m_3 \rangle\!\rangle$ is in P_u. We proceed by case disjunction on the edge after m_3 in P_g.

- ▷ <u>*Case $m_3 \multimap\ \in P_g$*</u>: this edge is also in P_r and P_u. Hence $\langle\!\langle m_3 \rangle\!\rangle = n$ is indeed in P_u.
- ▷ <u>*Case $m_3 \longleftarrow\ \in P_g$*</u>: Let m_2 be the first (**1**) node such that $\langle\!\langle m_2 \rangle\!\rangle$ appears in P_u and $m_3 \xleftarrow{+} m_2$ is in P_g; in particular, m_2 is above m_1 (**2**). It suffices to prove that $\langle\!\langle m_3 \rangle\!\rangle \longleftarrow \langle\!\langle m_2 \rangle\!\rangle$ is in fact in P_u.
  Consider a node m_2' of τ_g merged with m_2 in τ_u. By (2) and given the structure of τ_g, there exists m_1' merged with m_1 such that $m_2' \overset{*}{\multimap} m_1' \in \tau_g$ (m_1' is not necessarily unique, but this is unimportant here). Since $\langle\!\langle m_1 \rangle\!\rangle \longrightarrow \langle\!\langle m_3 \rangle\!\rangle \in \tau_u$ and m_1 is merged with m_1' in τ_u, by Lemma 7.4.2 there exists a unique node m_3' such that m_3' and m_3 are merged in τ_u, and $m_1' \xrightarrow{+} m_3'$. Any structure path to m_1' containing m_2' must go through m_3' by well-domination of m_1', and m_3' cannot be under m_2' in τ_g as it is above in τ_u; thus $m_3' \overset{+}{\multimap} m_2'$ holds. By Lemma 4.3.4 applied to $m_1' \xrightarrow{+} m_3' \overset{+}{\multimap} m_2' \overset{*}{\multimap} m_1'$, we have $m_2' \xrightarrow{+} m_3' \in \tau_g$. This result holds for any node m_2' merged with m_2; hence by Lemma 7.4.4, we have $\langle\!\langle m_2 \rangle\!\rangle \xrightarrow{+} \langle\!\langle m_3 \rangle\!\rangle \in \tau_u$. By (1), it is in fact $m_2 \longrightarrow m_3 \in \tau_u$, and m_3 is indeed in P_u (since $\longleftarrow m_2$ is in P_u).

Property 7.4.6 *The graphs $\tau_\uparrow$ and τ_r are types.* □

Proof: Lemma 7.4.3 shows that both graphs are pre-types. For well-domination of τ_r, let m be a node of τ_r, P be a mixed path of τ_r from the root to m. We must prove that $\hat{\tau_r}(m)$ is in P.

P is also a mixed path of τ_u, from the root to $\langle\langle m\rangle\rangle$. By well-domination of τ_u, the binder of m in τ_u is in P. Let m' be the node of P which extends into $\hat{\tau_u}(\langle\langle m\rangle\rangle)$. Let P' be the subset of P between m' and m. By definition of $\hat{\tau_r}$, we have $m' \overset{+}{\longrightarrow\!\circ}\, m \in \tau_r$. Thus, since $\langle\langle m\rangle\rangle \longrightarrow\!\triangleright \langle\langle m'\rangle\rangle \in \tau_u$ and by definition of $\hat{\tau_r}$, m' is the binder of m in τ_r. Since it is in P, we have the desired result.

For well-domination of $\tau_\uparrow$, a mixed path in $\tau_\uparrow$ is also a mixed path in τ_r. Thus the result is by well-domination of τ_r.

7.4.2 Soundness of Unif

In this section, we study the soundness of Rebind *and* Unif*. Hence, we assume that the computation of* $\tau_u = \mathsf{Rebind}(\tau, g)$ *succeeds.*

In order to prove that Rebind and Unif are sound, we use a few intermediary results showing that the permission checks performed by Rebind are correct.

Lemma 7.4.7 *The instance relation* $\tau \sqsubseteq^G \tau_g$ *holds.* □

Proof: All the grafting are authorized by step 2a of Rebind. The result holds by repeated applications of Graft.

Lemma 7.4.8 *The types* τ *and* τ_g *have the same set of red nodes.* □

Proof: By Lemma 7.4.7, the red nodes of τ are still in τ_g (instance preserves red nodes). By Lemma 5.4.1, grafting does not create red nodes in the existing structure. Finally, the nodes grafted in τ_g are not red, as they are all flexibly bound.

Lemma 7.4.9 *The relations* $\tau_g \sqsubseteq^R \tau_r$ *and* $\tau \sqsubseteq^R \tau_\uparrow$ *hold.* □

Proof: ▷ $\underline{\tau \sqsubseteq^R \tau_\uparrow}$: we have $\tau \sqsubseteq^{R\natural} \tau_\uparrow$ by Lemma 7.4.2 and the checks performed by step 2b of Rebind. We conclude by Lemma 6.3.4.

▷ $\underline{\tau_g \sqsubseteq^R \tau_r}$: as in the previous case, we show that $\tau_g \sqsubseteq^{R\natural} \tau_r$ holds. The fact that $\hat{\tau_r} \subseteq (\hat{\tau_g})^+$ is again by Lemma 7.4.2. The nodes of τ_r present in τ are checked for permissions in τ instead of τ_g; Lemma 7.4.8 shows that this is unimportant. The nodes present only in τ_g cannot be red, again by the same result.

Lemma 7.4.10 *The types* τ, τ_g, $\tau_\uparrow$ *and* τ_r *have the same set of red nodes.* □

Proof: Lemma 5.4.1 shows that raising preserves red nodes, and does not introduce new red nodes. The conclusion is thus immediate by Lemma 7.4.9 and Lemma 7.4.8.

This result shows in particular that it is sound to perform permissions checks on τ and $\tau_\uparrow$ instead of τ_g and τ_r respectively. Taking things one step further, we could in fact perform all the permissions checks of Rebind in τ only, and there is in fact no need to compute the permissions of the nodes in $\tau_\uparrow$. Indeed, $\tau_\uparrow$ is only really useful in step 2d, to decide whether $\hat{\tau_\uparrow}(m_1) = \hat{\tau_\uparrow}(m_2)$. We have chosen the current presentation only to make simpler the explanations of the permissions checks.

Lemma 7.4.11 *The instance relation $\tau_r \sqsubseteq^{MW} \tau_u$ holds.* □

Proof: Let us show that $\tau_r \sqsubseteq^{MW\natural} \tau_u$ holds; the conclusion is then by Lemma 6.3.8.

- ▷ Properties 1, 2, 3 and 5 of the definition of $\sqsubseteq^{MW\natural}$ (Definition 6.3.5) are immediate given the definition of Rebind.
- ▷ Step 2c of Rebind checks the nodes present in τ for property 6; while the checks are done in $\tau_\uparrow$ instead of τ_r, this is unimportant by Lemma 7.4.10. The nodes not present in τ cannot be red by the same lemma.
- ▷ For property 4, we consider two distinct nodes m_1 and m_2 of τ_r merged in τ_u, and bound to the same node in τ_r. We must show that none of them is red in τ_r.
  - ○ *Case* m_1 and m_2 are in τ: Step 2d checks that m_1 and m_2 are not red in τ, hence not in τ_r either by Lemma 7.4.10. This is the desired result.
  - ○ *Case* neither m_1 nor m_2 is in τ: m_1 and m_2 cannot be red in τ_r by Lemma 7.4.10.
  - ○ *Case* m_1 is in τ, m_2 is not in τ: m_2 is not red in τ_r by Lemma 7.4.10. Hence we must prove that m_1 is not red in τ_r.
    By hypothesis, in τ_r, m_1 and m_2 are bound on the same node m. Since m_2 is not in τ but m_1 is, there exists a bottom node m_2' of τ such that $m \overset{+}{\multimap} m_2' \overset{*}{\multimap} m_2 \in \tau_r$. Since m_2 is bound on m in τ, necessarily $m_2' \xrightarrow{+} m$ holds in τ. Since m_2' is grafted, it is green in τ; hence so is m, as green nodes are upwards-closed. Since grafting and raising increases permissions, m is green or inert in τ_r. Thus m_1, which is directly bound on m, is not red in τ_r.

As an immediate corollary of all the results above, both Rebind and Unif are sound, even on non-admissible problems.

Theorem 7.4.12 *If* Rebind(τ, g) *returns τ', the instance relation $\tau \sqsubseteq \tau'$ holds.* □

Theorem 7.4.13 (Soundness of unification) *If* Unif(τ, N) *returns τ_U, the instance relation $\tau \sqsubseteq \tau_U$ holds.* □

7.4.3 Relating admissibility and the binding trees of unifiers

The fact that some unification problems have non principal sets of solutions is a consequence of the fact that graphic types can be instantiated along two largely orthogonal axis:

- instances of the skeleton;
- instances of the binding tree.

In particular, on non-admissible problems, it is possible to have two unifiers τ_u and τ'_u such that the graph of τ'_u is an instance of the graph of τ_u, but without the same property for their binding trees—in particular we can have $\hat{\tau_u} \not\sqsubseteq \hat{\tau_u}'$. This is of course problematic for principality.

▶ **Example** In Figure 7.2.1, we have $\check{\tau_u} \sqsubseteq_{\mathsf{G}} \check{\tau_u}'$ but $\langle 11 \rangle$ is bound strictly higher in τ_u compared to τ'_u. Thus $\hat{\tau_u}'$ is not an instance of $\hat{\tau_u}$.

Given a unifier τ_v, the following definition characterizes the unifiers τ_u returned by rebind whose binding trees are more general than the one of τ_v. Interestingly, we simply verify that the binding edges of τ_v are included in the transitive closure of the ones of τ_u. There is no need to check for binding flags, as those are essentially determined by the skeletons of the unifiers.

Definition 7.4.14 (*Order on unifiers*) Let τ_u be $\mathsf{rebind}(\tau, g)$ and be τ_v be a unifier of (τ, N), such that $g \sqsubseteq_{\mathsf{G}} \check{\tau_v}$. We write $\tau_u \prec_{\mathsf{U}} \tau_v$ the property

$$\forall n \in g,\ n \xrightarrow{+} \hat{\tau_v}(n) \in \tau_u \qquad \blacksquare$$

As we will prove in the next sections, if $\tau_u \prec_{\mathsf{U}} \tau_v$ holds, then so does $\tau_u \sqsubseteq \tau_v$. Thus Unif is complete and principal when $\mathsf{Unif}_N(\tau)$ is more general for $\prec_{\mathsf{U}}$ than all the other unifiers. As we show in the lemma below, this is always the case when the unification problem is admissible.

Lemma 7.4.15 *Suppose that (τ, N) is admissible, and let τ_v be one unifier of (τ, N). Then $\tau_U \prec_{\mathsf{U}} \tau_v$.* □

(Notice that g_U exists and is more general than $\check{\tau_v}$, by completeness and principality of first-order unification. This ensures that τ_U exists.)

<u>Proof</u>: Let $\tau \sqsubseteq^G \tau'_g \sqsubseteq^R \tau'_r \sqsubseteq^{MW} \tau_v$ be a canonical derivation of $\tau \sqsubseteq \tau_v$. Let τ_G and τ_R be the two graphs τ_g and τ_r for τ_U. Let (**1**) be the fact that $\check{\tau_U} \sqsubseteq_{\mathsf{G}} \check{\tau_v}$. Let n be a node of τ_U. We must prove that $n \xrightarrow{+} \hat{\tau_v}(n) \in \tau_U$.

We suppose that we are not in the degenerate case where $|N| \leq 1$. Indeed, in this case we have $\mathsf{rebind}(\tau, g_U) = \tau$ as $g_U = \check{\tau}$, and the result is an immediate consequence of $\tau \sqsubseteq \tau_v$. By a slight abuse of notation, we call $\langle N \rangle$ the node of τ_U which is the result of merging all the nodes in N.

▷ <u>*Case* n is not under $\langle N \rangle$ in τ_U</u>: In this case, n is also a node of τ, as first-order unification does not merge nodes outside of the subgraphs under the nodes of N. Moreover, n is not partially grafted either, for the same reason. Hence Rebind binds n to its binder in τ, *i.e.* $\hat{\tau_U}(n) = \hat{\tau}(n)$. The conclusion is again by $\tau \sqsubseteq \tau_v$.

▷ <u>*Case* n is under $\langle N \rangle$ in τ_U</u>: Let $m_1, \ldots m_k$ be the nodes of τ_G that are merged together in n in τ_U. Notice that by (1) they are also nodes of τ'_g, and are merged together in τ_v. Moreover, by definition of first-order unification, there exists π and $m'_1, \ldots m'_k$ such that $m'_i \in N$ and $m_i = \langle m'_i \cdot \pi \rangle$ for $1 \leq i \leq k$ (**2**); this last relation is valid in τ_G—thus also in τ_R which has the same skeleton—but also in τ'_r by (1).
Let us first prove that the nodes $\hat{\tau_r}'(m_i)$ are merged together in τ_U (**3**).

- *Case n is bound under $\langle N\rangle$ in τ_v* (**4**): By (2), there exists mixed paths $\langle\epsilon\rangle \overset{*}{\multimap} m'_i \overset{\pi}{\multimap} m_i$ in τ'_r. By well-domination, this path contains $\hat{\tau_r}'(m_i)$. By (4), $\hat{\tau_r}'(m_i)$ is below m'_i in this path. Thus there exists a prefix π'_i of π such that $\hat{\tau_r}'(m_i) = \langle m'_i \cdot \pi'_i\rangle$. Since the m'_i are merged together in τ_v, the nodes $\hat{\tau_r}'(m_i)$ are merged together in τ_v. Thus all the π'_i are the same, as otherwise τ_v would be cyclic. Hence the $\hat{\tau_r}'(m_i)$ are of the form $\langle m'_i \cdot \pi'\rangle$ for a certain π'. This suffices to prove (3), as the m'_i are merged together in τ_U.
- *Case n is bound strictly above N in τ_v* (**5**): by (2), the nodes $\hat{\tau_r}'(m_i)$ are admissibility ancestors for N in τ'_r (**6**), as $\hat{\tau_r}'(m_i) \overset{+}{\multimap} m'_i \overset{*}{\multimap} m_i$ and $m'_i \in N$. Next, by Property 7.2.3, N is admissible for τ'_r. Hence, by (6), the nodes $\hat{\tau_r}'(m_i)$ are totally ordered by $\multimap_{\tau'_r}$. Since all those nodes are merged together in τ_v, they are all equal to a certain node m in τ'_r (as otherwise τ_v would be cyclic). (5) ensures that m is also a node of τ, hence of τ_U. This proves (3).

Let us now conclude by proving that $n \overset{+}{\rightarrowtriangle} \hat{\tau_v}(n)$ in τ_U. By definition of τ'_r, we have $m_i \overset{+}{\rightarrowtriangle} \hat{\tau_r}'(m_i) \in \tau'_g$. Since m_i is in τ_G and both $\tau \sqsubseteq^G \tau_G$ and $\tau \sqsubseteq^G \tau_g$ are from a canonical derivation, we obtain $m_i \overset{+}{\rightarrowtriangle} \hat{\tau_r}'(m_i) \in \tau_G$. The conclusion is by applying Lemma 7.4.4 to this fact and to (3).

Again, this result does *not* hold for some non-admissible problems, as evidenced by Figure 7.2.1.

In Part II of this document, we will generalize graphic types to graphic constraints, and will use a slightly different definition of admissibility. Since we prove the completeness and principality of Unif and Rebind w.r.t. $\prec_{\mathsf{U}}$, we will only need to reprove the result above with the new definition of admissibility.

7.4.4 Completeness of Unif

We start by stating a completeness result for Rebind. As explained in the previous section, we must assume the existence of an unifier with a binding tree sufficiently instantiated. Otherwise, some permission checks performed by Rebind might fail.

Theorem 7.4.16 *Suppose that there exists an unifier τ_v of (τ, N) verifying $\tau_u \prec_{\mathsf{U}} \tau_v$. Then the computation of* Rebind(τ, g) *does not fail.*[6] □

Proof: Let $\tau \sqsubseteq^G \tau'_g \sqsubseteq^R \tau'_r \sqsubseteq^{MW} \tau_v$ be an ordered derivation of $\tau \sqsubseteq \tau_v$. By Lemma 6.3.4 and 6.3.8, let (**1**) be $\tau'_g \sqsubseteq^{R\natural} \tau'_r$, and (**2**) be $\tau'_r \sqsubseteq^{MW\natural} \tau_v$. Let (**3**) be the hypothesis $\tau_u \prec_{\mathsf{U}} \tau_v$. Finally, let (**4**) be the fact that $\breve{\tau_u} \sqsubseteq_{\mathsf{G}} \breve{\tau_v}$ (which is a precondition of (3)).

We show that none of the checks in step 2 of Rebind fails.

- ▷ Step 2a: if a node is grafted in $\breve{\tau_u}$, it is also grafted in $\breve{\tau_v}$ by (3). Those grafting require the nodes to be green in τ. Hence this step does not fail.

For the three remaining checks, we use the following fact: by (1) or (2), any node raised, merged or weakened in the derivation $\tau \sqsubseteq \tau_v$ is not red in τ'_g or τ'_r. Since red nodes are preserved by instance, those nodes are not red in τ either (**5**).

- ▷ Step 2b: by (3), any node raised in τ_u must also be raised in τ_v, and cannot be red by (5). Thus this step does not fail.

[6] We recall that τ_u is defined as rebind(τ, g).

- ▷ Step 2c: by (4), two nodes merged in τ_u are also merged in τ_v; hence any node with a rigid binding in τ_u must also have a rigid binding in τ_v. Hence all the nodes weakened in τ_u must also be weakened in τ_v, and cannot be red in τ by (5). Thus they are not in $\tau_\uparrow$ by Lemma 7.4.10, and this step does not fail.
- ▷ Step 2d: for merging, let m_1 and m_2 be two nodes distinct in $\tau_\uparrow$, merged in τ_u, and bound to the same node m in $\tau_\uparrow$. We must show that those two nodes are not red in $\tau_\uparrow$. By (3), $m_i \xrightarrow{+} \hat{\tau_r}'(m_i) \in \tau_\uparrow$ holds for $1 \le i \le 2$, hence $m_i \longrightarrow m \xrightarrow{*} \hat{\tau_r}'(m_i) \in \tau_\uparrow$. This implies that $\hat{\tau_r}'(m_1) = \hat{\tau_r}'(m_2)$. Thus, m_1 and m_2 are bound at the same node in τ_r' and merged in τ_v by (4). By (2) and (5), they are not red in τ_v, hence not red in $\tau_\uparrow$ by Lemma 7.4.10. This is the desired result.

The completeness of Unif is an immediate consequence of the result above. We give the most general property below, and a simplified result involving admissibility afterwards.

Theorem 7.4.17 (Completeness of unification) *Suppose that τ_v is an unifier of (τ, N) such that $\tau_U \prec_{\mathsf{U}} \tau_v$. Then the computation of $\mathsf{Unif}_N(\tau)$ does not fail.* □

(Notice that g_U exists by completeness of first-order unification, as $\check{\tau_v}$ is a suitable unifier. Hence τ_U exists.)

Proof: We show that the steps of Unif do not fail.

- ▷ Step 1: since g_U exists, the unification of N in $\check{\tau}$ (which builds g_U), does not fail. The cyclicity check does not fail, as cyclicity is preserved by instance and $\check{\tau_v}$ is not cyclic.
- ▷ Step 2: this step does not fail by Theorem 7.4.16 and the hypothesis $\tau_U \prec_{\mathsf{U}} \tau_v$.

As an immediate consequence of this result and of Lemma 7.4.15:

Corollary 7.4.18 *Suppose that N is admissible for τ and that there exists a unifier τ_v of (τ, N). Then the computation of $\mathsf{Unif}_N(\tau)$ does not fail.* □

7.4.5 Principality of Unif

Before proving the principality of Rebind and Unif, we show that Rebind and Unif are stable by the instantiation of their main argument, provided that this argument remains more general than the unifier.

Lemma 7.4.19 *Suppose that $\tau_u = \mathsf{Rebind}(\tau, g)$ exists. Then for any τ' such that $\tau \sqsubseteq \tau' \sqsubseteq \tau_u$, we have $\mathsf{Rebind}(\tau', g) = \tau_u$.* □

Proof: Let us first show that $\mathsf{rebind}(\tau', g) = \tau_u$. We must prove that n_B and $\diamond_n$ are computed identically in τ and τ' for all the nodes of g. It suffices to show this result for one atomic step of a canonical instance derivation of $\tau \sqsubseteq \tau'$, as the result then follows by induction. We thus proceed by case disjunction on such an instance step $\tau \sqsubseteq_1 \tau'$.

- ▷ *Case $\tau' = \mathsf{Graft}(\tau'', m)$*: by hypothesis, τ'' is a constructor type. In step 1a of Rebind, the set M_n might increase for the nodes n such that $\langle\!\langle m \rangle\!\rangle \multimap n \in \tau_u$. The computation of $\diamond_n$ does not change, as the new nodes in M_n are flexibly bound. The computation of n_B does not change either, as the new binders in B_1^n were previously in B_2^n.

- ▷ *Case* $\tau' = \mathsf{Merge}(m_1, m_2)$: the computations of $\diamond_n$ and n_B does not change, as they involve sets that are unchanged by the merging.
- ▷ *Case* $\tau' = \mathsf{Weaken}(m)$: the computation of the new binders is unchanged, as $\hat{\tau} = \hat{\tau}'$. The computation of $\diamond_n$ is unchanged for all the nodes but $\langle\!\langle m \rangle\!\rangle$. For $\langle\!\langle m \rangle\!\rangle$, necessarily at least one of the nodes in $M_{\langle\!\langle m \rangle\!\rangle}$ was already rigid (otherwise m would not be weakened in τ_u), and the computation of $\diamond_{\langle\!\langle m \rangle\!\rangle}$ is also unchanged.
- ▷ *Case* $\tau' = \mathsf{Raise}(m)$: the computation of the binding flags is unchanged, as $\overset{\diamond}{\hat{\tau}} = \overset{\diamond}{\hat{\tau}}{}'$. The computation of n_B is unchanged for all the nodes but $\langle\!\langle m \rangle\!\rangle$. For $\langle\!\langle m \rangle\!\rangle$, $\langle\!\langle m \rangle\!\rangle_B$ is also unchanged, by definition of a least common ancestor (as $\mathsf{LCA}(n_1, n_2) = \mathsf{LCA}(\hat{n_1}, n_2)$ if $n_1 \longrightarrow \mathsf{LCA}(n_1, n_2)$).

It remains to prove that the computation of $\mathsf{Rebind}(\tau', g)$ does not fail. Notice that τ_u is an unifier of (τ', N), since $\tau' \sqsubseteq \tau_u$. The conclusion is then by Theorem 7.4.16 and the fact that $\prec_{\mathsf{U}}$ is reflexive.

Lemma 7.4.20 *Suppose that $\tau_U = \mathsf{Unif}_N(\tau)$ exists. Let τ' be such that $\tau \sqsubseteq \tau' \sqsubseteq \tau_U$. Then $\mathsf{Unif}_N(\tau') = \tau_U$.* □

Proof: Let g and g' be the first-order unifiers of N in τ and τ' respectively. Since $\tau \sqsubseteq \tau' \sqsubseteq \tau_U$, we have $\check{\tau} \sqsubseteq_{\mathsf{G}} \check{\tau}' \sqsubseteq_{\mathsf{G}} \check{\tau_U}$ by Property 5.3.10. By principality of first-order unification, we have $g = g'$. By definition of Unif we have $\mathsf{Unif}_N(\tau) = \mathsf{Rebind}(\tau, g)$ and $\mathsf{Unif}_N(\tau') = \mathsf{Rebind}(\tau', g')$. The conclusion is by Lemma 7.4.19.

Principality requires the same kind of hypothesis as completeness, *i.e.* the existence of an unifier τ_v, and the fact that $\tau_u \prec_{\mathsf{U}} \tau_v$. Indeed, otherwise the binding tree of τ_v could be less instantiated than the one of τ_u.

Theorem 7.4.21 *Suppose τ_v is an unifier of (τ, N) verifying $\tau_u \prec_{\mathsf{U}} \tau_v$. Then $\tau_u \sqsubseteq \tau_v$ holds.* □

Proof: Let (**1**) be the hypothesis $\tau_u \prec_{\mathsf{U}} \tau_v$, which implies in particular $\check{\tau_u} \sqsubseteq_{\mathsf{G}} \check{\tau_v}$ (**2**). We have the existence of $\mathsf{Rebind}(\tau, g)$ by Theorem 7.4.16. We consider a canonical derivation of $\tau \sqsubseteq \tau_u$ (Theorem 7.4.12); the result is by induction on this derivation. If $\tau = \tau_u$, the result is immediate. Otherwise, let τ' be such that $\tau \sqsubseteq_1 \tau' \sqsubseteq \tau_u$. We first prove by case disjunction on $\tau \sqsubseteq_1 \tau'$ that $\tau' \sqsubseteq \tau_v$ holds.

- ▷ *Case* $\tau' = \mathsf{Graft}(\tau'', m)(\tau)$: if τ'' is reduced to $\bot$, then $\tau' \sqsubseteq \tau_v$ holds as $\tau = \tau'$. Otherwise, by (2), $\tau_v(m) = \tau_u(m)$. We conclude by Lemma 6.6.1.
- ▷ *Case* $\tau' = \mathsf{Raise}(m)(\tau)$: by (1), m is also raised in τ_v, and $\hat{\tau_v}(m) \neq \hat{\tau}(m)$. We conclude by Lemma 6.6.2.
- ▷ *Case* $\tau' = \mathsf{Merge}(m_1, m_2)(\tau)$: by (2), m_1 and m_2 are merged in τ_v. We conclude by Lemma 6.6.3.
- ▷ *Case* $\tau' = \mathsf{Weaken}(m)(\tau)$: by construction of Rebind, there exists m' rigidly bound in τ such that m and m' are merged in τ_u. Since we are considering a canonical derivation of $\tau \sqsubseteq \tau_u$, the subgraphs under m and m' are equal, as they are both equal to the subgraph under $\langle\!\langle m \rangle\!\rangle$ in τ_u. We conclude by Lemma 6.6.4.

Thus we have proven that τ_v is an unifier of (τ', N). Next, by Lemma 7.4.19, we have $\mathsf{Rebind}(\tau', g) = \tau_u$. Thus, we can apply the induction hypothesis to τ', which proves the result.

As an immediate consequence (since step 3 of Unif returns the type returned by Rebind):

Theorem 7.4.22 (Principality of unification) *If τ_v is an unifier of (τ, N) verifying $\tau_U \prec_{\mathsf{U}} \tau_v$, then $\tau_U \sqsubseteq \tau_v$ holds.* $\square$

As a corollary of this last result and of Lemma 7.4.15:

Corollary 7.4.23 *Suppose that N is admissible for τ and that there exists an unifier τ_v of (τ, N). Then $\mathsf{Unif}_N(\tau) \sqsubseteq \tau_v$ holds.* $\square$

Notice that in both Theorem 7.4.22 and Corollary 7.4.23, the existence of $\mathsf{Unif}_N(\tau)$ is in fact implied by the corresponding completeness results.

7.4.6 Unification modulo similarity

The following two lemmas justify the fact that we do not need to study principality of unification up to similarity. Indeed, Unif is a morphism for $\sqsubseteq^{rmw}$ and $\approx$. Unlike in the previous sections, we do not give the more general results based on $\prec_{\mathsf{U}}$, as they are not significantly more interesting than the ones below.

Lemma 7.4.24 *Let τ_1 and τ_2 be two types, and N a set of nodes admissible for τ_1. Assume $\mathsf{Unif}_N(\tau_1)$ exists and $\tau_1 \sqsubseteq^{rmw} \tau_2$. Then $\mathsf{Unif}_N(\tau_2)$ exists and $\mathsf{Unif}_N(\tau_1) \sqsubseteq^{rmw} \mathsf{Unif}_N(\tau_2)$.* $\square$

(We recall that admissible problems are stable by instance (Property 7.2.3).)

Proof: By Theorem 7.4.13, $\tau_1 \sqsubseteq \mathsf{Unif}(\tau_1)$. By Lemma 6.7.9, $\sqsubseteq$ and $\sqsubseteq^{rmw}$ are confluent. Hence, there exists τ_{u_1} such that $\mathsf{Unif}(\tau_1) \sqsubseteq^{rmw} \tau_{u_1}$ (**1**) and $\tau_2 \sqsubseteq \tau_{u_1}$. By Corollary 7.4.18, $\mathsf{Unif}(\tau_2)$ exists (since $\tau_2 \sqsubseteq \tau_{u_1}$ and τ_{u_1} is a unifier of N). By Corollary 7.4.23, $\mathsf{Unif}(\tau_2) \sqsubseteq \tau_{u_1}$ (**2**). Now, $\mathsf{Unif}(\tau_2)$ is a unifier of τ_1. By Corollary 7.4.23 again, $\mathsf{Unif}(\tau_1) \sqsubseteq \mathsf{Unif}(\tau_2)$ (**3**). By (1), (2) and (3) and Lemma 5.3.13, $\mathsf{Unif}_N(\tau_1) \approx \mathsf{Unif}_N(\tau_2) \approx \tau_{u_1}$ (**4**). By (3) again and (4), $\mathsf{Unif}_n(\tau_1)$ $(\sqsubseteq \cap \approx)$ $\mathsf{Unif}_N(\tau_2)$, hence $\mathsf{Unif}_N(\tau_1) \sqsubseteq^{rmw} \mathsf{Unif}_N(\tau_2)$.

Lemma 7.4.25 *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 $\mathsf{Unif}_N(\tau_1) \approx \mathsf{Unif}_N(\tau_2)$.*$\square$

Proof: By Lemma 6.7.10, let τ_3 be such that $\tau_1 \sqsubseteq^{rmw} \tau_3$ and $\tau_2 \sqsubseteq^{rmw} \tau_3$. By Lemma 7.4.24, $\mathsf{Unif}_N(\tau_3)$ exists and $\mathsf{Unif}_N(\tau_1) \sqsubseteq^{rmw} \mathsf{Unif}_N(\tau_3)$. $\mathsf{Unif}_N(\tau_3)$ is an unifier of N in τ_2; thus by Corollary 7.4.18, $\mathsf{Unif}_N(\tau_2)$ exists. By Lemma 7.4.24 again, we obtain $\mathsf{Unif}_N(\tau_2) \sqsubseteq^{rmw} \mathsf{Unif}_N(\tau_3)$. As a consequence, $\mathsf{Unif}(\tau_1) \sqsubseteq^{rmw} ; \sqsupseteq^{rmw} \mathsf{Unif}(\tau_2)$ holds, which is the desired result.

Interestingly, these two proofs only use unification-related results, the confluence of $\sqsubseteq$ and $\sqsubseteq^{rmw}$, the decomposition of $\approx$ into $\sqsubseteq^{rmw} ; \sqsupseteq^{rmw}$ and the fact that the kernel of $\sqsubseteq^{\approx}$ is $\approx$. Since similar results hold for the abstraction relation, the result above also holds for $\boxminus$ instead of $\approx$.

Lemma 7.4.26 *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* $\tau_1 \boxminus \tau_2$*. Then* $\mathsf{Unif}_N(\tau_2)$ *exists and* $\mathsf{Unif}_N(\tau_1) \boxminus \mathsf{Unif}_N(\tau_2)$.□

7.5 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 $\langle\!\langle m \rangle\!\rangle \in \tau_u$;
- finding the list of nodes of τ that are merged into a node of τ_u, *i.e.* mapping n to M_n.

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 (Cole and Hariharan 2005).

Theorem 7.5.1 Rebind *is linear in the size of its arguments.* □

Proof: Consider the computation of $\mathsf{Rebind}(\tau, g)$. We first annotate the nodes of τ by their permissions, by walking $\hat{\tau}$ first from the root (for G, O and R permissions), and again bottom-up (for M and I permissions). This takes a linear time. We also mark partially grafted nodes by walking down along g starting from the root. The first time an instantiated bottom node is found, we mark all the nodes under it. When a marked node is found a second time, we stop the visit. The whole operation has a cost linear in the size of g.

1. A topological sort can be used to find a top-down ordering; this takes a linear time in the size of g.
   a) Constant time, by hypothesis.
   b) Linear in the size of M_n.
   c) B_1^n is smaller than M_n, and be computed linearly in its size.

      We let $b_2(n)$ be the size of B_2^n. B_2^n can be computed in a time proportional to $b_2(n)$. (This is an amortized bound; the computation can be done when marking partially grafted nodes).

      Each call of LCA on a pair of nodes takes a constant-time (Cole and Hariharan 2005). Hence the whole computation of $\mathsf{LCA}_{\hat{\tau}'}(B_1^n \cup B_2^n)$ takes a time linear in the size of $B_1^n \cup B_2^n$, *i.e.* linear in $|M_n| + b_2(n)$. Updating the structure used by LCA takes constant-time (Cole and Hariharan 2005) per node of τ'.
   d) In constant time.

   Thus each step takes a time linear in $|M_n| + b_2(n)$. The M_n are a partition of the nodes of τ, and the sum of $b_2(n)$ on all nodes is less than the number of edges of g. Hence the whole cost is linear in $|\tau| + |g|$.
2. Building $\hat{\tau_\dagger}$ can be done by an infix walk of $\hat{\tau}$, and takes a time linear in the time of τ. The permission of the nodes of $\tau_\dagger$ can be computed in linear time in its size.

a) This step can be done by a linear pass on the number of grafted bottom nodes, which can be marked during unification. Each check takes constant time, hence the cost is at most linear in the size of τ.

b) Each check takes a time at most linear in the size in M_n. Since the M_n partition the nodes of τ, the total cost is linear in the size of τ.

c) As the previous step.

d) We start by partitioning the nodes of τ merged together in g according to their binder in $\tau_\uparrow$. This can be done in linear time by attaching to a node m of $\tau_\uparrow$ the list of the nodes bound to m. Then, for each list containing more than one node, we check that no node of the list is red. Since the lists partition the nodes of τ, this step is linear in the size of τ.

Theorem 7.5.2 Unif *is linear in the size of its argument.* □

Proof: Consider the computation of $\mathsf{Unif}_N(\tau)$. For step 1, first-order unification and occur-check are linear in the size of $\breve{\tau}$ (Paterson and Wegman 1978). Moreover, the resulting term-graph g_u is at most linear in the size of τ (**1**). For step 2, by Theorem 7.5.1 and (1), the computation of $\mathsf{Rebind}(\tau, g_u)$ is also linear in the size of τ. Hence Unif has linear complexity.

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 (Huet 1976) 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 ML^F is not known, it performs many duplications and α-conversions. We believe it would not have scaled to larger inference problems, *e.g.* automatically generated code.

7.6 Generalized unification problems

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

Definition 7.6.1 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_u of τ such that $(\sim) \subseteq (\tilde{\tau_u})$. ■

The equivalence relation $\sim$ of a generalized unification problem $(\tau, \sim)$ 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 $\sim$.

7.6.1 Generalized 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 7.6.1: although it is clear that τ and τ' have the same solutions, τ

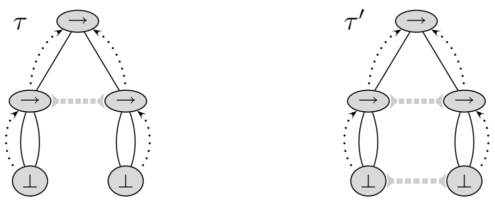


Figure 7.6.1 – Generalized unification problems and admissibility

would be admissible according to this criterion, while τ' would not. Thus, we use a slightly more powerful definition.

Definition 7.6.2 (*Generalized admissibility)* A generalized unification problem $(\tau/{\sim})$ is said to be admissible if there exists a set S of sets of nodes of τ such that each set N of S is an admissible problem for τ, and $\sim$ and S have the same first-order solutions on $\breve{\tau}$. (A first-order solution of S on $\breve{\tau}$ is an instance g of $\breve{\tau}$ such that, for any N of S, all nodes of N are merged in g.) ■

With this criterion, the generalized unification problems on both types τ and τ' of Figure 7.6.1 are admissible, by taking $S = \{\{\langle 1\rangle, \langle 2\rangle\}\}$ in both cases: the edge $\langle 11\rangle \mathrel{\rhd\!\cdots\!\lhd} \langle 22\rangle$ of τ' is redundant. More generally, congruence preserves generalized admissibility, a very desirable result.

7.6.2 Generalized unification algorithm

The Unif algorithm can be generalized in a straightforward manner to generalized unification problems, thanks to the clean separation between the computations of $\breve{\tau}_u$ and $\hat{\tau}_u$: 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 $\sim$.

Lemma 7.6.3 *The generalized unification algorithm is sound, complete, and principal on generalized admissible problems.* □

Proof: Let τ be a type, $\sim$ a generalized admissibility problem for τ. By definition of generalized admissibility, let S be a set of sets of nodes of τ with the same first-order solutions as $\sim$. The result is immediate by induction on the number k of sets in S, using the stability of admissibility by instance (Property 7.2.3), and Theorems 7.4.13, 7.4.17 and 7.4.22.

7.7 Unification in restrictions of ML^F

In the terminology of Le Botlan and Rémy (2007), the system presented in this document is Full ML^F, the most expressive of the ML^F variants (see also Appendix A). A natural restriction of Full ML^F exists, the system Shallow ML^F. As mentioned when defining shallow $\mathcal{F}$ types in §3.4.3, Shallow ML^F is obtained by restricting types: under a rigid edge, only variables can be flexibly bound. While less expressive than Full ML^F, it can be shown that Shallow ML^F is as expressive as System F. In Shallow ML^F, the type instance relation is the restriction of $\sqsubseteq$ to the types allowed in Shallow ML^F. Since the restriction on allowed types is preserved by unification, Unif can be used unchanged to perform unification in Shallow ML^F.

Alternatively, an even more drastic restriction is ML itself, in which all the nodes are flexibly bound to the root. However, in this completely degenerate case, most of the steps of Rebind can be removed: the computation of the new binding flag and the permissions checks are not needed, since all nodes are flexibly bound. Moreover the computation of the new binders can also be simplified, as Rebind will always bind all the nodes to the root. Thus, only first-order unification remains.

8

Relating the syntactic and graphic presentations of $\mathsf{ML^F}$ types

Abstract

We compare the syntactic and graphic presentations of the $\mathsf{ML^F}$ instance relation, and explain the choices we made in the design of this relation (§8.1). We also show that the graphic $\sqsubseteq$ and $\sqsubseteq$ relations are more expressive than their original syntactic counterpart (§8.1.3). We present two algorithms that translate to and from syntactic types (§8.2), both with linear time complexity. Finally, we introduce a simple syntactic sugar that can be used to display types in a much more readable way (§8.3).

8.1 An informal comparison of the syntactic and graphic instance relations

The rules of the original syntactic presentation of $\mathsf{ML^F}$ are recalled in Appendix B. Prior knowledge of this presentation is helpful to understand this section, but not strictly necessary.

8.1.1 Syntactic and graphic instance

The instance relation $\sqsubseteq$ on graphic types is noticeably simpler than its syntactic counterpart:

- A first difference is that it does not need not be defined under prefix; instead, it uses permission to operate deeply inside types. As a consequence, context rules such as I-Context-R or I-Context-L are superfluous.

- A second (seemingly inconsequential) difference stems from the way instance is defined. In the syntactic presentation, instance is defined as a super-relation of abstraction (through I-Abstract), itself a super-relation of equivalence (through A-Equiv). Conversely, on graphic types, abstraction is defined as a restriction of instance. While the two approaches are formally equivalent, we believe that ours is more lightweight. In particular, we avoid some redundancies, such as the duplication between I-Hyp and A-Hyp, or Eq-Context-R, A-Context-R and I-Context-R. Moreover, when proving a property on $\sqsubseteq$, we can focus on this relation, and need not prove this property for $\sqsubseteq$ or $\equiv$.

Of course, there are also links between the two presentations. In particular, the graphic atomic instance operations can be put in correspondence with some of the rules of the syntactic presentation:

- grafting corresponds to the rule I-Bot;
- weakening corresponds to the rule I-Rigid;
- raising corresponds to the derived rules I-Up and A-Up;

However, 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 more direct and simpler.

8.1.2 Syntactic equivalence and graphic similarity

Things are not so clear when comparing the syntactic equivalence relation $\equiv$ and the graphic similarity relation $\approx$. A careful study of both relations show that they capture the same transformations, but *only up to the differences in the representations*:

- The equivalences syntactically captured by the rules Eq-Comm, Eq-Free and Eq-Var are directly captured by the graphic representation (and have hence no equivalent on graphic types):
  - binders are not ordered in graphic types; hence there is no need for a rule to commute them;
  - unused quantification cannot be expressed;

  – quantifications of the form $\forall\,(\alpha \diamond \sigma)\ \alpha$, which can be simplified into σ by Eq-Var, are directly represented by σ in graphic types.

  In this respect, $\approx$ is much simpler than $\equiv$.

- Conversely, graphic types bind all nodes, while syntactic types allow monotypes. Thus the rule Eq-Mono must be "inlined" on graphic types, and we must allow raising, merging, weakening and the three symmetric operations on monomorphic nodes. While this complicates the definition of $\approx$ (compared to a system in which monomorphic nodes would be unbound), this actually simplifies the metatheoretical study. Indeed, $\sqsubseteq^r$, $\sqsubseteq^m$, $\sqsubseteq^w$ and their symmetric relations are actually restrictions of the corresponding relations in $\sqsubseteq^{\boxminus}$, and much of the work involved in studying those relations is already needed elsewhere.

  Moreover, we do not need to introduce special relations to remove binding edges, or merge unbound nodes. In previous presentations of graphic types (Rémy and Yakobowski 2007) we used not to bind monomorphic nodes. Therefore $\sqsubseteq^{rmw}$ was in fact equal to $\sqsubseteq^m$, which captured the merging of unbound graphs. Perhaps surprisingly, proofs involving $\sqsubseteq^m$ were very tedious. A good example is the proof of confluence of $\sqsubseteq^M$ and $\sqsubseteq^m$ (Lemma 6.7.2): without binding edges to "follow", it it very hard to tell whether $\tau' = \mathsf{Merge}(n_1, n_2)(\tau)$ holds, or merely $\tau' = \tau[n_1 = n_2]$.

The other, perhaps more important difference, is that the graphic instance relation only allows oriented similarity $\sqsubseteq^{rmw}$, while the syntactic instance allows the whole equivalence relation $\equiv$. This is a key design choice:

- it allows using first-order unification with dags, hence efficient algorithms;
- it drastically simplifies reasonings on instance, as $\sqsupseteq^{rmw}$ does not "pollute" all the proofs.

However, we still want to prove that similarity does not change the meaning of types. Thus, in a second step, we show that our important results (essentially principality of unification and principality of type inference) "commute" with similarity. Importantly, those commutations are only needed for a few key theorems, instead of for all the proofs involving $\sqsubseteq$, $\sqsubseteq$ or $\sqsubseteq^{rmw}$ (which is essentially the case of all proofs).

8.1.3 Comparison with the original syntactic relations

Disregarding the fact that (graphic) $\sqsubseteq$ and $\sqsubseteq$ do not allow $\sqsupseteq^{rmw}$, those two relations are larger than their syntactic counterparts in the original syntactic presentation of ML^F (Le Botlan and Rémy 2003). We have highlighted examples of the differences in Figure 8.1.1; all the transformations of this figure hold in graphic ML^F, but not in this original syntactic presentation. A cursory glance could give the impression that only $\sqsubseteq$ has been enriched; this is of course misleading, as $(\sqsubseteq) \subset (\sqsubseteq)$.

The first two transformations $\tau_1 \sqsubseteq \tau_1'$ and $\tau_2 \sqsubseteq \tau_2'$ involve the meaning of rigid edges, more precisely the idea that a single rigid edge is sufficient to freeze instantiation. In the original presentation of ML^F, abstraction was only possible in completely rigid contexts, of the form O^+, and those two transformations were not in $\sqsubseteq$. (The types τ_1 and τ_1' were in instance relation; however the transformation was thus irreversible.)

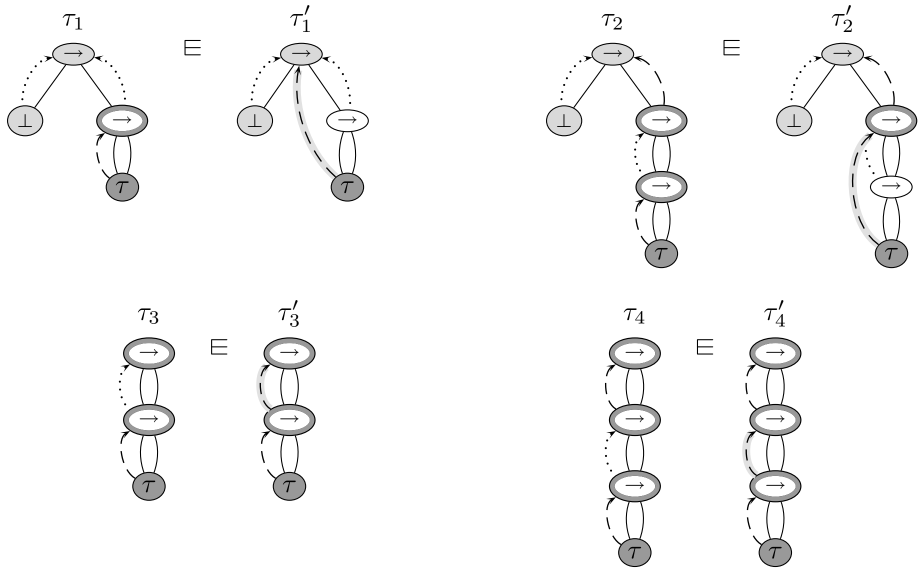


Figure 8.1.1 – Graphic relations not verified in the original syntactic presentation

This extension was first suggested by François Pottier, on the original syntactic presentation. However, naively changing $\sqsubseteq$ would have broken type soundness by an intricate interaction with context rules. Abstraction was extended simultaneously in this work, and in a second syntactic presentation by Le Botlan and Rémy (2007). However, in the latter, the introduction of a new relation, called *protected abstraction*, was necessary. Since graphic types do not use context rules, we only had to change permissions to extend $\sqsubseteq$.

The transformations $\tau_3 \sqsubseteq \tau_3'$ and $\tau_4 \sqsubseteq \tau_4'$ involve inert nodes. Again, only $\tau_3 \sqsubseteq \tau_3'$ was available in the original $\mathsf{ML^F}$, and τ_4 and τ_4' were not in instance relation at all. Originally, we had not introduced inert nodes on graphic types (Rémy and Yakobowski 2007). However, this broke a subtle invariant in the proof of completeness of the unification algorithm. In retrospect, although we first introduced inert nodes for technical reasons, their explanation in System $\mathcal{F}$ is very simple, and easily carries over to $\mathsf{ML^F}$—thus the semantic justification we gave in this document.

8.2 Translating graphic types to and from syntactic types

8.2.1 From graphic to syntactic types

Graphic types can be translated into syntactic types. In order to translate a node n, we first introduce syntactic quantifications for all the nodes bound on n, by recursively translating those nodes. Then we simply read the equation of n in the structure of the graph. The corresponding algorithm is shown in Figure 8.2.1, the translation of a graphic type τ being $\mathcal{S}_\tau(\langle\epsilon\rangle)$. The algorithm associates to a node n of τ a variable α_n.

$$\begin{array}{rcl} \mathcal{S}_\tau(n) & = & \forall\,(\alpha_{n_1}\ \overset{\diamond}{\tau}(n_1)\ \mathcal{S}_\tau(n_1))\ \dots\ \forall\,(\alpha_{n_i}\ \overset{\diamond}{\tau}(n_i)\ \mathcal{S}_\tau(n_i))\ \ \tau(n)(\alpha_{\langle n\cdot 1\rangle}, \dots, \alpha_{\langle n\cdot \mathsf{arity}(\tau(n))\rangle}) \\ & & \text{where } n_1, \dots\ n_i \text{ is one ordering of } (\relbar\!\!\!\rhd n) \text{ for } \overset{+}{\multimap}\text{, lowest nodes first} \end{array}$$

Figure 8.2.1 – Translation from graphic to syntactic types

Lemma 8.2.1 *Given a graphic type τ, $\mathcal{S}_\tau(\{\epsilon\})$ is a well-scoped syntactic type.* □

Proof: The proof is a consequence of well-domination. Suppose that the translation uses α_n. Then we are translating n_k, with $n = \langle n_k \cdot i\rangle$ for some i. We need to prove that α_n is in scope when n_k is translated. To translate n_k, we have followed a mixed path $\{\epsilon\} = n_1 \lhd\!\!\!\relbar \dots \lhd\!\!\!\relbar n_k$. Hence the path $n_1 \lhd\!\!\!\relbar \dots \lhd\!\!\!\relbar n_k \multimap n$ is correct. By well-domination, $\hat{n}$ is in $\{n_1, \dots, n_k\}$; Let us suppose it is n_l for a certain l.

- ▷ If l is not k: n is lower in the type than n_{l+1}. Hence α_n has been introduced earlier, when the nodes bound on n_l have been bound.
- ▷ If l is k: α_n is introduced just before the structure of n_k is translated.

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

Lemma 8.2.2 *Given a type τ, if σ_1 and σ_2 are two translations of τ for different orderings w.r.t. $\multimap$, then σ_1 and σ_2 are equal up to some permutation of binders (hence syntactically equivalent for $\equiv$).* □

Proof: Let $n_1, \dots, n_i$ and $n'_1, \dots, n'_i$ be two ordering of $(\relbar\!\!\!\rhd n)$ for a certain node n. We can transform one ordering into the other using only transpositions of two elements. Moreover there exists strategies that use only valid orderings. Each transposition is captured by one application of the rule Eq-Comm of syntactic $\mathsf{ML^F}$.

Lemma 8.2.3 *$\mathcal{S}_\tau(\{\epsilon\})$ can be computed in linear time in the size of τ.* □

Proof: Each structure and binding edge is visited once; thus it suffices to compute a possible ordering for $\multimap$ in linear time. This can be done by a depth-first traversal of τ (for $\multimap$). Each time all the successors of a node n have been visited, we add n at the end of a list attached to $\hat{n}$. Then the nodes bound on n must be translated in the order indicated by the list attached to n. Depth-first searches can be done in linear time in the size of the graph, hence the conclusion.

8.2.2 From syntactic to graphic types

Since graphic types bind all the nodes of a graph, we limit ourselves to *restricted* syntactic types, generated by the grammar

$$\begin{array}{rcl} t_r & ::= & \alpha \mid C(\alpha, ..., \alpha) \\ \sigma_r & ::= & t_r \mid \bot \mid \forall\,(\alpha \diamond \sigma_r)\ \sigma_r \end{array}$$

That is, we disallow monotypes of the form $C(C(...), ...)$. 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 corresponding function B is presented in Figure 8.2.2.

$$\begin{array}{rcl} \mathsf{B}(\alpha) & = & \alpha \\ \mathsf{B}(C(\tau_1, \dots \tau_n)) & = & \forall\,(\alpha_1 \geqslant \mathsf{B}(\tau_1)) \dots \forall\,(\alpha_n \geqslant \mathsf{B}(\tau_n))\ C(\alpha_1, \dots, \alpha_n) \\ & & \text{where } \alpha_1, \dots \alpha_n \text{ are distinct from } \mathsf{ftv}(C(\tau_1, \dots \tau_n)) \\ \mathsf{B}(\bot) & = & \bot \\ \mathsf{B}(\forall\,(\alpha \diamond \sigma)\ \sigma') & = & \forall\,(\alpha \diamond \mathsf{B}(\sigma))\ \mathsf{B}(\sigma') \end{array}$$

Figure 8.2.2 – Binding monomorphic subtypes

The following result, whose proof is immediate, ensures that this function is correct.

Lemma 8.2.4 *Given a syntactic $\mathsf{ML^F}$ type σ, the type $\mathsf{B}(\sigma)$ is a restricted type. Moreover, it is syntactically equivalent to σ (*i.e. $\sigma \equiv \mathsf{B}(\sigma)$*) and can be computed in linear time in the size of σ.* □

The translation of a syntactic type into a graphic one needs to take into account free variables. We introduce the notion of partial graph in order to model the objects returned by this algorithm when it is called on non-closed syntactic types.

Definition 8.2.5 (*Partial type*) A graph τ is a *partial type* if there exists a set V of bottom-labelled nodes of τ that are not bound in τ, and if τ is such that the graph obtained by binding the nodes of V to the root of τ is a well-formed graphic type. ■

The algorithm translating a restricted type σ into a graphic type $\mathcal{G}_\rho(\sigma)$ is given in Figure 8.2.3. It takes as input an environment ρ mapping any free variable of σ to a bottom-labelled node; the translation of a closed type σ is simply $\mathcal{G}_\epsilon(\sigma)$, where ϵ is the empty environment. The algorithm is defined inductively, and returns a partial type represented by a standard graph (§3.2.1.2). New nodes are taken all distinct from one another in a global pool of nodes (which is left implicit), using the notation "$\mathcal{V}(C)$" to mean the allocation of a fresh node labelled by C. Given a standard graph τ, we write $\mathsf{r}(\tau)$ for its root node. We use $+$ to aggregate elements (nodes, structure edges or binding edges) composing a standard graph.

Translating the type $\bot$ results in a type reduced to a single node labelled by $\bot$. In the case of a variable α, we simply return the node corresponding to α in the environment. For the application of a constructor C, we create a new node labelled by C whose children are the nodes corresponding to the variables that are the arguments of C. The most involved

$$\begin{aligned}
\mathcal{G}_\rho(\bot) &= \mathcal{V}(\bot)\\
\mathcal{G}_\rho(\alpha) &= \rho(\alpha)\\
\mathcal{G}_\rho(C(\alpha_i^{i\in I})) &= \mathsf{let}\ n = \mathcal{V}(C)\ \mathsf{in}\ n + \Big(\rho(\alpha_i) + n \xrightarrow{i} \rho(\alpha_i)\Big)^{i\in I}\\
\mathcal{G}_\rho(\forall\,(\alpha \diamond \sigma)\ \sigma') &= \mathsf{let}\ \tau = \mathcal{G}_\rho(\sigma)\ \mathsf{and}\ n_\alpha = \mathcal{V}(\bot)\ \mathsf{in}\\
&\ \mathsf{let}\ \tau' = \mathcal{G}_{\rho,\ \alpha\mapsto n_\alpha}(\sigma')\ \mathsf{in}\\
&\ \mathsf{if}\ \mathsf{r}(\tau') = n_\alpha\ \mathsf{then}\ \tau\ \mathsf{else}\\
&\ \mathsf{if}\ n_\alpha \notin \mathsf{dom}(\tau')\ \mathsf{then}\ \tau'\ \mathsf{else}\\
&\ \tau'[\tau/n_\alpha] + \mathsf{r}(\tau) \xrightarrow{\diamond} \mathsf{r}(\tau')
\end{aligned}$$

Figure 8.2.3 – Translation from syntactic types to types

case is $\forall\,(\alpha \diamond \sigma)\ \sigma'$. We start by translating the bound σ into τ. Then we translate σ' into a type τ', in an environment mapping α to a fresh variable node n_α. Then we graft τ at n_α in τ' and bind the root of τ (or equivalently n_α) to the root of τ', with the appropriate flag. There are however two special cases:

- When σ' is equivalent to α, which implies that τ' is n_α (hence also $\mathsf{r}(\tau') = n_\alpha$), we simply return the translation of σ. This is the equivalent of the syntactic rule Eq-Var, which states that $\forall\,(\alpha \geqslant \tau)\ \alpha \equiv \tau$.
- In some cases, the bound σ of α might not be useful. This happens if $\alpha \notin \mathsf{ftv}(\sigma')$, but also more generally if α does not appear free in $\mathsf{nf}(\sigma')$, where nf is the syntactic normal form of a type. We detect this case by checking whether n_α is in τ'. If it is not, we can directly return τ', without grafting τ or adding the binding edge.

This algorithm returns a correct type:

Lemma 8.2.6 *Given a closed syntactic restricted type σ, $\mathcal{G}_\epsilon(\sigma)$ returns a graphic type.* □

This result is an immediate consequence of the more general one below.

Lemma 8.2.7 *Given a syntactic restricted type σ and an environment ρ mapping at least* $\mathsf{ftv}(\sigma)$ *to bottom-labelled nodes, $\mathcal{G}_\rho(\sigma)$ returns a partial type in which only the nodes of* $\mathsf{codom}(\rho)$ *are unbound.* □

Proof: We slightly strengthen our result, and also prove that the nodes in the graph returned by $\mathcal{G}_\rho(\sigma)$ are fresh, except for those in $\mathsf{dom}(\rho)$. The proof is by induction on the shape of σ. The first three cases are immediate, and it remains to consider the case $\forall\,(\alpha \diamond \sigma)\ \sigma'$.

▷ *Case* $\mathcal{G}_\rho(\forall\,(\alpha \diamond \sigma)\ \sigma') = \tau$: this case is immediate by induction hypothesis.

▷ *Case* $\mathcal{G}_\rho(\forall\,(\alpha \diamond \sigma)\ \sigma') = \tau'$: by induction hypothesis, τ' is a partial graph in which only the nodes of $\mathsf{codom}(\rho)$ and n_α are not bound. However, n_α is not in τ' by hypothesis. Thus only the nodes of $\mathsf{codom}(\rho)$ are unbound, which is the desired result.

▷ *Case* $\mathcal{G}_\rho(\forall\,(\alpha \diamond \sigma)\ \sigma') = \tau'[\tau/n_\alpha] + \mathsf{r}(\tau) \xrightarrow{\diamond} \mathsf{r}(\tau')$: by induction hypothesis, $\tau = \mathcal{G}_\rho(\sigma)$ and $\tau' = \mathcal{G}_{\rho,\ \alpha\mapsto n_\alpha}(\sigma')$ are partial types, with no common nodes between them except

for those in the codomain of ρ and n_α (**1**). Moreover, n_α does not appear in τ and the nodes of $\mathsf{codom}(\rho) \cup \{n_\alpha\}$ are not bound in τ and τ'.
Let τ'' be $\tau'[\tau/n_\alpha] + \mathsf{r}(\tau) \xrightarrow{\diamond} \mathsf{r}(\tau')$. τ'' is a correct pre-type, up to the unbound nodes: the only non-immediate property is the acyclicity of $\check{\tau}''$ and $\hat{\tau}''$, which is ensured by (1) (as only leaves are shared between τ and τ'). It remains to prove that τ'' is well-dominated. For the nodes unbound in τ'', the result is immediate (as they would be well-dominated once they are bound to the root). Thus, consider a bound node n of τ''.

- <u>*Case $n \in \tau'$*</u>: all the mixed paths to n in τ'' are mixed paths in τ' by (1). Hence the result is by well-domination of τ'.
- <u>*Case $n = n_\alpha$*</u>: the binder of n is the root. Thus well-domination is immediate.
- <u>*Case $n \in \tau$*</u>: all the mixed paths to n contain n_α, by (1). Hence, all those mixed paths end by mixed paths of τ, and the result is by well-domination of τ.

Moreover, the translation can be performed in linear time.

Lemma 8.2.8 *Given a syntactic type σ, the computation of $\mathcal{G}(\sigma)$ can be done in linear time in the size of σ.* □

<u>Proof</u>: The proof is by induction on σ. All cases are structurally recursive on σ, except the check $n_\alpha \notin \mathsf{dom}(\tau')$. The easiest solution is not to perform this check at all during the translation. Instead, we add a cleaning operation at the end of the algorithm, which removes all parts of the type not reachable from the root by $\xrightarrow{+}$; the nodes that would have been eliminated by the check are those only accessible through $\xleftarrow{+}$.

8.3 A simple syntactic sugar to display types

MLF types are often cumbersome to read (and even more to write). Unfortunately, even simple functions might have complex principal MLF types. For example, the principal type of $\lambda(x)\ \lambda(y)\ \lambda(z)\ z$ in MLF is

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

compared to the one in ML

$$\forall\alpha.\ \forall\beta.\ \forall\gamma.\ \alpha \to \beta \to \gamma$$

The MLF type is much more complex, as it uses additional type variables (α' and β') for the bounded quantification. Furthermore, the structure of the type is not readily apparent: α and α' are quantified next to the other, even though one is used on the left of the arrow, and the other on the right.

Graphic types offer a partial remedy to this problem. In particular, the graphic representation of both types (given in Figure 8.3.1) makes apparent that they have exactly the same skeleton.[1] It is also easy to see that the ML type is an instance of the MLF one.

[1]We have intendedly not bound the monomorphic nodes in the ML type, in order to make the comparison fairer. Indeed, monomorphic nodes need not be presented to the programmer.

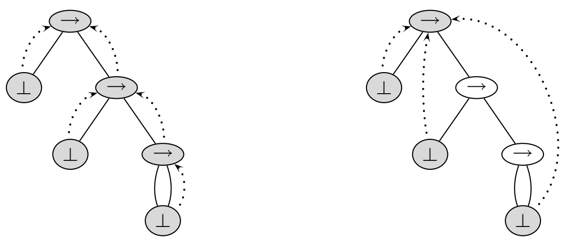


Figure 8.3.1 – $\mathsf{ML^F}$ and ML type for $\lambda(x)\ \lambda(y)\ \lambda(z)\ z$

Nevertheless, graphic types may look as plainly unfamiliar to the ML programmer as $\mathsf{ML^F}$ syntactic types do. Thus we propose to display $\mathsf{ML^F}$ types as syntactic types, using a simple syntactic sugar that removes in practice a lot (if not all) of the occurrences of bounded quantification.

8.3.1 Inlining bounds

The general idea is the following: when translating a graphic type into a syntactic type, we inline the bounds which can be rebuilt *unambiguously* when the inverse translation is performed. Indeed, we want our syntactic sugar to be bijective.

This restriction means in particular that we never inline the bound of a variable which is used twice.[2] Moreover, we must be able to reconstruct the binder and the binding flag of the bound. For the binder, we require from the node we translate to be bound exactly to its structural ancestor; otherwise, we do not inline it. For the binding flag, we use the variance of the constructor of this ancestor: we inline flexible edges on arguments in covariant positions, and rigid edges on arguments in other positions (nonvariant or contravariant).

► **Example** Consider the first graphic type of Figure 8.3.1, whose corresponding syntactic type was given at the beginning of the previous section. The bounds of the nodes $\langle 2\rangle$ and $\langle 22\rangle$ can be unambiguously inlined, as they verify the conditions above. Indeed:

- each node is "used" only once, *i.e.* there is a unique structure path from its bound to the node itself;
- the two nodes are bound to $\{\epsilon\}$ and $\langle 2\rangle$ respectively, which are their structural ancestors;
- they are flexibly bound, and the arrow constructor is covariant on its second argument.

As a result, we obtain the syntactic type

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

[2]As the examples below will show, this is weaker than saying that we do not inline shared nodes.

Of course, this type is still not as simple as the ML type. This is not entirely surprising: as MLF types generalize System F types, it is unlikely we would be able to remove inner quantification altogether. However, we believe that programmers willing to require second-order polymorphism will not be put off by System F types.

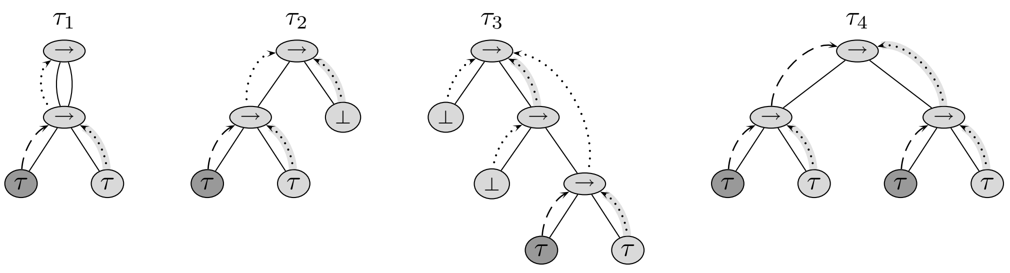


Figure 8.3.2 – Inlining bounds

▶ **Another example** Consider the types of Figure 8.3.2. In all four cases, the subtrees representing the type $\tau \to \tau$ can be inlined. However, this is not always the case for the other quantifications, and we have highlighted the edges that can be inlined. We detail each type below.

- in τ_1, the node $\langle 1\rangle$ is used twice, as $\langle 1\rangle = \langle 2\rangle$. Hence we cannot inline its bound. Thus τ_1 is written as
$$\forall\,(\alpha \geqslant \tau \to \tau)\ \alpha \to \alpha$$

- In τ_2, $\langle 1\rangle$ is used only once and is bound to its ancestor, but the variance does not match. Indeed, $\langle 1\rangle$ is flexibly bound, but it is the first argument of an arrow, which is contravariant in its first argument. Thus τ_2 is written
$$\forall\,(\alpha \geqslant \tau \to \tau)\ \alpha \to \bot$$

- In τ_3, while $\langle 22\rangle$ is used only once and flexibly bound, it is bound to $\langle\epsilon\rangle$ and not to its structural ancestor $\langle 2\rangle$. Hence we do not inline its bound. However we can inline the bound of $\langle 2\rangle$. This results in
$$\forall\,(\alpha)\ \forall\,(\beta \geqslant \tau \to \tau)\ \alpha \to (\forall\,(\gamma)\ \gamma \to \beta)$$

  Interestingly, a syntactic sugar similar to ours has been proposed by Leijen and Löh (2005, §2.6). However, theirs did not take the binder of the node to inline into account, and was thus not bijective. Indeed, it would have inlined β into the type above, resulting in the same translation as the type τ_3' derived from τ_3 by binding $\langle 22\rangle$ to $\langle 2\rangle$ instead of $\langle\epsilon\rangle$. (We also suggest to use the variance of all type constructors, while they restrain themselves to the arrow constructor but this is more anecdotical.)

- Finally, in τ_4, all the bounds can be inlined, and we simply write this type as

$$(\tau \to \tau) \to (\tau \to \tau)$$

8.3.1.1 In practice

The restrictions above can seem quite drastic, and one could think we rarely inline bounds. This is however not the case. In fact, in practice, we have found that most terms have a principal type which is in System F after inlining. In general, only partial application of two polymorphic terms—which are infrequent, especially at toplevel—have types that cannot be simplified.[3] For example, consider the term $\lambda(x)\ (\lambda(y : \sigma_{\mathsf{id}})\ y)$. Its principal type is

$$\forall\,(\alpha)\ \forall\,(\alpha' \geqslant \forall\,(\beta = \sigma_{\mathsf{id}})\ \forall\,(\gamma \geqslant \sigma_{\mathsf{id}})\ \beta \to \gamma)\ \alpha \to \alpha'$$

After inlining, we obtain the much more readable type

$$\forall\,(\alpha)\ \alpha \to \sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$$

More generally, in $\mathsf{ML^F}$, an annotated λ-abstraction of the form $\lambda(x : \tau)\ a$ receives a principal type of the form

$$\forall\,(\alpha = \tau)\ \forall\,(\beta \geqslant \tau')\ \alpha \to \beta$$

with $\alpha \notin \mathsf{ftv}(\tau')$. This means that we can *always* inline such a type into $\tau \to \tau'$.

Moreover, since our convention respects the variance of constructors, it is always clear when a type variable can be instantiated by $\sqsubseteq$ in an inlined type. This is quite useful, as the simplest instances of the type can be predicted without knowing the full definition of the $\mathsf{ML^F}$ instance relation. Indeed, disregarding non-arrow constructors for simplicity, the following simple rule holds:

a variable can be instantiated if the subtype at which it is introduced is reached by descending only on the right of arrows, and following only flexible quantification.

▶ **Example** Consider again the type $\forall\,(\alpha)\ \alpha \to \sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$ discussed above. If we expand the definition of σ_{id}, we obtain

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

In this type, β cannot be instantiated (since it appears at the left of an arrow), but both α and γ can.

8.3.2 Algorithm

Interestingly, graphic types are more suited to formally describe the algorithm for inlining bounds: on syntactic types, the order between quantifier—*e.g.* $\forall\,(\alpha \diamond_1 \sigma_1)\ \forall\,(\beta \diamond_2 \sigma_2)\ \sigma$ versus $\forall\,(\beta \diamond_2 \sigma_2)\ \forall\,(\alpha \diamond_1 \sigma_1)\ \sigma$—makes the definition cumbersome. However this is just a matter of convenience, and the algorithm can be adapted to the syntactic presentations of $\mathsf{ML^F}$.

[3] An example is the type $\forall\,(\alpha \geqslant \forall\,(\beta)\ \beta \to \beta)\ \alpha \to \alpha$ of choose id. As the bound of α is used twice, it cannot be inlined.

Let us write $\mathsf{Var}_i(C)$ for the variance of the type constructor C on its ith argument; for example, $\mathsf{Var}_1(\rightarrow) = -$ and $\mathsf{Var}_2(\rightarrow) = +$. We define a predicate that asserts whether or not a node can be inlined when translating a type τ.

$$\mathsf{Inline}(\tau, n) \quad \triangleq \quad \exists i \in \mathbb{N}, \wedge \begin{cases} \{\pi \mid \hat{\tau}(n) \xrightarrow{\pi}\!\!\circ\; n\} = \{i\} \\ \vee \left[\begin{array}{l} \overset{\diamond}{\tau}(n) = (\geqslant) \wedge \mathsf{Var}_i(\tau(n)) = + \\ \overset{\diamond}{\tau}(n) = (=) \wedge \mathsf{Var}_i(\tau(n)) \neq + \end{array} \right. \end{cases}$$

The first condition ensures that n is used exactly under the node to which it is bound, and only once. The second condition checks that the binding flag and the variance coincide. The (straightforward) adaption of the algorithm $\mathcal{S}$ of Figure 8.2.1 to take into account the fact that bounds can be inlined is presented in Figure 8.3.3. In fact, $\mathcal{S}$ is exactly $\mathcal{S}^I$ in which Inline is replaced by a predicate that is always false.

$$\mathcal{S}^I_\tau(n) \quad = \quad \mathsf{B}(n_1) \cdots \mathsf{B}(n_i)\ \tau(n)\Big(\mathsf{V}(n \cdot 1), \ldots, \mathsf{V}(n \cdot \mathsf{arity}(\tau(n)))\Big)$$

where n_1, ... n_i is one ordering of $(\longrightarrow\!\!\triangleright\ n)$ for $\xrightarrow{+}\!\!\circ$, lowest nodes first

$$\text{and } \mathsf{B}(n_k) \ \triangleq \ \begin{cases} \textit{nothing} \quad \text{if } \mathsf{Inline}(\tau, n_k) \\ \forall\,(\alpha_{n_k}\ \overset{\diamond}{\tau}(n_k)\ \mathcal{S}^I_\tau(n_k)) \quad \text{otherwise} \end{cases}$$

$$\text{and } \mathsf{V}(n \cdot k) \ \triangleq \ \begin{cases} \mathcal{S}^I_\tau(\langle n \cdot k \rangle) \quad \text{if } \mathsf{Inline}(\tau, \langle n \cdot k \rangle) \\ \alpha_{n \cdot k} \quad \text{otherwise} \end{cases}$$

Figure 8.3.3 – Translation from graphic to syntactic types with inlining

Interestingly, both the size of the resulting type and the complexity of the translation are linear in the size of the argument.

Lemma 8.3.1 *$\mathcal{S}^I(\tau)$ can be computed in linear time, and has linear size in the size of τ.*□

<ins>Proof</ins>: Only bounds used linearly are inlined. This ensures that no bound is inlined more than once, and the resulting syntactic type has a size linear in the size of the translated graphic type.

To show that the translation can be performed in linear time, let us first show that we can compute $\mathsf{Inline}(\tau, n)$ in linear time for a whole type τ. For this, we perform a depth-first traversal of τ. During the traversal, when encountering a node n (after having followed an edge $n' \xrightarrow{i}\!\!\circ\ n$), we mark n « non inlinable » if either:

- $\hat{n}$ is not n';
- n has already been encountered
- if $\mathsf{Var}_i(\tau(n'))$ does not match $\overset{\diamond}{\tau}(n)$

Then all nodes not marked « non-inlinable » can be inlined. Depth-first search has linear complexity, and each of the operation above can be done in constant time. Thus the entire operation can be done in linear time. The remainder of the proof is then the same as for Lemma 8.2.3.

8.3.3 Inlining monomorphic nodes

When presenting types to the user, inlining monomorphic bounds is almost mandatory. Indeed, they are only notational artifacts, and do not bring any useful information (§13.2 will give a formal meaning to this affirmation). Conversely, displaying them compromises the readability of the types.

An algorithm inlining such nodes can be obtained very easily from $\mathcal{S}^I$, by modifying the predicate Inline so that $\mathsf{Inline}(\tau, n)$ also holds for all nodes n monomorphic in τ. There are however two important consequences:

1. *$\mathcal{S}^I$ is no longer injective.*

   This is actually by design: similar types are mapped to the same syntactic type. Moreover the converse property also holds: two types mapped to the same syntactic type are similar. This justifies that this translation is still only a form of syntactic sugar, but this time up to similarity.

2. *The translation has exponential time worst-case complexity.*

   This is an unfortunate, but unavoidable consequence. The sharing of monomorphic nodes bring some conciseness to the display of graphs, which is lost when monomorphic bounds are inlined.

   Of course, this is not specific to ML^F graphic types. In syntactic ML^F, the normal form of a syntactic type can be exponentially bigger than the type itself, for exactly the same reason. Similarly, in ML, the textual representation of a type can be exponentially bigger than the internal representation of the type as a dag.

Part

II

Graphic constraints

9
Graphic constraints

Abstract

We extend graphic types to graphic constraints, which will be the basis for ML^F type inference. We first consider the basic constructs that would be needed to perform ML type inference with graphic types (§9.1). We generalize these to ML^F, and formally define ML^F constraints as a small extension of graphic types (§9.2). We isolate from those constraints the subset of ML constraints, and propose a ML type instance relation (§9.3). Finally we define how to translate a λ-term into a typing constraint (§9.4).

9.1 An informal presentation of graphic constraints

9.1.1 Our approach

We have shown in §7 how to perform unification on graphic ML^F types. Thus, we could theoretically perform ML^F type inference using the syntactic ML^F type inference algorithm (Le Botlan and Rémy 2003): for the most part, this algorithm is the adaptation of the ML type inference algorithm to the use of ML^F unification. However, this would involve repeatedly translating to and from graphic types; such an approach would be inelegant, and would make reasoning difficult.

Thus we propose an entirely graphic approach to type inference. Moreover, we do not limit ourselves to a type unification algorithm, but instead develop a constraint-based framework. The interest of using constraints to perform type inference is well-known (Pottier 2004; Pottier and Rémy 2005). In our case, we present in fact two very similar constraints systems, one for ML and one for ML^F. This allows us to easily compare the expressivity of both systems. Our approach can also easily be modified to accommodate other type systems based on graphic types (§17.2).

In order to be more gentle, we start by considering ML type inference, and explain what constructs are needed to encode ML typing problems into graphic constraints. The extension to MLF graphic constraints will then be straightforward. The remainder of this section is intendedly rather informal; formal definitions are given in the subsequent sections.

9.1.2 Graphic ML type inference without generalization

ML type inference is essentially based on first-order unification and type generalization. Let us focus on unification first. In order to type an application $a\ b$, a straightforward ML type inference algorithm typically proceeds as follows:

1. introduce two fresh type variables α and β;
2. recursively type a and b
3. unify the type found for a with $\alpha \to \beta$;
4. unify the type found for b with α (which has potentially been instantiated by the previous step)
5. return β (which has potentially been instantiated by the two previous steps).

Adapting this approach to graphic types is not really difficult once we add *existential nodes*, which are nodes only present to constrain other parts of graphic types.

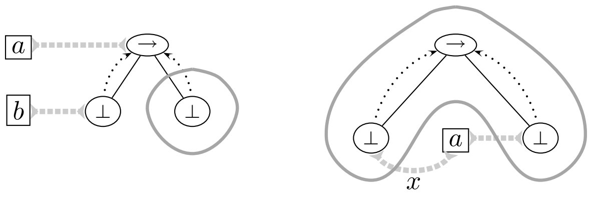


Figure 9.1.1 – Typing graphically an application or an abstraction

► **Example** Figure 9.1.1 shows possible graphic constraints, for the typing of an application $a\ b$ and for an abstraction $\lambda(x)\ a$.

Let us focus on the constraint for the application (on the left) first. We introduce a subconstraint for a, another one for b, and the graphic type for the type $\forall\alpha.\ \forall\beta.\ \alpha \to \beta$. Then we add unification edges (which have been introduced in §7.6 to represent generalized unification problems) to the subconstraints for a and b—thus requiring that the types of these expressions unify to $\alpha \to \beta$ and α respectively. The resulting type for the entire application is the node circled in blue, *i.e.* the codomain of the arrow, which represents β. All nodes but this last one can can be viewed as existential nodes, as they are not reachable from the result node through structure edges. Indeed, they exist merely for type inference purposes, and are not part of the result.[1]

Let us now consider the constraint for the abstraction $\lambda(x)\ a$. We introduce again the graphic type for $\forall\alpha.\ \forall\beta.\ \alpha \to \beta$. This time however, this type is the return type for the constraint. Then we constrain the type of a to be unified with β, and we constrain all the

[1]Of course, some sharing can occur, and some nodes will likely not be existential when the constraint is solved. This is typically the case for the node representing α, which will be shared with the one for β if a has a type instance of $\gamma \to \gamma$.

occurrences of x in a to be unified with α (thus the leftmost unification edge is actually a meta-edge, standing for possibly multiple edges). This time, only the subconstraint for a is existential.

9.1.3 (Graphic) type schemes and generalization

ML type inference cannot be done solely using unification. Indeed, when typing a construct let $x = a$ in a', the type found for a must be *generalized* when a' is typed, and each occurrence of x in a' is typed with a fresh instance of the type for a.

In ML, this is partly done by distinguishing between types and type schemes, since generalization can be seen as transforming a type into a type scheme. We use the same mechanism in our graphic constraints, and introduce a type constructor $\mathcal{G}$ for the purpose of representing type schemes. Nodes labelled by $\mathcal{G}$, which we call *gen nodes* for reasons explained below, are also used to delimit scopes and to bind variables.

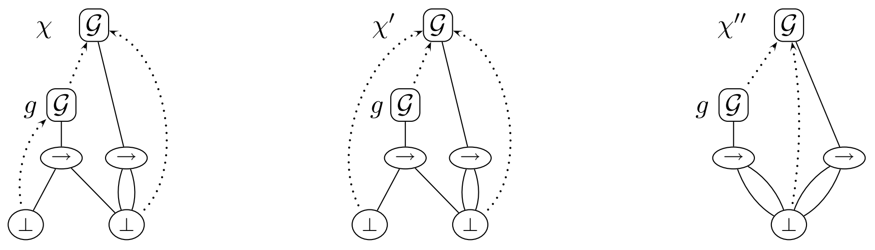


Figure 9.1.2 – Graphic type schemes

▶ **Example** Consider the first three constraints of Figure 9.1.2. Each constraint contains two gen nodes, the root $\langle\epsilon\rangle$ and the node g, bound at $\langle\epsilon\rangle$. We extend the syntax of paths to allow named nodes such as g. For example, in all three constraints the rightmost lowermost bottom node can be designated by either $\langle g12\rangle$, $\langle 11\rangle$ or $\langle 12\rangle$.

The node g of the constraint χ represents the type scheme $\forall\,(\alpha)\;\alpha \to \beta$: the node $\langle g11\rangle$ representing α is bound at g, while the node $\langle g12\rangle$ representing β is bound above g, and is thus the equivalent of a free variable for g. By contrast, in the constraint χ', both variables are bound above g; hence g represents the type scheme $\alpha \to \beta$, which is monomorphic in the context of g (and has thus no instances). The root node represents the same type scheme $\forall\,(\beta)\;\beta \to \beta$ in all three constraints. Notice the binding edge between g and the root, which is used to materialize the inclusion of scopes: g is in the scope of the root gen node.

9.1.3.1 Why gen nodes?

It might seem strange to introduce a special node for type schemes, as we have strongly argued against the addition of special nodes for quantifiers (§3.3). We thus briefly motivate our choice below.

First, we do not introduce a node for each quantifier: for example, the constraint χ of Figure 9.1.3 represents the type scheme $\forall\,(\alpha)\;\forall\,(\beta)\;\alpha \to \beta$. Similarly, we do not remove

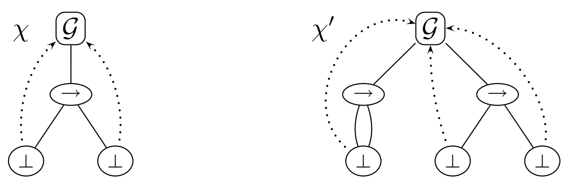


Figure 9.1.3 – More advanced examples of graphic types schemes

gen nodes when they become monomorphic, as examplified by g in the constraints χ' and χ'' of Figure 9.1.2. Thus the structure of the constraint will not change when variables are instantiated, unlike what would have been needed for $\forall$ constructors.

Secondly, and more importantly, gen nodes do not only represent type schemes, but also *generalization levels* (hence their names and the letter $\mathcal{G}$). Generalization levels are very similar to the notion of ranks used in efficient ML type inference algorithms (Pottier and Rémy 2005; Kuan and MacQueen 2007; Rémy 1992), a fact which will become more apparent once we present the translation from λ-terms to typing constraints.

Moreover, gen nodes are a mechanism to introduce *schemes*, not just a single scheme. Indeed, by adjusting the number of structural successors of the constructor $\mathcal{G}$, we can introduce more than one type scheme. Those schemes are essentially independent—as long as they do not share nodes—and are only linked by the fact that they are generalized at the same level. Such a construction can be useful for example to type a construct let $x = \ldots$ and $y = \ldots$ in $\ldots$.

► **Example** The root gen node of the constraint χ' of Figure 9.1.3 introduces two type schemes, $\forall\alpha.\ \alpha \rightarrow \alpha$ and $\forall\alpha.\ \forall\beta.\ \alpha \rightarrow \beta$ respectively.

Thus, in the following, we call *type scheme* a node of the form $\langle g \cdot i \rangle$. We let the letter s range over such nodes.

9.1.3.2 Merging variables

On graphic constraints, merging is restricted to variables bound at the same gen node: we can only merge variables of the same scope. If this is not the case, at least one of the variable must be raised beforehand. Notice the similarity with graphic types, in which nodes must be binding-congruent.

► **Example** Consider the nodes $\langle g11 \rangle$ and $\langle g12 \rangle$ in Figure 9.1.2. In χ, they cannot be merged. However, the node $\langle g11 \rangle$ can be raised, resulting in the constraint χ'. The merging is now possible, and results in the constraint χ''.

In graphic constraints, raising results in the extrusion of the polymorphism to the enclosing generalization level. Readers familiar with rank-based ML type inference can recognize the similarity between such a raising and adjusting the ranks of two variables about to be unified.

9.1.3.3 Binding nodes

In Figure 9.1.1, we have left some nodes (in particular the subconstraints for the expressions a and b) unbound. However, this was only to avoid complicating the explanations at this stage. Since variables need to be raised to a common binder before they are merged, the binding edges in a graphic constraint must verify at least two properties:

1. the binding edges must form a tree;
2. a node on which a variable is transitively bound must itself be bound.

Notice the similarity between the first point and the equivalent one in graphic types.

9.1.4 Type instantiation

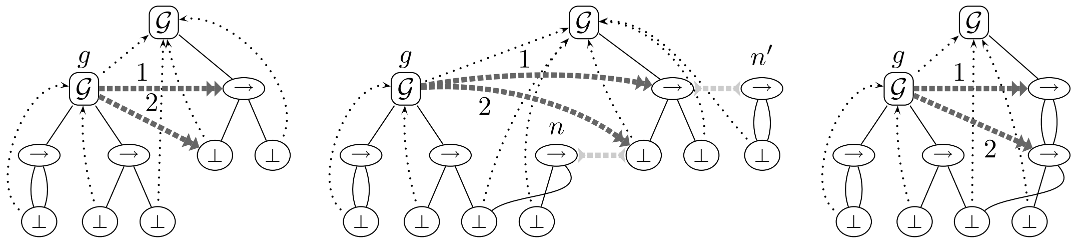


Figure 9.1.4 – Instantiation edges

Once we have introduced type schemes, we can add *instantiation constraints*, which require a type to be an instance of a type scheme. As we mentioned, this is in particular needed to type let constructs. We represent such an instantiation constraint by an oriented red edge ⇢ linking the type scheme to the constrained type. On gen nodes that introduce more than one type scheme, we add on the edge an integer i that indicates which scheme is meant.

▶ **Example** Consider the first constraint of Figure 9.1.4. The root gen node introduces the type scheme $\forall\alpha.\ \forall\beta.\ \alpha \to \beta$. The gen node g introduces the two type schemes $\forall\alpha.\ \alpha \to \alpha$ and $\forall\alpha.\ \alpha \to \gamma$, where γ is quantified at the level of the root.

The topmost instantiation edge (labelled by 1) constrains the node $\langle 1\rangle$ to be an instance of $\forall\alpha.\ \alpha \to \alpha$, while the lowermost one constrains $\langle 11\rangle$ to be an instance of $\forall\alpha.\ \alpha \to \gamma$. In the second constraint, we have created a fresh instance of each scheme, and added the unification edges corresponding to the instantiation ones. Notice that the node $\langle n2\rangle$ is shared with $\langle g22\rangle$. Indeed, $\langle g22\rangle$ cannot be generalized when we take an instance of $\langle g2\rangle$, as it is bound above g.

Solving the unification edges of the second constraint results in the third constraint of the figure, in which the root gen node now represents $\forall\alpha.\ \forall\gamma.\ (\alpha \to \gamma) \to (\alpha \to \gamma)$. In this graph, the constraints represented by the instantiation edgess are solved. Notice that we do not remove the edges: by instantiating g more, the edges could become unsolved again.

9.2 Graphic constraints as an extension of graphic types

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

There are some differences between the constraints we have presented informally in the examples of the previous section and the ones we define below. There are three kinds of reasons for these changes:

1. generalizing to $\mathsf{ML^F}$ constraints;
2. seeing graphic constraints as an extension of graphic types;
3. simplifying the reasoning on constraints.

In particular, following point 2, we bind all the nodes (except the root), exactly as in graphic types. The other most important difference, due to both points 2 and 3, is the fact that we stratify constraints. That is, we do not allow a constructor $\mathcal{G}$ to appear under a constructor such as $\rightarrow$ (such a construction is not useful to us anyway). Thus, constraints are split, with a «constraint» part on the top, and a «type» part on the bottom. Thanks to this separation, we will be able to reason about the type part exactly as on graphic types.

9.2.1 A formal definition of constraints

We extend the algebra of type constructors Σ by adding a family of symbols $\mathcal{G}_k$, for every integer k. We also introduce two sorts Gen and Type . A symbol $\mathcal{G}_k$ has signature

$$\mathcal{G}_k \ : \ \mathsf{Type}^k \Rightarrow \mathsf{Gen}$$

while all others constructors have signature

$$C \ : \ \mathsf{Type}^{\mathsf{arity}(C)} \Rightarrow \mathsf{Type}$$

Thus gen nodes cannot appear under nodes of sort Type, which are called *type nodes.* We let the letter g range over gen nodes. Since we can always deduce which symbol $\mathcal{G}_k$ is meant by looking at the number of its structural successors, we write all symbols $\mathcal{G}_k$ as $\mathcal{G}$.

The formal definition of graphic constraints is given below; we simultaneously define the nodes we call existential, and introduce a new definition of admissible unification problems. For brevity, the definitions reuse some notations and terminology of graphic types.

Definition 9.2.1 (*Graphic constraints)* A graphic constraint χ is a graph built from structure, binding, unification and instantiation edges, and which also associates to a node n a binding flag $\overset{\diamond}{\chi}(n)$. We write $\check{\chi}$ (*resp.* $\hat{\chi}$) the restriction of χ to structure edges (*resp.* binding edges). The graph χ must moreover verify the following properties:

1. for any node n of χ, the graph $\check{\chi}/n$ is a well-formed acyclic term-graph, in which constructors are well-sorted;
2. $\hat{\chi}$ forms an upside-down tree of domain $\mathsf{dom}(\check{\chi})$ rooted at $\langle\epsilon\rangle$;
3. $\mathsf{dom}(\overset{\diamond}{\chi}) = \mathsf{dom}(\chi) \setminus \{\langle\epsilon\rangle\}$
4. the root node of χ is a gen node;

5. all gen nodes are flexibly bound;
6. existential nodes (Definition 9.2.2) are bound on gen nodes;
7. unification edges link two type nodes;
8. unification edges are admissible (Definition 9.2.3);
9. an instantiation edge $g \overset{i}{\dashrightarrow\!\!\!\!\!\twoheadrightarrow} d$ links a gen node g of arity at least i to a type node;
10. any node destination of an instantiation edge is bound on a gen node;
11. χ is well-dominated, *i.e.* all mixed paths between the root and a node n contains $\hat{n}$.

Constraints are moreover quotiented by the addition and removal of unification edges whose two extremities are the same node. ■

Definition 9.2.2 (*Existential nodes)* A node n of a constraint χ is *existential* if n and $\hat{\chi}(n)$ are in two different partitions of $\check{\chi}$ for $\overset{*}{-\!\!\circ}$. A node different from the root is *purely existential* if there does not exist a node n' such that $n' \multimap n \in \chi$. ■

Definition 9.2.3 (*Admissible unification edges in constraints)* A unification edge $n_1 \rightarrowtail\!\!\!\leftarrowtail n_2$ is admissible on a constraint χ if either n_1 or n_2 is bound on a gen node. ■

Let us comment those definitions. Disregarding unification and instantiation edges, conditions 1, 2 and 3 ensure that the only difference between a graphic constraint and a graphic type is the fact that some nodes are not structurally reachable from the root. However, given the shape of $\hat{\chi}$, all nodes are reachable by a binding path, hence by a mixed path.

Condition 5 reflects the fact that the binding edges of gen nodes should not alter permissions. Indeed, those edges are only present to allow the extrusion of polymorphism from one generalization level to the enclosing one. We could instead have introduced a third kind of edges, «neutral» w.r.t. permissions, but they would have behaved the same way as flexible edges.

Condition 6 is technical, and will be discussed in §11.1.1.

Condition 7 forbids unification edges between two gen nodes. Such an edge would be unsolvable: gen nodes encode in particular the shape of the constraint, and this shape will remain invariant through instance. Moreover, while we could give a meaning to the merging of two generalization scopes, this meaning would be very different from the one of merging two type nodes.

Condition 8 limits unification edges to ones solvable in a principal way. We do not reuse the definition of admissibility on graphic types (Definition 7.2.1) because it is not entirely immediate to generalize it to graphic constraints (since $\check{\chi}$ is not necessarily a rooted graph, the relation $\twoheadrightarrow\!\!\!\multimap$ is not always defined on χ). Moreover, even after having been properly extended, this definition would still be too weak for the typing constraints we have in mind. The correctness of our new definition will be proven in §11.2.

Condition 9 requires instantiation edges to be well-sorted. That is, they must constrain a type to be an instance of a type scheme.

Condition 10 is also technical. Without it, our system would not be stable by the propagation operation defined in §10.3.

Condition 11 is the same as in graphic types, and ensures that the type part of constraints is correctly scoped.

Convention In the examples, we name the nodes to which we need to refer and that are not reachable from the root by structure edges. We extend the syntax of paths using those names, *e.g.* $\langle g11\rangle$. In drawings we never draw unification edges whose two extremities are the same node. (In fact, we often suppose that constraints do not contain such edges.) We also always omit the «1» on top of an instantiation edge $g \overset{1}{\twoheadrightarrow} d$ if g has a single structural successor.

9.2.1.1 Existential nodes

Our definition of existential nodes is mostly technical, and might seem a bit strange—in fact, purely existential nodes are more natural to comprehend. In essence, a node n is existential w.r.t. the subgraph it resides in. We can reason about a partition of $\check{\chi}$ for $\xrightarrow{*}$ as if it were a graphic type, except on existential nodes which form a frontier between the type part and the constraint part of constraints.

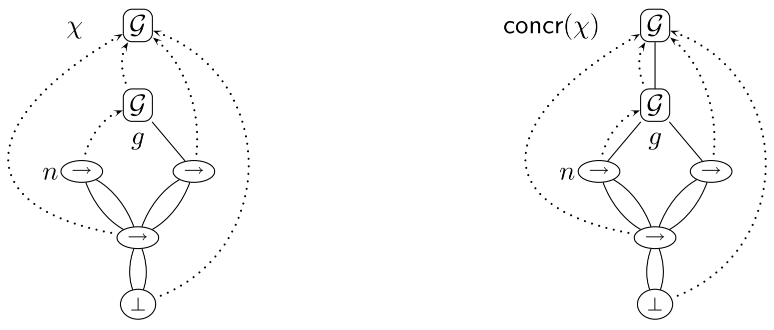


Figure 9.2.1 – Existential nodes

▶ **Example** Consider the constraint χ of Figure 9.2.1. The nodes n and g are both purely existential and existential, while $\langle g1\rangle$, $\langle n1\rangle$ and $\langle n11\rangle$ are only existential.

By construction, purely existential nodes are existential, but the converse does not hold. By well-sortedness, all gen nodes are purely existential. Notice that existential nodes, pure or not, can be of any sort. However, gen existential nodes are above type existential nodes, by condition 6. In the following we call *existential edge* a binding edge $n \longrightarrow n'$ when n is an existential node.

9.2.2 Properties of constraints

The well-formedness of constraints ensures some important invariants. We give three of them below.

Property 9.2.4 *All gen nodes have green or inert permissions. All existential nodes have green, orange or inert (hence non-red) permissions.* □

Proof: By well-sortedness, the binding tree above a gen node g is of the form $\langle\epsilon\rangle \mathrel{\lhd\!\!\relbar\!\!\relbar} g_0 \ldots \mathrel{\lhd\!\!\relbar\!\!\relbar} g$. The conclusion is then immediate by conditions 5 and 6 of Definition 9.2.1.

Lemma 9.2.5 *Any node reachable (by a mixed path) from a type node is a type node.* $\square$

Proof: Let n be a type node. Consider n' such that $n \overset{P}{\mathrel{\lhd\!\!\relbar\!\!\circ}} n'$. The proof is by induction on P.

- ▷ If P is the empty path: then $n' = n$ which is indeed a type node.
- ▷ If P is $n \mathrel{\relbar\!\!\relbar\!\!\circ} n'' \overset{P'}{\mathrel{\lhd\!\!\relbar\!\!\circ}} n'$: n'' cannot be a gen node, by well-sortedness. Hence n'' is a type node. The conclusion is by induction hypothesis.
- ▷ If P is $n \mathrel{\lhd\!\!\relbar\!\!\relbar} n'' \overset{P'}{\mathrel{\lhd\!\!\relbar\!\!\circ}} n'$: n is a type node, hence no gen node is bound on it. Consequently, n'' is a type node. The conclusion is again by induction hypothesis.

As a corollary of this second result, the existential edges of a constraint do not change the set of paths under a type node. This shows the small amount of difference between the type part of a constraint and a graphic type.

Corollary 9.2.6 *Consider a type node n of a constraint χ. No mixed path starting from n contains an existential edge.* $\square$

Proof: Consider a mixed path P starting from n, and suppose that it contains an existential edge $n' \mathrel{\relbar\!\!\relbar\!\!\rhd} n$. Then n is a gen node by condition 6 of Definition 9.2.1. This contradicts Lemma 9.2.5.

Finally, even though we only requested the acyclicity of the structural subgraphs of a graphic constraint, the superposition of the structure and of the binding tree is acyclic, as in graphic types.

Lemma 9.2.7 *Given a constraint χ, the relation $\mathrel{\lhd\!\!\relbar\!\!\circ}_\chi$ is acyclic.* $\square$

Proof: By contradiction, let n be a node of χ in a cycle. Let P be a non-empty mixed path from n to n. By condition 1 of Definition 9.2.1, at least one of the edges of P is an inverse binding edge $n' \mathrel{\lhd\!\!\relbar\!\!\relbar} n''$; we call this edge e. Without loss of generality we can suppose it is an existential one, as otherwise we could replace it by some structure edges, and we would have a cycle for $\mathrel{\relbar\!\!\relbar\!\!\circ}$. By condition 6, n' is a gen node. By well-sortedness, all the edges before e in P are binding edges between gen nodes. In particular, n is a gen node. The edges after e in P cannot all be binding edges: $\hat{\chi}$ would be cyclic, contradicting condition 2. Thus there exists a structure edge $n''' \mathrel{\relbar\!\!\relbar\!\!\circ} n''''$ in P, and n'''' is a type node. Together with Lemma 9.2.5, this means that n is a type node: contradiction.

9.2.3 Instance on graphic constraints

The instance operations on graphic constraints are the same as on graphic types, except that the transformations are only allowed on types nodes. We however do not duplicate all the definitions of §4 and §5, as there would be little point in doing so. Instead, a definition on a type τ is implicitly changed to a definition on a constraint χ by replacing $\breve{\tau}$ (*resp.* $\hat{\tau}$, $\mathring{\hat{\tau}}$) by $\breve{\chi}$ (*resp.* $\hat{\chi}$, $\mathring{\hat{\chi}}$). This is always meaningful: the instance operators are only concerned by the subgraphs under the nodes they transform and the permissions of the nodes they operate on. Moreover, the fact that we only operate on type nodes ensures (by Corollary 9.2.6) that existential edges are unimportant. We will formally justify this point in §9.2.5.

We usually reuse the symbol $\sqsubseteq$ to mean instance on constraints, as the differences with graphic types are quite small. In the rare cases where the distinction is important, we write ${}^{\mathrm{t}}\!\sqsubseteq$ and ${}^{\mathrm{c}}\!\sqsubseteq$ the instance relations on types and constraints respectively.

Definition 9.2.8 (*Instance on constraints*) A constraint χ' is an atomic instance of a constraint χ if:

- $\chi \sqsubseteq_1 \chi'$ holds according to the definitions of §5.3;
- constraints edges are preserved (*i.e.* if there is an instantiation or unification edge between two nodes n_1 and n_2 in χ, the same edge is present between the corresponding nodes in χ');
-
  - if $\chi' = \mathsf{Raise}(n)(\chi)$, n is not a gen node;
  - if $\chi' = \mathsf{Merge}(n_1, n_2)(\chi)$, n_1 and n_2 are not gen nodes;
  - if $\chi' = \mathsf{Weaken}(n)(\chi)$, n is not a gen node. ■

In particular, if $\chi \sqsubseteq \chi'$, the shape of the gen nodes is exactly the same in χ and χ'.

Property 9.2.9 *Instance preserves the well-formedness of constraints.* □

Proof: Let us first notice that the property «being bound on a gen node» is stable by instance (**1**). This is immediate for all the operations but raising; for raising, we simply conclude by the fact that the binder of a gen node is a gen node.

Let χ be a constraint, χ' derived from χ by an instance operation. We must show that χ' is a well-formed constraint. Most of the conditions of Definition 9.2.1 are immediate and we justify the others below.

▷ Condition 6: for the nodes that were existential in χ, the result is by (1) and well-formedness of χ. Otherwise, consider a node n existential in χ' but not in χ. Existential nodes can only appear through raising, and we had $n \longrightarrow n' \longrightarrow n''$ in χ with n' existential. Thus n'', which is the binder of n in χ', is a gen node, and the result holds.

▷ Condition 8: no new unification edge appears by instance. The result is thus immediate by (1), the well-formedness of χ and the definition of admissibility.

▷ Condition 10: no instantiation edge appears by instance, the conclusion is thus immediate by (1).

▷ Condition 11: well-domination is trivially preserved by all operations but raising. Suppose thus that $\chi' = \mathsf{Raise}(n)(\chi)$. Let n' be a node of χ', and P' a mixed path from $\langle\epsilon\rangle$ to n' in χ'. We must show that $\hat{\chi}'(n')$ is in P'.

- *Case* $\hat{\chi}'(n) \lhd\!\!\longleftarrow n$ is not in P': then P' is a valid path of χ.
  - *Case* n' is not n: P' contains $\hat{\chi}(n')$ by well-domination, and $\hat{\chi}'(n') = \hat{\chi}(n)$ by the subcase hypothesis. Thus the result holds.
  - *Case* n' is n: by well-domination of χ, P' contains $\hat{\chi}(\hat{\chi}(n'))$, which is $\hat{\chi}'(n')$ by construction.
- *Case* $\hat{\chi}'(n) \lhd\!\!\longleftarrow n$ is in P' (**2**): we replace this edge by $\hat{\chi}(\hat{\chi}(n)) \lhd\!\!\longleftarrow \hat{\chi}(n) \lhd\!\!\longleftarrow n$. This gives us a valid path P of χ.
  - *Case* n' is not bound on $\hat{\chi}(n)$ in χ: by this hypothesis $n \neq n'$; hence $\hat{\chi}'(n') = \hat{\chi}(n')$, which is contained in P by well-domination. Moreover $\hat{\chi}(n)$ is the only node of P that is not in P'. Hence $\hat{\chi}'(n')$ is in P'.
  - *Case* n' is n: this case is immediate by (2) and the subcase hypothesis.
  - *Case* n' is bound on $\hat{\chi}(n)$ in χ and is not n: by (2) and this subcase we have $n \overset{+}{\lhd\!\!-\!\!\circ} n' \in \chi$, hence $n \overset{+}{\multimap} n'$ by Corollary 9.2.6 (since n is raised, it is a type node). This contradicts the fact that n was raisable in χ.

Permissions in drawings Permissions are defined exactly the same way in types and constraints (as all the instance relation). In examples of graphic constraints, we however do not color nodes according to their permissions. Indeed, we rarely use rigid edges, and most nodes would be green or white. Instead, this leaves us the possibility to use colors to highlight some important nodes.

In the few examples where permissions are shown, we only color type nodes: Property 9.2.4 ensures that gen nodes have green or inert nodes. Moreover they are not really concerned with permissions since they cannot be transformed.

9.2.4 Transforming constraints

We introduce two operators to transform constraints, beyond instantiation.

Definition 9.2.10 (*Restriction*) Let χ be a constraint and N a subset of its nodes. The *restriction* of χ to N, written $\chi \restriction N$, is the subgraph composed of all the nodes of N and all edges between two nodes of N. ■

Definition 9.2.11 (*Constraint projection*) The *constraint projection* $\mathsf{cproj}(\chi)$ of χ is the constraint obtained by removing all unification and instantiation edges from χ. ■

9.2.5 From graphic constraints to graphic types

In this section, we go one step further in seeing graphic constraints as graphic types, and map the former into the latter. To do this, we remove all constraint edges, and add some new structure edges so that all existential nodes become structurally reachable.

Definition 9.2.12 (*Concretization*) A *concretization* of a constraint χ is a graphic type τ that differs from $\mathsf{cproj}(\chi)$ by the addition of some structure edges of the form $g \multimap n$, where n is bound on a gen node. Such an edge is called a *virtual edge.* ■

Concretization is purely technical. In particular, it does not respect well-sortedness, and changes the arity of gen nodes. Notice also that the definition above is highly non-deterministic: it specifies neither which virtual edges should be added, nor the order in which they should appear as the structural ancestor of the gen node from which they originate. This is indeed unimportant for the use we have in mind.

Let us first show that concretizations exist. Let concr be defined as

$$\mathsf{concr}(\chi) \quad \triangleq \quad \mathsf{cproj}(\chi) \cup \{\hat{\chi}(n) \multimap n \mid n \text{ is purely existential in } \chi\}$$

This definition is still not entirely deterministic, as we do not specify the order in which the virtual edges are added under gen nodes. Any arbitrary order is acceptable.

Lemma 9.2.13 *Given a constraint χ, $\mathsf{concr}(\chi)$ is a concretization of χ.* □

Proof: All existential nodes are bound on gen nodes. Hence it suffices to show that $\mathsf{concr}(\chi)$ it is a well-formed graphic type. The correctness of $\hat{\tau}$ and $\overset{\diamond}{\tau}$ is by conditions 2 and 3 of Definition 9.2.1. The correctness of $\breve{\chi}$ is by condition 1 and the structure edges we add:

- ▷ $\breve{\tau}$ is acyclic: immediate consequence of Lemma 9.2.7 and of the fact that we only add structure edges that follow binding edges.
- ▷ all the nodes of χ are structurally reachable by the root: we show a slightly stronger result: if there exists a mixed path from n to n' in χ, there exists a structure path from n to n' in τ. The proof is by induction on the length of the longest path P from n to n' in χ, which exists as $\leftarrow\!\!\multimap_\chi$ is acyclic (Lemma 9.2.7).
  - ∘ If P is of a length 0: we have $n = n'$ and the result is proven.
  - ∘ If P is of length 1 and $n \multimap n' \in \chi$: the result is immediate as $n \multimap n'$ is in τ.
  - ∘ If P is $n \leftarrow n'$ and $n \not\multimap n'$: let us justify that n' is purely existential (**1**). Otherwise there would exists n'' such that $n'' \multimap n'$. Then $n \overset{*}{\leftarrow\!\!\multimap} n''$ would hold by well-domination, and we would have a longer path between n and n', as n'' cannot be n (since $n \not\multimap n'$). By (1), the edge $n \multimap n'$ is in τ, which is the desired result.
  - ∘ If P is of length $k \geq 2$: we split P in two and conclude by induction hypothesis applied to each part,
- ▷ constructors respect arities: only the arities of the binders of the purely existential nodes change. By condition 6, they are gen nodes and the $\mathcal{G}$ constructor can have any arity.

Finally, let us show that τ is well-dominated. Given a node n of τ and a mixed path P from the root to n in τ, we can change it into a mixed path P' of χ by replacing the newly introduced structure edges by the corresponding binding edges. Then we have exactly the same nodes in P and P' by construction of τ. By condition 11. we have $\hat{\chi}(n)$ in P', and this node is also $\hat{\tau}n$, hence the conclusion.

▶ **Example** A concretization of the constraint χ of Figure 9.2.1 is the type $\mathsf{concr}(\chi)$ given in the same figure.

Let us briefly justify why we do not define concretization as a function, for example as the image of a constraint by concr. While this is technically possible, it would be of limited use: some problems arise because concretization does not commute with instance. For example, purely existential nodes can disappear when they are merged with non purely existential ones. Thus, if $\chi \sqsubseteq^M \chi'$, we do not necessarily have $\mathsf{concr}(\chi) \sqsubseteq^M \mathsf{concr}(\chi')$. Defining concretization as a relation is sufficient to sidestep this issue entirely, as will be shown by Lemma 9.2.15.

Importantly, virtual edges do not change the graph under type nodes. This results holds in particular for the structure paths under such a type node n.

Property 9.2.14 *Consider a constraint χ, and τ a concretization of χ. Let n be a type node of χ (or τ). The mixed paths under n are exactly the same in χ and τ.* □

Proof: Immediate consequence of the definition of concretization and Lemma 9.2.5.

We can now show that the instance operations on type nodes are really the same on graphic constraints and on graphic types.

Lemma 9.2.15 *Let χ be a constraint, τ a concretization of χ, and o an instance operation on a type node of χ. Then o can be applied to χ and $\chi \;{}^{\mathrm{c}}\!\sqsubseteq_1 o(\chi)$ holds if and only if o can be applied to τ and $\tau \;{}^{\mathrm{t}}\!\sqsubseteq_1 o(\tau)$ holds. Moreover $o(\tau)$ is a concretization of $o(\chi)$.* □

Proof: We first prove that o can be applied to τ iff it can be applied to χ.

- ▷ *Case* $o = \mathsf{Graft}(\tau', n)$ or $\mathsf{Weaken}(n)$: the result is immediate, as those two operations are not concerned with the new structure edges.
- ▷ *Case* $o = \mathsf{Raise}(n)$: the binding tree and permissions are the same in τ and χ. We thus focus on the fact that n is raisable.
  - ◦ raisable in τ implies raisable in χ: there are less structure edges in χ than in τ, which implies the result.
  - ◦ raisable in χ implies raisable in τ: by contradiction, suppose that n' is bound on $\hat{n}$ and $n \xrightarrow{+}\!\circ\, n' \in \tau$. By Property 9.2.14, this path is also is χ, as n is a type node. This contradicts the fact that n is raisable in χ.
- ▷ *Case* $o = \mathsf{Merge}(n_1, n_2)$: again, the permissions and binding trees are the same in χ and τ, so it suffices to show that n_1 and n_2 are congruent. Let us suppose that n_1 and n_2 are congruent in χ. By hypothesis, n_1 and n_2 are type nodes. By Property 9.2.14, no structure edge is added under them in τ and they are still congruent. In the other direction the proof is symmetric, as no edge under n_1 and n_2 is removed from τ to χ.

Let us now suppose that o applies to χ, and prove that $o(\tau)$ is a concretization of $o(\chi)$. Since we already know that $o(\tau)$ is a graphic type, and by definition of $o(\chi)$ and $o(\tau)$, it is sufficient to prove that the nodes of $o(\tau)$ on which a virtual edge arrives are bound on a gen node. («Being a virtual edge» has a well-defined meaning in $o(\tau)$: since we do not allow the merging of two gen nodes, a virtual edge is never merged with another edge.) Let n be such a node. Necessarily there exists a node n' of τ and χ such that $n' \subseteq n$, and a virtual edge arrives on n' in τ. Since τ is a concretization of χ, n' is bound on a gen node in χ. Then n is bound on a gen node in $o(\chi)$, as this property is preserved by instance.

This result generalizes immediately to an entire instance derivation. Thus we can freely reuse all the instance-related results established previously on graphic types.

9.2.6 Interiors

We introduce a last technical definition, which will be used in the next chapters.

Definition 9.2.16 (*Interiors)* The *constraint interior* of a node n, written $\mathcal{I}^c(n)$, is the set $(n \overset{*}{\lhd\!\!-\!\!-})$ of all the nodes transitively bound to n. The *structural interior*, written $\mathcal{I}^s(n)$, is the restriction of the constraint interior to the nodes structurally reachable from n, *i.e.* $\mathcal{I}^c(n) \cap (n \overset{*}{\multimap})$.

The *structural frontier* of a node n, written $\mathcal{F}^s(n)$, is the set $(\mathcal{I}^s(n) \multimap) \setminus \mathcal{I}^s(n)$ of the nodes outside $\mathcal{I}^s(n)$ with a structural immediate predecessor inside $\mathcal{I}^s(n)$. ■

We write $\mathcal{I}^c_\chi(n)$, $\mathcal{I}^s_\chi(n)$ and $\mathcal{F}^s_\chi(n)$ when there is an ambiguity on χ. Notice that $n \in \mathcal{I}^s(n)$ and $n \in \mathcal{I}^c(n)$. Moreover, $\mathcal{I}^s(n)$ is reduced to $\{n\}$ when all the children of n are bound strictly above n. If n is a type node, $\mathcal{I}^s(n)$ and $\mathcal{I}^c(n)$ coincide, as there is no existential node bound on n.

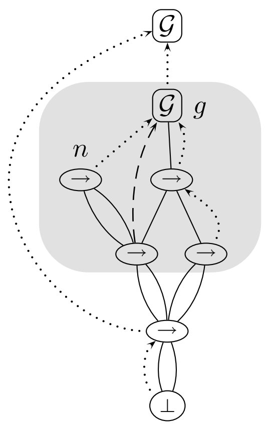


Figure 9.2.2 – Examples of interiors

▶ **Example** Consider the constraint of Figure 9.2.2. The constraint interior of g is the set of nodes highlighted in green, *i.e.* $\langle g\rangle$, $\langle g1\rangle$, $\langle g11\rangle$, $\langle g12\rangle$ and n. Its structural interior is composed of all those nodes but the purely existential node n. The structural frontier of g is the node $\langle g111\rangle$. The constraint and structural interiors of the node $\langle g1\rangle$ are composed of $\langle g1\rangle$ itself and $\langle g12\rangle$, as $\langle g11\rangle$ is bound above $\langle g1\rangle$ (and is in the structural frontier of $\langle g1\rangle$).

9.3 ML$^{\mathsf{F}}$ and ML constraints

Consider a constraint χ. We call *inner quantification* the fact that a type node can be bound on another type node. From now on, we distinguish ML$^{\mathsf{F}}$ constraints (that use the full range of ML$^{\mathsf{F}}$ graphic types), from ML constraints in which types cannot use inner or rigid quantification.

Definition 9.3.1 (*ML constraints)* A graphic constraint is an *ML constraint* if all the type nodes of the constraint are flexibly bound, and on a gen node. ■

Interestingly, while the instance relation on graphic ML constraints corresponds exactly to the restriction of the $\mathsf{ML^F}$ instance relation to ML constraints[2], this property does not hold for atomic instance steps. Indeed, unlike in $\mathsf{ML^F}$, we cannot graft closed types: the resulting constraint would not be an ML one. Thus we introduce a specific grafting operation, reminiscent of the one we defined for System F in §3.3.3.

Definition 9.3.2 (*ML grafting)* The ML grafting of a term-graph g at a bottom node n of an ML constraint χ is the constraint χ' which is χ everywhere, except under n where it is g, with all the nodes of g being bound on $\hat{n}$. ■

By construction of ML constraints, $\hat{n}$ is a gen node, and the resulting constraint is indeed an ML one.

Definition 9.3.3 (*ML instance)* The ML instance relation ${}^{\mathsf{ML}}\!\sqsubseteq$ is the relation

$$ {}^{\mathsf{ML}}\!\sqsubseteq_1^G \odot {}^{\mathsf{ML}}\!\sqsubseteq_1^R \odot {}^{\mathsf{ML}}\!\sqsubseteq_1^M $$

with ${}^{\mathsf{ML}}\!\sqsubseteq_1^X$ being the restriction of $\sqsubseteq_1^X$ to ML constraints for $X \in \{M, R\}$, and $\chi \; {}^{\mathsf{ML}}\!\sqsubseteq_1^G \; \chi'$ holding if χ' is an ML grafting of χ. ■

Notice that there is a raising operation in the ML instance relation, which is used to extrude polymorphism from one gen node to its binder.

Likewise we can define an ML similarity relation. This time, the definition is simpler, as the atomic operations are exactly the same in ML and $\mathsf{ML^F}$.

Definition 9.3.4 (*ML similarity)* The reversible ML instance relation ${}^{\mathsf{ML}}\!\sqsubseteq^{rm}$ is the relation

$$ {}^{\mathsf{ML}}\!\sqsubseteq^{rm} \;\triangleq\; \sqsubseteq^{rmw} \cap {}^{\mathsf{ML}}\!\sqsubseteq \;=\; {}^{\mathsf{ML}}\!\sqsubseteq_1^{r} \odot {}^{\mathsf{ML}}\!\sqsubseteq_1^{m} $$

where ${}^{\mathsf{ML}}\!\sqsubseteq_1^x \;\triangleq\; \sqsubseteq^x \cap {}^{\mathsf{ML}}\!\sqsubseteq$ for $x \in \{r, m\}$. The ML modulo similarity relation is the relation

$$ {}^{\mathsf{ML}}\!\sqsubseteq^{\approx} \;\triangleq\; {}^{\mathsf{ML}}\!\sqsubseteq \odot {}^{\mathsf{ML}}\!\sqsupseteq^{rm} $$ ■

It is immediate that ${}^{\mathsf{ML}}\!\sqsubseteq$ is a subrelation of $\sqsubseteq$: an ML grafting can be simulated using an $\mathsf{ML^F}$ grafting and some raisings. The same result holds for ML similarity.

Property 9.3.5 *Consider two ML constraints χ and χ'. If $\chi \; {}^{\mathsf{ML}}\!\sqsubseteq \; \chi'$ (*resp. *$\chi \; {}^{\mathsf{ML}}\!\sqsubseteq^{\approx} \; \chi'$), then $\chi \sqsubseteq \chi'$ (*resp. *$\chi \sqsubseteq^{\approx} \chi'$).* □

Proof: The result is by induction on the ML derivation. All atomic operations but ${}^{\mathsf{ML}}\!\sqsubseteq_1^G$ are in both the ML and $\mathsf{ML^F}$ relations. For ML grafting, suppose the grafted term-graph is g, and that the grafting occurs at node n. Let τ be the $\mathsf{ML^F}$ pre-type obtained by binding flexibly all the nodes of g at the root; this pre-type is trivially well-dominated, hence it is a type. Since χ is an ML constraint, n is green, and we can graft τ as an $\mathsf{ML^F}$ operation; we call χ_g the result. The constraint χ' is obtained by multi-raising n. All the nodes bound on n in χ_g are inert or green; thus $\chi \sqsubseteq_1^G \chi_g \sqsubseteq^R \chi'$ holds, which is the desired result.

[2]This result is proven at the end of this section.

In fact, the inverse property holds: if two ML constraints are in $\mathsf{ML^F}$ instance relation, they are also in ML instance relation: the differences of presentation in the definitions of grafting are not really significant. Thus we can reason about ML constraints using the $\mathsf{ML^F}$ instance relation, and reinterpret the obtained derivations as ML ones afterwards.

Lemma 9.3.6 *Consider two* ML *constraints* χ *and* χ'*. If* $\chi \sqsubseteq \chi'$ *holds, then* $\chi \;{}^{\mathsf{ML}}\!\sqsubseteq \chi'$ *also holds.* □

<u>Proof</u>: The proof is by noetherian induction on a canonical derivation $\chi \sqsubseteq|_{\chi'} \chi'$. If it is empty, the result is immediate. Otherwise, suppose that $\chi \sqsubseteq_1 \chi'' \sqsubseteq|_{\chi'} \chi'$. If $\sqsubseteq_1$ is $\sqsubseteq_1^M$ or $\sqsubseteq_1^R$, since χ is an ML constraint, $\chi \;{}^{\mathsf{ML}}\!\sqsubseteq_1 \chi'$ also holds, and the remainder of the result is by induction hypothesis. Moreover $\sqsubseteq_1$ cannot be a weakening, as the node would not be flexible in χ'.

If $\sqsubseteq_1$ is a grafting $\mathsf{Graft}(\tau, n)$, we must do some surgery. By hypothesis, τ is a constructor type. Let i be such that $1 \leq i \leq \mathsf{arity}(\tau(\langle\epsilon\rangle))$. Notice that $\langle n \cdot i\rangle$ can be raised in χ'': it is green, and raisable since there is no node under it. Moreover, since χ' is an ML constraint, the nodes $\langle n \cdot i\rangle$ cannot be bound at n in χ'. Let χ''' be the constraint obtained by raising once all the nodes $\langle n \cdot i\rangle$. By applying $\mathsf{arity}(\tau(\langle\epsilon\rangle))$ times Lemma 6.6.2, we obtain $\chi'' \sqsubseteq^R \chi''' \sqsubseteq \chi'$. By construction, we also have $\chi \;{}^{\mathsf{ML}}\!\sqsubseteq_1^G \chi'''$. The conclusion is then by induction hypothesis applied to $\chi''' \sqsubseteq \chi'$.

9.4 Typing constraints

As mentioned in §1.6, we consider the expressions of the λ-calculus

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

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

Expression nodes can be inductively transformed into simpler constraints using the rules presented in Figure 9.4.1. We follow the logical presentation of ML type inference, where generalization can be performed at every typing step, *i.e.* not only at let constructs. It is well-known that, for ML, both presentations are equivalent. However, this is not the case for $\mathsf{ML^F}$. Thus each basic expression is typed as a type scheme, and the root of a basic constraint will always be a gen node. We have drawn those nodes in Figure 9.4.1 on top of expression nodes, in order to disambiguate the origin and destination of edges.
Let us give some details on the translation:

- A variable x is typed as the universal type scheme $\forall\,(\alpha)\ \alpha$. Graphically this scheme is represented as a gen node g with a single child, this child being a bottom node bound on g. This bottom node is constrained by the unique edge annotated by x in

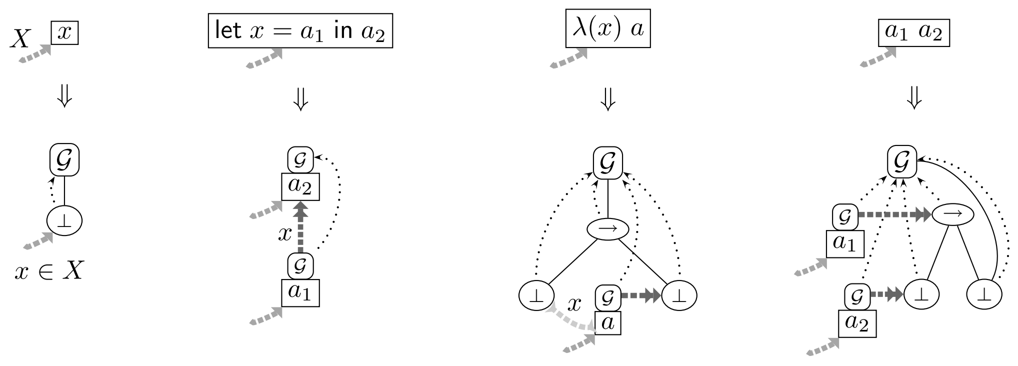


Figure 9.4.1 – Typing of primitive expressions

the typing environment; if there is no such edge, the constraint is not closed, thus untypable.

- A let-binding let $x = a_1$ in a_2 is simply typed as a_2 in which all the occurrences of x are constrained to be instances of a_1.
- An abstraction $\lambda(x)\ a$ is typed as a type scheme $\forall\,(\alpha \geqslant \bot)\ \forall\,(\beta \geqslant \bot)\ \alpha \to \beta$. The codomain of the arrow (*i.e.* β) is required to be an instance of the type of a, and the type of the occurrences of x in a must unify with the domain of the arrow (*i.e.* α).
- An application $a_1\ a_2$ is typed as the codomain of an arrow type existentially introduced. The domain of the arrow is constrained to be an instance of the type of a_2, while the arrow type itself is constrained be an instance of the type of a_1.

Definition 9.4.1 (*Typing constraints*) *Typing constraints* are the subset of constraints generated from λ-terms by the rules of Figure 9.4.1. ■

▶ **Example** Figure 9.4.2 shows a step by step transformation of the expression node for the term let $y = \lambda(x)\ x$ in $y\ y$ into the corresponding typing constraint. At each step, we develop the expression node highlighted in blue.

As shown below, typing constraints are well-formed constraints, and in particular ML ones. Thus, typing constraints are exactly the same in ML and ML^F; only the way in which the constraints are interpreted (§10) changes between ML and ML^F.

Property 9.4.2 *Typing constraints are well-formed ML and ML^F constraints.* □

Proof: All points but conditions 8 and 10 in Definition 9.2.1 are immediate. We detail them below:

▷ Condition 10: the property is immediate for the instantiation edges introduced in applications and abstractions. For the edge introduced for a let construct: the ending of this edge necessarily goes to the bottom node of the subconstraint for a variable, and this bottom node is bound on a gen node. Thus the condition holds for this edge.

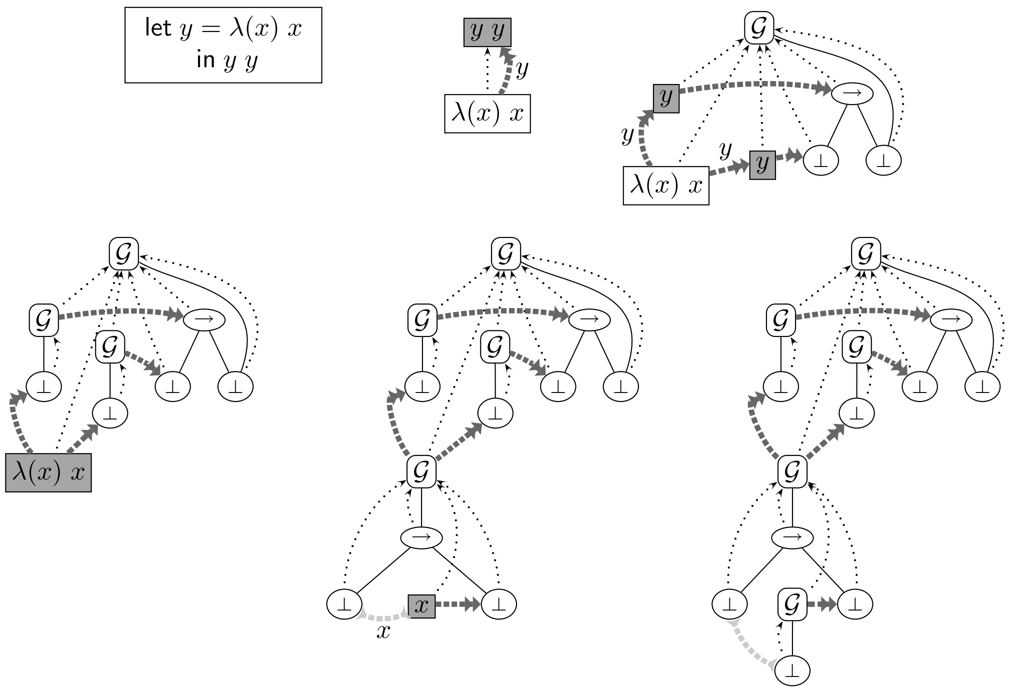


Figure 9.4.2 – Obtaining the typing constraint for let $y = \lambda(x)\ x$ in $y\ y$

▷ Condition 8: consider an unification edge resulting from the typing of an abstraction. Its right extremity (in the drawing) is transitively bound to the gen node for the abstraction. Hence this edge is admissible.

Finally, the fact that typing constraints are ML constraints is also immediate.

Notice that typing constraints only use gen nodes introducing a single type scheme, as the grammar of our λ-terms does not permit mutual (non-recursive) let constructs. The generalization to this construct is however immediate. Recursive let, mutual or not, are discussed in §12.1.1.

10

Semantics of constraints

Abstract

This chapter studies the semantics of constraints. Given a type scheme, we start by defining what it means to take a fresh instances of it (§10.1). Then we characterize the instantiation edges that are solved in a constraint (§10.3). We define what it means for a constraint to be solved, and for a type to be a solution of a constraint (§10.4). The meaning of a constraint is simply the set of its solutions (§10.5). Finally, we relate the meaning of ML and ML^{F} constraints (§10.6).

10.1 Expanding a type scheme

Type generalization is an essential part of ML type inference, and this is also the case in ML^{F}. Thus, a crucial and very common operation consists in taking a fresh instance of a type scheme. This requires taking into account the generalization level encoded by the gen node introducing the type scheme. In fact, the structural interior of a gen node g exactly contains the nodes generalizable at the level of g. Indeed, it would be unsafe to generalize a node n in the exterior of g: n can only be generalized higher in the constraint.

Thus, consider the i^{th} type scheme introduced by a gen node g. In order to take an instance of this scheme:

- we copy the nodes under $\langle g \cdot i \rangle$ that are in the structural interior of g. The exact shape of the binding tree of the copy is derived from the one of the structural interior, but depends on whether we create an ML^{F} or ML instance.
- for each node n in the structural frontier of g and under $\langle g \cdot i \rangle$, we introduce a fresh bottom node connected to the original node n by a unification edge. This ensures that all instances of $\langle g \cdot i \rangle$ will share n. We use unification edges because reusing n directly could result in ill-dominated constraints.

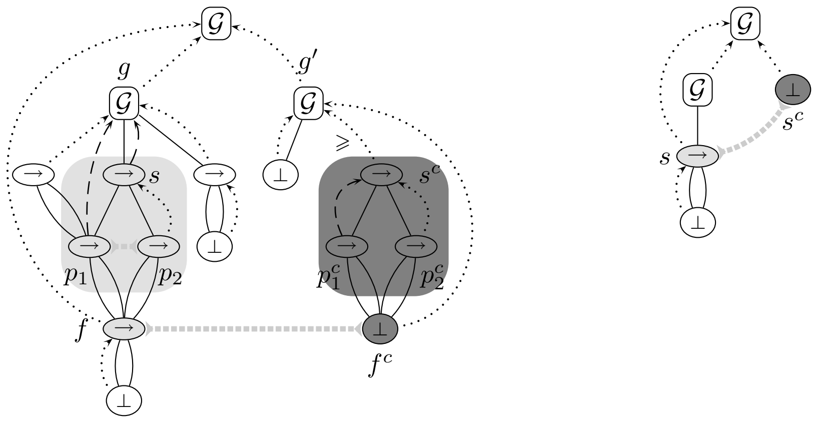


Figure 10.1.1 – Two examples of $\mathsf{ML^F}$ expansion

The creation of a fresh instance of a type scheme is called *expansion*. Of course, it must be given a destination gen node to which the nodes created by the expansion will be bound. Expansion is slightly less general in ML than in $\mathsf{ML^F}$, as ML types do not allow inner quantification; the difference will be explained through examples later.

Definition 10.1.1 (*$\mathsf{ML^F}$ and ML expansion*) Let χ be a constraint, g a gen node of χ, and s a type scheme introduced by g (*i.e.* $g \multimap s$). Let g' be a gen node of χ. The *expansion of s at g'* is derived from χ by:

- adding to χ a copy of $\mathsf{cproj}(\chi \restriction ((\mathcal{I}^s(g) \cup \mathcal{F}^s(g)) \cap (s \overset{+}{\multimap})))$.

  We write p^c the copy of a node p.

- for every copied node f of $\mathcal{F}^s(g)$, changing f^c into a bottom node flexibly bound at g', and adding the unification edge $f \rangle\!\cdots\!\langle f^c$;

- binding s^c flexibly to g';

- for every copied node p different from g and such that $p \xrightarrow{\diamond} g$, adding the binding edge $p^c \xrightarrow{\diamond} p'$, where p' is s^c in $\mathsf{ML^F}$ expansion, and g' in ML expansion.

We call *root of the expansion* the node s^c, and *frontier unification edges* the unification edges we introduce. ■

► **Example** An illustration of an $\mathsf{ML^F}$ expansion is given as the left constraint in Figure 10.1.1. The right-hand side of the constraint is the result of expanding the scheme $\langle g1\rangle$ at g'. We have highlighted the nodes to be copied (s, p_1, p_2 and f, on the left) in dark blue and their copies (s^c, p_1^c, p_2^c and f^c, on the right) in red.

Notice that existential nodes and inner constraints are ignored during expansion, as illustrated by the unification edge between p_1 and p_2 in Figure 10.1.1. Indeed, expansion is concerned with the type structure, not with the constraint structure.

Expansion preserves the well-formedness of constraints.

Property 10.1.2 *The* $\mathsf{ML^F}$ *(*resp. *ML) expansion χ' of a type scheme s at a gen node g' in an* $\mathsf{ML^F}$ *(*resp. *ML) constraint χ is an* $\mathsf{ML^F}$ *(*resp. *ML) constraint.* □

Proof: We first justify that χ is an $\mathsf{ML^F}$ constraint.

A copied node has a bound, either because its binder has been copied, or because it has been rebound explicitly. The new nodes form a valid-term graph: we essentially copy the top of $\check{\chi}/s$, and the nodes at which the copy stops are in the frontier, hence replaced by bottom nodes for which the arity is correct. Well-domination is also simple. Most of the other conditions of Definition 9.2.1 are immediate; we justify the others below:

- ▷ *Condition* 6: In ML, all the fresh nodes are existential. In $\mathsf{ML^F}$, only the root of the expansion and the copies of the nodes of the frontier are. However, all those nodes are bound on g', which is a gen node.
- ▷ *Condition* 8: The new unification edges are admissible, as their extremities inside the new structure are bound on g'.

Finally, suppose that we are considering an ML expansion in an ML constraint. All the nodes in the expansion are flexible, and bound on g', which is a gen node. Hence χ' is an ML constraint.

10.1.1 Degenerate type schemes

An interesting subcase occurs when s is not bound on g; in this case, the type scheme represented by s is monomorphic. We say that s is *degenerate*. In terms of expansion, when this scheme is expanded, only s is copied, and the copy is required to unify with s itself.

▶ **Example** The constraint on the right of Figure 10.1.1 is the result of expanding s at $\langle\epsilon\rangle$ in the constraint composed of the leftmost part of this constraint.

Interestingly, all the λ-bound variables of a constraint are degenerate once their unification constraint has been solved. This shows that, as is customary with type systems with second-order polymorphism and type inference (Peyton Jones *et al.* 2007; Leijen 2008), unannotated lambda abstractions cannot receive really polymorphic arguments.

10.1.2 Flag and binding reset

Let us consider a scheme s. When performing an $\mathsf{ML^F}$ expansion on s, the binding flag of s is entirely ignored: s^c is always flexibly bound. Similarly, if g is the gen node that introduces s, the nodes bound on g and s are all bound on s^c in the expansion. We call those two properties *flag* and *binding reset*

At first, it would seem that the flag of s, and the fact that a node is bound on g or s is indifferent. However, this is not the case, and is in fact deeply linked to the notion of generalization itself. In essence, the fact that a node is bound on s expresses that the polymorphism remains local to s. On the contrary, a node bound on g means that the polymorphism has been forced to leave the scope of s—although it will "reappear" through generalization.

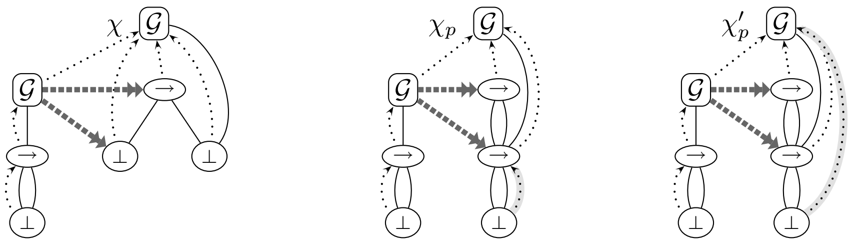


Figure 10.1.2 – Generalization and binding reset

▶ **Example** Consider the constraint χ of Figure 10.1.2, which represents a simplified typing constraint for $(\lambda(x)\ x)\ (\lambda(x)\ x)$. We have also shown two instances of χ, which are solved (*i.e.* the constraints created by the instantiation edges are satisfied). The difference between χ_p and χ'_p lies in the binder for the nodes $\langle 11\rangle$, which is bound on the type scheme $\langle 1\rangle$ in χ_p, and on the gen node $\langle\epsilon\rangle$ in χ'_p. As shown in Part III, we can translate those constraints into explicitly-typed terms, and we obtain the two terms

$$\mathsf{id}[\sigma_{\mathsf{id}}]\ \mathsf{id} \qquad \text{and} \qquad \Lambda(\alpha)\ \mathsf{id}[\alpha \to \alpha]\ \mathsf{id}[\alpha]$$

where $\mathsf{id} \ \triangleq\ \Lambda(\alpha)\ \lambda(x : \alpha)\ x$. In the first term, the polymorphism of the two identity functions is kept local. In the second, those functions are typed monomorphically, and the type variable α is generalized at the front of the type. Of course the first case is only possible in ML^{F}, as there is no no inner polymorphism in ML constraints.

The binding flag of s is also important, as it can be used to restrict the way in which the interior of s can be solved by instance. However, this is related to solving s, not to the type scheme it represents. Thus, when we take an instance of s, we can safely ignore this flag. In fact, we *must* ignore this flag, as we want to be able to instantiate the expanded nodes.

▶ **Example** In the leftmost constraint of Figure 10.1.1, p_2 is red, as s is rigidly bound. Thus the unification edge of the constraint is unsolvable. The binding flag of s is however ignored in the expansion, which represents s at this stage of the constraint solving.

10.2 An example

In the remainder of this chapter, we use as a threaded example the typing of the term

$$K \ \triangleq\ \lambda(x)\ \lambda(y)\ x$$

The typing constraint for K is the constraint χ_K presented in Figure 10.2.1. However, in order to simplify the examples, we will reason on the simpler constraint χ; §11.3 will show that the two constraints are in fact equivalent.

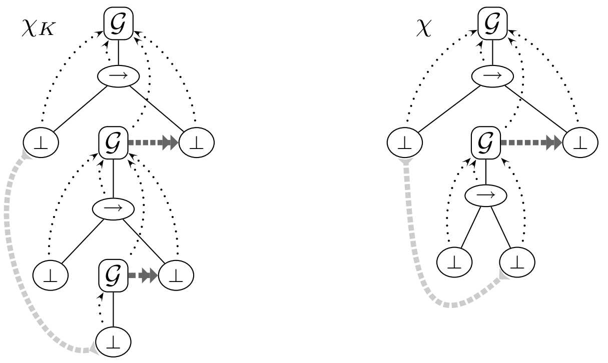


Figure 10.2.1 – Constraints for $\lambda(x)\ \lambda(y)\ x$

10.3 Solved constraint edges

Solved constraints are constraints in which edges are solved. For unification edges, this is the case when they relate two nodes already merged.

Definition 10.3.1 (*Solved unification edges*) A unification edge $n_1 \leftrightarrow n_2$ is solved if n_1 and n_2 are equal. ■

An instantiation edge is solved when a fresh instance of the corresponding type scheme *matches* the target of the edge, *i.e.* it unifies with the target without changing the constraint. This property can be checked by using an intermediate constraint that enforces this matching.

Definition 10.3.2 (*Propagation*) Let e be an edge $g \overset{i}{\twoheadrightarrow} n$ of a constraint χ. We call *propagation* of e in χ, written χ^e, the constraint obtained by expanding $\langle g \cdot i \rangle$ at $\hat{n}$, and adding a unification edge between n and the root of the expansion. ■

This definition is well-formed: by condition 10 of Definition 9.2.1, $\hat{n}$ is a gen node and the expansion is defined.[1]

Intuitively, propagation enforces the constraint imposed by the instantiation edge by forcing the unification of a fresh copy of the type scheme with the constrained node.

▶ **Example** The constraint χ'^e in Figure 10.3.1 results from performing both an ML and an ML$^\mathsf{F}$ propagation on the unique instantiation edge of χ'. The nodes created by the ML propagation are those in blue (under n_1), while those created by the ML$^\mathsf{F}$ propagation are the red ones (under n_2). Notice the highlighted binding edges, that show the difference between ML and ML$^\mathsf{F}$ expansion.

As for expansion, propagation preserves the well-formedness of constraints.

[1] We could in fact slightly relax condition 10, by changing the definition of propagation so that the expansion takes place at the first gen node g such that $n \xrightarrow{+} g$. However, this would not increase expressivity: solving the unification edge created by the propagation would raise n until it is bound on g.

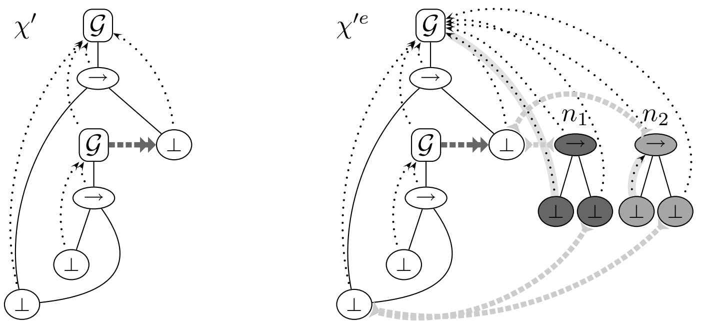


Figure 10.3.1 – Examples of propagation

Lemma 10.3.3 *Given an instantiation edge e of an* ML *(*resp. ML^F*) constraint χ, the constraint χ^e is a well-formed* ML *(*resp. ML^F*) constraint.* □

Proof: Let e be $g \dashrightarrow n$. By Property 10.1.2 it suffices to prove that the unification edge between n and the root of the expansion is admissible. This is the case, as the root of the expansion is bound on $\hat{n}$, which is a gen node by condition 10 of Definition 9.2.1.

An instantiation edge e is solved when the constrained node is already sufficiently instantiated, *i.e.* when the constraints introduced by propagating e can be solved without further instantiating the constraint. In order to be more general, we parameterize this definition by the instance relation $\sqsubset$ to use. In the remainder of Part II, and unless stated otherwise, $\sqsubset$ ranges over $\sqsubseteq$, $\sqsubseteq^{\approx}$, $\sqsubseteq^{\boxminus}$, ${}^{\mathsf{ML}}\sqsubseteq$ or ${}^{\mathsf{ML}}\sqsubseteq^{\approx}$.

Definition 10.3.4 (*Solved instantiation edge)* Given a constraint χ, an instantiation constraint e of χ is $\sqsubset$-*solved* if $\chi^e \sqsubset \chi$. ■

▶ **Examples** The instantiation edge of χ' in Figure 10.3.1 is not $\sqsubset$-solved: unifying the unification edges resulting from the propagations would change the skeleton under the node $\langle 11 \rangle$. Conversely, it is easy to check that this edge is $\sqsubseteq$-solved in the constraints $\chi_{\mathsf{ML^F}}$, χ_{ML} and χ'_{ML} of Figure 10.4.1. Moreover, it is ${}^{\mathsf{ML}}\sqsubseteq$-solved in χ_{ML} and χ'_{ML}.

10.4 Solutions and presolutions of constraints

The semantic of graphic constraints is given in two steps. We begin by characterizing the instances of a constraint that are solved, and use them to deduce the types that are solutions of the constraint.

Definition 10.4.1 (*Presolutions of constraints)* A $\sqsubset$-*presolution of* χ is an instance χ_p of χ for $\sqsubset$ (*i.e.* $\chi \sqsubset \chi_p$) in which unification edges are solved and instantiation edges are $\sqsubset$-solved. A $\sqsubset$-*presolution* is a constraint that is a $\sqsubset$-presolution of itself. ■

(Since instance is reflexive, a presolution is simply a constraint in which all edges are solved.)

Interestingly, since we view constraints up to solved unification edges (Definition 9.2.1), we can simply assume that presolutions do not contain unification edges. This is generally what we do in proofs.

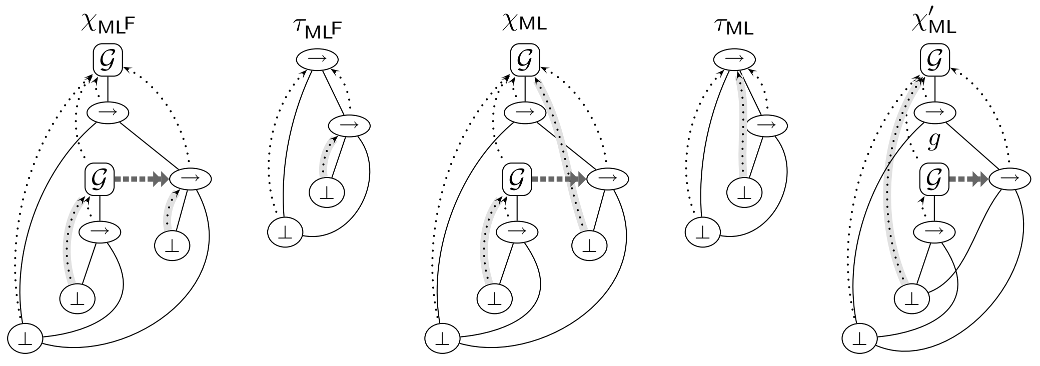


Figure 10.4.1 – Presolutions and solutions of K

▶ **Examples** Consider again the constraint χ' of Figure 10.3.1. It is an instance of the constraint χ of Figure 10.2.1, but it is not a presolution, as the instantiation edge is not solved. The constraint χ' is further instantiated into the constraints $\chi_{\mathsf{ML^F}}$, χ_{ML} and χ'_{ML} of Figure 10.4.1; the differences between these three constraints are highlighted. Those three constraints are $\sqsubseteq$-presolutions of χ, as can be verified by performing a propagation. Notice that $\chi_{\mathsf{ML^F}}$ is not a ${}^{\mathsf{ML}}\sqsubseteq$-presolution, as it contains inner quantification: the node $\langle 121\rangle$ is not bound on $\langle\epsilon\rangle$. However, both χ_{ML} and χ'_{ML} are ${}^{\mathsf{ML}}\sqsubseteq$-presolutions.

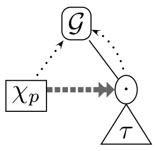
Interestingly, $\chi_{\mathsf{ML^F}} \sqsubseteq \chi_{\mathsf{ML}} \sqsubseteq \chi'_{\mathsf{ML}}$ holds. In fact, $\chi_{\mathsf{ML^F}}$ is the principal presolution of χ in $\mathsf{ML^F}$, as we will be able to prove in §12.

A presolution represents a fully solved, but also fully detailed form of a constraint, as all the nodes of the original constraint are retained. On the contrary, the solutions of a constraint are simply graphic types. More precisely, a solution is an instance of the type scheme at the root of a presolution. We formalize this property using the encoding presented in Figure 10.4.2, where τ is a solution of the constraint χ for which χ_p is a presolution.

Definition 10.4.2 (*Solutions of constraints*) A $\sqsubset$-*solution* of χ is a type τ such that there exists a $\sqsubset$-presolution χ_p of χ for which the instantiation edge in the constraint of Figure 10.4.2 is $\sqsubset$-solved. We say that χ_p *witnesses* τ. ■

This definition has the advantage of being compositional, as the root node and the inner gen nodes of χ_p are treated in a uniform way.

The solutions witnessed by a certain presolution χ_p are all the instances of the expansion τ of the scheme at the root of χ_p. Thus, the set of solutions witnessed by χ_p is, by construction, closed by instantiation; we refer to τ as «the principal solution witnessed by



In $\mathsf{ML^F}$, τ is a closed type.
In ML, the nodes of τ are all bound at $\langle\epsilon\rangle$.

Figure 10.4.2 – A constraint χ_p witnessing a type τ

χ_p». In order to show that two constraints have the same solutions, we often prove that their presolutions witness the same set of principal solutions, or equivalently that their presolutions expand to the same types. Of course, this does *not* implies that the set of all the solutions of χ is principal: there could exist two presolutions χ_p and χ'_p witnessing two principal solutions χ and χ' incomparable for $\sqsubset$.

▶ **Example** The principal solutions witnessed by the presolutions $\chi_{\mathsf{ML^F}}$, χ_{ML} and χ'_{ML} of χ are the types $\tau_{\mathsf{ML^F}}$, τ_{ML} and τ_{ML} respectively. In particular, χ_{ML} and χ'_{ML} witness exactly the same solutions. Moreover, since $\chi_{\mathsf{ML^F}} \sqsubseteq \chi_{\mathsf{ML}}$, all the solutions witnessed by χ_{ML} and χ'_{ML} are also witnessed by $\chi_{\mathsf{ML^F}}$.
Syntactically, τ_{ML} and $\tau_{\mathsf{ML^F}}$ are the types

$$\forall\,(\alpha)\;\forall\,(\beta)\;\alpha \to \beta \to \alpha \qquad\text{and}\qquad \forall\,(\alpha)\;\forall\,(\gamma \geqslant \forall\,(\beta)\;\beta \to \alpha)\;\alpha \to \gamma$$

Using our syntactic sugar, this second type can be written as

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

Interestingly, both types are correct System F types for K.

10.4.1 Presolutions and explicitly typed terms

Presolutions are interesting objects in their own right. First, they are the equivalent of an entire typing derivation. Moreover, given a λ-term a and a presolution χ_p of the typing constraint corresponding to a, χ_p can be used to obtain a version of a where all type information is fully explicit. This correspondence forms the basis of a translation into a Church-style version of $\mathsf{ML^F}$, which is developed in Part III of this document.

10.5 Meaning of constraints

We are now able to define the meaning of a constraint.

Definition 10.5.1 (*Meaning of constraints*) The $\sqsubset$-*meaning* of a constraint is the set of its $\sqsubset$-solutions. A constraint χ $\sqsubset$-*entails* a constraint χ', written $\chi \Vdash^{\sqsubset} \chi'$ if the $\sqsubset$-meaning of χ is a subset of the $\sqsubset$-meaning of χ'. Two constraints χ and χ' are $\sqsubset$-*equivalent*, written $\chi \dashv\Vdash^{\sqsubset} \chi'$, if they have the same $\sqsubset$-meaning. ■

It follows from the semantics of constraints that instantiation reduces the set of solutions, *i.e.* if $\chi \sqsubset \chi'$, then $\chi' \Vdash^{\sqsubset} \chi$. In general, the inclusion is strict; in fact, instantiation might change a solvable constraint into an unsolvable one. In certain cases, instantiation may however preserve the meaning. Conversely, many constraints not in instance relation have the same meaning. For example, two constraints with different gen nodes structure (*i.e.* constraint shape), cannot be in instance relation, as this structure is invariant by instantiation; however, they can still have the same meaning.

▶ **Examples** Consider the constraints χ_{ML} and χ_{MLF} of Figure 10.4.1 which verify $\chi_{\mathsf{MLF}} \sqsubseteq \chi_{\mathsf{ML}}$. In this case, the $\sqsubseteq$-meaning of the constraint χ_{ML} is strictly included in the one of χ_{MLF}, as τ_{MLF} is in the latter but not in the former. Considering now the constraints χ of Figure 10.2.1 and χ' of Figure 10.3.1 (which differ only by the unification edge which is solved in χ'), we again have $\chi \sqsubseteq \chi'$. However they have the same $\sqsubseteq$- and ${}^{\mathsf{ML}}\sqsubseteq$-meaning.[2] Finally, the constraints χ_K and χ of Figure 10.2.1 have different shapes, but will be proven $\sqsubseteq$- and ${}^{\mathsf{ML}}\sqsubseteq$-equivalent in §11.3.

Convention In the following, we focus mainly on $\sqsubseteq$ and ${}^{\mathsf{ML}}\sqsubseteq$-solutions. Thus, we allow to omit $\sqsubseteq$ in front of the relevant terms; for example a "presolution" must be understood as a $\sqsubseteq$-presolution. Similarly, we abbreviate ${}^{\mathsf{ML}}\sqsubseteq$ as ML. Thus $\chi \dashv\Vdash^{\mathsf{ML}} \chi'$ means that χ and χ' are ${}^{\mathsf{ML}}\sqsubseteq$-equivalent.

10.5.1 Preserving presolutions

While we are ultimately interested in proving that constraints have the same set of solutions, we often show the stronger result that *presolutions* are preserved by instantiation.

Definition 10.5.2 (*Preservation of presolutions*) We write $\chi \Vdash^{p} \chi'$ if every presolution of χ' is a presolution of χ, and $\chi \dashv\Vdash^{p} \chi'$ if both constraints have the same set of presolutions. ■

10.5.2 The different flavours of ML^F

The three instance relations $\sqsubseteq$, $\sqsubseteq^{\approx}$ and $\sqsubseteq^{\boxminus}$ define three different systems, in which the set of typable terms is not a priori the same. We name those three systems as follows:

Instance relation	$\sqsubseteq$	$\sqsubseteq^{\approx}$	$\sqsubseteq^{\boxminus}$
System	$g\mathsf{ML}^\mathsf{F}$	$e\mathsf{ML}^\mathsf{F}$	$i\mathsf{ML}^\mathsf{F}$

The system $i\mathsf{ML}^\mathsf{F}$ is the *implicit* version of ML^F, as it can be shown that adding type annotations to terms or λ-abstractions does not increase its expressivity. By contrast, $e\mathsf{ML}^\mathsf{F}$ is the explicit version, as type annotations are needed. Up to the difference in the set of types (§3.4.3), both systems correspond to the syntactic systems presented in (Le Botlan and Rémy 2007). The system $g\mathsf{ML}^\mathsf{F}$ is the *graphic* version of ML^F—this system has never been studied syntactically before.

Of course, since $(\sqsubseteq) \subseteq (\sqsubseteq^{\approx}) \subseteq (\sqsubseteq^{\boxminus})$, all terms typable in $g\mathsf{ML}^\mathsf{F}$ are typable in $e\mathsf{ML}^\mathsf{F}$, and similarly for $e\mathsf{ML}^\mathsf{F}$ and $i\mathsf{ML}^\mathsf{F}$. Thus the question is whether the inclusions between the

[2]We admit this result for now. It will be proven in §11.2.

sets of typable terms are strict or not. In the following we focus mainly in $g\mathsf{ML^F}$, as it is the most practical system to perform type inference in. We formally compare the expressivity of the three systems in §13.

10.6 Relating the meaning of ML and $\mathsf{ML^F}$ constraints

Consider the constraint χ' in Figure 10.3.1, and more precisely the nodes resulting from the ML expansion (in blue) and the $\mathsf{ML^F}$ one (in red). The difference lies in the binders of $\langle n_1 \cdot 1\rangle$ and $\langle n_2 \cdot 1\rangle$, which we have highlighted. In the ML expansion, $\langle n_1 \cdot 1\rangle$ is bound on $\langle \epsilon\rangle$. However, in the $\mathsf{ML^F}$ expansion $\langle n_2 \cdot 1\rangle$ is bound on n_2, creating inner polymorphism, forbidden in ML.

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

Property 10.6.1 *Consider an ML constraint χ, s a type scheme and g a gen node. Let χ_{ML} (*resp.* $\chi_{\mathsf{ML^F}}$) be the result of performing an ML (*resp.* $\mathsf{ML^F}$) expansion of s at g in χ. Then $\chi_{\mathsf{ML^F}} \sqsubseteq \chi_{\mathsf{ML}}$.* □

Proof: Let n be the root of the expansion in $\chi_{\mathsf{ML^F}}$. The constraint χ_{ML} is obtained by raising all the nodes bound on n, *i.e.* multi-raising n. By definition of an ML constraint and of expansion, all the nodes bound on n are flexibly bound. Since n itself is green (as it is flexibly bound to a gen node), the multi-raising is possible.

As the ML instance relation is a subrelation of the $\mathsf{ML^F}$ one, it is then immediate that $\mathsf{ML^F}$ extends ML.

Property 10.6.2 *Let $\sqsubset$ range over $\sqsubseteq$ and $\sqsubseteq^{\approx}$. Any ${}^{\mathsf{ML}}{\sqsubset}$-(pre)solution of an ML constraint is also a $\sqsubset$-(pre)solution.* □

Proof: Let χ be an ML constraint. Let χ_p be a $\sqsubset^{\mathsf{ML}}$-presolution of χ. By hypothesis, $\chi \; {}^{\mathsf{ML}}{\sqsubset} \; \chi_p$ holds. By Property 9.3.5, $\chi \sqsubset \chi_p$ holds and it remains to prove that the instantiation edges of χ_p are $\sqsubset$-solved. Consider such an edge e. Let χ_{ML} and $\chi_{\mathsf{ML^F}}$ be the ML and $\mathsf{ML^F}$ propagation of e in χ_p respectively. By hypothesis, $\chi_{\mathsf{ML}} \; {}^{\mathsf{ML}}{\sqsubset} \; \chi_p$ holds. Thus $\chi_{\mathsf{ML}} \sqsubset \chi_p$ holds by Property 9.3.5. Moreover, by Property 10.6.1, $\chi_{\mathsf{ML^F}} \sqsubseteq \chi_{\mathsf{ML}}$ holds, which implies $\chi_{\mathsf{ML^F}} \sqsubset \chi_{\mathsf{ML}}$. This shows $\chi_{\mathsf{ML^F}} \sqsubset \chi_p$, which is the desired result.

Interestingly, $\mathsf{ML^F}$ presolutions containing only flexible edges can always be transformed by raising into ML presolutions. Thus flexible quantification alone is not significantly more expressive than ML quantification; it just gives more general types—and more opportunities to use rigid quantification. This point is discussed further in §12.3.1.

Theorem 10.6.3 *Consider an ML constraint χ with an $\mathsf{ML^F}$ presolution χ_p in which all binding edges are flexible. Then there exists solutions of χ witnessed by χ_p that are ML types, and those types are also ML solutions of χ.* □

Proof: Let us call χ_r the constraint derived from χ_p by binding any type node n on the first gen node g such that $n \xrightarrow{+} g$ (if the binder of n is already a gen node, its binder is unchanged). Thus defined, χ_r is an ML constraint. Moreover χ_r can be derived from χ_p by multi-raising all the types nodes of χ_p. Since all the nodes of χ_p have green permissions, $\chi_p \sqsubseteq^R \chi_r$ holds (**1**).

Let us prove that χ_r is a ML presolution (**2**). We first show that $\chi \overset{\mathsf{ML}}{\sqsubseteq} \chi_r$ holds. We have $\chi \sqsubseteq \chi_r$ since $\chi \sqsubseteq \chi_p$ (as χ_p is a presolution) and by (1). Since both χ and χ_r are ML constraints, the subresult is by Lemma 9.3.6.

Next, consider an instantiation edge e. Let χ'_p be the ML^F propagation of e in χ_p. By hypothesis, $\chi'_p \sqsubseteq \chi_p$ holds (**3**). Let χ'_r be the ML propagation of e in χ_r. It remains to prove that e is ML-solved in χ_r, *i.e.* that $\chi'_r \overset{\mathsf{ML}}{\sqsubseteq} \chi_r$ holds, to prove (2).

Let us first prove that $\chi'_p \sqsubseteq^R \chi'_r$ holds (**4**). Given a gen node g, $\mathcal{I}^s_{\chi_p}(g) = \mathcal{I}^s_{\chi_r}(g)$; thus the expanded nodes in both χ'_p and χ'_r have the same shape. It is them immediate that χ'_r is the result of multi-raising all the type nodes of χ'_p. Since all the nodes of χ'_p are green, the raising is possible, which implies (4).

We can now repeatedly apply Lemma 6.6.2 to $\chi'_p \sqsubseteq \chi_r$ (proven by (1) and (3)) and to each node raised in (4). We obtain $\chi'_r \sqsubseteq \chi_r$. Lemma 9.3.6 proves in turn that $\chi'_r \overset{\mathsf{ML}}{\sqsubseteq} \chi_r$, as both constraints are ML constraints. Thus (2) holds.

Finally, the expansion of χ_r is an instance of the expansion of χ_p (obtained by raising all the nodes to the root), which shows that all solutions witnessed by χ_r are witnessed by χ_p. This is the desired result.

11

Reasoning on constraints

Abstract

In this section, we study some equivalence rules on gML$^\mathsf{F}$ constraints: an existentially introduced subpart of a constraint that is not constrained can be removed (§11.1); unification is sound, complete, and principal on constraints (§11.2); an instantiation edge leaving a degenerate gen node is itself degenerate, and can be replaced by a unification edge (§11.3); eager propagation of instantiation edges preserves the meaning of constraints (11.4).

We also show two important auxiliary results. First, an instance derivation showing that an instantiation edge is solved can be assumed to have a very specific shape (§11.5). Second, solved instantiation edges remain solved through instance steps that change unrelated parts of the constraint (11.6).

11.1 Removing unconstrained existential nodes

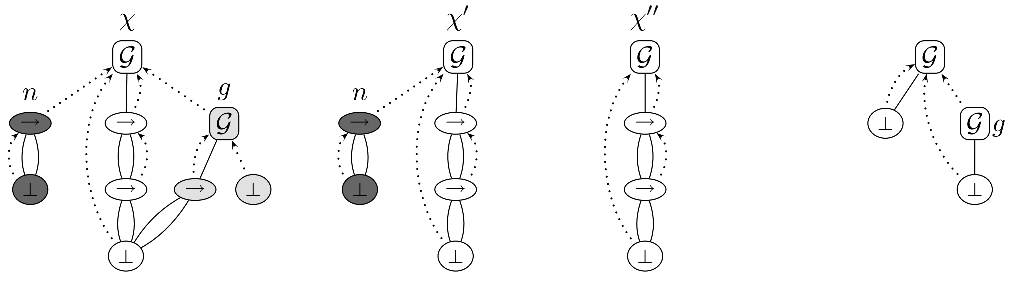


Figure 11.1.1 – Simplifying unconstrained existential nodes

Existential nodes are meant to introduce constraint edges. If it is not the case, for example because those constraint edges have been solved or removed, the existence of the existential nodes does not add constraints. Hence they are useless, and can be eliminated. Implementation-wise, this allows saving memory; from a theoretical standpoint, it permits reasoning on simpler constraints.

Definition 11.1.1 (*Existential elimination*) Let n be a purely existential node of a constraint χ such that no node in $\mathcal{I}^c(n)$ is the origin or the target of an instantiation or unification edge. We call *existential elimination of* n *in* χ the constraint $\chi{\restriction}(\mathsf{dom}(\chi) \setminus \mathcal{I}^c(n))$ obtained by removing from χ all the nodes of $\mathcal{I}^c(n)$ and all the dangling edges. We refer to the rule transforming χ into this constraint as Exists-Elim. ■

▶ **Example** In Figure 11.1.1, existentially eliminating g in χ results in χ'; eliminating n in this constraint gives χ''. The interiors of the eliminated nodes are highlighted. However we cannot eliminate $\langle g1\rangle$ in the fourth constraint: while it is existential, it is not purely existential because of the edge $g \multimap \langle g1\rangle$, and eliminating it would result in an ill-formed constraint.

Property 11.1.2 *Existential elimination preserves the well-formedness of constraints.* □

<u>Proof</u>: Let χ be a constraint, and χ' obtaining by existentially eliminating n in χ. The only difficult property is the fact that constructors still have the correct arity in χ'. Thus, let n'' be a node of $\mathcal{I}^c(n)$, and n' such that $n' \multimap n'' \in \chi$; n' could have an incorrect arity in χ'.

Consider a mixed path P equal to $\langle\epsilon\rangle \leftarrow n' \multimap n''$ in χ. By well-domination, since by hypothesis $n' \xrightarrow{*} n$, we know that n is contained in P. Moreover, the case $n'' = n$ is impossible, as n is purely existential. This means that $n' \xrightarrow{*} n$, and n' is eliminated when n is eliminated.

The remainder of the section shows that existential elimination preserves solutions. We write $\exists^E$ for an existential elimination, and $\exists^I$ for the inverse operation, called existential introduction.

Lemma 11.1.3 *Existential introduction and atomic instance commute. That is, let χ be a constraint, χ' be such that $\chi \;\exists^I\; \chi'$, and χ'' be such that $\chi \sqsubseteq_1 \chi''$ with $\chi'' = o(\chi)$. Then $\chi' \sqsubseteq_1 o(\chi')$ and $\chi'' \;\exists^I\; o(\chi')$.* □

<u>Proof</u>: Let r be the root of the structure existentially introduced. We first prove that o can be applied to χ', by case disjunction on o. In each case, the nodes existing in χ are unchanged in χ', including their permissions (**1**).

▷ <u>*Case* o is a grafting or a weakening</u>: the result is immediate by (1).

▷ <u>*Case* $o = \mathsf{Merge}(n_1, n_2)$</u>: n_1 and n_2 are still locally congruent in χ', as both are type nodes, and the new nodes are introduced under a gen node. By (1) the result holds.

▷ <u>*Case* $o = \mathsf{Raise}(n)$</u>: by (1) it suffices to show that n is raisable in χ'. Since it was raisable in χ, it suffices to consider new structure paths below n (that were not present in χ). All new paths contain the edge $\hat{r} \leftarrow r$. However, since r is purely existential, there is thus no new structure path starting from n. Thus n is still raisable in χ'.

Let now χ''' be $o(\chi')$. It remains to prove $\chi''' \ \exists^E \ \chi''$ to conclude.

▷ <u>r is purely existential in χ'''</u>: (this ensures that r can be eliminated). This node is purely existential in χ'. The only operation removing a purely existential node is the merging of this node with another node. This is not the case, as o does not change r.

▷ <u>$\mathcal{I}^c_{\chi'''}(r) = \mathcal{I}^c_{\chi'}(r)$</u>: (this ensures that we eliminate exactly the nodes that have been introduced). This result holds, as interiors change only by the raising of a node of the interior, and o does not change $\mathcal{I}^s_{\chi'}(r)$.

The commutation of instance with existential elimination is not as simple, as we must distinguish the cases where the instance transformation occurs inside the part being eliminated. Also, if the interior of the node being eliminated is changed (by a raising), more than one elimination might be needed.

Lemma 11.1.4 *Let χ be a constraint, χ' and χ'' such that $\chi \sqsubseteq_1 \chi'$ and $\chi \ \exists^E \ \chi''$ Then $\chi' = \chi''$, or $\chi' \ (\exists^E)^+ \ \chi''$, or the two operations commute. Moreover the two operations commute if the instance operation does not change the nodes being eliminated.* □

<u>Proof</u>: Let n the root of the existential elimination, o the instance operation. The proof is by case analysis on o.

▷ <u>*Case* $o = \mathsf{Graft}(\tau, n')$ or $o = \mathsf{Weaken}(n')$</u>: If $n' \in \mathcal{I}^c(n)$, eliminating n in χ' gives χ''. Otherwise the two operations commute.

▷ <u>*Case* $o = \mathsf{Raise}(n')$</u>: If $n' \in \mathcal{I}^c(n)$ and n' is not bound on n, eliminating n in χ' gives χ'. If n' is bound on n, eliminating n then n' in χ' gives χ'' (by well-domination, n' is purely existential after the elimination of n). If $n' \notin \mathcal{I}^c(n)$, the two operations commute.

▷ <u>*Case* $o = \mathsf{Merge}(n_1, n_2)$</u>: If n_1 and n_2 are both in $\mathcal{I}^c(n)$, eliminating n in χ' gives χ''. If none are in $\mathcal{I}^c(n)$, the two operations commute. If one node is in $\mathcal{I}^c(n)$, and the other is not, either n_1 or n_2 is n. Then $\chi' = \chi''$.

Lemma 11.1.5 *Consider a constraint χ which is a presolution, and χ' derived from χ by performing an existential introduction or elimination (EEI). Then χ' is a presolution witnessing the same solutions as χ.* □

<u>Proof</u>: Let us first show that χ' is a presolution. It suffices to show that its instantiation edges are solved. Let e be such an edge $g \dashrightarrow d$. It is also an edge of χ, as EEI preserves constraint edges, and e is solved in χ since χ is a presolution. We prove the dashed lines of the diagram below, the leftmost one being is the desired subresult.

$$\begin{tikzcd}
\chi \arrow[r, "o \in \exists^{IE}"] \arrow[d, "e"'] & \chi' \arrow[d, "e"] \\
\chi^e \arrow[r, dashed, "o"'] \arrow[u, bend left=60, "I \in \sqsubseteq"] & \chi'^e \arrow[u, dashed, bend right=60, "I"']
\end{tikzcd}$$

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

Consider now the instance operations I solving the unification edges introduced in χ^e:

- ▷ If o is an introduction: Lemma 11.1.3 shows that this operation and I commute. Thus I solves the unification edges of $\chi_p'^e$, and the resulting constraint is χ plus the EEI, which is exactly χ'. Hence e is solved in χ'.
- ▷ If o is an elimination: I transforms only the subgraph introduced by the expansion and the nodes structurally under d. Necessarily those nodes are disjoint with the ones eliminated by o, as the former are constrained by e. Thus, in this case, existential elimination and instance commute (Lemma 11.1.4). The conclusion is then similar to the previous case.

For the remainder of the full result: χ' and χ expand to the same type, since expansion ignores existential nodes. Hence they witness exactly the same solutions.

As a direct consequence of the previous lemmas:

Lemma 11.1.6 *Existential elimination preserves solutions.* □

Proof: Consider a constraint χ, and a constraint χ' obtained by performing an existential elimination in χ.

Suppose that χ has a presolution χ_p. Lemma 11.1.4 shows that there exists χ_p' such that $\chi_p\ (\exists^E)^*\ \chi_p'$ and $\chi' \sqsubseteq \chi_p'$. Lemma 11.1.5 ensures that χ_p' is a presolution witnessing the same solutions as χ_p. Thus χ_p' is a presolution of χ'. Since this holds for any presolution, any solution of χ is a solution of χ'.

In the other direction, the reasoning is the same, using Lemma 11.1.3 instead of Lemma 11.1.4.

11.1.1 Raising and existential nodes

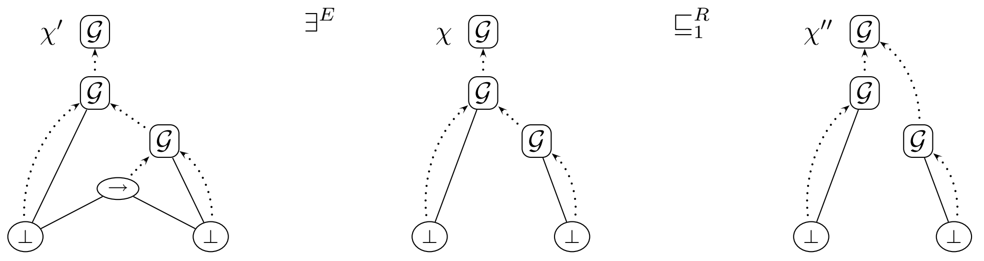


Figure 11.1.2 – Instance and existential introduction

We do not allow transforming gen nodes along the instance relation $^{c}\sqsubseteq$ on constraints, mainly because gen nodes encode the structure of the constraint and reflect the shape of the λ-term being typed. However, even if we chose to abandon that interpretation, allowing the raising of gen nodes would have very profound consequences. In particular, it would no longer be the case that existential introduction and instance commute (Lemma 11.1.3)—thus making reasoning on constraints much more complicated.

▶ **Example** Consider Figure 11.1.2, in which $\chi \exists^I \chi'$ and $\chi \sqsubseteq \chi''$ hold for an instance relation that would allow raising gen nodes. The two operations transforming χ into χ' and χ'' do not commute: the resulting constraint would be ill-dominated. Worse, the only way to close the diagram would be to raise the two bottom nodes of the constraints. By complicating slightly the example, we could in fact require the raising of nodes that were untouched by both the instance operation and the existential introduction!

Notice that this is also the main reason why we do not allow existential nodes to be bound on non-gen nodes.[1] Indeed, in order to allow reusing all the results on graphic types, it is crucial that type nodes can be raised independently of the constraint structure around them, including existential nodes.

11.2 Solving unification edges

The level of generalization we brought to our graphic representation is small enough that the unification algorithm on graphic types can be reused almost entirely unchanged: we only suppose that it maintains the unification and instantiation edges that are present on the graph it receives as argument. We show below that the resulting algorithm (which we still call Unif) is sound, complete, and principal for ${}^{c}\sqsubseteq$.

Step 1 of Unif performs first-order unification on its argument. In the case of a constraint χ, this may seem a bit puzzling: $\check{\chi}$ is not really a term-graph, but a forest of term-graphs. We can either perform unification on this forest (first-order unification does not require the graph on which it operates to be rooted), or on a concretization of χ. We choose the second option, as this allows us to remain within the framework of term-graphs. However, this is only a technicality. In particular, if we quotient constraints by the addition or removal of virtual edges, the result of Unif does not depend on the concretization: neither Unif nor Rebind "see" the virtual edges added by the concretization, as those edges are above the nodes to unify. In order to simplify the statements of the results of this section, we thus proceed as follows: we reason on a concretization of the constraints, but implicitly read the results up to removal of the virtual edges.

Lemma 11.2.1 *Consider a constraint χ and e an admissible unification edge $n_1 \rightarrowtail\!\!\leftarrowtail n_2$ of χ. Let χ_v be an unifier of e in χ for ${}^{c}\sqsubseteq$, τ a concretization of χ. Let g_U be the first-order principal unifier of n_1 and n_2 in $\check{\chi}$. Then* rebind$(\chi, g_U) \prec_{\mathsf{U}} \chi_v$. □

Proof: By definition, n_1 or n_2 is bound on a gen node (**1**). The proof is almost exactly the same as the one of Lemma 7.4.15 (page 103). In one subcase we had concluded using the fact that admissibility ancestors cannot be merged; here we conclude because gen nodes cannot be merged by ${}^{c}\sqsubseteq$.

We prove this subcase below, reusing some of the notations of the original proof. Let χ_U be rebind(χ, g_U) and n a node of this graph. We want to prove that $n \xrightarrow{+} \hat{\chi_v}(n) \in \chi_U$. Let N be the node resulting from the merging of n_1 and n_2 in χ_U, $m_1, \ldots, m_k$ be the nodes that are merged together in n in χ_U, and χ_r be such that $\chi \sqsubseteq^{GR} \chi_r \sqsubseteq^{MW} \chi_v$ is a

[1]The other reason is related to permissions. Existential nodes mainly belong to the constraint structure, and it is unclear whether they should follow the restrictions imposed by permissions. By requiring existential nodes to be bound on gen nodes, we guarantee that they are never red.

canonical derivation of $\chi \mathrel{{}^{\mathrm{c}}\!\sqsubseteq} \chi_v$. The interesting subcase is the one in which n is under N in χ_U, and is bound strictly above N in χ_v (**2**). By (2) the nodes $\hat{\chi_r}(m_i)$ are bound strictly above n_1 or n_2 in χ_r. Hence at least one of them is bound on a gen node by (1). Since they are merged on χ_v, they are all bound on a gen node. Since gen node cannot be merged by ${}^{\mathrm{c}}\!\sqsubseteq$, they are in fact all bound on the same gen node m. From there, the end of the proof is unchanged.

(Although we do not use this generalization in typing constraints, it is immediate to extend this result to a set N of nodes to unify, all but perhaps one being bound on a gen node.)

The result above is sufficient to conclude, as we have established the completeness and principality results of Unif using $\prec_{\mathsf{U}}$. As a consequence, unification is sound, complete and principal, and is an equivalence on constraints.

Lemma 11.2.2 *On graphic constraints, the unification algorithm is sound; it is also complete and principal when the unification edge it unifies is admissible.* □

<u>Proof</u>: Let χ be a constraint, e an unification edge $n_1 \rightarrowtail\!\cdots\!\leftarrowtail n_2$ of χ.

- ▷ <u>Soundness</u>: we assume that $\mathsf{Unif}_e(\chi)$ returns χ_u, and need to prove that $\chi \mathrel{{}^{\mathrm{c}}\!\sqsubseteq} \mathsf{Unif}_e(\chi)$ holds. By Theorem 7.4.13, we have $\chi \mathrel{{}^{\mathrm{t}}\!\sqsubseteq} \mathsf{Unif}_e(\chi)$. However Unif only merges nodes above n_1 and n_2, by definition of first-order unification; similarly, Rebind only changes the binding edges of nodes under n_1 and n_2. Thus, by Lemma 9.2.5, no operation of $\chi \mathrel{{}^{\mathrm{t}}\!\sqsubseteq} \mathsf{Unif}_e(\chi)$ occurs on a gen node and $\chi \mathrel{{}^{\mathrm{c}}\!\sqsubseteq} \mathsf{Unif}_e(\chi)$ also holds.

For completeness and principality, we assume that e is admissible and that a ${}^{\mathrm{c}}\!\sqsubseteq$-unifier χ_v of e in χ exists. It is also a ${}^{\mathrm{t}}\!\sqsubseteq$-unifier (**1**), as $({}^{\mathrm{c}}\!\sqsubseteq) \subset ({}^{\mathrm{t}}\!\sqsubseteq)$ up to constraint edges.

- ▷ <u>Completeness</u>: by Theorem 7.4.17, Lemma 11.2.1 and (1) the computation of $\mathsf{Unif}_e(\chi)$ does not fail; this is the desired result.
- ▷ <u>Principality</u>: by Theorem 7.4.22, Lemma 11.2.1 and (1) we have $\chi_u \mathrel{{}^{\mathrm{t}}\!\sqsubseteq} \chi_v$. By soundness we have $\chi \mathrel{{}^{\mathrm{c}}\!\sqsubseteq} \chi_u$, and we have $\chi \mathrel{{}^{\mathrm{c}}\!\sqsubseteq} \chi_v$ by hypothesis. Hence the shape of gen nodes is exactly the same in χ_u and χ_v, and the relation $\chi_u \mathrel{{}^{\mathrm{c}}\!\sqsubseteq} \chi_v$ also holds.

Lemma 11.2.3 *Let e be an admissible unification edge of a constraint χ. If unifying e in χ fails, χ has no solution. Otherwise, let χ' be the principal unifier of e in χ. Then $\chi \dashv\vdash^{p} \chi'$* □

<u>Proof</u>: Suppose that the unification of e fails, but that χ has a presolution χ_p. By definition of presolutions, e is solved in χ_p, hence χ_p is a unifier of e in χ. This contradicts the completeness of the unification algorithm (Lemma 11.2.2).

Consider now a presolution χ_p of χ. By definition, $\chi \sqsubseteq \chi_p$ holds. Since e is solved in χ_p (by definition of presolutions), we have $\chi \sqsubseteq \chi' \sqsubseteq \chi_p$ by principality of unification (Lemma 11.2.2). Thus the presolutions of χ are presolutions of χ'. This implies the result, as we have $\chi \sqsubseteq \chi'$ by soundness (which implies that the presolutions of χ' are presolutions of χ).

11.2.1 Unification in ML constraints

It is immediate that unification on ML graphic types can be solved with the unification algorithm for $\mathsf{ML^F}$ graphic types. This follows from the fact that the unification algorithm of $\mathsf{ML^F}$ applied to ML graphic constraints returns ML graphic constraints, and Lemma 9.3.6.

Unlike in graphic types (§7.7), unification might raise some nodes, from one gen node to a gen node above. However those raisings amount to updating generalization levels when variables are merged, exactly as done in efficient implementations of ML type inference based on ranks and term dags (Pottier and Rémy 2005; Rémy 1992).

11.3 Removing degenerate instantiation edges

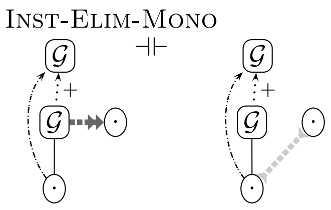


Figure 11.3.1 – Simplifying degenerate instantiation edges

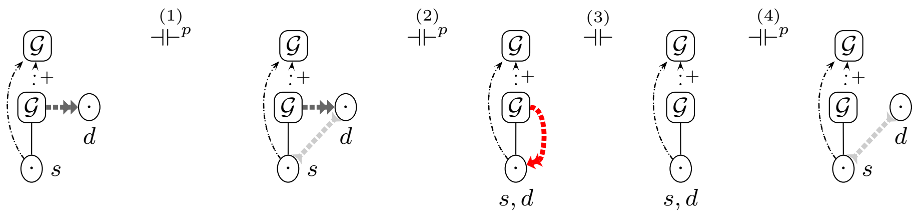
A degenerate type scheme contains no polymorphism, as witnessed by the fact that the fresh node created during its expansion must be unified with pre-existing structure (Figure 10.1.1). An instantiation edge for a degenerate type scheme is itself degenerate, in the sense that it is equivalent to a unification edge.

Definition 11.3.1 (*Elimination of degenerate instantiation edges*) Let χ be a constraint, $g \overset{i}{\twoheadrightarrow} d$ be an instantiation edge e of χ such that $\langle g \cdot i \rangle$ is degenerate in χ. We call Inst-Elim-Mono the rule removing e from χ and replacing it by a unification edge $\langle g \cdot i \rangle \leftrightarrow d$. ■

▶ **Example** A graphical depiction of Inst-Elim-Mono on the case where g has a single successor is given Figure 11.3.1.

Lemma 11.3.2 *Rule* Inst-Elim-Mono *preserves solutions.* □

Proof: The proof steps are given below for the case $\mathsf{arity}(\chi(g)) = 1$; the generalization to greater arities is immediate.



Step (1) is by equality of presolutions. Indeed, s is degenerate in any instance of both constraints. Thus, the only way to solve the instantiation edge is by merging n and d: in all the expansions, d will have to be merged with s^c, which itself must be merged with s.

Step (2) is by unification (Lemma 11.2.3).

Step (3) is by equality of presolutions. In all the presolutions, s and d are merged, hence the instantiation edge is not constraining.

Step (4) is the inverse operation of step (2); the presence or the absence of the instantiation edge does not change the execution of the unification algorithm.

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

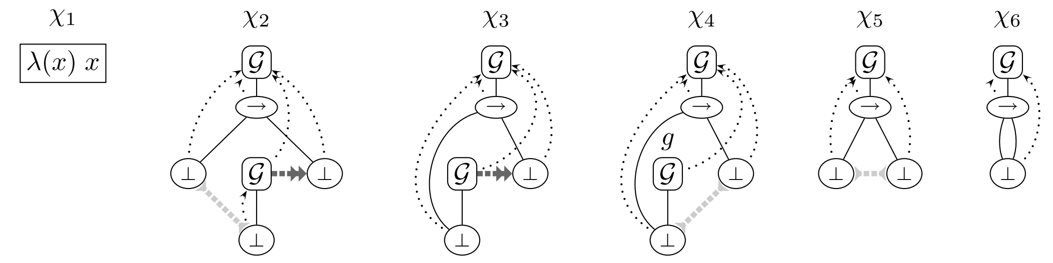


Figure 11.3.2 – Typing $\lambda(x)\ x$

► **Example: typing of the identity function** Figure 11.3.2 presents a detailed solving of the typing constraint for the identity function. On such a simple example, all steps are valid in both ML and ML$^{\mathsf{F}}$. The first step (from χ_2 to χ_3) is by unification. χ_4 is by Inst-Elim-Mono on the instantiation edge. χ_5 is by Exists-Elim on g (the only node removed being g itself). χ_6 is by unification. Hence all the solutions are witnessed by χ_6, and the meaning of all the constraints of this figure is the set of instances of the type τ_{id} equal to $\forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha$.

► **Another example** We can now prove that the constraint χ of Figure 10.2.1 is equivalent to the typing constraint χ_K for $\lambda(x)\ \lambda(y)\ x$ presented in the same figure. Let us call g the lowermost gen node, which corresponds to the occurrence of x in $\lambda(y)\ x$. Moreover, let e be the instantiation edge on g. The constraint χ is obtained from χ_K by successively:

1. solving by unification the constraint edge on the node $\langle 11\rangle$;
2. performing Inst-Elim-Mono on e, as $\langle g1\rangle$ is degenerate after the unification;
3. existentially eliminating g (whose interior is reduced to $\{g\}$);
4. performing an inverse unification step.

Thus the equivalence is by Lemmas 11.2.3, 11.3.2 and 11.1.6.

11.4 Eager propagation

A crucial property of $g\mathsf{ML}^\mathsf{F}$ is the fact that expansion and propagation are essentially monotonic w.r.t. to instance. Indeed, while the property $\chi \sqsubseteq \chi' \implies \chi^e \sqsubseteq \chi'^e$ does not hold, if $\mathcal{U}(\chi^e)$ and $\mathcal{U}(\chi'^e)$ are the constraints resulting from solving the unification edges generated by the propagation in χ^e and χ'^e, $\chi \sqsubseteq \chi' \implies \mathcal{U}(\chi^e) \sqsubseteq \mathcal{U}(\chi'^e)$ does hold. We prove a slightly weaker version of this monotony property below.

Lemma 11.4.1 *Consider a constraint χ' such that $\chi \sqsubseteq_1 \chi'$, and e an instantiation edge of χ. Then $\chi'^e \Vdash^p \chi^e$.* □

Proof: Let e be $g \overset{i}{\dashrightarrow} d$, s be $\langle g \cdot i \rangle$, and N be $\mathcal{I}^s(g) \cap (s \xrightarrow{\pm} \circ)$. We proceed by case analysis on the operation o such that $\chi' = o(\chi)$.

In all but one case, we can write χ'^e as $I(\chi^e)$, as shown below. Those equalities are actually quite simple: either o changes N and we duplicate the operation (except if o involves s itself, in which case the operation might have no effect in the expanded nodes), or o and the expansion commute.

- ▷ *Case* $o = \mathsf{Graft}(\tau, n)$: if $n \in N$, then $\chi'^e = (o\,;\mathsf{Graft}(\tau, n^c))(\chi^e)$; else $\chi'^e = o(\chi^e)$
- ▷ *Case* $o = \mathsf{Weaken}(n)$: if $n \in N$ and $n \neq s$ then $\chi'^e = (o\,;\mathsf{Weaken}(n^c))(\chi^e)$; otherwise $\chi'^e = o(\chi^e)$. (The second hypothesis handles flag reset.)
- ▷ *Case* $o = \mathsf{Merge}(n_1, n_2)$:
  - ∘ If $n_1, n_2 \in N$: $\chi'^e = (o\,;\mathsf{Merge}({n_1}^c, {n_2}^c))(\chi^e)$
  - ∘ If $n_1 \in N$, $n_2 \notin n$ (or the symmetrical case): necessarily $n_1 \longrightarrow g$ and $n_2 \longrightarrow g$. The nodes under s are unchanged, and $\chi'^e = o(\chi^e)$.
  - ∘ If $n_1, n_2 \notin N$: $\chi'^e = o(\chi^e)$.
- ▷ *Case* $o = \mathsf{Raise}(d)$:
  - ∘ If $d \notin N$: then $\chi'^e = (\mathsf{Raise}(d);\mathsf{Raise}(F^c);\mathsf{Raise}(s^c))(\chi^e)$ where $F = \mathcal{F}^s(g) \cap (s \xrightarrow{\pm} \circ)$.
  - ∘ If $d \in N$: by well-formedness of instantiation edges, $d \longrightarrow g$ must hold. Then $\chi'^e = (\mathsf{Raise}(d)\ ;\mathsf{Raise}(F^c)\ ;\mathsf{Raise}(s^c)\ ;\mathsf{Raise}(d^c)\ ;\mathsf{Merge}(d^c, d))(\chi^e)$ with the same notations as above.
- ▷ *Case* $o = \mathsf{Raise}(n)$, with $n \neq d$:
  - ∘ If $n \in N$ and $n \not\longrightarrow g$ and $\neg(n \longrightarrow s \longrightarrow g)$: $\chi'^e = (o\,;\mathsf{Raise}(n^c))(\chi^e)$.
  - ∘ If $n \notin N$, or $n \in I \wedge n \longrightarrow s \longrightarrow g$: $\chi'^e = o(\chi^e)$. (The second hypothesis handles the case of a binding reset.)
  - ∘ If $n \in N$ and $n \longrightarrow g$: this is the non-trivial case, which is treated at the end of this proof.

All the nodes transformed in the instance operations I that prove $\chi^e = I(\chi'^e)$ have enough permissions to be transformed, either because they are copies of nodes already transformed by $\chi \sqsubseteq \chi'$ (and a copy has always more permissions in the expansion than the original node), or because they are bound on a gen node, hence non-red. Thus the relation $\chi^e \sqsubseteq \chi'^e$ holds, which implies the result.

Let us come back to the remaining case, *i.e.* $o = \mathsf{Raise}(n)$, $n \in N$ and $n \longrightarrow g$. In χ, n is in the interior, while it is in the exterior in χ'. In order to show that $\chi'^e = I(\chi)$, we would need to merge the copied nodes under n^c in χ^e with the nodes under n. However, this cannot be done if n has not been raised high enough.

Thus we proceed otherwise. Consider χ^e. The node n^c can be raised in this constraint: it is raisable, as n was raisable in χ, and n^c is not red as it is bound on the root of the expansion. We call χ'' the constraint $\mathsf{Raise}(n)(\chi^e)$ in which we add an unification edge u from n^c to n. This edge is admissible, as $\hat{n^c}$ is a gen node in χ''. We have $\chi'' \Vdash^p \chi^e$ by dropping of constraints and instance (**1**).

Notice next that n is in the structural frontier of g in χ'. Thus, by definition of expansion, the only difference between χ'^e and χ'' is that there are some nodes under n^c in χ''. In fact, since n^c is a green bottom node in χ'^e and the nodes under n and n^c are the same in χ'', unifying u in χ'' and χ'^e results in the same constraint. Thus, by Lemma 11.2.3, we have $\chi'^e \dashv\Vdash^p \chi''$. Together with (1), this is the desired result.

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

Lemma 11.4.2 *Propagation preserves presolutions.* □

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

Consider a presolution χ_p of χ. We must show that χ_p is a presolution of χ^e. Since χ_p is a presolution, it suffices to show that $\chi^e \sqsubseteq \chi_p$. By Lemma 11.4.1, we have $\chi_p^e \Vdash^p \chi^e$ (**1**), since $\chi \sqsubseteq \chi_p$. By definition of χ_p and of solved instantiation edges, χ_p is a presolution of χ_p^e. Thus χ_p is a presolution of χ^e by (1). This is the desired result.

Consider now a presolution χ'_p of χ^e. We must show it is a presolution of χ. Since it is a presolution, it suffices to show that $\chi \sqsubseteq \chi'_p$. By definition of χ'_p, all the unification edges of χ^e are solved in χ'_p, in particular those resulting from the expansion. All the existential nodes introduced by the propagation are thus merged in χ'_p, and $\chi \sqsubseteq \chi'_p$ holds (a derivation of this fact can be obtained by removing all the operations on a node of the expansion in the derivation $\chi^e \sqsubseteq \chi'_p$).

This result is of course essential to reduce type inference to propagation (*i.e.* type scheme instance) and unification. We have shown here that it holds for $\sqsubseteq$, hence for $g\mathsf{ML^F}$. However, as we will see in §13.3, this is not the case for $\sqsubseteq^{\boxminus}$, hence for $i\mathsf{ML^F}$—in which type inference is undecidable.

11.5 Normalized expansion solving

In this section, we consider an instantiation edge e of a presolution χ_p equal to $g \dashrightarrow\!\!\!\!\twoheadrightarrow d$, and r the root of the type resulting from the expansion of e.

Inside a presolution, the instance steps needed to solve the propagation of an instantiation edge are very specific. Indeed, since the constraint is by hypothesis solved, the nodes outside of the interior of the expanded nodes cannot be changed. We define a set of instance operations that reflect this property.

Definition 11.5.1 (*Normalized instance derivation*) We say that an instance $\chi \sqsubseteq \chi'$ (for a constraint χ such that $\chi_p^e \sqsubseteq \chi$ holds) is *normalized* if it can be decomposed into subsequences of the form

1. $\mathsf{Graft}(\tau, n)$ or $\mathsf{Weaken}(n)$ with n in $\mathcal{I}^s(r)$;
2. $\mathsf{Merge}(n_1, n_2)$ with n_1 and n_2 in $\mathcal{I}^s(r)$;
3. $\mathsf{Raise}(n)$ with $n \overset{+}{\longrightarrow}\!\triangleright\!\longrightarrow\!\triangleright\, r$;
4. a sequence $(\mathsf{Raise}(n))^k$; $\mathsf{Merge}(n, n')$, with $n \in \mathcal{I}^s(r)$ and $n' \notin \mathcal{I}^s(r)$. We write those operations $\mathsf{RaiseMerge}(n, n')$. ■

The first three kinds of operations preserve the interior of r. Conversely, in an operation $\mathsf{RaiseMerge}(n, n')$, n leaves the interior of r and is merged atomically with some preexisting structure of the exterior of r.

In part III of this document, we will also use the fact the weakenings in such a derivation are done as late as possible, so that nodes remain green as long as possible.

Definition 11.5.2 (*Delayed weakenings*) A normalized instance operation is said to transform n if it is either $\mathsf{Graft}(\tau, n)$, $\mathsf{Raise}(n)$, $\mathsf{Weaken}(n)$, $\mathsf{Merge}(n, n')$, $\mathsf{Merge}(n', n)$ or $\mathsf{RaiseMerge}(n, n')$. The weakenings in a normalized instance derivation $o_1 ; \ldots ; o_k$ are said to be delayed if

$$\forall i, \forall j > i,\ o_i = \mathsf{Weaken}(n) \wedge o_j \text{ transforms } n' \implies \neg(n' \overset{+}{\longrightarrow}\!\triangleright\, n) \qquad ■$$

Given a derivation of $\chi_p^e \sqsubseteq \chi_p$, we cannot always assume it is normalized. Indeed, the instance operations on the nodes of the expansion corresponding to the frontier of g do not respect the conditions of Definition 11.5.1. However, it is always possible to perform those steps first, and the remainder of the derivation can be normalized, as shown by the result below.

Lemma 11.5.3 *Consider a presolution χ_p, and e an instantiation edge $g \dashrightarrow d$ of χ_p. There exists instance derivations of $\chi_p^e \sqsubseteq \chi_p$ of the form $I_u ; I$ with*

- *$I_u(\chi_p^e)$ is exactly the result of solving in χ_p^e all the frontier unification edges of χ_p^e;*
- *I is normalized, and the weakenings it performs are delayed.* □

Proof: We let χ be the constraint obtained by unifying all the unification edges from the frontier in χ_p^e. Since χ_p is a unifier of those edges (as $\chi_p^e \sqsubseteq \chi_p$ holds), we have $\chi_p^e \sqsubseteq \chi \sqsubseteq \chi_p$ by principality of unification.

Let χ be $I_u(\chi_p^e)$. Let r be the root of the expansion in this graph. Given a node n under r, we write r^d the corresponding node under d, *i.e.* the unique node of χ_p in which r is merged by the derivation $\chi_p^e \sqsubseteq \chi_p$. We build I by noetherian induction on $\chi \sqsubseteq\!|_{\chi_p} \chi_p$. If χ is not χ_p, we create a constraint χ' such that $\chi \sqsubseteq \chi' \sqsubseteq \chi_p$ **(1)** with $\chi \neq \chi'$, and conclude by induction hypothesis applied to χ'.

1: if there exists n such that $\chi(n) \neq \chi(n^d)$, then necessarily $\chi(n) = \bot$. We let χ' be $\mathsf{Graft}(\tau, n^d)(\chi)$, with τ being the constructor type for $\chi(n^d)$. We have (1) by Lemma 6.6.1

1': let N be the nodes such that for $n \in N$, $n \longrightarrow\!\triangleright\!\overset{+}{\longrightarrow}\!\triangleright\, r$, and n is not bound on $\hat{\chi_p}(n)$ in χ. We choose a node n in N as low as possible for $\multimap$. It is routine to prove that n is raisable (as otherwise χ_p would not be well-dominated). Necessarily n cannot be red, as otherwise $\chi \sqsubseteq \chi_p$ could not hold. Thus we have (1) by Lemma 6.6.2.

1'': if there exists two nodes n_1 and n_2 such that $n_1, n_2 \longrightarrow n \xrightarrow{*} r$, with n_1 and n_2 mergeable, and n_1^d and n_2^d merged, we let χ' be $\mathsf{Merge}(n_1, n_2)(\chi)$. As above, n_1 and n_2 cannot be red, and (1) holds by Lemma 6.6.3.

2: if none of the conditions above are verified, let N be the nodes such that for $n \in N$, we have $n \neq r$, $\mathring{\chi}(n) = (\geqslant)$ and $\mathring{\chi}(n^d) = (=)$. We choose $n \in N$ as low as possible for $\longleftarrow$ (or equivalently $\multimap$) and let χ' be $\mathsf{Weaken}(n)(\chi)$. Again, n cannot be red, as $\chi \sqsubseteq \chi_p$ would not hold. At this stage, the interiors of n and n' are identical (although the frontiers might not be). We thus have (1) by Lemma 6.6.4.

3: once we have exhaustively applied the cases above, the only differences between the subgraphs under d and r is the fact that, for some nodes bound on r, the corresponding nodes are bound above d, and possibly also the binding flag of r. Importantly, the bounds of the corresponding nodes under d and under r are exactly the same. Thus, let N be the set of nodes such that if $n \in N$, we have $n \longrightarrow r$ and $n^d \not\longrightarrow d$. We choose $n \in N$ as low as possible for $\multimap$; as usual, this ensures that n is raisable. Then we let χ' be $\mathsf{RaiseMerge}(n, n^d)(\chi)$, and we have (1) by Lemmas 6.6.2 and 6.6.3.

4: once all the nodes of the case above have been raised, n and d can be merged. Thus, if d is rigid we let χ' be $(\mathsf{Weaken}(r); \mathsf{RaiseMerge}(r, d))(\chi)$; otherwise we let χ' be $\mathsf{RaiseMerge}(r, d)(\chi)$. (There is in fact no need to raise r, but our use of $\mathsf{RaiseMerge}$ avoids the need to introduce a specific merging operation for r.) By construction, χ' is actually χ_p.

The operations above define I, and it remains to justify that all the weakenings it contains are delayed. The weakenings of step 2 are inserted bottom-up, hence delayed. The operations we insert after step 2 concern r or some nodes bound on r; those nodes are above or at the same level as the nodes that are weakened in step 2. Thus the result holds.

11.6 Stability of solved instantiation edges

Another important reasoning tool is the stability of solved edges by unrelated transformations. Consider indeed a type scheme s. Expansion is only concerned with the nodes that are both under s and in the structural interior of the corresponding gen node; a transformation that does not change those nodes leaves the expansion unchanged. We can in fact lift this property to propagation, and by extension, to solved instantiation edges.

Lemma 11.6.1 *An instantiation edge $g \overset{i}{\dashrightarrow} d$ that is solved in a constraint χ remains solved in any instance of χ that leaves $\mathcal{I}^s(g) \cap (s \overset{\pm}{\multimap})$ unchanged.* □

Proof: We let e be $g \dashrightarrow d$, N be $\mathcal{I}^s(g) \cap (s \overset{\pm}{\multimap})$ and r be the root of the expansion in χ^e and χ'^e. It suffices to show the result of one single step of a canonical instance derivation, as the result then follows by induction. We thus consider an operation o such that $\chi \sqsubseteq_1 \chi'$ and $\chi' = o(\chi)$, and show that $\chi'^e \sqsubseteq \chi'$. There are two cases, depending on whether o is $\mathsf{Raise}(d)$ or not.

▷ Case $o \neq \mathsf{Raise}(d)$: we consider a derivation of $\chi^e \sqsubseteq \chi$ of the form I_u ; I as per Lemma 11.5.3. By hypothesis, I is normalized and we modify it into I' to prove that I'_u ; I' witnesses $\chi'^e \sqsubseteq \chi'$ instead (where I'_u is simply obtained by solving the frontier unification edges in $o(\chi)^e$; this always succeeds, as the instantiated nodes are green bottom nodes). The first three kinds of operation of Definition 11.5.1 need not be changed. Indeed they operate on $\mathcal{I}^s(r)$, which is the same in χ^e and χ'^e by hypothesis. Thus we consider an operation $o' = \mathsf{RaiseMerge}(n, n')$ in the normalized derivation, and change o' when it modifies the node on which o operates.

- *Case* $o = \mathsf{Graft}(\tau, n')$: we rewrite o' into $\mathsf{Graft}(\tau, n)$; $\mathsf{RaiseMerge}(n, n')$
- *Case* $o = \mathsf{Weaken}(n')$: we rewrite o' into $\mathsf{Weaken}(n)$; $\mathsf{RaiseMerge}(n, n')$
- *Case* $o = \mathsf{Raise}(n')$: we do not change o'. Indeed, there exists k such that $o' = ((\mathsf{Raise}(n))^k ; \mathsf{Merge}(n, n'))$, and after o we must instead apply $((\mathsf{Raise}(n))^{k+1} ; \mathsf{Merge}(n, n'))$; however this is still written $\mathsf{RaiseMerge}(n, n')$.
- In all the other cases: we do not change o'.

▷ *Case* $o = \mathsf{Raise}(d)$: then $\chi'^e = (\mathsf{Raise}(d) ; \mathsf{Raise}(F^c) ; \mathsf{Raise}(r))(\chi^e)$, where $F = \mathcal{F}^s(g) \cap (s \overset{\pm}{\multimap})$ and r is the root of the expanded part. The nodes in F^c and r are flexibly bound to the root, hence not red. Thus $\chi^e \sqsubseteq^R \chi'^e$ (**1**).
Let us next prove that all nodes of F are bound strictly above $\hat{d}$ in χ (**2**). Consider $n \in F$. Let π be such that $s \overset{\pi}{\multimap} n$. Since e is solved, $d \overset{\pi}{\multimap} n$ also holds. By well-domination, all nodes of F must be bound at least as high as d. But, since d can be raised in χ, it is raisable and no node structurally under d can have been bound on $\hat{d}$ in χ. This proves the desired subresult.
Let us now conclude. We have $\chi^e \sqsubseteq \chi'$ (**3**), as both $\chi^e \sqsubseteq \chi$ and $\chi \sqsubseteq_1 \chi'$ hold by hypothesis. The nodes d and r are bound lower in χ^e than in χ'^e, as d is raised in χ. It is also the case for the nodes of F^c by (2). By applying Lemma 6.6.2 $|F| + 2$ times to (3) and (1), we obtain $\chi'^e \sqsubseteq \chi'$.

12

Type inference in ML^{F}

Abstract

In this section, we show how to solve typing constraints, hence how to perform type inference in ML^{F}. We restrict our attention to a subset of constraints, called *acyclic*, as type inference for the full set of constraints is undecidable (§12.1.1). We show that acyclic constraints, which include all typing constraints, have principal decidable presolutions and solutions (§12.1.3). We deduce from the strategy used to find the principal presolution some simplification rules on acyclic constraints (§12.2). We also show that the set of terms typable without type annotation in ML^{F} and ML is the same (§12.3.1), and formalize how type annotations are added to source terms—thus giving ML^{F} its expressivity (§12.3.2). We identify superfluous gen nodes inside typing constraints, and propose some rules to remove them (§12.4). Finally, we study the complexity of type inference (§12.5); our constraint framework can be used to perform type inference with optimal theoretical complexity, and with excellent practical complexity. We conclude by discussing implementation-related issues (§12.6).

12.1 Solving acyclic constraints

12.1.1 Acyclic constraints

In their full generality, our constraints system can be used to encode typing problems with polymorphic recursion, for which the typing problem is already undecidable in ML (Henglein 1993). Indeed, the constraint corresponding to a term such as «$\mathsf{val\ rec}\ x = a$» is simply obtained by building the typing constraint χ for a, and constraining all the occurrences of x in a to be an instance of χ itself (using an instantiation edge).

Thus, in order to preserve decidability, we need to restrict ourselves to a subset of constraints. In order to rule out all cases—direct or indirect—of polymorphic recursion, we only consider constraints in which instantiation edges induce an *acyclic* relation.

Definition 12.1.1 (*Acyclic constraints)* A gen node g *depends* on another gen node g' if g' is in the scope of g, or if g' constrains the constraint interior of g, *i.e.*

$$g' \xrightarrow{+} g \;\vee\; \exists n \in \mathcal{I}^c(g),\, g' \dashrightarrow n$$

A constraint χ is *acyclic* if the dependency relation on its gen nodes is acyclic, *i.e.* a strict partial order. ■

Typing constraints are acyclic: the instantiation edges follow the scopes of the variables of the expression, which are themselves nested.

Property 12.1.2 *A typing constraint χ is acyclic.* □

Proof: Let a be a term such that χ is its typing constraint. Let $\mathcal{R}_1$ be the relation "is a subterm" and $\mathcal{R}_2$ be the relation "is in the scope of", both relations being restricted to subterms of a. The relations $\mathcal{R}_1$, $\mathcal{R}_1^{-1}$, $\mathcal{R}_2$ and $\mathcal{R}_2^{-1}$ are well-founded and finite. Thus, so is the transitive closure of $\mathcal{R}_1 \times \mathcal{R}_2^{-1}$ (**1**).

Consider two gen nodes g and g' such that g depends on g'. We identify gen nodes with the corresponding subterms of a. By definition of typing constraints and dependency, either (1) g' is a subterm of g (for abstractions or applications), or (2) g' is the bound of a let bound variable and g is the right-hand side of the let or a variable of this right-hand side. Thus the dependency relation is a subrelation of $\mathcal{R}_1 \times \mathcal{R}_2^{-1}$, and the conclusion is by (1).

Crucially, acyclicity is stable by instantiation.

Property 12.1.3 *Let χ and χ' be two constraints. If $\chi \sqsubseteq \chi'$, the dependency relation of χ' is a subrelation of the dependency relation of χ.* □

Proof: We show the property for an atomic instantiation step; the general case follows by induction. Since gen nodes cannot be raised, we only need to consider the dependency relation induced by instantiation edges.

- ▷ Weakening: The relation remains completely unchanged.
- ▷ Grafting: The interiors of gen nodes can grow, but with new nodes that do not contain instantiation edges. Hence, the relation remains unchanged.
- ▷ Raising: The interiors of gen nodes may only decrease: if the node raised is bound on a gen node, its subgraph leaves the interior of this node, otherwise all gen nodes interiors are left unchanged. Hence the relation diminishes or remains unchanged.
- ▷ Merging: The interiors of gen nodes are left unchanged, as gen nodes cannot be merged. Thus merging can only occur *inside* the interior of gen nodes, and the relation remains unchanged.

12.1.2 Solving an instantiation edge

In acyclic constraints, propagating an instantiation edge and solving the resulting unification edges is sufficient to solve the instantiation edge. Of course, this result is the key to reducing type inference to unification, exactly as for ML type inference.

Lemma 12.1.4 *Let e be an instantiation edge $g \dashrightarrow\!\!\!\!\twoheadrightarrow d$ of a constraint χ, with d not in $\mathcal{I}^c(g)$. Let χ' be the principal unifier of the unification edges introduced in χ^e (if this unifier exists). Then χ' is an instance of χ in which e is solved.* □

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

<ins>Proof</ins>: Let E be the set of unification edges resulting from the propagation. We prove the three points below, the first and the third being the result.

1. <ins>χ' is an instance of χ</ins>: The existential structure introduced during the propagation is merged with the existing structure under d when the unification edges are solved.
2. <ins>$\mathcal{I}^c(g)$ has not been instantiated in χ'</ins>: by definition of the unification algorithm, the nodes changed by the unification are those reachable by $\xrightarrow{*}\!\circ$ from a node constrained by an edge in E. These nodes are either fresh (*i.e.* created by the propagation), under d, or under a node in the frontier (hence the exterior) of g. Hence, it suffices to prove that no node structurally under d is in $\mathcal{I}^c(g)$.
   By contradiction, suppose there exists n such that $n \in \mathcal{I}^c(g)$ and $d \xrightarrow{*}\!\circ\, n$. By well-domination, since $n \xrightarrow{+}\!\triangleright\, g$, either $g \xrightarrow{*}\!\triangleright\, d$, or $d \xrightarrow{*}\!\triangleright\, g$. The second case is forbidden by the hypothesis on d and g. The first case is impossible because d is a type node, and g a gen node.
3. <ins>e is solved in χ'</ins>: we show that the diagram below holds:

$$\begin{tikzcd}
\chi \arrow[r, "e"] & \chi^e \arrow[r, "e"] \arrow[d, "\mathsf{Unif}"'] & \chi^{ee} \arrow[l, bend right, "\sqsubseteq_1^M"'] \arrow[d, dashed, "\mathsf{Unif}"'] \\
& \chi' \arrow[r, "e"] & \chi'^e \arrow[l, bend left, dashed, "\sqsubseteq"]
\end{tikzcd}$$

   Let us first justify the edge $\chi^{ee} \overset{\mathsf{Unif}}{\longrightarrow} \chi'^e$. Propagating e a second time in χ^e is similar to doing an existential introduction and adding some unification edges. The new nodes and edges are not considered by the unification algorithm, so the first expanded nodes are merged exactly as in χ'. The square closes because the interior of g have not changed, so the propagation creates the same nodes in χ' and χ^e.
   Next, $\chi^{ee} \sqsubseteq^M \chi^e$ holds, as it is possible to merge the copies of the nodes of the structural frontier, then the root of the two propagations. By soundness of unification, this implies $\chi^{ee} \sqsubseteq \chi'$. Hence χ' is an unifier of the edge introduced by the second propagation in χ^{ee}. by Lemma 11.2.3, this means that it is an instance of the principal unifier of this edge, which is χ'^e. (Indeed, the two propagated parts being equal, solving one propagation edge or the other yields the same graph.) Hence the relation $\chi'^e \sqsubseteq \chi'$ holds, which is exactly the desired result.

Notice that this result does not require the acyclicity of the constraint itself, only that gen nodes are not "self-cyclic".

12.1.3 Solving an acyclic constraint

Using the result above, solving an acyclic constraint is immediate: we merely need to solve the instantiation edges in the correct order, *i.e.* according to the dependencies between gen nodes. The resulting algorithm SolveConstraint is shown in Figure 12.1.1.

Input: an acyclic constraint χ
Output: a presolution of χ

1. Compute an ordering for the instantiation edges of χ such that $g \dashrightarrow d$ is visited before $g' \dashrightarrow d'$ if d' depends on d in χ. This order can be found by a topological sort of χ.
2. Solve all the unification edges by unification.
3. Visit the instantiation edges according to the order found in step 1. On each edge e:
   a) perform a propagation on e;
   b) unify the resulting unification edges.

Figure 12.1.1 – SolveConstraint algorithm

In order to show that this algorithm indeed solves the constraint, we introduce one auxiliary definition.

Definition 12.1.5 A gen node g is *recursively-solved* if its interior is not the target of a unification edge and for any edge $e = g' \dashrightarrow d$ with $d \in \mathcal{I}^c(g)$, e is solved and g' is recursively solved. ■

Recursively solved gen nodes are completely solved: as shown in the proof below, their interiors will remain invariant throughout the traversal of the instantiation edges by SolveConstraint.

Theorem 12.1.6 *Given an acyclic constraint χ, the algorithm SolveConstraint returns a principal presolution of χ if χ has solutions, or fail if χ has no solutions.* □

<u>Proof</u>: The preservation of presolutions follows from Lemma 11.2.3 for steps 2 and 3b and from Lemma 11.4.2 for step 3a. Hence we only need to show that the type χ_p returned by the algorithm if it does not fail is indeed a presolution of χ.

By step 2, χ_p only contains solved unification edges (**1**). We prove by induction on the iterations of step 2 that all the instantiation edges of χ_p are solved. The invariant is that all the already visited instantiation edges are solved in the current constraint, and that their originating gen node is recursively solved (**2**). If all the edges have been visited, χ_p is solved by (1) and (2).

Otherwise, let χ' be the current constraint at the beginning of an iteration of step 2, $e = g \dashrightarrow d$ be the current instantiation edge to solve, E the edges visited before e, G their gen nodes. By (1), the induction hypothesis, and the definition of the order used to choose e, g is recursively solved in χ'. Let χ'' be the constraint obtained by propagating e and solving the resulting unification edges (by construction this step must succeed, as we consider only the cases where SolveConstraint returns). By Lemma 12.1.4, e is solved in χ''; it remains to prove that g is still recursively solved, that the gen nodes of G are still recursively-solved, and that the edges of E are still solved. To do so, we prove that the constraint interiors of the gen nodes of $G \cup \{g\}$ are unchanged in χ'', which implies the result by Lemma 11.6.1.

By definition of the unification algorithm, the nodes of χ changed by the propagation and the unification are those structurally under d, or under the structural frontier of g. By well-domination, the latter nodes can only be in the interior of gen nodes on which g is transitively bound, which thus depend on g in χ'' (**3**). Let now g_d be the gen node on which d is bound (by condition 10 of Definition 9.2.1). By definition, g' depends on g_d. The nodes under d are also in $\mathcal{I}^c(g_d)$, or on one of the gen node on which g_d is transitively bound, which all depend on g_d (**4**). Thus, by (3) and (4), the nodes changed are only in interiors of gen node which depend on g in χ''. By Property 12.1.3, those nodes also depend on g in χ. Hence, they are not in $G \cup \{g\}$ by acyclicity, which is the desired result.

As an immediate corollary, we can deduce that an acyclic constraint has a principal solution, which is the solution witnessed by its principal presolution.

Corollary 12.1.7 *Acyclic constraints have principal decidable (pre)solutions.* □

Interestingly, the constraint returned by SolveConstraint does not depend on the ordering chosen to enumerate the instantiation edges. Indeed, given two possible constraints returned by the algorithm, they are both a principal presolution of χ, and verify $\chi_p \sqsubseteq \chi'_p$ and $\chi'_p \sqsubseteq \chi_p$. The kernel of instance is equality (Lemma 5.3.11), which proves that they are in fact equal.

Efficiency Using the order induced by the dependency relation ensures that an instantiation edge never needs to be propagated more than once. Hence, the number of unification steps that must be performed during inference is bounded by the number of instantiation edges plus one (for the initial unification edges). A formal study of the complexity of this algorithm on typing constraints will be given in §12.5.

12.2 Simplifying acyclic constraints

The proof of correctness of SolveConstraint (Theorem 12.1.6) gives us some insights on the shape of the principal presolution of an acyclic constraint. We present three important consequences in the next sections.

12.2.1 Removing solved instantiation edges

Once a gen node g is recursively solved, its interior will never need to be instantiated more to obtain the principal presolution. Thus, after an outgoing instantiation edge of g has been propagated, Lemmas 11.6.1 and 12.1.4 ensure that this edge can be soundly removed.

Corollary 12.2.1 *Consider a recursively solved gen node g of an acyclic constraint χ. For any edge $e = g \dashrightarrow\!\!\!\!\twoheadrightarrow d$, the constraints χ and $\chi^e \setminus \{e\}$ are equivalent.* □

In particular, this holds for unconstrained gen nodes, which are trivially recursively solved.

Corollary 12.2.2 *Let e be an edge $g \twoheadrightarrow n$ of an acyclic constraint χ. If $\mathcal{I}^c(g)$ is not the target of a constraint edge, then χ and $\chi^e \setminus \{e\}$ are equivalent. Under those hypotheses, we call* Inst-Expand *the rule replacing χ by $\chi^e \setminus \{e\}$.* □

Interestingly, this gives us another proof of the correctness of rule Inst-Elim-Mono. However, Lemma 11.3.2 is more general, as it does not require the constraint to be acyclic.

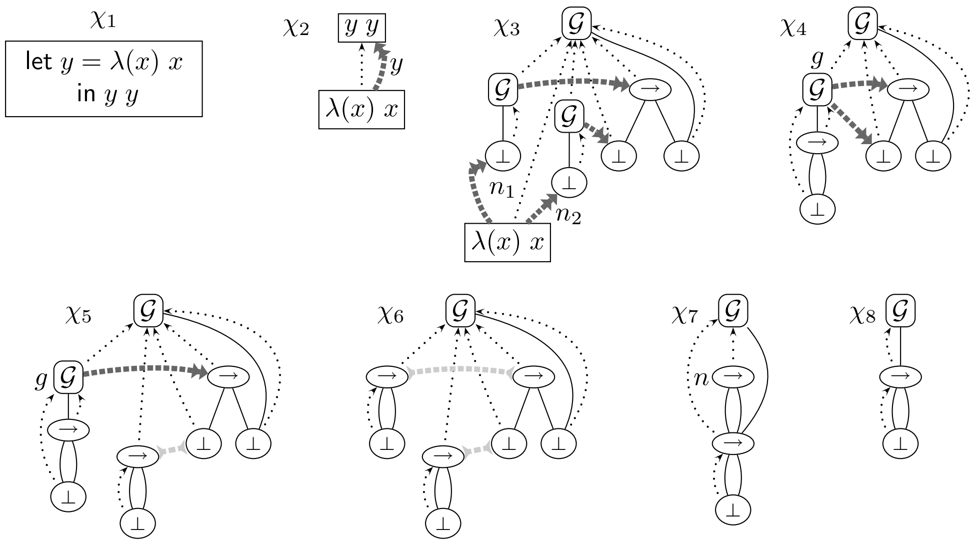


Figure 12.2.1 – Typing let $y = \lambda(x)\ x$ in $y\ y$

▶ **Example** Figure 12.2.1 presents the typing of let $y = \lambda(x)\ x$ in $y\ y$ in MLF. In χ_3 we have developed the expression node for $y\ y$. In χ_4 we have solved the box for $\lambda(x)\ x$ as in Figure 11.3.2, and applied the rule Var-Let (which will be presented in §12.4) to both n_1 and n_2. χ_5 is by Inst-Expand on the lowermost instantiation edge. χ_6 is by Inst-Expand on the remaining instantiation edge, then by Exists-Elim on g. χ_7 is by unification and χ_8 by Exists-Elim on n. Thus the principal solution is the type τ_{id}. The derivation is essentially the same in ML, except that all the nodes not under g are bound at $\langle\epsilon\rangle$ in χ_5 to χ_8.

Efficiency of constraint solving Often, we are not interested in the presolutions of a constraint—only in its solutions. Finding the principal presolution then deducing the principal solution can be costly memory-wise, as the former can be arbitrarily bigger than the latter. Indeed, presolutions retain the shape of the initial constraint. Hence a better approach is to apply Inst-Expand to perform the propagation, then existential elimination to the gen nodes that are no longer constrained. While this does not change time complexity, it ensures that constraints remain as small as possible and can improve space complexity.

12.2.2 Solving closed subconstraints

Since solving acyclic constraints amounts to performing unification and propagation steps, gen nodes with no escaping edges can be solved locally.

Definition 12.2.3 (*Closed subconstraint*) The subconstraint under a gen node g is said to be *closed* if

- $(n \overset{*}{\lhd\!\!-\!\!\!\circ}\ ) = \mathcal{I}^c(g)$
- $(\mathcal{I}^c(g) \rightarrowtail\!\!\dashrightarrow\!\!\leftarrowtail\ ) \cup (\mathcal{I}^c(g) \twoheadleftarrow\!\!\dashleftarrow\ ) \cup ((\mathcal{I}^c(g) \setminus \{g\} \dashrightarrow\!\!\twoheadrightarrow\ ) \subseteq \mathcal{I}^c(g)$ ■

Closed subconstraints only allow constraint edges within $\mathcal{I}^c(g)$, or outgoing instantiation edges from g itself. The typing constraints generated from closed λ-terms are closed subconstraints.

Lemma 12.2.4 *Let χ be an acyclic constraint, g a gen node of χ whose subconstraint is closed. Suppose χ is solvable, and let χ_p be its principal presolution. Then the subconstraint under g is closed in χ_p.* □

Proof: The proof is by induction on the construction of the principal presolution; the invariant is that the subconstraint under g remains closed after each step. The unification of two nodes under g will never raise a binding edge above g, by definition of a least common ancestor. The propagation of an instantiation edge from a gen node different from g will create unification edges only inside $\mathcal{I}^c(g)$, as the destination node is inside $\mathcal{I}^c(g)$. The propagation of an instantiation edge from g will not create unification edges that go outside $\mathcal{I}^c(g)$, since $\mathcal{F}^s(g)$ is empty.

12.2.3 Splitting gen nodes

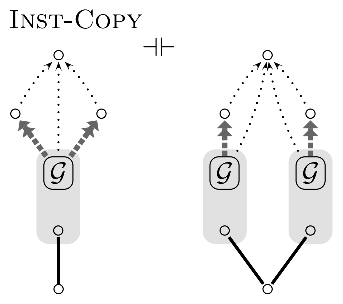


Figure 12.2.2 – Rule Inst-Copy

An interesting rule to consider on acyclic constraints is Inst-Copy, presented schematically in Figure 12.2.2. It can be applied whenever a gen node has one or more outgoing instantiation edge. The edges may be arbitrarily partitioned into two sequences, and the interior nodes and edges of the gen node are duplicated (as well as the edges between the interior and the frontier).

Intuitively, one could think that the constraint in which the gen node has been split has more solutions. Indeed, each scheme could seemingly pick a different type. However, this is not the case on acyclic constraints. Indeed, since they have principal presolutions, the two schemes can only pick instances of the most general solution. We sketch the proof of this result below. However, since it is never used in the remainder of this document, we admit an intermediary result (which can be proven by a tedious case disjunction on the unification algorithm).

Lemma 12.2.5 *Let χ be a constraint, g one of its gen nodes. Let C be one application of* Inst-Copy *on g. The following holds:*

$$\begin{tikzcd}
\chi \arrow[r, "C"] \arrow[d, "\sqsubseteq_1"'] & \chi_C \arrow[d, dashed, "\sqsubseteq"] \\
\chi' \arrow[r, dashed, "C"'] & {}
\end{tikzcd}$$

□

Proof: Let g' be the name of the copy of g in χ_C. The proof is by case disjunction on the instance operation transforming χ into χ'. Each case is immediate: if the operation involves a node of $\mathcal{I}^c(g)$, we duplicate it so that the copy occurs on the corresponding node in $\mathcal{I}^c(g')$; otherwise we do not change the operation.

Lemma 12.2.6 *Let χ be a constraint, g one of its gen nodes, C an application of* Inst-Copy. *Let U be the application of* Unif *on a given unification edge e, U' the application of this algorithm to e if it is not duplicated by C, and to e and its copy otherwise. The following holds:*

$$\begin{tikzcd}
\chi \arrow[r, "C"] \arrow[d, "U"'] & \chi_C \arrow[d, dashed, "U'"] \\
\chi' \arrow[r, dashed, "C"'] & \chi'_C
\end{tikzcd}$$

□

Lemma 12.2.7 *Rule* Inst-Copy *preserves the meaning of acyclic constraints.* □

Proof (sketch): Let χ be a constraint, C an application of Inst-Copy, χ_C the constraint $C(\chi)$.

▷ Consider a presolution χ_p of χ. Let χ_{C_p} be $C(\chi_p)$. By Lemma 12.2.5, $\chi_C \sqsubseteq \chi_{C_p}$ holds. By hypothesis, χ_p is solved; hence it does not contain unification edges, and all instantiation edges are solved. Thus χ_{C_p} does not contain unification edges either. Moreover, all its instantiation edges can be solved, by using the steps that solve them in χ_p. Finally, χ_p and χ_{C_p} witness the same solutions, as C does not change the expansion of the root node. Hence all solutions of χ are solutions of χ_C.

▷ In the other direction, we cannot apply the same approach. Indeed, the two copies of g could seemingly be instantiated in different ways. Instead, we instrument the steps of $\mathsf{SolveConstraint}$ on χ to build the principal presolution of χ_C. After each atomic step of $\mathsf{SolveConstraint}$ on χ, we apply the same step to χ_C, as well as on the copy of the transformed edge if it exists. Lemma 12.2.6 ensures that after each propagation and unification step, χ and χ_C can be maintained synchronized. Thus the principal presolution of χ_C is $C(\chi_p)$, and all solutions of χ_{C_p} are solutions of χ.

12.3 Typability in annotated and unannotated $\mathsf{ML^F}$

12.3.1 Unannotated terms

The strategy for solving an acyclic constraint also gives us some insights on the expressiveness of $\mathsf{ML^F}$. Consider a typing constraint. By Property 9.4.2, it is an ML constraint; in particular it contains only flexible edges. If it is solvable in $\mathsf{ML^F}$, its principal presolution will contain only flexible edges. Then, by Theorem 10.6.3, it will have an ML solution. Thus, a program without type annotations is typable in $\mathsf{ML^F}$ if and only if it is typable in ML. (In general, its principal type in ML will however be a strict instance of its principal type in $\mathsf{ML^F}$.)

Theorem 12.3.1 *Any expression typable without type annotations in* $\mathsf{ML^F}$ *is also typable in* ML □

This result is a direct consequence of the following more general result:

Lemma 12.3.2 *Consider an* ML *acyclic constraint. It is typable in* ML *if and only if it is typable in* $\mathsf{ML^F}$. □

Proof: Let χ be the constraint. The direction "typable in ML implies typable in $\mathsf{ML^F}$" is proven in Property 10.6.2. Suppose then that χ is typable in $\mathsf{ML^F}$. By definition of $\mathsf{SolveConstraint}$, the principal presolution χ_p contains only flexible edges: unification does not introduce fresh binding edges, and propagation only copies existing subgraphs. Thus we can apply Theorem 10.6.3 to χ_p, which gives us an ML solution to χ.

12.3.2 Type annotations

Type annotations are thus the key to the expressivity of $\mathsf{ML^F}$. Conveniently, there is no need to introduce special constructs or typing rules for annotated expressions. Instead, we add to the typing environment a denumerable set of retyping, or *coercion*, functions. More precisely, to any closed type τ we associate a constant c_τ. Then $(a : \tau)$ and $\lambda(x : \tau)\ a$ are both syntactic sugar:

$$(a : \tau) \;\triangleq\; \mathsf{c}_\tau\ a \qquad\qquad \lambda(x : \tau)\ a \;\triangleq\; \lambda(x)\ \mathsf{let}\ x = (x : \tau)\ \mathsf{in}\ a$$

Notice that type annotations are part of expressions: two terms with different annotations are really different terms and do not usually have a common, most general type.

12.3.2.1 Annotations in source terms

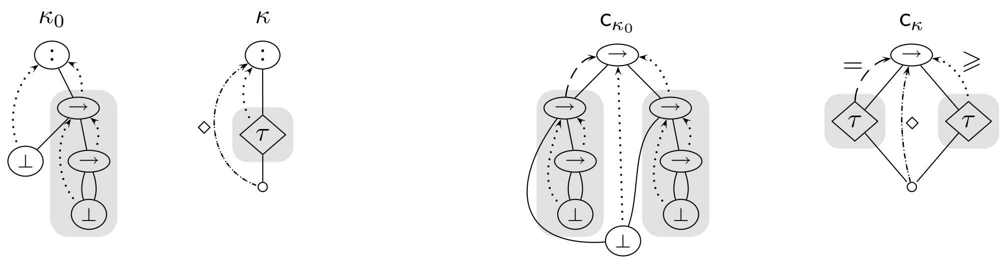


Figure 12.3.1 – Types of coercion functions

In fact, we go one step further and introduce annotations that are more general than a simple type. As an example, consider the annotation

$$(a : \exists \beta \forall (\alpha)\ \beta \to (\alpha \to \alpha))$$

It contains both *universal* and *existential* quantification, and expresses that a must be a function, the type of its first argument being left unspecified, and its return type being exactly $\alpha \to \alpha$. The existential part will be found (*i.e.* potentially instantiated) by type inference.

The annotation above can be represented by the graph κ_0 of Figure 12.3.1. The existential part is the node $\langle 11 \rangle$, which is bound at the root ":" node. Conversely, the nodes corresponding to the universal part are $\langle 1 \rangle$, $\langle 12 \rangle$ and $\langle 121 \rangle$. More general annotations are depicted by the pseudo-type κ of the same figure. Again, the universal part is the interior of $\langle 1 \rangle$. Conversely, the other nodes of κ, represented by the $\circ$ meta-node and bound on the root, are those existentially quantified.

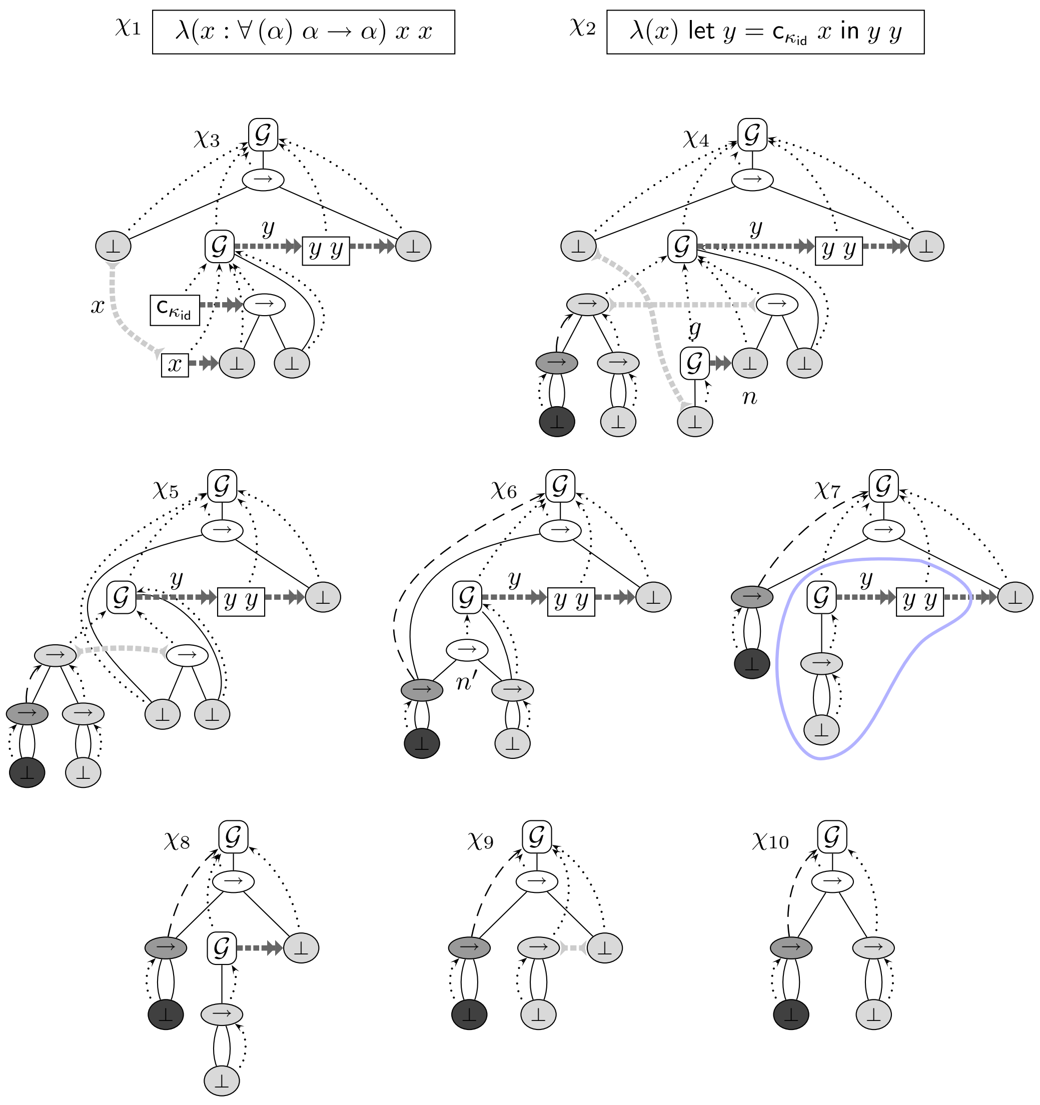
12.3.2.2 Coercion functions

We associate to an annotation κ a coercion function c_κ which has the type depicted in Figure 12.3.1. Each side of the arrow is a copy of τ; hence, they could a priori be instantiated independently. However, the domain is rigidly bound. Hence, its polymorphism is requested, and thus cannot be weakened by instantiation: a must be of type τ. On the contrary, the codomain is flexibly bound, meaning that the polymorphism is provided, and can freely instantiated. In parallel, the nodes corresponding to the existential part of κ are not duplicated: they are shared between the domain and the codomain, and will be instantiated simultaneously on both sides.

▶ **Example** In Figure 12.3.1, the type of the coercion function corresponding to κ_0 is c_{κ_0}.

▶ **Example: typing of an annotated term** Figure 12.3.2 shows the typing of the term $\lambda(x : \forall (\alpha)\ \alpha \to \alpha)\ x\ x$. In this example we have colored the nodes according to their permissions, as some nodes are orange and red.

The term we consider is desugared into the expression described in the constraint χ_2. In χ_3 we have developed the expression nodes for the abstraction, the let, and the application $\mathsf{c}_{\kappa_{\mathsf{id}}}\ x$. In χ_4 we have developed the expression node for x, and simplified by INST-EXPAND



χ_1
$\lambda(x : \forall\,(\alpha)\ \alpha \to \alpha)\ x\ x$
χ_2
$\lambda(x)$ let $y = \mathsf{c}_{\kappa_{\mathrm{id}}}\ x$ in $y\ y$
χ_3
y
$y\ y$
x
$\mathsf{c}_{\kappa_{\mathrm{id}}}$
χ_4
g
n
χ_5
χ_6
n'
χ_7
χ_8
χ_9
χ_{10}

Figure 12.3.2 – Typing $\lambda(x : \forall\,(\alpha)\ \alpha \to \alpha)\ x\ x$

and EXISTS-ELIM the instantiation edge on $\mathsf{c}_{\kappa_{\mathsf{id}}}$ into a unification one. χ_5 is by unification on $\langle g1\rangle$, INST-ELIM-MONO on g, unification on the edge introduced by INST-ELIM-MONO between $\langle g1\rangle$ and n, and existential elimination on g. χ_6 is by unification on the remaining unification edge. χ_7 is by EXISTS-ELIM on n'. Up to a few unimportant differences, the circled nodes correspond to the constraint χ_3 of Figure 12.2.1; simplifying those nodes thus results in χ_8. χ_9 is by INST-EXPAND on the instantiation edge, then by EXISTS-ELIM. χ_{10} is by unification.

The principal solution is thus the type $\forall\,(\alpha = \tau_{\mathsf{id}})\;\forall\,(\beta \geqslant \tau_{\mathsf{id}})\;\alpha \to \beta$. Using our syntactic sugar, this type is written $\tau_{\mathsf{id}} \to \tau_{\mathsf{id}}$. The leftmost occurrence of τ_{id} occurs on the right of an arrow, and can thus be instantiated.

Definition 12.3.3 (*Initial typing environment)* The initial typing environment for $\mathsf{ML^F}$ is composed of all the coercion functions c_κ for all annotations κ. ■

12.3.2.3 Soundness of coercion functions

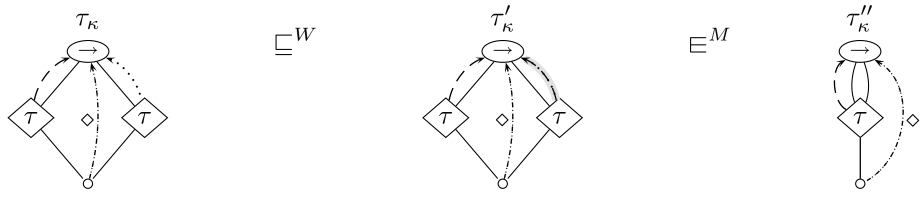


Figure 12.3.3 – Soundness of coercion functions

Let us informally justify why the coercion functions are sound. We can instantiate the type τ_κ of c_κ into the types τ'_κ and τ''_κ of Figure 12.3.3. It is immediate that the type τ''_κ is sound, as it is an instance of the type τ_{id} of the identity function. Thus, by definition of abstraction (which does not change the semantic of types), τ'_κ is also sound. Finally, τ_κ is actually the principal type of $\lambda(x)\ \mathsf{c}_\kappa\ x$, *i.e.* the η-expansion of c_κ, and is as such sound.

Interestingly, we could give to coercion functions the type τ'_κ above (instead of τ_κ) without altering the expressivity of $g\mathsf{ML^F}$. Indeed, in an application $\mathsf{c}_\kappa\ a$, the binding edge for the codomain of c_κ is reset when the gen node for the application is expanded; this can for example be seen by weakening the node $\langle n2\rangle$ in the constraint χ_6 of Figure 12.3.2.

Given the remark above, the key point in Figure 12.3.3 is the step $\tau''_\kappa \sqsupseteq \tau'_k$. While it is correct from a soundness point of view, this step is not possible in $g\mathsf{ML^F}$, as it is not part of the instance relation $\sqsubseteq$. Indeed, allowing such an operation in $\sqsubseteq$ would permit—hence require, for principality—to infer all the possible uses of coercion functions, making type inference undecidable.

12.4 Simplifying typing constraints

Mainly for homogeneity reasons, typing constraints introduce a gen node for every sub-expression. In a few cases, those nodes are superfluous, as they do not increase the expressiveness of the constraint. In this section we propose a few simplifications rules to remove them.

Importantly, even though the rules below are tailored for typing constraints, they are not restricted to them. Indeed, they can be used on any constraint, including cyclic ones.

12.4.1 Simplifying the typing of variables

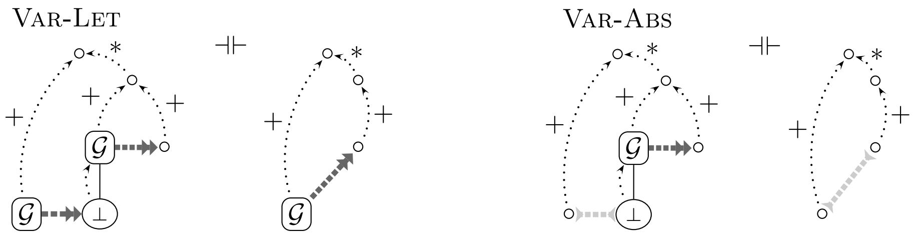


Figure 12.4.1 – Simplifying the typing of variables

The gen nodes introduced in the typing of variables are superfluous:

- let-bound variables only generate indirections. Indeed, consider the typing constraint for let $x = a$ in a', and a subconstraint for an occurrence of x in a'. This subconstraint is constrained to have an instance of the type of a, and can be entirely removed
- λ-bound variables create gen nodes that will ultimately be degenerate. Indeed, those variables can only be typed monomorphically.[1]

We introduce two simplifications rules to remove those gen nodes, presented in Figure 12.4.1, and illustrated by the following examples:

- In Figure 11.3.2, the rule VAR-ABS can be used to directly transform χ_2 into χ_5.
- In Figure 12.3.2, the step from χ_4 to χ_5 is by VAR-ABS on the gen node containing n, then by unification.
- In Figure 12.2.1, the step from χ_3 to χ_4 is by VAR-LET applied twice, on each of the instantiation edges for the constraint corresponding to $\lambda(x)\ x$.

Both rules are correct.

Lemma 12.4.1 *Rule* VAR-ABS *preserves solutions.* □

Proof: By unification, INST-ELIM-MONO and existential elimination.

[1]Interestingly, this will not be the case in iMLF, in which a λ-bound variable can "guess" it has a polymorphic type (§13.2).

Lemma 12.4.2 *Rule* VAR-LET *preserves solutions.* □

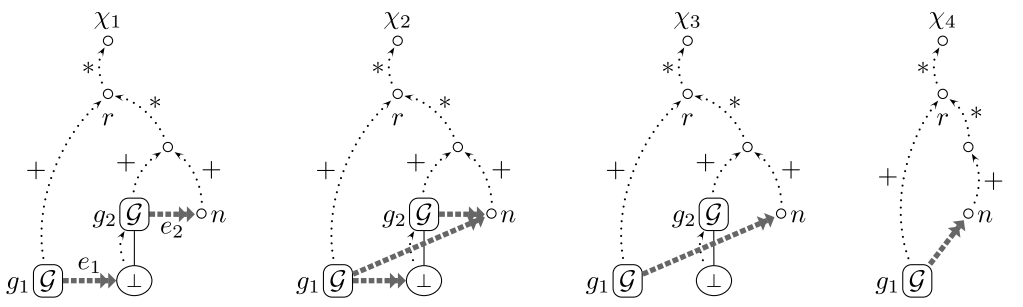


Figure 12.4.2 – Steps proving the correctness of VAR-LET

Proof: Consider the constraint of Figure 12.4.2. We have $\chi_4 \dashv\Vdash \chi_3$ by EXISTS-ELIM and $\chi_2 \Vdash \chi_1, \chi_3$ by dropping of constraints. The entailments shown below are sufficient to prove $\chi_1 \dashv\Vdash^p \chi_2 \dashv\Vdash^p \chi_3 \dashv\Vdash^p \chi_4$, which implies the result.

▷ $\chi_1 \Vdash \chi_2$: : let χ_p be a presolution of χ_1 in which we add an edge $g_1 \twoheadrightarrow n$. We are going to prove that χ_p is a presolution of χ_2. It suffices to show that the instantiation edge $g_1 \twoheadrightarrow n$ (which we call e_3) is solved in that constraint: it is immediate that χ_p is an instance of χ_2, and all the other instantiation edges are solved.
Up to e_3, χ_p is a presolution of χ_1. Thus, let $I_1^u; I_1$ and $I_2^u; I_2$ be two decompositions of $\chi_p^{e_1} \sqsubseteq \chi_p$ and $\chi_p^{e_2} \sqsubseteq \chi_p$, as given by Lemma 11.5.3. Let us call χ_p' the constraint $\chi_p^{e_3}$; we need to show that $\chi_p' \sqsubseteq \chi_p$ to prove that e_3 is solved. For this, we are essentially going to do some glueing, and prove that $\chi_p = (I \,;\, I_1' \,;\, I_2)(\chi_p')$, where I will be defined below, and I_1' is derived from I_1. We call s_1 the node $\langle g_1 1\rangle$ and s_2 the node $\langle s_2 1\rangle$.

First, consider a path π such that $\langle s_1\pi\rangle \notin \mathcal{I}^s(g_1)$. By definition of presolutions and propagation, $\langle s_1\pi\rangle = \langle s_2\pi\rangle$. Given the shape of the binding tree in χ_1, those nodes are bound above r, and $\langle s_2\pi\rangle \notin \mathcal{I}^s(g_2)$. Thus, again again by definition of presolutions and propagation, we have $\langle s_2\pi\rangle = \langle n\pi\rangle$. Thus, the nodes of the structural frontier of g_1 under s_1 are shared with s_2 and n. This implies that, in χ_p', the unification edges resulting from the frontier nodes under s_1 can be trivially solved, which gives us I; moreover I does not instantiate the node that have not been created by the expansion (**1**).

Next, let χ_p'' be $I(\chi_p')$. By construction, the expanded nodes are exactly the same in $I_1^u(\chi_p^{e_1})$ and χ_p''. The differences concern the root r' of the expansion: this node is bound to $\hat{g_2}$ in $I_1^u(\chi_p^{e_1})$, and to $\hat{n}$ in χ_p''. There is another difference, unimportant for the time being, the destination of the unification edge for r'.
Thus the operations of I_1 can be applied mostly unchanged to χ_p''. Our goal is modify I_1 into I_1' so that $I_1'(\chi_p'')$ contains exactly an expansion of s_2 after the unification of the frontier nodes. Since the expanded nodes of χ_p'' must be unified with $\langle s_2\rangle$, there is little work to do. We proceed by case disjunction on an operation o of I_1.

- *Case* $o = \mathsf{Weaken}(r')$: we remove this operation: by flag reset, an expansion of s_2 would be flexibly bound.
- *Case* $o = \mathsf{Merge}(r', s_2)$: we remove this operation.

- <u>*Case o* is another weakening, a merging, a raising or a grafting:</u> by definition this operation involves only the interior of r'. Thus we leave it unchanged.
- <u>*Case* $o = \mathsf{RaiseMerge}(n', n'')$ with $n'' \rightarrowtriangle g_2$:</u> we transform this operation in a sequence of raise such as afterwards $n' \rightarrowtriangle r'$. Indeed, by binding reset, n'' would be bound to the root of the expansion in an expansion of s_2.
- <u>*Case* $o = \mathsf{RaiseMerge}(n', n'')$ with $n'' \not\rightarrowtriangle g_2$:</u> by definition of normalized derivations, n'' is in the exterior of the expansion. By the second hypothesis, it is also in the exterior of g_2, as $n'' \rightarrowtriangle \overset{+}{\rightarrowtriangle} g_2$ cannot hold: the merging operation would not be possible, as n' is not in the interior of g_2. Moreover n'' is structurally under s_2, as n' is structurally under r' which is merged with s_2. This means that n'' is in the structural exterior of g_2, and under s_2. Since e_2 is solved in χ_p, and, n'' is shared between s_2 and n. Thus we can also apply the operation $\mathsf{RaiseMerge}(n', n'')$ to χ_p'', and we do not change this operation at all.

We call I_1' the result of changing the operations of I_1 as specified above, and let χ_p''' be $I_1'(\chi_p)$. By construction $\chi_p' \sqsubseteq \chi_p''$, and I_1' does not change the original nodes of χ_p (**2**). Since e_2 was solved in χ_p, by (1) and (2) the nodes of the structural frontier of g_2 and under s_2 are shared between g_2 and n. By construction of the operations defining I_1', the nodes of the expansion in χ_p''' are exactly those of $I_2^u(\chi_p^{e_2})$, *i.e.* the really generic part of s_2. In χ_p''', r' is bound to $\hat{n}$ and constrained to be unified with n, exactly as in $\chi_p^{e_2}$. Thus we have $\chi_p''' \sqsubseteq \chi_p$ by I_2. Since we already had $\chi_p^e \sqsubseteq \chi_p' \sqsubseteq \chi_p''$ by I and I_1', e_3 is solved in χ_p.

▷ <u>$\chi_4 \Vdash \chi_2$:</u> let χ_p be a presolution of χ_4. Let χ_p' be χ_p plus g_2, the bottom node under it, and the two missing instantiation edges. The unification edges introduced by propagating $g_1 \dashrightarrow \langle g_2 \cdot 1\rangle$ can be solved, as the bottom node under g_2 creates no constraint for the skeleton or the term-graph. Moreover, since χ_p is a presolution, all the nodes in $\mathcal{F}^s(g_1)$ are already bound above the least common binder of g_1 and n, which is r. Thus solving the unification edge does not raise a node already in χ_4.
We call χ_p'' the resulting constraint. In this constraint, g_1 and g_2 have exactly the same structural frontier, and their structural interior is the same up to expansion reset. Thus $g_2 \dashrightarrow n$ is solved, and χ_p'' is a presolution. Given the nodes and edges we have added, it is an instance of χ_2, hence a presolution of this constraint. Moreover χ_p'' witnesses the same solutions as χ_4, since the expansion of the root node is unchanged. Thus all solutions of χ_4 are solutions of χ_2.

The non-trivial cases of this proof show two very simple results:

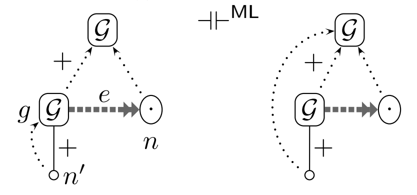
1. successively instantiating, generalizing, and instantiating again a type scheme results in an instance of this scheme;
2. trivial type schemes (*i.e.* a gen node with a single bottom child) do not really add constraints.

The most interesting result is the first one. Unfortunately, the proof is tedious, because we cannot just compose the instance derivations; instead, there is a lot of surgery to do. Interestingly, in Part III of this document we will present another form of instance witnesses, simpler than graphic instance derivations, and which can be composed.

Notice that we could have proceeded otherwise, and only shown the correctness of Var-Let on acyclic constraints. To see this, consider the leftmost constraint of Figure 12.4.1. In the principal presolution for this constraint, the middle gen node will always have the same structural interior as the leftmost one (up to flag and binding reset). Indeed, there is no reason to instantiate it further. However the proof above is more interesting, more general (it applies to cyclic constraints), and generalizes to $i\mathsf{ML^F}$, in which there are no principal presolutions.

12.4.2 Simplifying ML typing constraints

ML-Extrude



Condition:
e is the only instantiation edge leaving g

Figure 12.4.3 – Simplifying ML constraints

In ML typing constraints, we can extrude some binding edges from one gen node to its ancestor without loss of generality. This is shown by the rule ML-Extrude presented in Figure 12.4.3. Of course, this is not always possible:

- the equivalence only holds in ML. In $\mathsf{ML^F}$, raising n' outside of the interior of g means that it will not be reset during expansion. This results in either untypable constraints, or constraints with weaker principal solutions.
- if there is more than one instantiation edge leaving g, n' might be instantiated in incompatible ways at the various points it is used; thus we cannot raise n', as once it has left the interior of g, it must "choose" one of the types.
- we must have $g \xrightarrow{+} \hat{n}$. Otherwise, since n' is no longer in the interior of g, solving e might force a node under n to be raised.

Lemma 12.4.3 *Rule* ML-Extrude *preserves the solutions of* ML *constraints.* □

Proof: Let χ and χ' be the left and right-hand sides of the rule respectively. One entailment is immediate, as $\chi \ {}^{\mathsf{ML}}\!\sqsubseteq \chi'$

Thus, suppose that χ_p is an ML presolution of χ. We are going to prove that n' and n can be unified in χ_p, and that the result is a presolution. We let r be $\hat{n}$ and N be $\mathcal{I}^s(g)$.

The nodes in the structural frontier of g and below are bound on r or above (**1**), since e is solved. Consider χ_p^e in which we have solved the unification edges for the frontier by unification; we call this constraint χ'. By construction of presolutions, those unifications do not change a node of χ_p (**2**), only the nodes corresponding to $\mathcal{F}^s(g)$ in the expansion. In χ' the nodes of the expansion have exactly the shape of the nodes of N, except for the gen node itself which is not copied. Moreover those nodes are bound on r, while those in N

are bound on g. Hence we can unify the root of the expansion with n' itself, by changing only the nodes of N (**3**): it suffices to raise sufficiently all the nodes of $\mathcal{I}^s(g)$ (which is possible by (1) and because they are not red, as we are considering an ML constraint) and merging the nodes with the expansion afterwards.

Let us call χ'' the result those operations. By (2) and (3) the nodes of the expansion are unchanged, and we can apply the steps I proving $\chi_p^e \sqsubseteq \chi_p$ to χ'', which merges n and n'; we call χ''' this constraint. By definition of I this only changes the nodes which used to be the expansion, *i.e.* now the nodes of N (**4**).

Finally, let us prove that χ''' is a presolution of χ witnessing the same solutions as χ_p:

- ▷ $\underline{\chi_p \overset{\mathsf{ML}}{\sqsubseteq} \chi'''}$: $\chi_p \sqsubseteq \chi'''$ holds as all the existentially introduced nodes have been merged with existing structure. To obtain χ''' we have performed unification on χ_p which is an ML constraint, raised nodes, and merged nodes. Thus χ''' is an ML constraint. The conclusion is by Lemma 9.3.6 applied to $\chi_p \sqsubseteq \chi'''$.
- ▷ $\underline{\chi''' \text{ witness the same solutions as } \chi_p}$: by (2), (3) and (4) we have only changed the nodes of N. Thus the nodes under the root node $\langle\epsilon\rangle$ are unchanged, and both χ''' and χ_p expand to the same type.
- ▷ $\underline{\text{all the instantiation edges are solved in } \chi'''}$: by construction e is solved in χ''', as $n = n'$. For all the other instantiation edges: by (2), (3) and (4) the structural interior of their gen nodes are unchanged. By Lemma 11.6.1, those edges are thus still solved.

Rule ML-Extrude can be used as follows: in the typing constraint for an application $a_1\ a_2$, we can raise the nodes bound on the gen nodes g_1 and g_2 for a_1 and a_2. Similarly, in an abstraction $\lambda(x)\ a$, we can raise the nodes bound on the gen node for a. After the raising, the corresponding type schemes become degenerate, and we can transform the instantiation edges into unification ones. Hence gen nodes are only needed for let-bound expressions, as in the usual presentations of ML.

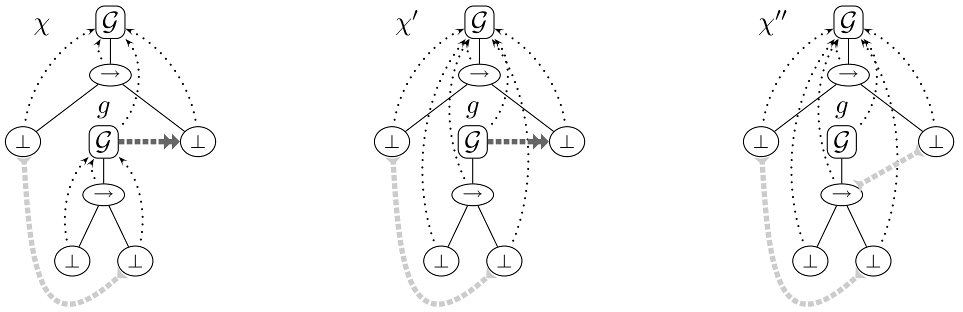


Figure 12.4.4 – Constraints for $\lambda(x)\ \lambda(y)\ x$ with ML-Extrude

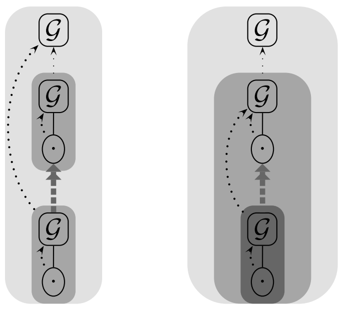
▶ **Example** The constraint χ in Figure 12.4.4 is the typing constraint for $\lambda(x)\ \lambda(y)\ x$, except that we have used Var-Abs on the variable x. We can simplify this constraint by applying ML-Extrude to the nodes $\langle g1\rangle$, $\langle g11\rangle$ and $\langle g12\rangle$, resulting in χ''. In this constraint $\langle g1\rangle$ is degenerate, and we can apply Inst-Elim-Mono to the instantiation edge, resulting in χ''. Thus all three constraints are equivalent in ML. As expected, unifying the various edges of χ'' result in $\forall\alpha.\ \forall\beta.\ \alpha \to \beta \to \alpha$ as the principal solution for $\lambda(x)\ \lambda(y)\ x$.

Notice that χ and χ' are *not* equivalent in $\mathsf{ML^F}$: the principal solution $\forall\,(\alpha)\ \forall\,(\gamma \geqslant \forall\,(\beta)\ \beta \to \alpha)\ \alpha \to \gamma$ for $\lambda(x)\ \lambda(y)\ x$ is not a solution of χ'. By raising $\langle g11\rangle$, the binder of this node is not reset during expansion, and we lose the inner quantification of the $\mathsf{ML^F}$ type.

12.4.3 Using the simplifications rules

All three simplifications rules (Var-Let, Var-Abs and ML-Extrude) can be applied to existing constraints. Alternatively, they can be performed on-the-fly during the generation of typing constraints. From an algorithmic standpoint, the second approach is actually simpler. Indeed, depending on the representation of graphic constraints, detecting whether a rule applies can be difficult. Conversely, simplifying the constraints while they are generated is immediate, as their structure is simple and well-known; this is no longer the case after some resolutions steps have occurred.

12.5 Analyzing the complexity of type inference



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

Figure 12.5.1 – Type schemes embedding

While type inference for ML is DExp-Time complete (when types need not be output), McAllester (2003) has shown that type inference has complexity $O(kn(d + \alpha(kn)))$ where α is the inverse of the Ackermann function, k is the maximum size of type schemes during inference, and d is the maximum embedding of type schemes—Figure 12.5.1 describes what is meant by this expression in terms of $\mathsf{ML^F}$ gen nodes.

In McAllester's analysis, d corresponds to the maximum left-nesting of let constructs, *i.e.* nestings of the form $\mathsf{let}\ x = (\mathsf{let}\ y = \ldots\ \mathsf{in}\ \ldots)\ \mathsf{in}\ \ldots$; we refer the reader to the original article as to why this is so, as the analysis is quite technical. McAllester argues that, in practice, d is bounded by 5, and that k does not increase with the size of the program. Under those assumptions, type inference in ML has $O(n\alpha(n))$ complexity (*i.e.* linear complexity, the term $\alpha(n)$ being completely negligible).

Our strategy for solving constraints is quite similar to the one used in efficient implementations of type inference for ML (Rémy 1994; McAllester 2003; Pottier and Rémy 2005; Kuan and MacQueen 2007), including the one proposed by McAllester. In particular, type schemes are also simplified in an innermost fashion. Moreover, unification in $\mathsf{ML^F}$ can be performed in linear time, as in ML. Thus the complexity analysis of McAllester for ML can be transferred to our constraints setting. However, since $\mathsf{ML^F}$ constraints use type generalization (*i.e.* gen nodes) for every construct and not only for let ones, we must reason on the embedding of gen nodes, not on the left-nesting of let constructs.

12.5.1 Practical complexity bound for $\mathsf{ML^F}$ type inference

More precisely, for our typing constraints, an examination of Figure 9.4.1 shows that the embedding of gen nodes d verifies

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

If we apply Var-Let and Var-Abs during the generation of constraints, d moreover verifies $d(x)=0$.

Importantly, as for ML, d does not increase with the right-nesting of let bindings. In fact, a large upper bound of d is the height of the biggest function of the program when it is written as an abstract syntax tree. Thus we can transpose McAllester's result to $\mathsf{ML^F}$:

Theorem 12.5.1 *Under the two assumptions that*

- *large programs are composed of cascades of right-nested toplevel* let *declarations*
- *k does not increase with the size of the program,*

type inference in our constraints system (thus in $\mathsf{ML^F}$) has linear complexity. □

Notice that annotated functions $\lambda(x:\tau)\ a$ in the surface language are syntactic sugar for core language expressions $\lambda(x)\ \mathsf{let}\ x=(x:\tau)\ \mathsf{in}\ a$ and should thus also be treated as let-constructs. Hence, uses of let-construct in ML programs and $\mathsf{ML^F}$ programs might differ, but we expect those differences to remain small.

12.5.2 Practical complexity bound for ML type inference in our system

Interestingly, if we restrict ourselves to ML, using rule ML-Extrude (Figure 12.4.3) will eliminate the gen nodes for all the sub-expressions but the left-hand side of let constructs.[2] Thus the function d verifies

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

Hence, for ML, we obtain exactly the same complexity bound as McAllester.

[2]More precisely, the gen nodes are not removed, but the type schemes they introduce become degenerate. Hence they are no longer important in the complexity analysis.

12.5.3 Exact complexity bound for MLF type inference

The complexity bound $O(kn(\alpha(kn) + d))$ of McAllester for type inference, and our adaptation to MLF gen nodes also provide an upper bound for the complexity of type inference.

Theorem 12.5.2 *MLF type inference has $2^{O(n)}$ complexity, where n is the size of the program.* □

Proof: Let us first consider the size of the principal presolution of a typing constraint. The typing constraint has a size linear in n. Unification does not increase the size of a constraint; only expansion does. At each expansion step, the size of the constraint can at most double since we only copy existing nodes. Inside a typing constraint of size n, the number of instantiation edges is bounded by $O(n)$. Since each instantiation edge needs to be expanded only once, the size of the principal presolution itself is bounded by $2^{O(n)}$.

This shows in particular that the maximum size k of type schemes is bounded by $2^{O(n)}$. In parallel, by definition of typing constraints, the maximum depth of gen nodes d is bounded by n. Following the analysis of McAllester, the complexity is thus in $2^{O(n)} \times n \times (\alpha(2^{O(n)} \times n) + n)$, *i.e.* in $2^{O(n)}$.

As ML programs are typable in MLF if and only if they are typable in ML, the complexity bound for MLF cannot be better than the one for ML (Kanellakis *et al.* 1991), which establishes the result.

12.6 Implementation

We have implemented an MLF type checker in OCaml (Leroy *et al.* 2007). This prototype faithfully follows the type inference algorithm we have presented, and we detail its more interesting points below.

Representing graphic types Much of the difficulty in implementing graphic constraints lies in finding a good representation for graphic types, and implementing the unification algorithm. Indeed, in graphic types, the graph structure and the binding tree are interwoven and their edges go in inverse directions. Finding an efficient functional representation of such a structure is not obvious, and we have used an imperative implementation. This only causes problems when unification fails, *i.e.* when a type inference error occurs—it is then too late too explain what caused the error, as most of the original structure of the graphic types has been changed. In this case, we type the expression a second time, and explain the error in terms of the last valid constraint.

Representing graphic constraints Interestingly, the representation we have chosen does not store the nodes bound on a given node. Maintaining this set while preserving the linear complexity of unification seems very difficult, and this information is not needed for inference anyway. Going even further, we do not store the set of gen nodes present in a constraint either: this set can almost entirely be deduced from the instantiation and unification edges of the constraint. The gen nodes that cannot be found this way are unconstrained, and

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

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

Figure 12.6.1 – Examples of error messages

can be removed by Exists-Elim. Thus we do not implement this rule explicitly, as it is implicitly performed by the OCaml garbage collector.

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

Type errors Explaining type inference errors raises two challenges: (**1**) explaining unification clashes caused by $\mathsf{ML^F}$ polymorphism; (**2**) associating a type inference error with the corresponding part of the source term.

Point (2) may not seem obvious in our constraint setting. However the difficulty is only apparent. Indeed, it is straightforward to associate a constraint edge with the expression that lead to its creation. Our constraint-based approach also allows choosing a strategy to solve the instantiation edges[3]; for example, the function in an application can be typed before or after the argument. While our implementation already gives quite readable error messages, we are also experimenting with other strategies.

Point (1) is trickier. In simple examples that do not result from encodings, a message such as *« function f expects an argument of type τ but receives a value of type τ'; type τ' is probably not polymorphic enough »* is often sufficient. More ambitiously, we could try to explain precisely why the subterm that has type τ' is not not polymorphic enough. We however leave this for future work.

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

13

Constraints up to similarity or abstraction

Abstract

We study in more details *e*ML$^{\mathsf{F}}$ and *i*ML$^{\mathsf{F}}$. Both systems share some characteristics, which we detail in §13.1. We then study typability in *e*ML$^{\mathsf{F}}$, and show that this system is not more expressive than *g*ML$^{\mathsf{F}}$ (§13.2). Finally we show that *i*ML$^{\mathsf{F}}$ is strictly more expressive than the two other systems; as a counterpart, it does not have principal presolutions (§13.3).

13.1 Constraints and inverse instance

Studying *e*ML$^{\mathsf{F}}$ and *e*ML$^{\mathsf{F}}$ requires considering constraints modulo $\approx$ and $\boxminus$. From a purely operational standpoint, there are some similarities between *e*ML$^{\mathsf{F}}$ and *i*ML$^{\mathsf{F}}$, as $\sqsupseteq^{rmw}$ and $\mathrel{\sqsupset\!\!\!-}$ share some properties. Thus, in this section, we partially abstract over the system considered. We let $\sqsubset$ range over $\mathrel{-\!\!\!\sqsubset}$ and $\sqsubseteq^{rmw}$, and $\Box$ be $\sqsubset \odot \sqsupset$. We say that a node n has $\sqsubset$ permissions if $\sqsubset$ is $\sqsubseteq^{rmw}$ and n is monomorphic, or if $\sqsubset$ is $\mathrel{-\!\!\!\sqsubset}$ and n is orange or inert. Finally, we let $\Box$ML$^{\mathsf{F}}$ be *e*ML$^{\mathsf{F}}$ if $\sqsubset$ is $\sqsubseteq^{rmw}$, and *i*ML$^{\mathsf{F}}$ if $\sqsubset$ is $\mathrel{-\!\!\!\sqsubset}$.

13.1.1 Inverse instance operations

Considering constraints up to inverse instance operations is not entirely obvious. Indeed, while the splitting and lowering operations can be transposed from graphic types to graphic constraints, there are a few potential pitfalls.

Splitting Consider the constraint χ in Figure 13.1.1 and let n be $\langle 111\rangle$. As it is rigidly bound, it can be unshared in *i*ML$^{\mathsf{F}}$. This results in χ_m, in which the instantiation edge is duplicated on the two resulting nodes. However, perhaps surprisingly, it can also be

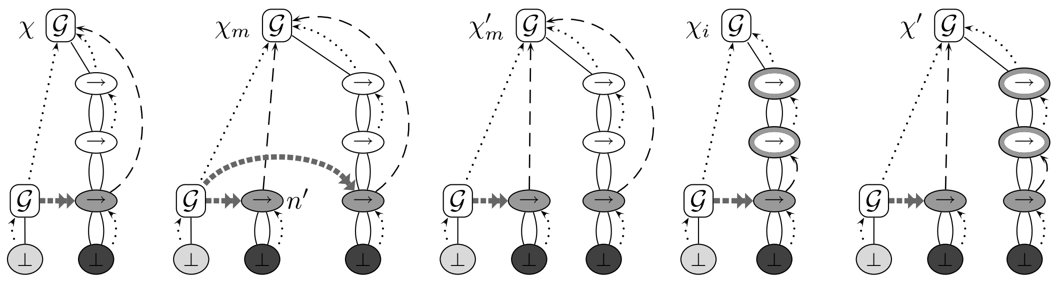


Figure 13.1.1 – Lowering and unmerging nodes

unshared into χ'_m. Let us indeed call n' the existential node resulting from the unmerging. The relation $\chi = \mathsf{Merge}(n, n')(\chi'_m)$ holds, thus $\chi \sqsupseteq_1^M \chi'_m$ must hold.

While potentially surprising, this property is entirely consistent with the intuition behind $\sqsubseteq$ and $\sqsubseteq^{rmw}$. Even though n and n' are distinct in χ'_m, they will never be able to pick different instances for $\boxminus$; thus we only need to constrain one of the two in order to ensure the soundness of the constraint.

This example also shows that fresh existential nodes can be introduced by $\sqsupset^M$. Thus it becomes slightly more difficult to keep track of nodes through (inverse) instance operations—graphic constraints are more suited to transformations that increase sharing. In the following, we say that a node n' of a constraint χ' such that $\chi \sqsubseteq^{\Box} \chi'$ is *derived* from a node n of χ if they share a path in a constraint of this derivation. For example, the nodes n and $\langle 111 \rangle$ of χ_m are derived from the node $\langle 111 \rangle$ of χ, as $\langle 111 \rangle$ and n are shared in χ.

Lowering Some nodes that could be lowered (w.r.t. permissions and well-domination) in graphic types cannot be lowered in graphic constraints. Indeed, the well-formedness of constraints prevents nodes constrained by an instantiation edge to be bound on a type node. Thus, for example n cannot be lowered in χ (and the constraint χ_i is invalid). Interestingly, if lowering $\langle 111 \rangle$ is really needed, we can first unshare n into χ'_m, then lower 111 in this constraint.

13.1.2 Properties of the modulo systems

There are some simple but important properties common to *e*ML$^{\mathsf{F}}$ and *i*ML$^{\mathsf{F}}$, which ease reasoning on these systems.

Property 13.1.1 *Let χ be a constraint, χ' be such that $\chi \sqsubseteq^{\Box} \chi'$. If n has $\sqsubset$ permissions in χ and n' is derived from n in χ', then n' has $\sqsubset$ permissions in χ'.* $\Box$

Proof: Immediate consequence of Lemma 5.4.1.

Property 13.1.2 *Let χ and χ' be such that $\chi \sqsubseteq^{\Box} \chi'$. Then there exists χ'' such that $\chi \sqsubseteq \chi'' \sqsupset \chi'$.* $\Box$

Proof: By Lemma 6.7.10, this relation holds for the instance relations on graphic types. By supposing that $\mathcal{G}$ constructors are polymorphic, the structure of gen nodes are the same in χ, χ'' and χ' (as an instance operation on them could not be cancelled afterwards by $\sqsupset$). This ensures that $\chi \sqsubseteq \chi''$ holds for the instance relation on constraints ${}^{c}\sqsubseteq$. It remains to show that $\chi'' \sqsupset \chi'$ holds. The only operation on graphic types in $\sqsupset$ that cannot be transposed to graphic constraints is the lowering of a node constrained by an instantiation edge. However, the property "failing condition 10 of Definition 9.2.1" is stable by $\sqsupset$. Thus, if an operation of $\chi'' \sqsupset \chi'$ created an ill-formed constraint, χ' would also be ill-formed. This ensures that all operations of $\chi'' \sqsupset \chi'$ are correct on graphic constraints.

The result above means that we can leverage the principality of unification in *g*ML$^{\mathsf{F}}$ to *e*ML$^{\mathsf{F}}$ and *i*ML$^{\mathsf{F}}$.

Property 13.1.3 *Let χ be a $\sqsubset$-presolution, e an instantiation edge of χ. There exists instance derivations of $\chi^e \sqsubseteq^{\Box} \chi$ of the form $\chi \sqsubseteq \chi' \sqsubseteq \chi'' \sqsupset \chi$, where χ' is the result of solving of the unification edges of χ^e by unification.* $\Box$

Proof: By definition, $\chi^e \sqsubseteq^{\Box} \chi$ holds. By Property 13.1.2, there exists χ'' such that $\chi^e \sqsubseteq \chi'' \sqsupset \chi$. In particular, χ'' is a unifier of all the unification edges of χ. Thus by completeness and principality of unification for $\sqsubseteq$, we have $\chi^e \sqsubseteq \chi' \sqsubseteq \chi''$, hence the result.

Another useful property is the stability of nodes with $\sqsubset$ permissions by propagation. However, this does not hold for the type scheme nodes themselves, because of flag and binding reset.

Lemma 13.1.4 *Let $e = g \overset{i}{\twoheadrightarrow} d$ be an instantiation edge of a constraint χ, n a node of $\mathcal{I}^s(g)$ with $n \neq \langle g \cdot i \rangle$. If n has $\sqsubseteq^{\Box}$ permissions in χ, n^c has $\sqsubseteq^{\Box}$ permissions in χ^e.* $\Box$

Proof: Since n is not $\langle g{\cdot}i \rangle$, the binding structure under n is exactly copied in the expansion. Thus, if n is monomorphic or inert, n^c is monomorphic or inert. The same reasoning holds if n is inert. Finally, if n is rigid, n^c has the same flag (since n is not $\langle g \cdot i \rangle$). Otherwise the flag path that shows that n is not inert is copied identically in the expansion, and the permissions of n and n^c are computed identically, since both n and n^c are bound on gen nodes. (There is no flag reset, since n is not n^c.)

13.1.3 Shape of presolutions

Compared to *g*ML$^{\mathsf{F}}$, the presolutions of a constraint can have quite different shapes in *e*ML$^{\mathsf{F}}$ and *i*ML$^{\mathsf{F}}$. As a first example, consider Figure 13.1.2. In *g*ML$^{\mathsf{F}}$, the principal solution of χ is χ_p. Thus, in all the $\sqsubseteq$-presolutions of χ, the node $\langle g1 \rangle$ is shared with $\langle 1 \rangle$. This is not the case in *e*ML$^{\mathsf{F}}$ and *i*ML$^{\mathsf{F}}$: as $\langle g1 \rangle$ is monomorphic, it can be freely unshared and lowered in both. Thus χ'_p is a valid presolution of χ in both systems.

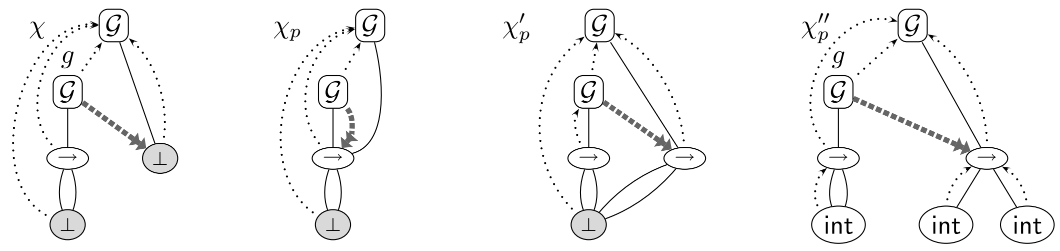


Figure 13.1.2 – Presolutions in $\Box\mathsf{ML^F}$

Even more strikingly, the constraint χ''_p is a presolution in $\Box\mathsf{ML^F}$. Indeed, even though $\langle g1\rangle$ and $\langle 1\rangle$ are not shared in this constraint, the propagation of the instantiation edge can be solved: those two nodes can be unified, and then unshared by $\sqsupseteq^{rmw}$ afterwards.

Importantly, this kind of freedom only concerns $\sqsubset$ nodes, as shown by the lemma below.

Lemma 13.1.5 *Let $e = g \overset{i}{\twoheadrightarrow} d$ be an instantiation edge of a constraint χ. Let π and n be such that $g \xrightarrow{i\cdot\pi}\!\circ\; n$ and $n \notin \mathcal{I}^s(g)$. If χ is a $\sqsubseteq^{\Box}$-presolution, then either $d \xrightarrow{\pi}\!\circ\; n$, or n has $\sqsubset$ permissions.* $\Box$

Proof: By hypothesis, $\chi^e \sqsubseteq^{\Box} \chi$. Thus there exists χ' such that $\chi^e \sqsubseteq \chi' \sqsupset \chi$. Necessarily $\langle gi\pi\rangle$ and $\langle d\pi\rangle$ are merged in χ', since the unification edges introduced by the propagation are solved. $\sqsubseteq^{\Box}$-permissions are preserved by $\sqsubset^{-1}$ (Property 13.1.1), hence the conclusion.

13.1.4 Stability of presolutions

Interestingly, presolutions of $\Box\mathsf{ML^F}$ are stable by almost all the operations in $\Box$. We start by proving this result for the operations that do not change the shape of gen nodes interiors (merging, weakening and the symmetric operations), which are the easy cases.

Lemma 13.1.6 *Let χ be a presolution in $\Box\mathsf{ML^F}$, $\chi' = o(\chi)$ with $o \in \Box_1^{MW}$. Then χ' is a presolution in $\Box\mathsf{ML^F}$.* $\Box$

Proof: Let $e = g \overset{i}{\twoheadrightarrow} d$ be an instantiation edge of χ; we must prove that $\chi'^e \sqsubseteq^{\Box} \chi'$.
We let O be the steps transforming χ^e into χ, and write $\mathsf{Flex}(n)$ the strengthening of a node n, $\mathsf{Split}(n_1, n_2)$ the splitting of a node $n = n_1 \cup n_2$ into two nodes n_1 and n_2 (by local congruence, this defines a unique operation). We let N be $\mathcal{I}^s(g) \cup (\xrightarrow{*}\!\circ\; \langle g\cdot i\rangle)$. The proof is by case disjunction on o. When it involves a node copied in the expansion, we let o' be the same operation as o, but acting on n^c when it acted on n (*i.e.* if o is $\mathsf{Weaken}(n)$, o' is $\mathsf{Weaken}(n^c)$). We start by writing χ' as a transformation on χ'^e.

▷ <u>*Case* $o = \mathsf{Weaken}(\langle g\cdot i\rangle)$ or $o = \mathsf{Flex}(\langle g\cdot i\rangle)$:</u> by flag reset, the expanded part is unchanged. Then $\chi' = (o^{-1}\,;O\,;o)(\chi'^e)$. (**1**)

- ▷ <u>*Case* $o = \mathsf{Weaken}(n)$ or $o = \mathsf{Flex}(n)$, $n \in N \setminus \{\langle g \cdot i \rangle\}$:</u> then $\chi'^e = (o\,; o')(\chi^e)$ and $(o^{-1}\,; o'^{-1}\,; O\,; o'\,; o)(\chi'^e) = \chi'$. **(2)**
- ▷ <u>*Case* $o = \mathsf{Merge}(n_1, n_2)$ or $o = \mathsf{Split}(n_1, n_2)$ with n_1 and n_2 both in N:</u> by definition of merging and splitting, n_1 and n_2 cannot be both $\langle g \cdot i \rangle$. The same equalities as for case (2) hold.
- ▷ <u>in the other cases</u>: N is unchanged, χ'^e is $o(\chi^e)$ and $\chi' = (o^{-1}\,; O\,; o)(\chi'^e)$ holds. **(3)**

In each case the operations are on $\sqsubset$ nodes, either by construction for the nodes changed by o, or by Lemma 13.1.4, for the nodes changed by o'. Thus $\chi'^e \sqsubseteq^{\square} \chi$ holds.

Presolutions are also stable by lowering of $\sqsubset$ nodes. The cases where the interiors of gen nodes remain unchanged are easy, as above. Otherwise more generalization occur, but this does not change the fact that the constraint is solved.

Lemma 13.1.7 *Let χ be a presolution in $\square\mathsf{ML^F}$, χ' such that $\chi = \mathsf{Raise}(n)(\chi')$, n having $\sqsubset$ permissions. Then χ' is a presolution in $\square\mathsf{ML^F}$.* □

<u>Proof</u>: Let $e = g \overset{i}{\dashrightarrow} d$ be an instantiation edge; we must prove it is solved in χ'. We let O be the operations transforming χ^e into χ, and N be $\mathcal{I}^s_{\chi'}(g) \cap (\langle g \cdot i \rangle \xrightarrow{*}\!\circ)$ The proof is by case disjunction on o.

- ▷ <u>*Case* $n \notin N$ and $n \neq d$:</u> n is not in $\mathcal{I}^s_{\chi}(g) \cap (\langle g \cdot i \rangle \xrightarrow{*}\!\circ)$ either, and the expanded part is unchanged between the two constraints. We conclude as in case (3) of the proof of Lemma 13.1.6.
- ▷ <u>*Case* $n = d$ and $n \in N$:</u> , the expanded part is bound lower in χ'^e than in χ^e. Let S be the copies of the nodes of $\mathcal{F}^s(g)$ in χ'^e, n' the root of the expansion. Then $\chi^e = (\mathsf{Raise}(n)\,; \mathsf{Raise}(S)\,; \mathsf{Raise}(n'))(\chi'^e)$ holds. The copies of the frontiers nodes are green, n is an $\sqsubset$-node, and n' cannot be red as it is the root of an expansion. Thus $\chi'^e \sqsubseteq \chi^e$ holds. Moreover, since $\chi = O(\chi^e)$ and $\chi' = \mathsf{Lower}(n)(\chi)$, we have $\chi'^e \sqsubseteq^{\square} \chi'$, which is the desired result.
- ▷ <u>*Case* $n \neq d$ and $n \in N$:</u> there are three subcases:
  - ∘ <u>*Case* $n \longrightarrow\!\triangleright\, g$ in χ:</u> then the expanded part between χ and χ' are the same, by binding height reset. This case is similar to subcase (1) in the proof of Lemma 13.1.6.
  - ∘ <u>*Case* $n \longrightarrow\!\triangleright \xrightarrow{+}\!\triangleright\, g$ in χ:</u> the lowering occurs "deep" inside the expansion. This case is similar to subcase (2) in the proof of Lemma 13.1.6.
  - ∘ <u>*Case* $g \longrightarrow\!\triangleright\, \hat{\chi}(n)$ in χ:</u> in this case the binding edge of n "enters" the interior of g in χ'. The main difference between χ^e and χ'^e is that n^c is a bottom node bound on $\hat{d}$ (and with a unification edge to n) in χ^e, and a whole subgraph identical to the one under n in χ'^e.
    Let χ'' be such that $\chi^e \sqsubseteq \chi'' \sqsubset^{-1} \chi$ **(4)**. Let χ''' be χ^e in which n^c is grafted the subgraph under n^c in χ'^e then weakened if n is rigid (this is possible as n^c has flexible permissions). Since there is a unification edge between n^c and n, solved in χ'', it is routine to prove (using the results of Section 6.6) that $\chi^e \sqsubseteq \chi''' \sqsubseteq \chi''$ **(5)**. Now, up to the unification edge between n and n^c, $\chi''' = (\mathsf{Raise}(n)\,; \mathsf{Raise}(n^c))(\chi'^e)$, both nodes being non-red. By definition n has $\sqsubset$ permissions, and n^c has either the same permissions by Lemma 13.1.4, or it is non-red if n is $\langle g \cdot i \rangle$. Thus $\chi'^e \sqsubseteq \chi'''$ holds. By this result, (4), (5) and the definition of χ' we have $\chi'^e \sqsubseteq \chi''' \sqsubseteq \chi'' \sqsubset^{-1} \chi \sqsubset^{-1} \chi'$. Hence $\chi'^e \sqsubseteq^{\square} \chi'$ which is the desired result.

- ▷ <u>*Case $n = d$* and $n \in N$</u>: the only possible case is $n = \langle g \cdot 1 \rangle$; otherwise, since n is not in $\mathcal{I}^s(g)$ in χ, solving the frontier unification edge that would be created when e is propagated would require χ to be cyclic. Thus n can only be bound on g in χ'. The expansion only create copies of the nodes under n (until the frontier of g, which are copied as green bottom nodes), and n^c is bound on g. Thus all the unification edges can simply be solved by unification in χ'^e, and this does not change the nodes under g. Thus $\chi'^e \sqsubseteq \chi'$ holds. (And thus so does $\chi'^e \sqsubseteq^{\Box} \chi'$.)

Presolutions are thus stable by $\sqsupseteq^{RMW}$. This means that we can always maximally unshare and lower nodes with $\sqsubset$ permissions in presolutions of $\Box$ML$^\mathsf{F}$. In particular, we can always suppose that a $\sqsubset$ node n structurally under a gen node g is in the interior of g. Indeed, once it is sufficiently unshared, it is always possible to lower n until it is bound under g. This significantly simplifies reasonings on the shape of presolutions.

13.2 Constraints up to similarity

In *e*ML$^\mathsf{F}$, raising a monomorphic node outside of a scheme interior cannot make a constraint unsolvable. Indeed, all monomorphic similar subgraphs could be shared. Thus we can prove that presolutions are stable by $\sqsubseteq^{rmw}$ (and are thus stable by $\approx$, since we have already proven that they are stable by $\sqsupseteq^{rmw}$).

Lemma 13.2.1 *Consider two constraints χ and χ' such that χ is a presolution in* eML$^\mathsf{F}$ *and $\chi \sqsubseteq^r_1 \chi'$. Then χ' is a presolution in* eML$^\mathsf{F}$. □

<u>Proof</u>: We let n be the node raised. We must prove that all instantiation edges are solved in χ'. Let $g \dashrightarrow\!\!\!\!\rightarrow d$ be such an edge e, s be $\langle g1 \rangle$. Finally, let N be $\mathcal{I}^s(g) \cap (s \xrightarrow{+}\!\circ)$.

- ▷ <u>*Case $n \notin N$, $n \neq d$*</u>: then $\chi'^e = \mathsf{Raise}(n)(\chi^e)$. Thus $\chi'^e \sqsupseteq^r_1 \chi^e \sqsubseteq^{\approx} \chi \sqsubseteq^r_1 \chi'$; hence e is solved.
- ▷ <u>*Case $n \in N$, $n \longrightarrow s \longrightarrow g \in \chi$, $n \neq d$*</u>: by binding reset, $\chi'^e = \mathsf{Raise}(n)(\chi^e)$. The conclusion is as above.
- ▷ <u>*Case $n \in N$, $n \longrightarrow\!\longrightarrow g \in \chi$, $\hat{\chi}(n) \neq s$, $n \neq d$*</u>: then $\chi'^e = (\mathsf{Raise}(n); \mathsf{Raise}(n^c))(\chi^e)$. By Lemma 13.1.4, n^c is also monomorphic. Thus $\chi'^e \sqsupseteq^r \chi^e \sqsubseteq^{\approx} \chi \sqsubseteq^r_1 \chi'$ holds, hence the conclusion.
- ▷ <u>*Case $n \notin N$, $n = d$*</u>: let g_d be $\hat{\chi}(d)$. χ^e and χ'^e differ only by the fact that the root of the expansion, and the copies of the frontier nodes, are bound on $\hat{g_d}$ in χ'^e, instead of on g_d in χ^e. By Property 13.1.3, all the unification edges in χ^e can be solved by unification. Since χ'^e only differ from χ^e by some raising, the unification edges can also be solved; we call χ'' the resulting constraint. Compared to χ', some nodes can have been raised in χ''. By definition of unification, those nodes are those structurally under g and not copied in the expansion (hence bound above g). Consider such a non-$\sqsubset$ node n'. By Lemma 13.1.5 n' and the corresponding node under d are merged in χ. Thus, n' is bound at least on the least common ancestor of g and $\hat{d}$. Let g' be this node. Notice that g' cannot be g_d: d would not have been raisable. Thus, g' is at least $\hat{g_d}$. Since the nodes created by the expansion in χ'^e are bound also bound on $\hat{g_d}$, solving the unification edges in χ'^e will not raise n' above the node on which it is bound on χ. Thus, compared to χ, all the nodes raised in χ'' are $\sqsubset$. We can lower them, and finish solving $\chi'^e \sqsubseteq^{\Box} \chi'$ by the $\sqsupseteq$ steps of $\chi^e \sqsubseteq^{\Box} \chi$.

- ▷ <u>*Case* $n \in N$, $n = d$</u>: the only possible case is $n = s$, as otherwise χ would need to be cyclic. Then n is necessarily bound on g, and s is degenerate in χ'. Thus χ'^e is χ' plus a fresh existential bottom node bound on $\hat{g}$, and constrained to be unified with s. This edge can simply be solved by unification. Moreover, all the nodes under s are bound under g or above $\hat{g}$ by well-domination, and solving the unification does not raise nodes, hence does not change the nodes under s at all. Thus $\chi'^e \sqsubseteq \chi'$ holds.
- ▷ <u>*Case* $n \in N$, $n \longrightarrow\!\!\triangleright\, g \in \chi$</u>: this is the most involved case. Let π be such that $n = \langle s\pi \rangle$. Let n' be $\langle s^c\pi \rangle$ in χ^e and χ'^e. The difference between χ^e and χ'^e (which we call χ_1 and χ'_1 from now on) lies on the subgraph under n'. In χ_1 it is a real subgraph, while in χ'_1 it is a bottom node constrained by an unification edge to unify with n. We let χ'_2 be χ'_1 in which we have grafted the same graph as under n' in χ_1. The constraints χ_1 and χ'_2 only differ by the unification edge between n and n', and possibly some frontier unification edges on the nodes below n and n' (**1**); those edges will be solved by congruence when the first one is solved.
  Let us justify that n' is monomorphic in χ_1 (**2**). If n is not $\langle s1 \rangle$, the result is by Lemma 13.1.4. Otherwise, since n can be raised, this means that no other node in N is bound on g. Hence the nodes bound on s^c are the copy of the ones bound on $\langle g1 \rangle$. Since $\langle g1 \rangle$ is monomorphic, so is s^c.[1]
  As a first step, we solve by unification all the frontier edges of χ_1, resulting in χ_3. Unlike in *g*ML$^{\mathsf{F}}$, this can raise some nodes under n. However, by slightly adapting Lemma 13.1.5, those nodes are monomorphic. Hence, on the nodes that are present in χ_1 (*i.e.* not in the expansion), χ_1 and χ_3 differ only by the raising of some monomorphic nodes.
  Next, let g' be the first gen node such that $g \xrightarrow{+}\!\!\triangleright\, g'$ and $\hat{\chi}(d) \xrightarrow{*}\!\!\triangleright\, g'$. In order to merge n and n', we must raise them until they are both bound at the same node, *i.e.* at least on χ'. Since n is raisable in χ, we can also raise it once in χ_3; this also ensures that n' can be raised once. Afterwards, there is no node under n or n' bound on a node between g and g', or between $\hat{s}^c$ and g', as those nodes have been raised to g' when the unification edges of g_1 have been solved. Hence both n and n' can be raised until g'. Moreover, those operations are in $\sqsubseteq^r$, as n is monomorphic by hypothesis, and n' is monomorphic by (2).
  We are almost ready to merge n and n'. By constructions of the raisings done, the subgraphs under n and n' (which were identical in χ_3) are still identical. Thus the only remaining difference is the binding flag of the two nodes. If n is not $\langle g1 \rangle$, the bindings flags are identical. Otherwise n can be rigid, and n' can be flexible. However, by (2) we can weaken n' by $\sqsubseteq^w$. Thus we have proven that n and n' can be unified in χ_3, the operations being in $\sqsubseteq^{rmw}$. We call χ_4 the resulting graph.
  We call I_1 the sequence of instance operations transforming χ_1 into χ_3. By Property 13.1.3, $\chi_3 \sqsubseteq \chi$. By this result, $\chi_3 \sqsubseteq^{rmw} \chi_4$, and Lemma 6.7.9 there exists χ_5 such that $\chi \sqsubseteq^{rmw} \chi_5$ and $\chi_4 \sqsubseteq \chi_5$.
  By (1), we can apply I_1 to χ'_2, which solves all the unification edges of χ'_2 except the edge between d and s^c; in fact we obtain exactly χ_3. Thus we have $\chi'^e = \chi'_1 \sqsubseteq \chi'_2 \sqsubseteq \chi_3 \sqsubseteq^{rmw} \chi_4 \sqsubseteq \chi_5 \sqsupseteq^{rmw} \chi \sqsubseteq \chi'$. This shows that $\chi'^e \sqsubseteq^{\approx} \chi'$, which is the desired result.

As a corollary of the result above and of Lemma 13.1.6, in *e*ML$^{\mathsf{F}}$, we can maximally instantiate of presolution.

[1]Interestingly, the remainder of the proof would work unchanged in *i*ML$^{\mathsf{F}}$. However, this property does not carry over to *i*ML$^{\mathsf{F}}$, as n can be orange while n^c can be green.

Lemma 13.2.2 *Consider an $e\mathsf{ML^F}$ presolution χ. The maximal instance of χ for $\sqsubseteq^{rmw}$ is a presolution in $e\mathsf{ML^F}$.* □

We are now almost ready to conclude: since in maximally instantiated presolutions there is no need for inverse similarity steps, those presolutions are also $g\mathsf{ML^F}$ presolutions.

Lemma 13.2.3 *Let χ_p be a presolution in $e\mathsf{ML^F}$, χ'_p its maximal instance for $\sqsubseteq^{rmw}$. Then χ'_p is a presolution in $g\mathsf{ML^F}$.* □

Proof: Let us show that the instantiation edges of χ'_p are $\sqsubseteq$-solved. Let e be an instantiation edge. By Lemma 13.2.2, χ'_p is a presolution in $e\mathsf{ML^F}$; thus $\chi'^e_p \sqsubseteq^{\approx} \chi'_p$. Hence there exists χ' such that $\chi'^e_p \sqsubseteq \chi' \sqsupseteq^{rmw} \chi'_p$. By maximality of χ'_p for $\sqsubseteq^{rmw}$, we must have $\chi' = \chi'_p$. Hence χ'_p is also a presolution in $g\mathsf{ML^F}$.

Using the results above, we can finally show that $e\mathsf{ML^F}$ is not more expressive than $g\mathsf{ML^F}$: all terms typable in $e\mathsf{ML^F}$ are typable in $g\mathsf{ML^F}$ (and conversely). Moreover, up to similarity, terms have the same set of (pre)solutions.

Theorem 13.2.4 *Let χ be a constraint. Any $g\mathsf{ML^F}$ (pre)solution of χ is an $e\mathsf{ML^F}$ (pre)solution. Given an $e\mathsf{ML^F}$ (pre)solution S of χ, there exists an $e\mathsf{ML^F}$ (pre)solution S' of χ similar to S.* □

Proof: It suffices to prove the result for presolutions. Moreover the first part of the result is immediate, since $(\sqsubseteq) \subset (\sqsubseteq^{\approx})$. Thus let χ_p be a $e\mathsf{ML^F}$ presolution of χ. By definition, $\chi \sqsubseteq^{\approx} \chi_p$. Let χ'_p be the maximal instance of χ_p for $\sqsubseteq^{rmw}$. By construction, $\chi_p \sqsubseteq^{rmw} \chi'_p$, hence χ_p and χ'_p are similar. By Lemma 13.2.2, it is a $g\mathsf{ML^F}$ presolution. Thus it remains to prove that it is a $g\mathsf{ML^F}$ presolution of χ. We have $\chi \sqsubseteq^{\approx} \chi_p \sqsubseteq^{rmw} \chi'_p$, hence $\chi \sqsubseteq^{\approx} \chi'_p$. Thus, by Property 13.1.2, $\chi \sqsubseteq ; \sqsupseteq^{rmw} \chi'_p$ holds. Since χ'_p is maximally shared, we have in fact $\chi \sqsubseteq \chi'_p$. This is the desired result.

For typing constraints, this justifies the fact that studying inference up to similarity is not needed. Instead, we can limit ourselves to $\sqsubseteq$ solutions, as the $\sqsubseteq^{\approx}$ solutions can be easily deduced.

Corollary 13.2.5 *Let χ be a solvable typing constraint, S its principal $g\mathsf{ML^F}$ (pre)solution. The set of its $e\mathsf{ML^F}$ (pre)solutions is $\{S' \mid S \sqsubseteq^{\approx} S'\}$.* □

13.3 Constraints up to abstraction

13.3.1 Typability in $i\mathsf{ML^F}$

Let us show on an example why $i\mathsf{ML^F}$ is the *implicit* version of $\mathsf{ML^F}$, in which type annotations are not needed. We consider the problem of typing ω defined as $\lambda(x)\ x\,x$. The typing constraint for ω is shown as χ in Figure 13.3.1.

We first simplify χ into χ' by solving the unification edges. Next, in χ'', which is also an instance of χ, we have guessed the type of x to be $\forall\,(\alpha)\ \alpha \to \alpha$ and have instantiated the

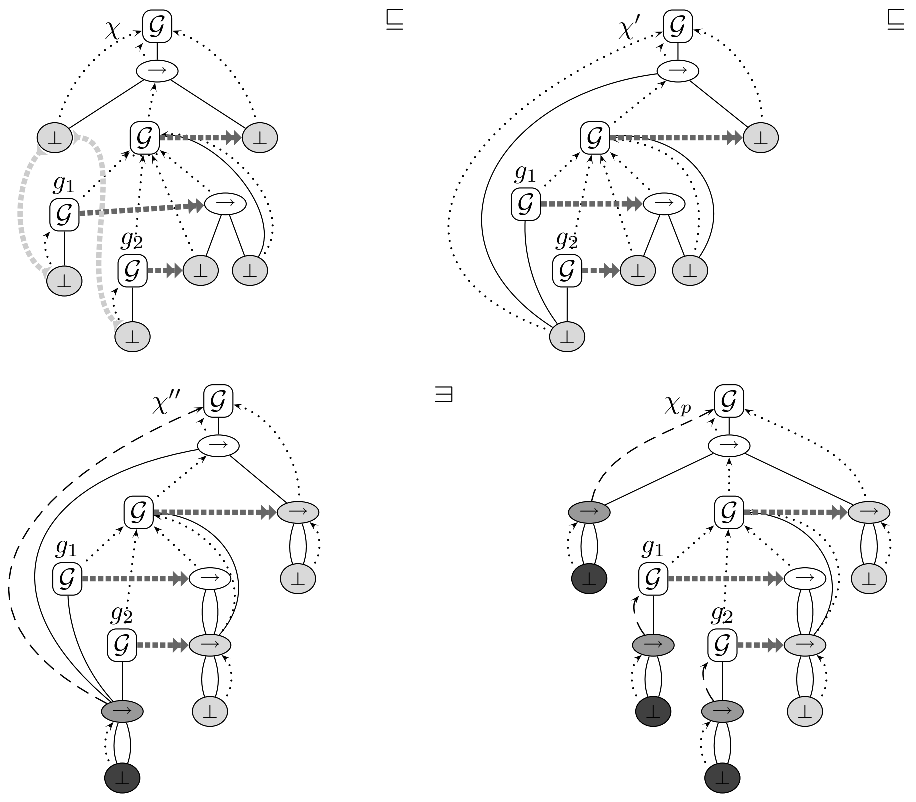


Figure 13.3.1 – One typing of $\lambda(x)\; x\, x$ in iML$^{\mathsf{F}}$

remainder of the constraint accordingly. We have also turned the flexible flag of the node $\langle 11\rangle$ into a rigid one.

This rigidification enables us to unshare the nodes $\langle 11\rangle$, $\langle g_1 1\rangle$ and $\langle g_2 1\rangle$ by inverse abstraction. Moreover, we can also lower the binding edge of $\langle g_1 1\rangle$ and $\langle g_2 1\rangle$ until they are bound at the same position as in χ—albeit rigidly instead of flexibly. In the resulting presolution χ_p, $\langle 11\rangle$ is still rigid. Hence the arrow type requires its argument to be polymorphic. In fact, there is no way to turn this rigid binder into a flexible one; this is of course essential for type soundness. On the other hand, for $\langle g_1 1\rangle$ and $\langle g_2 1\rangle$, the flag will be reset during expansion, and each occurrence of x is free to pick a different instance of $\forall\,(\alpha)\;\alpha \to \alpha$. Here, the two occurrences receive the types $\forall\,(\beta)\;\beta \to \beta$ and $\forall\,(\gamma \geqslant \forall\,(\beta)\;\beta \to \beta)\;\gamma \to \gamma$ respectively.

13.3.2 Properties of *i*ML$^{\mathsf{F}}$ presolutions

Although *e*ML$^{\mathsf{F}}$ and *i*ML$^{\mathsf{F}}$ share some similarities in their presentation, they have very different properties. In particular, constraints do not in general have principal solutions or presolutions. Moreover, raising inert or orange nodes does not preserve presolutions.

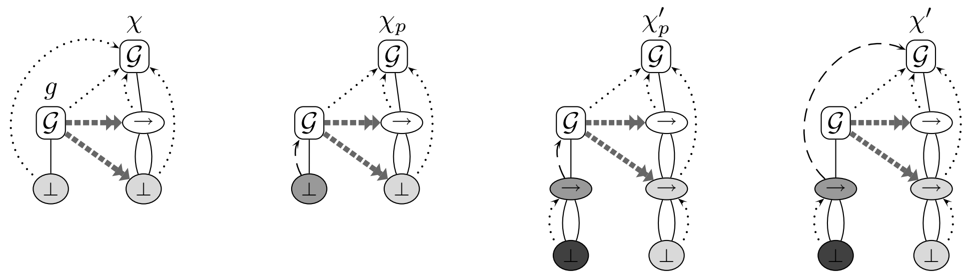


Figure 13.3.2 – Presolutions in iML$^{\mathsf{F}}$

Non-principal solutions Consider Figure 13.3.2. The constraint χ is unsolvable in gML$^{\mathsf{F}}$, as the bottom node $\langle g \cdot 1\rangle$ is outside the interior of g, and would need to be unified with both $\langle 1\rangle$ and $\langle 11\rangle$. However, it has an infinity of presolutions in iML$^{\mathsf{F}}$, of which two are shown: χ_p and χ'_p. Notice that there are no principal presolution: necessarily, $\langle g \cdot 1\rangle$ must be rigid, and the different ways of instantiating this node makes the presolutions incomparable in general. (In this very simple example, the set of *solutions* is principal, the principal solution being $\forall\,(\alpha)\ \alpha \to \alpha$. This is not the case in general.)

Raising The constraint χ' of the same figure also shows that presolutions are not stable by raising. Indeed, χ' is not a presolution in iML$^{\mathsf{F}}$, although it is derived from the presolution χ'_p by the raising of $\langle g1\rangle$, which is orange. Before the raising, each instantiation edge can pick its own copy of $\langle g1\rangle$. After the raising, this is no longer possible. Moreover, unlike in eML$^{\mathsf{F}}$, we cannot share the constrained nodes, as they have taken different instances of the type $\forall\,(\alpha)\ \alpha \to \alpha$.

Eager propagation This example shows that the equivalent of Lemma 11.4.2 (*i.e.* the fact that eager propagation preserves presolutions) does not hold in iML$^{\mathsf{F}}$. Indeed, propagating eagerly the topmost unification edge of χ results in an unsolvable constraint, as there will be no way to solve the second instantiation edge afterwards. However, χ'_p which is derived from χ' by an inverse abstraction step, has solutions.

It is worth mentioning that splitting (and lowering) the nodes of a constraint along $\exists$ as much as possible does not help, as χ exemplifies. Indeed, one must somehow "guess" the polymorphic part, make it rigid by weakening, and then unshare it by inverse abstraction. In ML$^{\mathsf{F}}$ source terms, hence in gML$^{\mathsf{F}}$ this "guessing" part is explicitly done by the programmer, through the use of type annotations.

13.3.3 Reasoning in Implicit ML$^{\mathsf{F}}$

Since iML$^{\mathsf{F}}$ allows polymorphism to be "guessed", it does not permit type inference. We have also seen that it does not have principal presolutions or solutions. Thus, it is unsurprising that the transformation rules for $g\mathsf{ML^F}$ are unsound or incomplete in $i\mathsf{ML^F}$.

In particular, rule Inst-Copy is no longer sound. Indeed, by splitting the gen node for the term $\lambda(x)\ x\ x$, we could find a typing for the term

$$\mathsf{let}\ \omega = \lambda(x)\ x\ x\ \mathsf{in}\ (\omega\ (\lambda(y)\ y),\ \omega\ (\lambda(y)\ \lambda(z)\ y))$$

However, this term is not typable in $i\mathsf{ML^F}$. Indeed, it is not possible to find a type for ω that makes both $\omega\ (\lambda(y)\ y)$ and $\omega\ (\lambda(y)\ \lambda(z)\ y))$ simultaneously typable.

Conversely, rule Var-Abs is not complete in $i\mathsf{ML^F}$: this can be seen by applying this rule to the constraint χ of Figure 13.3.1, which makes this constraint untypable. Indeed, using rule Var-Abs forfeits the possibility to type the variables polymorphically. Finally, as can be seen by considering χ' in Figure 13.3.2, rule Inst-Elim-Mono is incomplete.

13.3.4 Expressivity of $i\mathsf{ML^F}$

Any term typable in System F is typable in $i\mathsf{ML^F}$—this is essentially immediate, since $\sqsubseteq^{\boxminus}$ is a superrelation of $\sqsubseteq_{\mathsf{F}}$, up to inlining of rigid edges in $\mathsf{ML^F}$ types. Thus typability in $i\mathsf{ML^F}$ is likely to be undecidable, as it is undecidable in System F (Wells 1994).

In parallel, since type inference in $g\mathsf{ML^F}$ is decidable and $g\mathsf{ML^F}$ solutions are also $i\mathsf{ML^F}$ solutions, $i\mathsf{ML^F}$ is strictly more expressive than $g\mathsf{ML^F}$. An example of one term typable in a system but not in the other is $\lambda(x)\ x\ x$: in $g\mathsf{ML^F}$ we must add some type annotations. Interestingly, type annotations are sufficient to recover all the missing expressivity. That is, any term a typable in $i\mathsf{ML^F}$ can be transformed in a term a' typable in $g\mathsf{ML^F}$, which differ from a only by the addition of some type annotations. This result has already been proved on the syntactic presentation of $\mathsf{ML^F}$ (Le Botlan 2004; Le Botlan and Rémy 2007). We leave a graphic proof for future work.

Part

III

An explicit language for ML^F

14

*x*ML$^\mathsf{F}$, a Church-style language for ML$^\mathsf{F}$

Abstract

We introduce *x*ML$^\mathsf{F}$, a fully explicit version of ML$^\mathsf{F}$ suitable for use as an internal language. We explain in more detail the need for such a language, as well as its characteristics (§14.1). We present the syntax of the language and the typing rules in §14.2, while the reduction rules are studied in §14.3. We show the soundness of the language in both a call-by-value and a call-by-name setting (§14.4). Finally, we show that reduction is confluent (§14.5).

14.1 Why another explicit language for ML$^\mathsf{F}$?

The language *g*ML$^\mathsf{F}$ we have presented is partially in Church-style, as λ-abstractions (or even entire expressions) can be annotated. However, a large amount of type information is still inferred, and partial type information cannot be easily maintained during reduction. Hence, while *g*ML$^\mathsf{F}$ is a good surface language, it is not a good candidate for use as an internal language during the compilation process, where some simple program transformations and perhaps some reduction steps are being performed. This has been a problem for the adoption of ML$^\mathsf{F}$ in the Haskell community (Peyton Jones 2003): the GHC compilation chain uses an internal explicitly typed language, especially (but not only) for evidence translation due to the use of qualified types (Jones 1994).

This is also an obstacle to proving subject reduction, which does not hold in *g*ML$^\mathsf{F}$ or *e*ML$^\mathsf{F}$. In a way, this is unavoidable in a language with non-trivial partial type inference. Indeed, type annotations cannot be completely dropped, but must at least be transformed and reorganized during reduction. Still, one could expect that *g*ML$^\mathsf{F}$ could be equipped with reduction rules for type annotations that would preserve subject reduction. This has actually been considered in the original presentation of ML$^\mathsf{F}$ (Le Botlan 2004), but with only partial success. The reduction kept track of annotation sites during reduction. This showed, in particular, that no new annotation site needs to be introduced during reduction. Unfortunately, the exact form of annotations could not be maintained during reduction by lack of an appropriate language to describe their computation. As a result, it has only been shown that some type derivation can be rebuilt after the reduction of a well-typed program, but without exhibiting an algorithm to compute them during reduction.

***x*MLF** In this chapter, we present a Church-style version of MLF, called *x*MLF, which contains full type information. In fact, type checking becomes a simple and local verification process—by contrast with type inference in *g*MLF, which is based on unification. In *x*MLF, all parameters of functions are explicitly typed, and both type abstraction and type instantiation are explicit, as in System F. However, type instantiation is more general and more atomic than type application in System F: type instantiations are actually proof evidences for the type instance relations in MLF.

In addition to the usual β-reduction, we give a series of reduction rules for simplifying type instantiations. These rules are confluent when allowed in any context. They are also sufficient to reduce all type instantiations when used in either a call-by-value or call-by-name setting. Moreover, reduction preserves typings.

Interestingly, the difficulties in defining an internal language for MLF is not reflected in the internal language itself, which we believe is quite simple to understand. Instead, most of the pitfalls lie in the translation itself. Hence, we present *x*MLF in this chapter, and defer the translation to the next one (§15).

14.2 Types and typing rules of *x*MLF

14.2.1 Types, terms, and environments

The types of *x*MLF are defined in Figure 14.2.1. They include flexible MLF quantification, but also types of the form $\sigma \to \sigma$, even when σ is a second-order type. Rigid quantification is only useful for type inference purposes, and is thus not present in *x*MLF: rigid bounds are directly inlined inside types.

$\alpha, \beta, \gamma, \delta$			**Type variables**
σ	$::=$		**Types**
	$\mid$	α	Type variable
	$\mid$	$\sigma \to \sigma$	Arrow type
	$\mid$	$C\,\overline{\sigma}$	Constructor type
	$\mid$	$\forall\,(\alpha \geqslant \sigma)\,\sigma$	Instance-bounded quantification
	$\mid$	$\bot$	Bottom type

Figure 14.2.1 – Types of *x*MLF

Expressions are given in Figure 14.2.2. They are those of the λ-calculus enriched with let constructs, with two small differences. Type instantiation is of the form $a[\varphi]$, where φ

is a *type computation*, and generalizes the type application of System F; we discuss type computations in the next section. Moreover, a type abstraction $\Lambda(\alpha \geqslant \sigma)\ a$ is now given an instance bound σ; α is bound in a, but not in σ. The type abstraction construction $\Lambda(\alpha)\ a$ of System F can be simulated by $\Lambda(\alpha \geqslant \bot)\ a$.

x, y, z		**Expression variables**
a ::=		**Expressions**
	$\mid\ x$	Variable
	$\mid\ \lambda(x : \sigma)\ a$	Function
	$\mid\ a\ a$	Application
	$\mid\ \Lambda(\alpha \geqslant \sigma)\ a$	Type abstraction
	$\mid\ a[\varphi]$	Type instantiation
	$\mid\ \mathsf{let}\ x = a\ \mathsf{in}\ a$	Let-binding

Figure 14.2.2 – Terms of $x\mathsf{ML}^{\mathsf{F}}$

Γ ::=		**Environments**
	$\mid\ \varnothing$	Empty environment
	$\mid\ \Gamma, \alpha \geqslant \sigma$	Type variable
	$\mid\ \Gamma, x : \sigma$	Expression variable

Figure 14.2.3 – Environments of $x\mathsf{ML}^{\mathsf{F}}$

Environments, defined in Figure 14.2.3 are standard. We call $\mathsf{dom}(\Gamma)$ the type or expression variables bound by Γ. As usual, all the variables appearing in either a judgment or the environment itself must be bound in the environment. Moreover, we assume that an environment does not bind twice the same variable. Both conditions can be ensured by the well-formedness predicate of Figure 14.2.4. All environments are implicitly well-formed.

14.2.2 Type instance

Type instantiation in $x\mathsf{ML}^{\mathsf{F}}$ is explicit and details every instantiation step along the instance relation $\leq$ of $x\mathsf{ML}^{\mathsf{F}}$. This departs from System F where type instantiation is more atomic. Instead, type instantiation in $x\mathsf{ML}^{\mathsf{F}}$ is fully determined by type computations: a type computation φ is an explicit witness for the fact that a type σ can be instantiated into another type σ'. We let φ range over type computations, which are described in Figure 14.2.5.

Type instance judgments, in Figure 14.2.6, are of the form $\Gamma \vdash \varphi : \sigma \leq \sigma'$, and state that in environment Γ, the type computation φ witnesses that σ' is an instance of σ. The identity computation ε witnesses the fact that type instance is reflexive, while the composition operator ";" witnesses the transitivity of type instance. The computation $\triangleright\sigma$ says that any type σ is an instance of the type $\bot$. Under the hypothesis $\alpha \geqslant \sigma$ in the environment, the computation $\alpha\triangleleft$ says that α is an instance of type σ.

$$\frac{}{\mathsf{wf}\,(\varnothing)} \qquad \frac{\mathsf{wf}\,(\Gamma) \qquad \alpha \notin \mathsf{dom}(\Gamma) \qquad \mathsf{ftv}(\sigma) \subseteq \mathsf{dom}(\Gamma)}{\mathsf{wf}\,(\Gamma, \alpha \geqslant \sigma)}$$

$$\frac{\mathsf{wf}\,(\Gamma) \qquad x \notin \mathsf{dom}(\Gamma) \qquad \mathsf{ftv}(\sigma) \subseteq \mathsf{dom}(\Gamma)}{\mathsf{wf}\,(\Gamma, x : \sigma)}$$

Figure 14.2.4 – Well-formed environments

φ	::=		**Type computations**
	|	ε	Reflex
	|	$\triangleright \sigma$	Bot
	|	$\alpha \triangleleft$	Hyp
	|	$\forall\,(\geqslant \varphi)$	Inner
	|	$\forall\,(\alpha \geqslant)\ \varphi$	Outer
	|	$\&$	Quantification elimination
	|	$\rotatebox[origin=c]{180}{\&}$	Quantification introduction
	|	$\varphi; \varphi$	Trans

Figure 14.2.5 – Type computations

The inner computation $\forall\,(\geqslant \varphi)$ applies the computation φ to instantiate the bound σ' of an instance-bounded quantification $\forall\,(\alpha' \geqslant \sigma')\ \sigma$. Conversely, the outer computation $\forall\,(\alpha \geqslant)\ \varphi$ applies the computation φ to instantiate the type σ of the quantification. The type variable α is bound in φ and the premise of the rule Inst-Outer is increased accordingly.

The computation $\rotatebox[origin=c]{180}{\&}$ introduces a trivial quantification $\forall\,(\alpha \geqslant \bot)$. The computation $\&$ eliminates the bound of a type of the form $\forall\,(\alpha \geqslant \sigma)\ \sigma'$ by substituting α by σ in σ'. This amounts to definitively choosing the present bound σ for α.

Convention In the following, we identify types, expressions and computations up to renaming of their bound variables. The capture-avoiding substitution of a (*resp.* expression, type) variable v by a (*resp.* expression, type) p inside a term q (which can, independently, be an expression, a type, a type computation, or an environment) is written $q\{v \leftarrow p\}$. By a slight abuse of notation, we also write $q\{\alpha \triangleleft \leftarrow \varphi\}$ for the capture-avoiding replacement of all the occurrences of the computation $\alpha \triangleleft$ by the computation φ in q.

► **Example** Consider the following four types

$$\begin{array}{rclcrcl} \sigma_K & \triangleq & \forall\,(\alpha \geqslant \bot)\,\forall\,(\beta \geqslant \bot)\ \alpha \to \beta \to \alpha & \quad & \sigma_{\mathsf{min}} & \triangleq & \forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha \to \alpha \\ \sigma_{\mathsf{cmp}} & \triangleq & \forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha \to \mathsf{bool} & & \sigma_{\mathsf{and}} & \triangleq & \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \end{array}$$

Let φ be the type computation $\forall\,(\alpha \geqslant)\ (\forall\,(\geqslant \triangleright \alpha); \&)$ and φ' be the type computation $\forall\,(\geqslant \triangleright \mathsf{bool}); \&$. Then we have $\vdash \varphi : \sigma_K \leq \sigma_{\mathsf{min}}$, and both $\vdash \varphi' : \sigma_{\mathsf{min}} \leq \sigma_{\mathsf{and}}$ and $\vdash \varphi' :$

$$\textsc{Inst-Reflex}\quad \frac{}{\Gamma \vdash \varepsilon : \sigma \leq \sigma} \qquad \textsc{Inst-Trans}\quad \frac{\Gamma \vdash \varphi_1 : \sigma_1 \leq \sigma_2 \qquad \Gamma \vdash \varphi_2 : \sigma_2 \leq \sigma_3}{\Gamma \vdash \varphi_1; \varphi_2 : \sigma_1 \leq \sigma_3}$$

$$\textsc{Inst-Bot}\quad \frac{}{\Gamma \vdash \triangleright\sigma : \bot \leq \sigma} \qquad \textsc{Inst-Hyp}\quad \frac{\alpha \geqslant \sigma \in \Gamma}{\Gamma \vdash \alpha\triangleleft : \sigma \leq \alpha}$$

$$\textsc{Inst-Inner}\quad \frac{\Gamma \vdash \varphi : \sigma_1 \leq \sigma_2}{\Gamma \vdash \forall\,(\geqslant \varphi) \colon \forall\,(\alpha \geqslant \sigma_1)\ \sigma \leq \forall\,(\alpha \geqslant \sigma_2)\ \sigma} \qquad \textsc{Inst-Outer}\quad \frac{\Gamma, \varphi : \alpha \geqslant \sigma \vdash \varphi : \sigma_1 \leq \sigma_2}{\Gamma \vdash \forall\,(\alpha \geqslant)\ \varphi : \forall\,(\alpha \geqslant \sigma)\ \sigma_1 \leq \forall\,(\alpha \geqslant \sigma)\ \sigma_2}$$

$$\textsc{Inst-Quant-Elim}\quad \frac{}{\Gamma \vdash \& : \forall\,(\alpha \geqslant \sigma)\ \sigma' \leq \sigma'\{\alpha \leftarrow \sigma\}} \qquad \textsc{Inst-Quant-Intro}\quad \frac{\alpha \notin \mathsf{ftv}(\sigma)}{\Gamma \vdash \invamp : \sigma \leq \forall\,(\alpha \geqslant \bot)\ \sigma}$$

Figure 14.2.6 – Type instantiation

$\sigma_{\mathsf{cmp}} \leq \sigma_{\mathsf{and}}$.

14.2.2.1 A derived computation

As the example above shows, we often instantiate a quantification over $\bot$ and immediately substitutes the result—exactly as type application in System F. As this is a common pattern, we define the computation σ as syntactic sugar for $\forall\,(\geqslant \triangleright \sigma); \&$. Then, the substitution φ above can be written more concisely as $\forall\,(\alpha \geqslant)\ \alpha$. Similarly, φ' is just bool.

14.2.2.2 Computing on types

All type instance operations are made entirely explicit, through the use of computations. In fact, type instantiation is deterministic.

Lemma 14.2.1 *If $\Gamma \vdash \varphi : \sigma \leq \sigma_1$ and $\Gamma \vdash \varphi : \sigma \leq \sigma_2$, then $\sigma_1 = \sigma_2$.* □

Proof: Immediate induction on the instance derivation.

Hence, type instance judgments also define a function that computes the application of a computation to a type. Another more explicit definition is given in Figure 14.2.7. Notice that this function is complete but not sound for the type instance judgment, as it does not check the premise of rule Inst-Hyp. This is intentional, as it avoids parameterizing the function by the type environment.

Property 14.2.2 *If $\Gamma \vdash \varphi : \sigma \leq \sigma'$, then $\sigma[\varphi] = \sigma'$.* □

$$\begin{array}{rcll}
(\forall\,(\alpha \geqslant \sigma)\ \sigma')[\&] & = & \sigma'\{\alpha \leftarrow \sigma\} & \\
\sigma[\otimes] & = & \forall\,(\alpha \geqslant \bot)\ \sigma & \alpha \notin \mathsf{ftv}(\sigma) \\
\sigma[\varphi_1; \varphi_2] & = & (\sigma[\varphi_1])[\varphi_2] & \\
(\forall\,(\alpha \geqslant \sigma)\ \sigma')[\forall\,(\geqslant \varphi)] & = & \forall\,(\alpha \geqslant \sigma[\varphi])\ \sigma' & \\
(\forall\,(\alpha \geqslant \sigma)\ \sigma')[\forall\,(\alpha \geqslant)\ \varphi] & = & \forall\,(\alpha \geqslant \sigma)\ \sigma'[\varphi] & \\
\sigma[\alpha \lhd] & = & \alpha & \\
\bot[\rhd \sigma] & = & \sigma & \\
\sigma[\varepsilon] & = & \sigma &
\end{array}$$

Figure 14.2.7 – Application of a computation to a type

Proof: Immediate by induction on φ and the definition of instance.

14.2.3 Typing rules for xMLF

The typing rules of xMLF are given in Figure 14.2.8. They are quite standard: the only novelty is rule TApp, which checks that the type computation φ transforms σ into σ'.

$$\textsc{Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \textsc{Abs}\ \frac{\Gamma, x : \sigma \vdash a : \sigma'}{\Gamma \vdash \lambda(x : \sigma)\ a : \sigma \rightarrow \sigma'} \qquad \textsc{App}\ \frac{\Gamma \vdash a_1 : \sigma_2 \rightarrow \sigma_1 \qquad \Gamma \vdash a_2 : \sigma_2}{\Gamma \vdash a_1\ a_2 : \sigma_1}$$

$$\textsc{TAbs}\ \frac{\Gamma, \alpha \geqslant \sigma' \vdash a : \sigma \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \Lambda(\alpha \geqslant \sigma')\ a : \forall\,(\alpha \geqslant \sigma')\ \sigma} \qquad \textsc{TApp}\ \frac{\Gamma \vdash a : \sigma \qquad \Gamma \vdash \varphi : \sigma \leq \sigma'}{\Gamma \vdash a[\varphi] : \sigma'}$$

$$\textsc{Let}\ \frac{\Gamma \vdash a : \sigma \qquad \Gamma, x : \sigma \vdash a' : \sigma'}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \sigma'}$$

Figure 14.2.8 – Typing rules for xMLF

Notice that, as in System F, a closed expression has a unique type.

Lemma 14.2.3 *If* $\Gamma \vdash a : \sigma_1$ *and* $\Gamma \vdash a : \sigma_2$*, then* $\sigma_1 = \sigma_2$*.* □

Proof: Immediate by induction on the derivation and Lemma 14.2.1.

(In fact, the typing derivation is itself completely determined by the expression.)

Notice that Rule Let is not formally derivable via a syntactic translation: if we see a let-binding let $x = a_1$ in a_2 as an abstraction $(\lambda(x : \sigma_1)\ a_2)\ a_1$, we do not know what the type σ_1 is—although it is entirely determined by a_1. This is not new however, as it would also be the case in System F extended with let-bindings.

▶ **Example** Let id be the identity function $\Lambda(\alpha \geqslant \bot)\ \lambda(x : \alpha)\ x$ and σ_{id} the type $\forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha$. We have $\vdash \mathsf{id} : \sigma_{\mathsf{id}}$. A definition of the function choose mentioned in the introduction of this document may be

$$\Lambda(\beta \geqslant \bot)\ \lambda(x : \beta)\ \lambda(y : \beta)\ \mathsf{if}\ \mathit{true}\ \mathsf{then}\ x\ \mathsf{else}\ y$$

assuming the existence of booleans and a conditional construct in the language, with the obvious corresponding typing rules. We then have $\vdash \mathsf{choose} : \forall\,(\beta \geqslant \bot)\ \beta \to \beta \to \beta$. We can define the application of choose to id as the expression choose_id equal to:

$$\Lambda(\beta \geqslant \sigma_{\mathsf{id}})\ \mathsf{choose}[\beta]\ \mathsf{id}[\beta \lhd]$$

which has type $\forall\,(\beta \geqslant \sigma_{\mathsf{id}})\ \beta \to \beta$.

The expression choose_id may also be given the weaker types below, by applying the corresponding computations:

$$\begin{array}{lcl} (\mathsf{choose\_id})[\parr; \forall\,(\gamma \geqslant)\ (\forall\,(\geqslant \gamma); \&)] & : & \forall\,(\gamma \geqslant \bot)\ (\gamma \to \gamma) \to \gamma \to \gamma \\ (\mathsf{choose\_id})[\&] & : & (\forall\,(\beta \geqslant \bot)\ \beta \to \beta) \to (\forall\,(\beta \geqslant \bot)\ \beta \to \beta) \end{array}$$

▶ **Another example** *This example is adapted from (Leijen and Löh 2005)*

Let us continue the example of §14.2.2. Consider three values K, min and cmp of types σ_K, σ_{min}, σ_{cmp} respectively. Let us write nil the empty list, and cons the list concatenation operator. We give below some fully explicit versions of the lists

$$x_1 = \mathsf{nil} \qquad x_2 = \mathsf{cons}\ K\ x_1 \qquad x_3 = \mathsf{cons}\ \mathsf{min}\ x_2 \qquad x_4 = \mathsf{cons}\ \mathsf{cmp}\ x_3$$

That is, each list being obtained by adding one element at the beginning of the previous one—thus refining its type:

$$\begin{array}{lcl} x_1 & : & \forall\,(\alpha \geqslant \bot)\ \mathsf{list}\ \alpha \\ & = & \mathsf{nil} \\ x_2 & : & \forall\,(\beta \geqslant \sigma_K)\ \mathsf{list}\ \beta \\ & = & \Lambda(\beta \geqslant \sigma_K)\ \mathsf{cons}\ [\beta]\ K[\beta \lhd]\ x_1[\forall\,(\geqslant \rhd \beta)] \\ x_3 & : & \forall\,(\gamma \geqslant \sigma_{\mathsf{min}})\ \mathsf{list}\ \gamma \\ & = & \Lambda(\gamma \geqslant \sigma_{\mathsf{min}})\ \mathsf{cons}\ [\gamma]\ \mathsf{min}[\gamma \lhd]\ x_2[\forall\,(\geqslant \varphi; \gamma \lhd)] \\ x_4 & : & \mathsf{list}\ (\mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}) \\ & = & \mathsf{cons}\ [\sigma_{\mathsf{and}}]\ \mathsf{cmp}[\mathsf{bool}]\ x_3[\forall\,(\geqslant \varphi'); \&] \end{array}$$

▶ **System F like type application** The derived typing rule for the type application $a[\sigma]$ is simply

$$\frac{\Gamma \vdash a : \forall\,(\alpha \geqslant \bot)\ \sigma'}{\Gamma \vdash a[\sigma] : \sigma'\{\alpha \leftarrow \sigma\}}$$

14.3 Reduction in $x\mathsf{ML}^\mathsf{F}$

We define a non deterministic reduction semantics, where reduction rules can be applied in any context (including under abstractions). More precisely, reduction can be performed as per the following evaluation contexts:

$$\begin{array}{rcl}
E & ::= & \{\cdot\} \\
 & \mid & E[\varphi] \\
 & \mid & \lambda(x:\sigma)\ E \\
 & \mid & \Lambda(\alpha \geqslant \sigma)\ E \\
 & \mid & E\ a \ \mid\ a\ E \\
 & \mid & \mathsf{let}\ x = E\ \mathsf{in}\ a \ \mid\ \mathsf{let}\ x = a\ \mathsf{in}\ E
\end{array}$$

The reduction rules are described in Figure 14.3.1. Rules (β) and (β_{LET}) are the usual β-reduction rules. Rule CONTEXT is also standard. The other rules deal with the reduction of type computations and are described next.

$$\begin{array}{rcll}
(\lambda(x:\sigma)\ a_1)\ a_2 & \longrightarrow & a_1\{x \leftarrow a_2\} & (\beta) \\
\mathsf{let}\ x = a_2\ \mathsf{in}\ a_1 & \longrightarrow & a_1\{x \leftarrow a_2\} & (\beta_{\text{LET}}) \\
 & & & \\
a[\varepsilon] & \longrightarrow & a & \text{REFLEX} \\
a[\varphi;\varphi'] & \longrightarrow & a[\varphi][\varphi'] & \text{TRANS} \\
a[\text{⅋}] & \longrightarrow & \Lambda(\alpha \geqslant \bot)\ a \qquad \text{if } \alpha \notin \mathsf{ftv}(a) & \text{QUANT-INTRO} \\
(\Lambda(\alpha \geqslant \sigma)\ a)[\&] & \longrightarrow & a\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} & \text{QUANT-ELIM} \\
(\Lambda(\alpha \geqslant \sigma)\ a)[\forall\,(\geqslant \varphi)] & \longrightarrow & \Lambda(\alpha \geqslant \sigma[\varphi])\ a\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\} & \text{INNER} \\
(\Lambda(\alpha \geqslant \sigma)\ a)[\forall\,(\alpha \geqslant)\ \varphi] & \longrightarrow & \Lambda(\alpha \geqslant \sigma)\ (a[\varphi]) & \text{OUTER} \\
 & & & \\
E\{a\} & \longrightarrow & E\{a'\} \qquad \text{if } a \longrightarrow a' & \text{CONTEXT}
\end{array}$$

Figure 14.3.1 – Reduction rules

We define $\longrightarrow_\Lambda$ as the type reduction subrelation, obtained by removing the rules (β) and (β_{LET}) from Figure 14.3.1. Similarly, we defined $\longrightarrow_\beta$ as the expression reduction subrelation, obtained by considering only (β), (β_{LET}) and CONTEXT.

14.3.1 Type reduction rules

The rules REFLEX, TRANS and QUANT-INTRO reduce type applications of the computations ε, ";" and ⅋ respectively. As expected, an application of the identity computation can be discarded, and the application of a composition of type computations reduces into successive applications of each type computation. Rule QUANT-INTRO introduces a new type abstraction. Rules INNER, OUTER and QUANT-ELIM are more involved. They also reduce type applications, but only to type abstractions $\Lambda(\alpha \geqslant \sigma)\ a$. Rule OUTER is the simplest of the three, and it just propagates the type computation inside the body a.

Rule QUANT-ELIM reduces the application of the computation & to a type abstraction $\Lambda(\alpha \geqslant \sigma)\ a$. This is done by removing the type abstraction altogether, and substituting α

by σ everywhere in a. Moreover, computations $\alpha\triangleleft$ —which coerce expressions of type σ into expressions of type α—become vacuous as α is now σ, and are replaced by the identity computation (which will eventually be eliminated by Rule REFLEX).

▶ **Example** Let a be defined as

$$\Lambda(\alpha \geqslant \sigma)\ \lambda(x : \alpha \to \alpha)\ \lambda(y : \bot)\ y[\triangleright(\alpha \to \alpha)]\ z[\alpha\triangleleft]$$

Then $a[\&]$ reduces to

$$\lambda(x : \sigma \to \sigma)\ \lambda(y : \bot)\ y[\triangleright(\sigma \to \sigma)]\ z[\varepsilon]$$

Finally, rule INNER reduces the application of an inner substitution $\forall(\geqslant \varphi)$ to a type abstraction $\Lambda(\alpha \geqslant \sigma)\ a$. This is done by rewriting the bound σ to $\sigma[\varphi]$, as one could expect, and also updating uses of $\alpha\triangleleft$ inside a. Indeed, the occurrences of $\alpha\triangleleft$ inside a assume that the bound of α is σ. After the reduction, this hypothesis becomes false, as α ranges over instances of $\sigma[\varphi]$. Hence, we replace all occurrences of $\alpha\triangleleft$ by the composition of type computations $(\varphi; \alpha\triangleleft)$. This transforms an expression being instantiated by $\alpha\triangleleft$, which is necessarily of type σ, first into an expression of type $\sigma[\varphi]$, then into an expression of type α.

▶ **Example *(continued)*** Reusing the expression a of the previous example, the type application $a[\forall(\geqslant \varphi)]$ reduces to

$$\Lambda(\alpha \geqslant \sigma[\varphi])\ \lambda(x : \alpha \to \alpha)\ \lambda(y : \bot)\ y[\triangleright(\alpha \to \alpha)]\ z[\varphi; \alpha\triangleleft]$$

Notice that type applications of the form $a[\triangleright\sigma]$ or $a[\alpha\triangleleft]$ are not reducible. However the latter can disappear by an outer application of Rule QUANT-ELIM.

▶ **A longer example** Consider again the expression choose_id defined in §14.2.3 as

$$\Lambda(\beta \geqslant \sigma_{\mathsf{id}})\ \mathsf{choose}[\beta]\ \mathsf{id}[\beta\triangleleft]$$

The type application choose$[\beta]$ reduces to

$$\lambda(x : \beta)\ \lambda(y : \beta)\ \mathsf{if}\ \mathit{true}\ \mathsf{then}\ x\ \mathsf{else}\ y$$

Thus, choose_id reduces (after the reduction above and a β-reduction step) to:

$$\Lambda(\beta \geqslant \sigma_{\mathsf{id}})\ \lambda(y : \beta)\ \mathsf{if}\ \mathit{true}\ \mathsf{then}\ \mathsf{id}[\beta\triangleleft]\ \mathsf{else}\ y$$

Consider the following three specialized versions of choose_id.

choose_id$[\forall(\geqslant \mathsf{int}); \&]$:	$(\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int}$
choose_id$[\&]$:	$(\forall(\alpha \geqslant \bot)\ \alpha \to \alpha) \to (\forall(\alpha \geqslant \bot)\ \alpha \to \alpha)$
choose_id$[\text{⅋}; \forall(\alpha \geqslant)\ (\forall(\geqslant \alpha); \&)]$:	$\forall(\alpha \geqslant \bot)\ (\alpha \to \alpha) \to \alpha \to \alpha$

They respectively reduce to the terms

$$\lambda(y : \mathsf{int} \to \mathsf{int})\ \mathsf{if}\ \mathit{true}\ \mathsf{then}\ (\lambda(x : \mathsf{int})\ x)\ \mathsf{else}\ y$$
$$\lambda(y : \forall(\alpha \geqslant \bot)\ \alpha \to \alpha)\ \mathsf{if}\ \mathit{true}\ \mathsf{then}\ (\Lambda(\alpha \geqslant \bot)\ \lambda(x : \alpha)\ x)\ \mathsf{else}\ y$$
$$\Lambda(\alpha \geqslant \bot)\ \lambda(y : \alpha \to \alpha)\ \mathsf{if}\ \mathit{true}\ \mathsf{then}\ (\lambda(x : \alpha)\ x)\ \mathsf{else}\ y$$

In all three cases, all type instantiations can be eliminated by reduction.

14.3.2 Reducing only type applications

Let the type erasure of $x\mathsf{ML}^\mathsf{F}$ expressions be the untyped λ-term defined inductively by

$$\begin{aligned}
\lceil x \rceil &= x \\
\lceil \lambda(x : \sigma)\ a \rceil &= \lambda(x)\ \lceil a \rceil \\
\lceil a_1\ a_2 \rceil &= \lceil a_1 \rceil\ \lceil a_2 \rceil \\
\lceil \Lambda(\alpha \geqslant \sigma)\ a \rceil &= \lceil a \rceil \\
\lceil a[\varphi] \rceil &= \lceil a \rceil \\
\lceil \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 \rceil &= \mathsf{let}\ x = \lceil a_1 \rceil\ \mathsf{in}\ \lceil a_2 \rceil
\end{aligned}$$

Type reduction preserves the shape of terms:

Lemma 14.3.1 *If $a \longrightarrow_\Lambda a'$, then $\lceil a \rceil = \lceil a' \rceil$.* □

Proof: By induction on the reduction. The only non-immediate cases are QUANT-ELIM and INNER, and the substitutions in these two rules only change the types or the computations in the terms, which are removed by type erasure.

14.3.3 System F as a subsystem of $x\mathsf{ML}^\mathsf{F}$

System F can be seen as a restriction of $x\mathsf{ML}^\mathsf{F}$, using the following syntactic restrictions:

- all type computations are of the form σ;
- all type abstractions are of the form $\Lambda(\alpha \geqslant \bot)\ a$. We may then write $\Lambda(\alpha)\ a$ instead of $\Lambda(\alpha \geqslant \bot)\ a$.
- all types inside term abstractions and type computations must be of the form

  $$\sigma \quad ::= \quad \alpha \mid \sigma \to \sigma \mid C\ \overline{\sigma} \mid \forall\,(\alpha \geqslant \bot)\ \sigma$$

  We may thus write $\forall\,(\alpha)\ \sigma$ instead of $\forall\,(\alpha \geqslant \bot)\ \sigma$. Notice that $\bot$ is not a valid type anymore; however, as in System F, $\forall\,(\alpha)\ \alpha$ is.

Lemma 14.3.2 *Given a System F type derivation, it is also a correct $x\mathsf{ML}^\mathsf{F}$ derivation.* □

(See *e.g.* (Pierce 2002, §23) for the typing rules of System F.)

Proof: The proof is by induction on the derivation. Each case is immediate, as the typing rules coincide in the two systems.

Moreover, let us consider a reducible System F like type application, which is of the form $(\Lambda(\alpha)\ a)[\sigma]$. In $x\mathsf{ML}^\mathsf{F}$, it can be reduced as follows:

$$\begin{aligned}
(\Lambda(\alpha)\ a)[\sigma] \ &= \ (\Lambda(\alpha \geqslant \bot)\ a)[\forall\,(\geqslant \triangleright \sigma); \&] && (14.1)\\
&\longrightarrow \ (\Lambda(\alpha \geqslant \bot)\ a)[\forall\,(\geqslant \triangleright \sigma)][\&] && (14.2)\\
&\longrightarrow \ (\Lambda(\alpha \geqslant \sigma)\ a)[\&] && (14.3)\\
&\longrightarrow \ a\{\alpha \leftarrow \sigma\} && (14.4)
\end{aligned}$$

Step 14.1 is by definition; step 14.2 is by TRANS; step 14.3 is by INNER, and the fact that $a\{\alpha\triangleleft \leftarrow \triangleright\sigma; \alpha\triangleleft\}$ is a, since by construction $\alpha\triangleleft$ does not occur in a; step 14.4 is by QUANT-ELIM, and the fact that $a\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$ is $a\{\alpha \leftarrow \sigma\}$, for the same reason. Notice that $a\{\alpha \leftarrow \sigma\}$ belongs to System F, as σ and a do. Thus, we get the System F type application reduction rule as a derived rule in xML$^\mathsf{F}$.

14.4 Type soundness

In this section, we show that reduction in xML$^\mathsf{F}$ preserves typings. This holds in particular when reduction can be performed in any context, *i.e.* for strong reduction. We also show that weak reduction, *i.e.* when reduction is not performed under λ-abstraction is sufficient to reduce expressions to values. We consider both cases of a call-by-value and a call-by-name setting—importantly, this is the first time that a variant of ML$^\mathsf{F}$ is proven sound for call-by-name evaluation. Hence, xML$^\mathsf{F}$ can be used as a general calculus, but also as the core of a programming language, with either strict or lazy semantics.

14.4.1 Preservation of typings

As usual, the subject reduction property relies on the weakening and substitution lemmas, which hold for both instance and typing judgments.

Lemma 14.4.1 (Weakening) *Assume that* $\mathit{wf}(\Gamma, \Gamma', \Gamma'')$ *holds.*

- *If* $\Gamma, \Gamma'' \vdash \varphi : \sigma_1 \leq \sigma_2$, *then* $\Gamma, \Gamma', \Gamma'' \vdash \varphi : \sigma_1 \leq \sigma_2$
- *If* $\Gamma, \Gamma'' \vdash a : \sigma'$, *then* $\Gamma, \Gamma', \Gamma'' \vdash a : \sigma'$. $\square$

Proof: Immediate induction on each derivation.

Lemma 14.4.2 (Term substitution) *Assume that* $\Gamma \vdash a' : \sigma'$ *holds.*

- *If* $\Gamma, x : \sigma', \Gamma' \vdash \varphi : \sigma_1 \leq \sigma_2$ *then* $\Gamma, \Gamma' \vdash \varphi : \sigma_1 \leq \sigma_2$
- *If* $\Gamma, x : \sigma', \Gamma' \vdash a : \sigma$, *then* $\Gamma, \Gamma' \vdash a\{x \leftarrow a'\} : \sigma$ $\square$

Proof: By induction on each derivation. The result is immediate for the instance relation, which never consider term variables in the environment. For the typing relation, the only non-immediate case is the one for the variable x itself, which we develop below.

By hypothesis, we have $\Gamma, x : \sigma', \Gamma' \vdash x : \sigma$. Only rule VAR applies, thus $\sigma = \sigma'$ (**1**). By well-formedness of $\Gamma, x : \sigma, \Gamma'$, the environment Γ, Γ' is well-formed. Thus, by Lemma 14.4.1, $\Gamma, \Gamma' \vdash a' : \sigma'$ (**2**). The conclusion is then by (1), (2) and the fact that $a' = x\{x \leftarrow a'\}$.

The next lemma, which expresses that we can substitute an instance bound inside judgments, ensures the correctness of rule QUANT-ELIM.

Lemma 14.4.3 (Bound substitution)

- *If* $\Gamma, \alpha \geqslant \sigma, \Gamma' \vdash \varphi : \sigma_1 \leq \sigma_2$ *then*
  $\Gamma, \Gamma'\{\alpha \leftarrow \sigma\} \vdash \varphi\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} : \sigma_1\{\alpha \leftarrow \sigma\} \leq \sigma_2\{\alpha \leftarrow \sigma\}$
- *If* $\Gamma, \alpha \geqslant \sigma, \Gamma' \vdash a : \sigma'$ *then*
  $\Gamma, \Gamma'\{\alpha \leftarrow \sigma\} \vdash a\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} : \sigma'\{\alpha \leftarrow \sigma\}$ □

Proof: We let Γ'' be $\Gamma, \Gamma'\{\alpha \leftarrow \sigma\}$.

For type instance, the proof is by induction on the shape of φ. Let φ' be $\varphi\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$. In each case we show $\Gamma'' \vdash \varphi' : \sigma_1\{\alpha \leftarrow \sigma\} \leq \sigma_2\{\alpha \leftarrow \sigma\}$ (**1**).

▷ *Case ε*: then $\sigma_1 = \sigma_2$ and $\varphi' = \varepsilon$. Then (1) holds by Inst-Reflex

▷ *Case $\triangleright\sigma'$*: then $\sigma_1 = \bot$, $\sigma_2 = \sigma'$, $\varphi' = \triangleright\sigma'\{\alpha \leftarrow \sigma\}$ and $\sigma_1\{\alpha \leftarrow \sigma\} = \bot$. Then (1) holds by Inst-Bot.

▷ *Case $\alpha\triangleleft$*: then $\sigma_1 = \sigma$, $\sigma_2 = \alpha$, $\varphi' = \varepsilon$, $\sigma_1\{\alpha \leftarrow \sigma\} = \sigma$ (α cannot be free in σ, by well-formedness of environments) and $\sigma_2\{\alpha \leftarrow \sigma\} = \sigma$. Thus (1) holds by rule Inst-Reflex.

▷ *Case $\beta\triangleleft$ (with $\alpha \neq \beta$)*: then the bound of β in Γ, Γ' is σ_1, σ_2 is β, φ' is φ and $\sigma_2\{\alpha \leftarrow \sigma\}$ is β. There are two subcases.

  ◦ *Case $\beta \in \mathsf{dom}(\Gamma)$*: $\sigma_1\{\alpha \leftarrow \sigma\}$ is σ_1 as $\alpha \notin \mathsf{ftv}(\sigma')$. Thus (1) holds by Inst-Hyp, as $\Gamma''(\beta) = \Gamma(\beta) = \sigma_1$.

  ◦ *Case $\beta \in \mathsf{dom}(\Gamma')$*: we have $\Gamma''(\beta) = (\Gamma'\{\alpha \leftarrow \sigma\})(\beta) = (\Gamma'(\beta))\{\alpha \leftarrow \sigma\} = \sigma_1\{\alpha \leftarrow \sigma\}$, hence (1) holds by Inst-Hyp.

▷ *Case $⅋$*: then $\sigma_2 = \forall\,(\beta \geqslant \bot)\ \sigma_1$ with $\beta \notin \mathsf{ftv}(\sigma_1)$, $\varphi' = ⅋$ and $\sigma_2\{\alpha \leftarrow \sigma\} = \forall\,(\gamma \geqslant \bot)$ $\sigma_1\{\alpha \leftarrow \sigma\}$, for a type variable γ fresh for both σ_1 and σ. Hence the conclusion holds by Inst-Quant-Intro.

▷ *Case $\&$*: then $\sigma_1 = \forall\,(\beta \geqslant \sigma_1')\ \sigma_1''$ for some σ_1' and σ_1'', $\sigma_2 = \sigma_1''\{\beta \leftarrow \sigma_1'\}$, $\varphi' = \&$. Without loss of generality we can assume that β does not appear free in σ. Then $\sigma_1\{\alpha \leftarrow \sigma\} = \forall\,(\beta \geqslant \sigma_1'\{\alpha \leftarrow \sigma\})\ \sigma_1''\{\alpha \leftarrow \sigma\}$ and $\sigma_2\{\alpha \leftarrow \sigma\}$ is $\sigma_1''\{\alpha \leftarrow \sigma\}\{\beta \leftarrow \sigma_1'\{\alpha \leftarrow \sigma\}\}$. Hence (1) holds by Inst-Quant-Elim.

▷ *Case $\forall\,(\beta \geqslant)\ \varphi''$*: then $\sigma_1 = \forall\,(\beta \geqslant \sigma_1')\ \sigma_1''$, and $\sigma_2 = \forall\,(\beta \geqslant \sigma_1')\ \sigma_2''$ with $\Gamma, \alpha \geqslant \sigma, \Gamma', \beta \geqslant \sigma_1' \vdash \varphi'' : \sigma_1'' \leq \sigma_2''$. By induction hypothesis, $\Gamma'', \beta \geqslant \sigma_1'\{\alpha \leftarrow \sigma\} \vdash \varphi''\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} : \sigma_1''\{\alpha \leftarrow \sigma\} \leq \sigma_2''\{\alpha \leftarrow \sigma\}$ holds. Since φ' is $\forall\,(\beta \geqslant)\ \varphi''\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$, (1) holds by Inst-Outer.

▷ *Case $\forall\,(\geqslant \varphi'')$*: then $\sigma_1 = \forall\,(\beta \geqslant \sigma_1')\ \sigma_1''$, and $\sigma_2 = \forall\,(\beta \geqslant \sigma_2')\ \sigma_1''$ with $\Gamma, \alpha \geqslant \sigma, \Gamma' \vdash \varphi'' : \sigma_1' \leq \sigma_2'$. By induction hypothesis, $\Gamma'' \vdash \varphi''\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} : \sigma_1'\{\alpha \leftarrow \sigma\} \leq \sigma_2'\{\alpha \leftarrow \sigma\}$ holds. Since φ' is $\forall\,(\geqslant \varphi''\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\})$, (1) holds by Inst-Inner.

▷ *Case $\varphi = \varphi_1; \varphi_2$*: by inversion of instance there exists σ_0 such that $\Gamma, \alpha \geqslant \sigma, \Gamma' \vdash \varphi_1 : \sigma_1 \leq \sigma_0$ and $\Gamma, \alpha \geqslant \sigma, \Gamma' \vdash \varphi_2 : \sigma_0 \leq \sigma_2$. By induction hypothesis, $\Gamma'' \vdash \varphi_1\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} : \sigma_1\{\alpha \leftarrow \sigma\} \leq \sigma_0\{\alpha \leftarrow \sigma\}$ and $\Gamma'' \vdash \varphi_2\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} : \sigma_0\{\alpha \leftarrow \sigma\} \leq \sigma_2\{\alpha \leftarrow \sigma\}$; hence (1) holds by Inst-Trans.

(Notice that the only really interesting cases are those for $\alpha\triangleleft$ and $\beta\triangleleft$.)

For the typing relation, the proof is by induction on the shape of a. We let a' be $a\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$.

▷ *Case x*: Then σ' is the type of x in the environment, and a' is x. Thus $\sigma'\{\alpha \leftarrow \sigma\}$ is indeed the type of x in Γ'' (whether x is in Γ or Γ'), and the conclusion holds by Var.

▷ Case $\lambda(x : \sigma_1)\ b$: Then a' is $\lambda(x : \sigma_1\{\alpha \leftarrow \sigma\})\ b\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$. The type σ' is $\sigma_1 \to \sigma_2$, with $\Gamma, \alpha \geqslant \sigma, \Gamma', x : \sigma_1 \vdash b : \sigma_2$. By induction hypothesis, $\Gamma'', x : \sigma_1\{\alpha \leftarrow \sigma\} \vdash b\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} : \sigma_2\{\alpha \leftarrow \sigma\}$ holds. Then by Abs, $\Gamma'' \vdash a' : \sigma_1\{\alpha \leftarrow \sigma\} \to \sigma_2\{\alpha \leftarrow \sigma\}$; this last type is exactly $\sigma'\{\alpha \leftarrow \sigma\}$, hence the result holds.

▷ The cases for applications, type abstractions and let are similar.

▷ Case $a = b[\varphi]$: Then a' is $b\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}[\varphi\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}]$, and there exists σ'' such that $\Gamma, \alpha \geqslant \sigma, \Gamma' \vdash b : \sigma''$ and $\Gamma, \alpha \geqslant \sigma, \Gamma' \vdash \varphi : \sigma'' \leq \sigma'$ holds. By induction hypothesis, $\Gamma'' \vdash b\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} : \sigma''\{\alpha \leftarrow \sigma\}$. By the result of instance, $\Gamma'' \vdash \varphi\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} : \sigma''\{\alpha \leftarrow \sigma\} \leq \sigma'\{\alpha \leftarrow \sigma\}$. Hence the result holds by TApp.

The following lemma, which expresses that an instance bound can be instantiated, ensures in turn the correctness of rule Inner.

Lemma 14.4.4 (Narrowing) *Assume that* $\Gamma \vdash \varphi : \sigma \leq \sigma'$ *holds.*

- *If* $\Gamma, \alpha \geqslant \sigma, \Gamma' \vdash \varphi' : \sigma_1 \leq \sigma_2$ *then*

  $\Gamma, \alpha \geqslant \sigma', \Gamma' \vdash \varphi'\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\} : \sigma_1 \leq \sigma_2$

- *If* $\Gamma, \alpha \geqslant \sigma, \Gamma' \vdash a : \sigma''$ *then*

  $\Gamma, \alpha \geqslant \sigma', \Gamma' \vdash a\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\} : \sigma''$ □

Proof: Let Γ'' be $\Gamma, \alpha \geqslant \sigma', \Gamma'$. For type instance, the proof is by induction on the shape of φ. We let φ' be $\varphi\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\}$.

▷ Case $\alpha\triangleleft$: then $\sigma_1 = \sigma$, $\sigma_2 = \alpha$ and $\varphi' = (\varphi; \alpha\triangleleft)$. By hypothesis and Lemma 14.4.1, we have $\Gamma'' \vdash \varphi : \sigma \leq \sigma'$, and $\Gamma'' \vdash \alpha : \sigma' \leq \alpha$ by Inst-Hyp. Hence the result holds by Inst-Trans.

▷ Case $\beta\triangleleft$ with $\alpha \neq \beta$: Then the bound of β in in $\Gamma, \alpha \geqslant \sigma, \Gamma'$ is σ_1, σ_2 is β and φ' is φ. Since $\alpha \neq \beta$, the result is simply by Inst-Hyp.

As in the proof Lemma 14.4.3, all the other cases are directly by induction hypothesis. This is also the case for the typing relation.

Subject reduction is an easy consequence of all these results.

Theorem 14.4.5 (Subject reduction) *If* $\Gamma \vdash a : \sigma$ *and* $a \longrightarrow a'$ *then,* $\Gamma \vdash a' : \sigma$. □

Proof: The proof is by induction on the reduction $a \longrightarrow a'$.

▷ Case (β): then $a = (\lambda(x : \sigma')\ a_1)\ a_2$ and $a' = a_1\{x \leftarrow a_2\}$ for some σ, a_1 and a_2. By inversion of typing, $\Gamma, x : \sigma' \vdash a_1 : \sigma$ and $\Gamma \vdash a_2 : \sigma'$. The conclusion is by Lemma 14.4.2.

▷ Case (β_{Let}): then $a = \text{let } x = a_2 \text{ in } a_1$ for some a_1 and a_2, with $a' = a_1\{x \leftarrow a_2\}$. By inversion of typing there exists σ' such that $\Gamma \vdash a_2 : \sigma'$ and $\Gamma, x : \sigma' \vdash a_1 : \sigma$. The conclusion is by Lemma 14.4.2.

- ▷ *Case* Inner: then $a = (\Lambda(\alpha \geqslant \sigma_1)\ a'')[\forall\,(\geqslant \varphi)]$, a' is $\Lambda(\alpha \geqslant \sigma_1[\varphi])\ a''\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\}$ for some a'', σ_1 and φ. By inversion of typing, there exists σ_2 and σ' such that $\sigma = \forall\,(\alpha \geqslant \sigma_2)\ \sigma'$, $\Gamma \vdash \varphi' : \sigma_1 \leq \sigma_2$ and $\Gamma, \alpha \geqslant \sigma_1 \vdash a'' : \sigma'$. By Lemma 14.4.4, $\Gamma, \alpha \geqslant \sigma_2 \vdash a''\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\} : \sigma'$. By rule TAbs $\Gamma \vdash \Lambda(\alpha \geqslant \sigma_2)\ a''\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\} : \forall\,(\alpha \geqslant \sigma_2)\ \sigma'$. The conclusion is immediate, as $\sigma_2 = \sigma_1[\varphi]$.
- ▷ *Case* Outer: then $a = (\Lambda(\alpha \geqslant \sigma')\ a'')[\forall\,(\alpha \geqslant)\ \varphi]$, a' is $\Lambda(\alpha \geqslant \sigma')\ (a''[\varphi])$ for some a'', σ' and φ. By inversion of typing, there exists σ_1 and σ_2 such that $\sigma = \forall\,(\alpha \geqslant \sigma')\ \sigma_2$, $\Gamma \vdash \varphi' : \sigma_1 \leq \sigma_2$ and $\Gamma, \alpha \geqslant \sigma' \vdash a'' : \sigma_1$. By TApp and TAbs, $\Gamma \vdash \Lambda(\alpha \geqslant \sigma')\ (a''[\varphi]) : \forall\,(\alpha \geqslant \sigma')\ \sigma_2$.
- ▷ *Case* Quant-Elim: then $a = (\Lambda(\alpha \geqslant \sigma')\ a'')[\&]$, a' is $a''\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$ for some a'' and σ'. By inversion of typing, there exists σ'' such that $\sigma = \sigma''\{\alpha \leftarrow \sigma'\}$ and $\Gamma, \alpha \geqslant \sigma' \vdash a'' : \sigma''$. By Lemma 14.4.3, $\Gamma \vdash a''\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} : \sigma''\{\alpha \leftarrow \sigma'\}$ which is the desired result.
- ▷ *Case* Reflex: then $a = a''[\varepsilon]$ and $a' = a''$. By inversion of typing, a'' has type σ, which is the desired result.
- ▷ *Case* Trans: then $a = a''[\varphi_1; \varphi_2]$ and $a' = a''[\varphi_1][\varphi_2]$. By inversion of typing, there exists σ_0 and σ_0' such that $\Gamma \vdash \varphi_1 : \sigma_0 \leq \sigma_0'$, $\Gamma \vdash \varphi_2 : \sigma_0' \leq \sigma$ and $\Gamma \vdash a'' : \sigma_0$. We conclude by applying TApp twice.
- ▷ *Case* Quant-Intro: then $a = a''[\otimes]$ and $a' = \Lambda(\alpha \geqslant \bot)\ a''$ with $\alpha \notin \mathsf{fv}(a'')$. By inversion of typing, there exists σ' such that $\Gamma \vdash a'' : \sigma'$ and $\sigma = \forall\,(\alpha \geqslant \bot)\ \sigma'$ with $\alpha \notin \mathsf{ftv}(\sigma')$. We conclude by TAbs.
- ▷ *Case* Context: then there exists E such that $a = E[b]$, $a' = E[b']$ and $b \longrightarrow b'$. The proof is immediate by structural induction on E.

14.4.2 Progress with call-by-value and call-by-name semantics

In order to have a more general setting, we introduce a set of constants into our language; each constant κ comes with its arity (written $\mathsf{arity}(\kappa)$). We partition constants into *constructors* c and *primitives* f. The difference lies in their semantics: primitives (such as $+$) are reduced when fully applied, while constructors (such as `cons`) can only be reduced by some other primitives.

We also assume given an initial context Γ_0 that assigns to every constant κ of arity n a type σ of the form $\forall\,(\overline{\alpha \geqslant \sigma})\ \sigma_1 \to \ldots \sigma_n \to \sigma_0$ where σ_0 is of the form $C\ \overline{\sigma}$ with $C \neq \to$ when κ is a constructor.[1]

Values The grammar of values is given in Figure 14.4.1. A value v is either an expression abstraction, a type abstraction, a constructor application, or a partially applied primitive. Applications of constants can involve a series of type computations, but only of a certain form, and before all other arguments. The set W over which w ranges depends on whether we are considering call-by-value or call-by-name reduction, and will be defined later. Notice that $a[\triangleright \sigma]$ and $a[\alpha \triangleleft]$ are *never* values, even though they are irreducible expressions.

Importantly, values have non-bottom types.

Lemma 14.4.6 *Let v be a value; if $\Gamma_0 \vdash v : \sigma$, σ is not $\bot$.* □

[1]It is possible to give more general types such as $\forall\,(\overline{\alpha \geqslant \sigma})\ \sigma_1 \to \forall\,(\overline{\beta \geqslant \sigma})\ \sigma_2 \to \ldots$ to constants, but this complicates the set of values and the definition of δ-rules.

$$\begin{array}{rcll} v & ::= & \lambda(x:\sigma)\ a & \\ & \mid & \Lambda(\alpha:\sigma)\ a & \\ & \mid & c\ [\theta_1]\ldots[\theta_k]\ w_1\ldots w_n & n \leq \mathsf{arity}(c) \\ & \mid & f\ [\theta_1]\ldots[\theta_k]\ w_1\ldots w_n & n < \mathsf{arity}(f) \\ \theta & ::= & \forall\,(\geqslant\varphi) \mid \forall\,(\alpha\geqslant)\ \varphi \mid \& & \end{array}$$

Figure 14.4.1 – Grammar of values

Proof: The proof is by case disjunction on the shape of v and unicity of typing. If v is an abstraction, it has an arrow type. It v is type abstraction, it has a type of the form $\forall\,(\alpha \geqslant \sigma')\ \sigma''$. If v is a non fully applied constant, it has a type of the form $\forall\,(\alpha \geqslant \sigma)\ \sigma'$ or $\sigma \to \sigma'$. If it is a fully applied constructor it cannot have type $\bot$ by the hypothesis on the return type σ_0.

Reduction rules for primitives We finally assume that there exists a set of reduction rules $\longrightarrow_\delta$, called δ-rules, that reduce fully applied primitives, as per the following hypothesis:

(H) Well-typed full applications of primitive can be reduced.

This is, for any expression a of the form $f\ [\theta_1]\ \ldots\ [\theta_k]\ w_1 \ldots w_n$ with $n = \mathsf{arity}(f)$ and verifying $\Gamma_0 \vdash a : \sigma$, then there exists an expression a' such that $a \longrightarrow_\delta a'$.

Of course, in order to ensure type soundness, we also assume that δ-rules preserve typings. However, we will not use this last hypothesis in this section.

14.4.2.1 Call-by-value reduction

For call-by-value reduction, constructors and primitives must be applied to values. Hence the set W is the set V of values itself, and the grammar of values is of the form

$$\begin{array}{rcll} v & ::= & \ldots & \\ & \mid & c\ [\theta_1]\ldots[\theta_k]\ v_1\ldots v_n & n \leq \mathsf{arity}(c) \\ & \mid & f\ [\theta_1]\ldots[\theta_k]\ v_1\ldots v_n & n < \mathsf{arity}(f) \end{array}$$

The rules (β) and (β_{LET}) are also limited to the substitution of values:

$$(\lambda(x:\sigma)\ a)\ v \longrightarrow a[v/x] \qquad\qquad \mathsf{let}\ x = v\ \mathsf{in}\ a \longrightarrow a[v/x]$$

The rules Reflex, Trans and Quant-Intro are also restricted so that they only apply on values (*e.g.* $v[\varepsilon]$ can be reduced by Reflex, but not $(a_1\ a_2)[\varepsilon]$). Finally, we restrict rule Context to contexts of the form

$$\begin{array}{rcl} E & ::= & \{\cdot\} \\ & \mid & E\ a \\ & \mid & v\ E \\ & \mid & E[\varphi] \\ & \mid & \mathsf{let}\ x = E\ \mathsf{in}\ a \end{array}$$

We write $\longrightarrow_{\mathsf{v}}$ the resulting reduction relation. It follows from the above restrictions that values are irreducible, and that the reduction is deterministic.

Property 14.4.7 *If v is a value, $v \not\longrightarrow_{\mathsf{v}}$.* □

Proof: Immediate induction on the shape of v.

Lemma 14.4.8 *If $a \longrightarrow_{\mathsf{v}} a'$ and $a \longrightarrow_{\mathsf{v}} a''$, then $a' = a''$.* □

Proof: For all the rules but Context it suffices to show that the reductions of a to a' and a'' are done by the same rule, as the rules are deterministic. The proof is by induction on the rule such that $a \longrightarrow_{\mathsf{v}} a'$.

- ▷ *Case* (β): a is of the form $(\lambda(x : \sigma)\ a')\ v$, and only (β) applies as we cannot reduce v or inside the abstraction.
- ▷ *Case* (β_{Let}): : a is of the form $\mathsf{let}\ x = v\ \mathsf{in}\ a'$. We cannot reduce in v or in a', hence only (β_{Let}) applies.
- ▷ *Case* Reflex: : then a is of the form $v[\varepsilon]$. Since v is irreducible, only Reflex applies.
- ▷ *Case* Trans or Quant-Intro: : the reasoning is the same as in the previous case
- ▷ *Case* Quant-Elim: : a is of the form $\Lambda(\alpha \geqslant \sigma)\ a'$. We cannot reduce a' as we cannot reduce under type abstractions, hence only Quant-Elim applies.
- ▷ *Case* Outer or Inner: : the reasoning is the same as above.
- ▷ *Case* Context: by the case disjunction above, $a \longrightarrow_{\mathsf{v}} a''$ is also by Context. Without loss of generality, given the results above we can suppose that the reduction does not occur in the context $\{\cdot\}$. Property 14.4.7 ensures that both reductions occur at the same subcontext, as an application can be reduced in only one way, and there is a single context for all the other type of terms. The conclusion is then by induction hypothesis.

We may now show progress for call-by-value, which in combination with subject-reduction ensures that evaluation of well-typed expressions "cannot go wrong".

Theorem 14.4.9 *If $\Gamma_0 \vdash a : \sigma$, either a is a value or there exists a' such that $a \longrightarrow_{\mathsf{v}} a'$.* □

Proof: By induction on the shape of a:

- ▷ *Case* x: variables are not typable in Γ_0;
- ▷ *Case* $\lambda(x : \sigma)\ a'$ or $\Lambda(\alpha \geqslant \sigma)\ a'$: then a is a value
- ▷ *Case* κ: if κ is a primitive f with $\mathsf{arity}(f) = 0$, it can be reduced by the appropriate δ-rule. All the other constants are values.
- ▷ *Case* $a_1\ a_2$: by inversion of typing, a_1 and a_2 are typable in Γ_0, of type $\sigma' \to \sigma$ and σ' for a certain σ' respectively. If a_1 is not a value, by induction hypothesis it can be reduced, and so can a by Context. If a_1 is a value but not a_2, a_2 can be reduced, and so can a by Context again. Otherwise, if both a_1 and a_2 are values, we proceed by case disjunction on the shape of a_1 (which, we remind, is a value of type $\sigma' \to \sigma$)
  - ◦ if $a' = \lambda(x : \sigma')\ a_1'$, a can be reduced by (β).

- a_1 cannot be a type abstraction, as it would not have an arrow type.
- if a_1 is a (partially) applied primitive, either a is a fully applied primitive and it can be reduced by the appropriate δ-rule, or a is a value.
- if a_1 is a partially applied constructor: by hypothesis on the typing of constructors, a_1 is of the form $c\,[\theta]\ldots[\theta]\;v_1\ldots v_n$ with $n < \mathsf{arity}(c)$ (as a full application would not have an arrow type). Then a is a value.

▷ *Case* $\mathsf{let}\;x = a_2\;\mathsf{in}\;a_1$: by inversion of typing a_2 is typable in Γ_0. If it is not a value, by induction hypothesis it can be reduced. Hence a can be reduced by rule CONTEXT. Otherwise a can be reduced by rule (β_{LET}).

▷ *Case* $a'[\varphi]$: by inversion of typing a' is typable in Γ_0. If a' is not a value, it can be further reduced by induction hypothesis, and so can a by CONTEXT. Otherwise we proceed by case analysis on φ:

- *Case* ε, $\mathbin{\rotatebox[origin=c]{180}{$\&$}}$ or $\varphi_1; \varphi_2$: a can be reduced by rules REFLEX, QUANT-INTRO or TRANS
- *Case* $\alpha\triangleleft$: this case is impossible in Γ_0, as a would not be well-typed.
- *Case* $\triangleright\sigma$: a' is a value which cannot have type $\bot$ by Lemma 14.4.6; hence this case is impossible.
- *Case* $\&$, $\forall\,(\alpha \geqslant)\;\varphi'$ or $\forall\,(\geqslant \varphi')$: a' must have type $\forall\,(\alpha \geqslant \sigma')\;\sigma''$ for some σ' and σ''. Since it is a value, by shape analysis on a' there are only two possible cases:
  - *Case* $a' = \Lambda(\alpha \geqslant \sigma)\;a''$: a can be reduced by INNER, OUTER or QUANT-ELIM.
  - *Case* $a' = \kappa\;[\theta_1]\ldots[\theta_k]$: then $a = \kappa\;[\theta_1]\ldots[\theta_k][\varphi]$, and it is a value, as φ is of the form θ.

14.4.2.2 Value-restriction

The *value-restriction* (Wright and Felleisen 1994) is the standard way to add side effects in a call-by-value language. It is thus important to verify that it can be transposed to $x\mathsf{ML}^{\mathsf{F}}$. This question may be surprising, as it is usually taken for granted. However, the type instantiation rules of $x\mathsf{ML}^{\mathsf{F}}$ are quite unusual, as they permit the creation of type abstraction by type reduction.

Typically, the *value restriction* amounts to restricting type generalizations to non-expansive expressions, which contain at least value-forms, *i.e.* values and term variables, as well as their type-instantiations. In an explicitly typed language such as ours, we in fact limit the source terms. Hence, we obtain the grammar of Figure 14.4.2 for *restricted* expressions r and *non-expansive* expressions u. As usual, we require let-bound expressions to be non-expansive, since they implicitly contain a type generalization.

Of course, we must suppose that δ-rules are well-behaved w.r.t. the value restriction:

(H') δ-rules reduce fully applied primitives into non-expansive expressions.

We can now show that the evaluation of expansive expressions cannot create polymorphism, a condition that is sufficient to ensure the soundness with side-effects. Indeed, the value-restriction is closed by reduction.

Lemma 14.4.10 *Non-expansive expressions are closed by call-by-value reduction.* □

Proof: Let u be a non-expansive expression. We prove by induction on the rule reducing u that it reduces to a non-expansive expression.

$$
\begin{array}{rcll}
r & ::= & u & \\
  & \mid & r\ r & \\
  & \mid & \mathsf{let}\ x = u\ \mathsf{in}\ r & \\
  & \mid & r[\varphi] & \\
u & ::= & x & \\
  & \mid & \lambda(x : \sigma)\ r & \\
  & \mid & \Lambda(\alpha : \sigma)\ u & \\
  & \mid & u\ [\varphi] & \\
  & \mid & c\ [\theta_1] \ldots [\theta_k]\ u_1 \ldots u_n & n \leq \mathsf{arity}(c) \\
  & \mid & f\ [\theta_1] \ldots [\theta_k]\ u_1 \ldots u_n & n < \mathsf{arity}(f)
\end{array}
$$

Figure 14.4.2 – $\mathsf{xML^F}$ with value-restriction

▷ Variables and abstractions are irreducible.

▷ Reductions of fully-applied primitives by δ-rules are by hypothesis **(H')**.

▷ Reductions of type applications or of arguments of constant by rule CONTEXT are by induction hypothesis.

▷ it remains to consider the reduction of a type application by a rule of $\longrightarrow_\Lambda$. Thus u is of the form $u'[\varphi]$, and we proceed by case disjunction on φ. If φ is of the form ε, ⅋ or $\varphi_1; \varphi_2$, it is immediate that u reduces to a non-expansive expression by rules REFLEX, QUANT-INTRO and TRANS. For the three other reduction rules, u is of the form $(\Lambda(\alpha \geqslant \sigma)\ u')[\theta]$.

∘ *Case* $\forall\,(\alpha \geqslant)\ \varphi$: then u reduces by OUTER to $\Lambda(\alpha \geqslant \sigma)\ u'[\varphi]$, which is indeed a non-expansive expression.

∘ *Case* &: then u reduces by QUANT-ELIM to $u'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$. Moreover $u'\{\alpha\triangleleft \leftarrow \varepsilon\}$ is a non-expansive expression; this is immediate for all the non-expansive expressions but constant applications. However, for this subcases, the substitutions θ do not contain $\alpha\triangleleft$ and are unchanged by the substitution. Since non-expansive expression are also stable by type substitution, this shows that $u'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$ is indeed an non-expansive expression.

∘ *Case* $\forall\,(\geqslant \varphi)$: then a reduces by INNER to $u'\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\}$. The reasoning is similar to the case above.

Lemma 14.4.11 *The value-restriction is closed by call-by-value reduction.* □

Proof: Let r be a restricted expression. We prove by induction on the rule reducing r that it reduces into a restricted expression. We only consider the cases when r is not an non-expansive expression, as this case is already proven by Lemma 14.4.10.

▷ *Case* CONTEXT: the result is by induction hypothesis.

▷ *Case* (β): then r is of the form $(\lambda(x : \sigma)\ r_1)\ v$, which reduces to $r_1\{x \leftarrow v\}$. Restricted expressions are stable by the substitution of values, which ensure the result.

▷ *Case* (β_{LET}): then r is of the form $\mathsf{let}\ x = v\ \mathsf{in}\ r_1$, which reduces to $r_1\{x \leftarrow v\}$. The conclusion is as above.

- ▷ <u>*Case δ*</u>: the result is by hypothesis **(H')**, as non-expansive expressions are restricted expressions.
- ▷ in the remaining case, r is a type application $r'[\varphi]$, and reduces by a rule of $\longrightarrow_{\mathsf{v}} \cap \longrightarrow_{\Lambda}$. This implies that r' is a value. Hence r is non expansive and the conclusion is by Lemma 14.4.10.

Hence, subject reduction holds with the value-restriction as well. It is then routine work to extend the semantics with a global store to model side effects and verify type soundness for this extension.

14.4.2.3 Call-by-name reduction

For call-by-name reduction semantics, partially applied constants do not require that their arguments are values, and the set W is actually the set A of all values

$$\begin{array}{rcll} v & ::= & \dots & \\ & \mid & c\ [\theta_1] \dots [\theta_k]\ a_1 \dots a_n & n \le \mathsf{arity}(c) \\ & \mid & f\ [\theta_1] \dots [\theta_k]\ a_1 \dots a_n & n < \mathsf{arity}(f) \end{array}$$

Rules Reflex, Trans and Quant-Intro are restricted to values, as for call-by-value reduction. Evaluation contexts are of the form

$$\begin{array}{rcl} E & ::= & \{\cdot\} \\ & \mid & E\ a \\ & \mid & E[\varphi] \end{array}$$

As usual, we can introduce strict type constructors or primitives (which require their arguments to be evaluated) by changing the set of values *and* adding the necessary evaluation contexts.

We write $\longrightarrow_{\mathsf{n}}$ the reduction relation obtained through the restrictions above. As for call-by-value, values are irreducible and, reduction is deterministic, and progress holds. The proofs are the same as for call-by-value for type reduction, and for ML for β-reduction; thus we omit them.

Property 14.4.12 *If v is a value, $v \not\longrightarrow_{\mathsf{n}}$.* □

Lemma 14.4.13 *If $a \longrightarrow_{\mathsf{n}} a'$ and $a \longrightarrow_{\mathsf{n}} a''$, then $a' = a''$.* □

Theorem 14.4.14 *If $\Gamma_0 \vdash a : \sigma$, either a is a value or there exists a' such that $a \longrightarrow_{\mathsf{n}} a'$.*□

14.5 Confluence of reduction

In this section, we show that strong reduction is a confluent relation on well-typed terms, ensuring that reductions can be performed in any order. We use the standard technique of parallel reductions (Barendregt 1984).

We first define a *parallel reduction* relation $\xrightarrow{/\!/}$, given in Figure 14.5.1. Most rules are immediate adaptations of the rules of Figure 14.3.1. For example, rule PBeta allows reducing both the argument and the function during β-reduction. The congruence rules PAbs, PApp, PTAbs and PTApp are essentially the result of inlining rule Context; notice however that rule PApp allows reducing both terms of the application simultaneously, unlike Context.

PBeta
$$\frac{a_1 \xrightarrow{//} a_1' \qquad a_2 \xrightarrow{//} a_2'}{(\lambda(x : \sigma)\ a_1)\ a_2 \xrightarrow{//} a_1'\{x \leftarrow a_2'\}}$$

PBetaLet
$$\frac{a_1 \xrightarrow{//} a_1' \qquad a_2 \xrightarrow{//} a_2'}{\mathsf{let}\ x = a_2\ \mathsf{in}\ a_1 \xrightarrow{//} a_1'\{x \leftarrow a_2'\}}$$

PVarRefl
$$\frac{}{x \xrightarrow{//} x}$$

PAbs
$$\frac{a \xrightarrow{//} a'}{\lambda(x : \sigma)\ a \xrightarrow{//} \lambda(x : \sigma)\ a'}$$

PApp
$$\frac{a_1 \xrightarrow{//} a_1' \qquad a_2 \xrightarrow{//} a_2'}{a_1\ a_2 \xrightarrow{//} a_1'\ a_2'}$$

PTApp
$$\frac{a \xrightarrow{//} a'}{a[\varphi] \xrightarrow{//} a'[\varphi]}$$

PTAbs
$$\frac{a \xrightarrow{//} a'}{\Lambda(\alpha \geqslant \sigma)\ a \xrightarrow{//} \Lambda(\alpha \geqslant \sigma)\ a'}$$

PLet
$$\frac{a_1 \xrightarrow{//} a_1' \qquad a_2 \xrightarrow{//} a_2'}{\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 \xrightarrow{//} \mathsf{let}\ x = a_1'\ \mathsf{in}\ a_2'}$$

PQuant-Intro
$$\frac{a \xrightarrow{//} a' \qquad \alpha \notin \mathsf{fv}(a)}{a[\invamp; \forall\,(\geqslant \triangleright \sigma)] \xrightarrow{//} \Lambda(\alpha \geqslant \bot)\ a'}$$

PQuant-Elim
$$\frac{a \xrightarrow{//} a'}{(\Lambda(\alpha \geqslant \sigma)\ a)[\&] \xrightarrow{//} a'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}}$$

POuter
$$\frac{a \xrightarrow{//} a'}{(\Lambda(\alpha \geqslant \sigma)\ a)[\forall\,(\alpha \geqslant)\ \varphi] \xrightarrow{//} \Lambda(\alpha \geqslant \sigma)\ (a'[\varphi])}$$

PTrans
$$\frac{a \xrightarrow{//} a'}{a[\varphi_1; \varphi_2] \xrightarrow{//} (a'[\varphi_1])[\varphi_2]}$$

PInner
$$\frac{a \xrightarrow{//} a'}{(\Lambda(\alpha \geqslant \sigma)\ a)[\forall\,(\geqslant \varphi)] \xrightarrow{//} \Lambda(\alpha \geqslant \sigma[\varphi])\ (a'\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\})}$$

PReflex
$$\frac{a \xrightarrow{//} a'}{a[\varepsilon] \xrightarrow{//} a'[\varepsilon]}$$

Figure 14.5.1 – Parallel reduction

Rule PVarRefl expresses that a variable reduces to itself. This ensures that $\xrightarrow{//}$ is reflexive.

Lemma 14.5.1 *The relation $\xrightarrow{//}$ is reflexive.* □

Proof: Immediate induction on the shape of the expression, using PVarRefl and the congruence rules.

The relation $\xrightarrow{//}$ is a subrelation of $\longrightarrow^\star$, and is designed so that its reflexive transitive closure is the same as the one of $\longrightarrow$.

Lemma 14.5.2 *The relations $\longrightarrow \subseteq \xrightarrow{//} \subseteq \longrightarrow^\star$ hold.* □

Proof: The inclusion $(\longrightarrow) \subseteq (\xrightarrow{/\!/})$ is immediate by definition of $\xrightarrow{/\!/}$ and Lemma 14.5.1. The inclusion $(\xrightarrow{/\!/}) \subseteq (\longrightarrow^\star)$ is immediate by induction on a derivation $a \xrightarrow{/\!/} a'$.

Corollary 14.5.3 *The relation $(\longrightarrow)^\star = (\xrightarrow{/\!/})^\star$ hold.* □

Our final result is the confluence of $\xrightarrow{/\!/}$, which implies the confluence of $\longrightarrow$ by the result above. However, we first need stating a few intermediary results for the compatibility of $\xrightarrow{/\!/}$ with the various forms of substitutions. The proofs simple (if tedious).

Lemma 14.5.4 *If $a \xrightarrow{/\!/} a'$ and $b \xrightarrow{/\!/} b'$, then $a\{x \leftarrow b\} \xrightarrow{/\!/} a'\{x \leftarrow b'\}$.* □

Proof: The proof is by induction on $a \xrightarrow{/\!/} a'$. All the cases are immediate by induction hypothesis; we detail two significant ones below.

▷ *Case* PBeta: then $a = (\lambda(y)\ a_1)\ a_2$; without loss of generality we suppose that $y \notin \mathsf{fv}(b) \cup \mathsf{fv}(b')$ (**1**). Then $a' = a'_1\{y \leftarrow a'_2\}$ with $a_i \xrightarrow{/\!/} a'_i$. We have $a\{x \leftarrow b\} = (\lambda(y)\ a_1\{x \leftarrow b\})\ (a_2\{x \leftarrow b\})$ by (1), which reduces by induction hypothesis and PBeta to $a'_1\{x \leftarrow b'\}\{y \leftarrow a'_2\{x \leftarrow b'\}\}$, which is equal to $a'_1\{y \leftarrow a'_2\}\{x \leftarrow b'\}$ by (1).

▷ *Case* PInner: then $a = (\Lambda(\alpha \geqslant \sigma)\ a_1)[\forall\,(\geqslant \varphi)]$; without loss of generality we suppose that $\alpha \notin \mathsf{fv}(b) \cup \mathsf{fv}(b')$ (**2**). Then $a' = \Lambda(\alpha \geqslant \sigma[\varphi])\ a'_1\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\}$ with $a_1 \xrightarrow{/\!/} a'_1$. We have $a\{x \leftarrow b\} = (\Lambda(\alpha \geqslant \sigma)\ a_1\{x \leftarrow b\})[\forall\,(\geqslant \varphi)]$ by (2), which reduces to $\Lambda(\alpha \geqslant \sigma[\varphi])\ a'_1\{x \leftarrow b'\}\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\}$ by PInner and induction hypothesis. This term is equal to $(\Lambda(\alpha \geqslant \sigma[\varphi])\ a'_1\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\})\{x \leftarrow b'\}$ by (2), which is the desired result.

Lemma 14.5.5 *Let a be a term such that there exists a context $\Gamma', \alpha \geqslant \sigma, \Gamma''$ under which a is well-typed. Suppose $a \xrightarrow{/\!/} a'$. Then*

1. *$a\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} \xrightarrow{/\!/} a'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$.*

2. *if $\Gamma' \vdash \varphi : \sigma \leq \sigma'$ holds for a certain σ', then $a\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\} \xrightarrow{/\!/} a'\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\}$.* □

Proof: We let Γ be $\Gamma', \alpha \geqslant \sigma, \Gamma''$. In both cases, the proof is by induction on $a \xrightarrow{/\!/} a'$; we detail only the cases for PInner and PQuant-Elim, as all the other cases are simply by induction hypothesis.

For $a\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\} \xrightarrow{/\!/} a'\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\}$:

▷ *Case* PInner: then $a = (\Lambda(\beta \geqslant \sigma'')\ b)[\forall\,(\geqslant \varphi')]$ and $a' = \Lambda(\beta \geqslant \sigma''[\varphi'])\ (b'\{\beta\triangleleft \leftarrow \varphi'; \beta\triangleleft\})$ with $b \xrightarrow{/\!/} b'$. Without loss of generality we suppose that β does not appear free in φ and σ, and that α and β are distinct (**1**). We have $a\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\} = (\Lambda(\beta \geqslant \sigma'')\ b\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\})[\forall\,(\geqslant \varphi'\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\})]$ by (1). By induction hypothesis, $b\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\} \xrightarrow{/\!/} b'\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\}$. Thus $a\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\} \xrightarrow{/\!/} \Lambda(\beta \geqslant \sigma''[\varphi'\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\}])\ b'\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\}\{\beta\triangleleft \leftarrow \varphi'\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\}; \beta\triangleleft\}$ by PInner (**2**). Since β does not appear in φ and $\alpha \neq \beta$, we have $b'\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\}\{\beta\triangleleft \leftarrow \varphi'\{\alpha\triangleleft \leftarrow \varphi; \alpha\triangleleft\}; \beta\triangleleft\} =$

$b'\{\beta\triangleleft \leftarrow \varphi';\beta\triangleleft\}\{\alpha\triangleleft \leftarrow \varphi;\alpha\triangleleft\}$ (**3**). Let σ''' be such that $\Gamma \vdash \varphi' : \sigma'' \leq \sigma'''$. By Lemma 14.4.4, $\Gamma \vdash \varphi'\{\alpha\triangleleft \leftarrow \varphi;\alpha\triangleleft\} : \sigma'' \leq \sigma'''$. Thus $\sigma''[\varphi'\{\alpha\triangleleft \leftarrow \varphi;\alpha\triangleleft\}] = \sigma''[\varphi']$ (**4**). By (2), (3) and (4), we have $a\{\alpha\triangleleft \leftarrow \varphi;\alpha\triangleleft\} \xrightarrow{/\!/} a'\{\alpha\triangleleft \leftarrow \varphi;\alpha\triangleleft\}$.

▷ *Case* PQuant-Elim: then $a = (\Lambda(\beta \geqslant \sigma'')\ b)[\&]$ and $a' = b'\{\beta\triangleleft \leftarrow \varepsilon\}\{\beta \leftarrow \sigma''\}$ with $b \xrightarrow{/\!/} b'$. Without loss of generality we can suppose that β does not appear free in φ and σ, and that α and β are distinct. Then we have $a\{\alpha\triangleleft \leftarrow \varphi;\alpha\triangleleft\} = (\Lambda(\alpha \geqslant \sigma'')\ b\{\alpha\triangleleft \leftarrow \varphi;\alpha\triangleleft\})[\&]$. By induction hypothesis and PQuant-Elim, we have $a\{\alpha\triangleleft \leftarrow \varphi;\alpha\triangleleft\} \xrightarrow{/\!/} b'\{\alpha\triangleleft \leftarrow \varphi;\alpha\triangleleft\}\{\beta\triangleleft \leftarrow \varepsilon\}\{\beta \leftarrow \sigma''\}$. Since β does not appear in φ and $\alpha \neq \beta$, this last term is $b'\{\beta\triangleleft \leftarrow \varepsilon\}\{\beta \leftarrow \sigma''\}\{\alpha\triangleleft \leftarrow \varphi;\alpha\triangleleft\}$, hence the result.

For $a \xrightarrow{/\!/} a'$, $a\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} \xrightarrow{/\!/} a'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$ we use the same notations and freshness conventions as above.

▷ *Case* PInner: by definition, we have $a\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} = (\Lambda(\beta \geqslant \sigma''\{\alpha \leftarrow \sigma\})\ b\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\})[\forall\,(\geqslant \varphi'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\})]$. By induction hypothesis, $b\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} \xrightarrow{/\!/} b'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$. Thus $a\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} \xrightarrow{/\!/} \Lambda(\beta \geqslant \sigma''\{\alpha \leftarrow \sigma\}[\varphi'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}])\ b'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}\{\beta\triangleleft \leftarrow \varphi'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\};\beta\triangleleft\}$ holds by PInner (**5**). By definition of substitutions, we have $b'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}\{\beta\triangleleft \leftarrow \varphi'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\};\beta\triangleleft\} = b'\{\beta\triangleleft \leftarrow \varphi';\beta\triangleleft\}\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$ (**6**). Let σ''' be such that $\Gamma \vdash \varphi' : \sigma'' \leq \sigma'''$. By Lemma 14.4.3, $\Gamma', \Gamma''\{\alpha \leftarrow \sigma\} \vdash \varphi'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} : \sigma''\{\alpha \leftarrow \sigma\} \leq \sigma'''\{\alpha \leftarrow \sigma\}$. Thus $\sigma''\{\alpha \leftarrow \sigma\}[\varphi'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}] = \sigma''[\varphi']\{\alpha \leftarrow \sigma\}$ (**7**). We have $a\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} \xrightarrow{/\!/} a'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$ by (5), (6) and (7).

▷ *Case* PQuant-Elim: by definition we have $a\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} = (\Lambda(\alpha \geqslant \sigma''\{\alpha \leftarrow \sigma\})\ b\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\})[\&]$. By induction hypothesis and PQuant-Elim, we have $a\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\} \xrightarrow{/\!/} b'\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}\{\beta\triangleleft \leftarrow \varepsilon\}\{\beta \leftarrow \sigma''\{\alpha \leftarrow \sigma\}\}$. Since β does not appear in σ and $\alpha \neq \beta$, this last term is $b'\{\beta\triangleleft \leftarrow \varepsilon\}\{\beta \leftarrow \sigma''\}\{\alpha\triangleleft \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$, which is the desired result.

Notice the well-typedness hypotheses, made necessary by the fact that the function applying a computation to a type is partial. Otherwise, if we rewrite a computation in a way incompatible with the types on which it is used, the reduction could get stuck at different points, depending on the reduction strategy we follow.

▶ **Example** Consider the expression

$$(\Lambda(\alpha \geqslant \forall\,(\gamma \geqslant \bot)\ \gamma)\ (\Lambda(\beta \geqslant \mathsf{int})\ x)[\forall\,(\geqslant \alpha\triangleleft)])[\forall\,(\geqslant \&)]$$

It is ill-typed in any context, because $\alpha\triangleleft$ coerces terms of type $\forall\,(\gamma \geqslant \bot)\ \gamma$ into α, but here it is indirectly applied on an expression of type int. If we reduce the innermost type application, we obtain the expression

$$(\Lambda(\alpha \geqslant \forall\,(\gamma \geqslant \bot)\ \gamma)\ (\Lambda(\beta \geqslant \alpha)\ x))[\forall\,(\geqslant \&)]$$

which in turn reduces to

$$(\Lambda(\alpha \geqslant \bot)\ (\Lambda(\beta \geqslant \alpha)\ x))$$

Meanwhile, if we reduce the rightmost type application in the first expression, we obtain the expression

$$\Lambda(\alpha \geqslant \bot)\ (\Lambda(\beta \geqslant \mathsf{int})\ x)[\forall\,(\geqslant \&;\alpha\triangleleft)]$$

This time, we cannot reduce the remaining type application, since $\mathsf{int}[\&]$ is undefined. Thus we have to rule out those ill-typed terms.

Our main result is the fact that $\xrightarrow{/\!/}$ has the diamond property.

Theorem 14.5.6 (Confluence of $\xrightarrow{/\!/}$) *Let a be a term, well-typed in some environment Γ. If $a \xrightarrow{/\!/} a'$ and $a \xrightarrow{/\!/} a''$, then there exists a''' such that $a' \xrightarrow{/\!/} a'''$ and $a'' \xrightarrow{/\!/} a'''$.* □

Proof: The proof is by induction on $a \xrightarrow{/\!/} a'$.

- ▷ *Case* PBeta: Then $a = (\lambda(x : \sigma)\ a_1)\ a_2$ and a' is $a'_1\{x \leftarrow a'_2\}$ with $a_i \xrightarrow{/\!/} a'_i$. We proceed by case analysis on $a \xrightarrow{/\!/} a''$.
  - ○ *Case* PBeta: then $a'' = a''_1\{x \leftarrow a''_2\}$ with $a_i \xrightarrow{/\!/} a''_i$. By induction hypothesis there exists a'''_i such that $a'_i, a''_i \xrightarrow{/\!/} a'''_i$. By Lemma 14.5.4, $a', a'' \xrightarrow{/\!/} a'''_1\{x \leftarrow a'''_2\}$.
  - ○ *Case* PApp: Then there exists b and a''_2 such that $\lambda(x : \sigma)\ a_1 \xrightarrow{/\!/} b$ and $a_2 \xrightarrow{/\!/} a''_2$. For the first reduction, the only possibility is by PAbs, and there exists a''_1 such that $a_1 \xrightarrow{/\!/} a''_1$. By induction hypothesis (IH), there exists a'''_1 and a'''_2 such that $a'_i, a''_i \xrightarrow{/\!/} a'''_i$. Thus $a' \xrightarrow{/\!/} a'''_1\{x \leftarrow a''''_2\}$ by Lemma 14.5.4 and $a'' \xrightarrow{/\!/} a'''_1\{x \leftarrow a''''_2\}$ by PBeta.
- ▷ *Case* PBetaLet: then $a = \mathsf{let}\ x = a_2\ \mathsf{in}\ a_1$ and a' is $a'_1\{x \leftarrow a'_2\}$ with $a_i \xrightarrow{/\!/} a'_i$. We proceed by case analysis on $a \xrightarrow{/\!/} a''$.
  - ○ *Case* PBetaLet: this case is similar to the case PBeta/PBeta above
  - ○ *Case* PLet: then a'' is $\mathsf{let}\ x = a''_1\ \mathsf{in}\ a''_2$ with $a_i \xrightarrow{/\!/} a''_i$. By IH, there exists a'''_i such that $a_i, a''_i \xrightarrow{/\!/} a'''_i$. Then $a' \xrightarrow{/\!/} a'''_1\{x \leftarrow a'''_2\}$ by Lemma 14.5.4 and $a'' \xrightarrow{/\!/} a'''_2\{x \leftarrow a'''_2\}$ by PLet.
- ▷ *Case* PVarRefl: In this case, $a = a' = a''$, and the conclusion is by Lemma 14.5.1.
- ▷ *Case* PAbs: Then $a = \lambda(x : \sigma)\ b$ and $a' = \lambda(x : \sigma)\ b'$. Moreover, $a \xrightarrow{/\!/} a''$ must be by PAbs, and there exists b'' such that $a'' = \lambda(x : \sigma)\ b''$. By IH there exists b''' such that $b', b'' \xrightarrow{/\!/} b'''$. Hence $a', a'' \xrightarrow{/\!/} \lambda(x : \sigma)\ b'''$ by PAbs.
- ▷ *Case* PTAbs: the reasoning is the same as in the previous case.
- ▷ *Case* PApp: Then $a = a_1\ a_2$ and $a' = a'_1\ a'_2$. We proceed by case analysis on $a \xrightarrow{/\!/} a''$.
  - ○ *Case* PBeta: the symmetrical case has been handled above.
  - ○ *Case* PApp: Then $a'' = a''_1\ a''_2$. By IH, there exists a'''_1 and a'''_2 such that $a'_i, a''_i \xrightarrow{/\!/} a'''_i$. Hence $a', a'' \xrightarrow{/\!/} a'''_1\ a'''_2$ by PApp.
- ▷ *Case* Plet: this case is similar to the one above, with PBeta instead of PBetaLet.
- ▷ *Case* PTApp: Then $a = b[\varphi]$ and $a' = b'[\varphi]$. We proceed by case disjunction on $a \xrightarrow{/\!/} a''$.
  - ○ *Case* PTApp: the case is similar to the case PAbs/PAbs above.
  - ○ *Case* PReflex: Then φ is ε and a'' is b''. By IH, there exists b''' such that $b', b'' \xrightarrow{/\!/} b'''$. Thus $a' \xrightarrow{/\!/} b'''$ by PReflex and $a'' \xrightarrow{/\!/} b'''$ by hypothesis.
  - ○ *Case* PTrans: Then φ is $\varphi_1; \varphi_2$ and a'' is $(b''[\varphi_1])[\varphi_2]$. By IH, there exists b''' such that $b', b'' \xrightarrow{/\!/} b'''$ and $a', a'' \xrightarrow{/\!/} (b'''[\varphi_1])[\varphi_2]$ by PTrans and congruence respectively.

- *Case* PQUANT-INTRO: Then φ is $\parr$ and a'' is $\Lambda(\alpha \geqslant \bot)\ b''$ with $\alpha \notin \mathsf{fv}(b)$. By IH, there exists b''' such that $b', b'' \xrightarrow{/\!/} b'''$ and $a', a'' \xrightarrow{/\!/} \Lambda(\alpha \geqslant \bot)\ b'''$ by PFRESH and congruence respectively (in the second case, the relation $\alpha \notin \mathsf{fv}(b')$ holds, since reduction does not introduce free variables).
- *Case* PQUANT-ELIM: Then φ is $\&$, b is $\Lambda(\alpha \geqslant \sigma)\ c$, a'' is $c''\{\alpha\lhd \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$ and b' is $\Lambda(\alpha \geqslant \sigma)\ c'$ (as only PTABS applies for $b \xrightarrow{/\!/} b'$), with $c \xrightarrow{/\!/} c', c''$. By IH, there exists c''' such that $c', c'' \xrightarrow{/\!/} c'''$. Thus $a' \xrightarrow{/\!/} c'''\{\alpha\lhd \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$ by PQUANT-ELIM and $a'' \xrightarrow{/\!/} c'''\{\alpha\lhd \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$ by Lemma 14.5.5, point 1.
- *Case* POUTER: Then φ is $\forall\,(\alpha \geqslant)\ \varphi'$, b is $\Lambda(\alpha \geqslant \sigma)\ c$, a'' is $\Lambda(\alpha \geqslant \sigma)\ (c''[\varphi'])$ and b' is $\Lambda(\alpha \geqslant \sigma)\ c'$ (again, only PTABS applies), with $c \xrightarrow{/\!/} c', c''$. By IH, there exists c''' such that $c', c'' \xrightarrow{/\!/} c'''$. Thus $a', a'' \xrightarrow{/\!/} \Lambda(\alpha \geqslant \sigma)\ (c'''[\varphi'])$ by POUTER and congruence respectively.
- *Case* PINNER: Then φ is $\forall\,(\geqslant \varphi')$, b is $\Lambda(\alpha \geqslant \sigma)\ c$, a'' is $\Lambda(\alpha \geqslant \sigma[\varphi'])\ c''\{\alpha\lhd \leftarrow \sigma; \alpha\lhd\}$ and b' is $\Lambda(\alpha \geqslant \sigma)\ c'$ (only PTABS applies), with $c \xrightarrow{/\!/} c', c''$. By IH, there exists c''' such that $c', c'' \xrightarrow{/\!/} c'''$. Thus $a', a'' \xrightarrow{/\!/} \Lambda(\alpha \geqslant \sigma[\varphi])\ c'''\{\alpha\lhd \leftarrow \sigma; \alpha\lhd\}$ by PINNER for a', and congruence and Lemma 14.5.5, point 2 for a''.

▷ *Case* PREFLEX: The case $a \xrightarrow{/\!/} a''$ by PTAPP is symmetrical to a case already handled. It remains $a \xrightarrow{/\!/} a''$ by PREFLEX. Thus $a = b[\varepsilon]$ and $a' = b'$, $a'' = b''$ with $a \xrightarrow{/\!/} a', a''$. The result is by induction hypothesis on b, b' and b''.

▷ *Case* PTRANS: The case $a \xrightarrow{/\!/} a''$ by PTAPP is symmetrical to a case already handled. It remains $a \xrightarrow{/\!/} a''$ by PTRANS. Thus $a = b[\varphi_1; \varphi_2]$ and $a' = (b'[\varphi_1])[\varphi_2]$, $a'' = (b''[\varphi_1])[\varphi_2]$ with $a \xrightarrow{/\!/} a', a''$. By induction hypothesis, there exists b''' such that $b', b'' \xrightarrow{/\!/} b'''$. Thus $a', a'' \xrightarrow{/\!/} (b'''[\varphi_1])[\varphi_2]$ by congruence.

▷ *Case* PQUANT-INTRO: it is similar to the case above.

▷ *Case* PSUBST: then $a = (\Lambda(\alpha \geqslant \sigma)\ b)[\&]$ and a' is $b'\{\alpha\lhd \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$ with $b \xrightarrow{/\!/} b'$. We proceed by case analysis on $a \xrightarrow{/\!/} a''$.

- *Case* PTAPP: it is symmetrical to a case already handled.
- *Case* PSUBST: then $a'' = b''\{\alpha\lhd \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$, with $b \xrightarrow{/\!/} b''$. By induction hypothesis, there exists b''' such that $b', b'' \xrightarrow{/\!/} b'''$. Thus $a', a'' \xrightarrow{/\!/} b'''\{\alpha\lhd \leftarrow \varepsilon\}\{\alpha \leftarrow \sigma\}$ by Lemma 14.5.5, point 1.

▷ *Case* POUTER: is it similar to the subcase above, except that the conclusion is by congruence instead of Lemma 14.5.5.

▷ *Case* PINNER: it is also similar to the subcase above, except that the conclusion is by point 2 instead of point 1.

As an immediate corollary:

Theorem 14.5.7 *The relation $\longrightarrow$ is confluent on well-typed terms.* □

Proof: By Lemma 14.5.2, we have $\longrightarrow^* = \xrightarrow{/\!/}^*$. Theorem 14.5.6 ensures that $\xrightarrow{/\!/}^*$ is confluent, hence the result.

Moreover, although is it not a corollary of the previous results *per se*, an examination of the proofs above shows that both $\longrightarrow_\beta$ and $\longrightarrow_\Lambda$ are also confluent.

Theorem 14.5.8 *The relation* $\longrightarrow_\beta$ *is confluent. The relation* $\longrightarrow_\Lambda$ *is confluent on well-typed terms.* $\square$

In particular, this result shows that β-reduction and type reduction are mostly independent. This is of particular interest in a language with side-effects, in which we could reduce type applications without interference with β-reduction.

14.6 A formal proof of *x*ML$^{\mathsf{F}}$?

We have used the Coq proof assistant (Coq development team 2007) to formally prove the type soundness of a preliminary version of *x*ML$^{\mathsf{F}}$. The main difference with the current presentation of the system was that the computations ε, $\triangleright\sigma$ and $\alpha\triangleleft$ were merged into a single computation $\rightarrow\sigma$, which witnessed either $\sigma \leq \sigma$, $\bot \leq \sigma$ or $\sigma \leq \alpha$.

Our development uses a locally nameless representation of terms, and cofinite quantification for free variable names, following Aydemir *et al.* (2008a). As suggested by the authors, the development is split in three parts:

1. The trusted definitions contains the syntax, typing rules and reductions rules of *x*ML$^{\mathsf{F}}$, as well as the statements of the type soundness theorems.

2. The *infrastructure* sets up the machinery for the core proofs. It includes auxiliary results about substitutions, proofs on the well-formedness of terms and environments, and a fair amount of Coq tactics to partly automatize the development.

3. The core lemmas part contains the lemmas that would normally be stated in an informal presentation.

We have found the proofs of the main results to be very natural, and very close to the pen-and-paper proofs. However, in our experience, the size of the infrastructure is very important. In fact, as shown by the table below, it represents significantly more than half the size of the whole development

Trusted definitions	Infrastructure	Main results
421 lines	1514 lines	652 lines

Infrastructure results are often simple, but also very tedious to state and prove; so it is a bit disappointing to spend so much time on them. Moreover, as noted by Aydemir *et al.* (2008b), the number of substitution-related results, hence the size of the infrastructure itself, grows quadratically with the number of sorts of variables in the language. In *x*ML$^{\mathsf{F}}$ we have type, term and substitution variables, and the number of needed results is quite high. Moreover, this hinders the readability of the proofs: while Coq allows defining custom syntax, for example for substitution, we need a different syntax for *e.g.* the substitution of a type variable in a type, a term, or a computation.

Another issue concerns the handling of renaming lemmas. Those lemmas allow changing the name of a variable in a typing derivation. As noticed by Aydemir *et al.* (2008a), they can be derived for free from the weakening and substitution lemmas. However, in one case we had no need for the substitution lemma (which did not even exist, as its statement would have been ill-sorted). In this case, an external tool deriving a renaming lemma, such as the Nominal package for Isabelle (Urban 2008), would have been very handy.

15

Translating gML$^\mathsf{F}$ into xML$^\mathsf{F}$

Abstract

Pursuing our goal to use xML$^\mathsf{F}$ as an internal language for ML$^\mathsf{F}$, we explain how to translate a presolution of gML$^\mathsf{F}$, eML$^\mathsf{F}$ or iML$^\mathsf{F}$ into a well-typed xML$^\mathsf{F}$ term. We start by presenting some examples (§15.1), then detail which presolutions can be translated, and which cannot (§15.2). The translation for gML$^\mathsf{F}$ is presented in §15.3. We explain how the translation could be improved, so as to obtain smaller xML$^\mathsf{F}$ terms (§15.4). We also adapt our approach to eML$^\mathsf{F}$ and iML$^\mathsf{F}$ (§15.5). Finally, we discuss the work needed to translate the syntactic presentations of iML$^\mathsf{F}$ and eML$^\mathsf{F}$ into xML$^\mathsf{F}$ (§15.6).

15.1 An introductory example

15.1.1 Our approach

Typable terms of gML$^\mathsf{F}$, eML$^\mathsf{F}$ or even iML$^\mathsf{F}$ can be translated into xML$^\mathsf{F}$ terms. However we do not instrument the type inference and unification algorithms to return an xML$^\mathsf{F}$ term—there are some complications in doing so, which we develop in §15.3.10. Instead, we elaborate *presolutions*, which contain all the needed information, into xML$^\mathsf{F}$ terms. More precisely:

- following the interpretation of gen nodes in terms of generalization levels, we translate the nodes bound on a gen node into xML$^\mathsf{F}$ type abstractions.
- the fact that an instantiation edge $g \overset{i}{\dashrightarrow} d$ is solved means that the type τ' under d is an instance of the type scheme τ represented by $\langle g \cdot i \rangle$. If σ and σ' are xML$^\mathsf{F}$ translation of τ and τ', we can find a type computation φ witnessing $\sigma \leq \sigma'$, and insert it at the appropriate place in the elaborated term.

In this section, we however do not explain in detail these two steps. Instead, we simultaneously present some already elaborated terms and the presolutions they correspond to. Hopefully, those examples should provide good intuitions for the formal development in the next sections.

15.1.2 Example

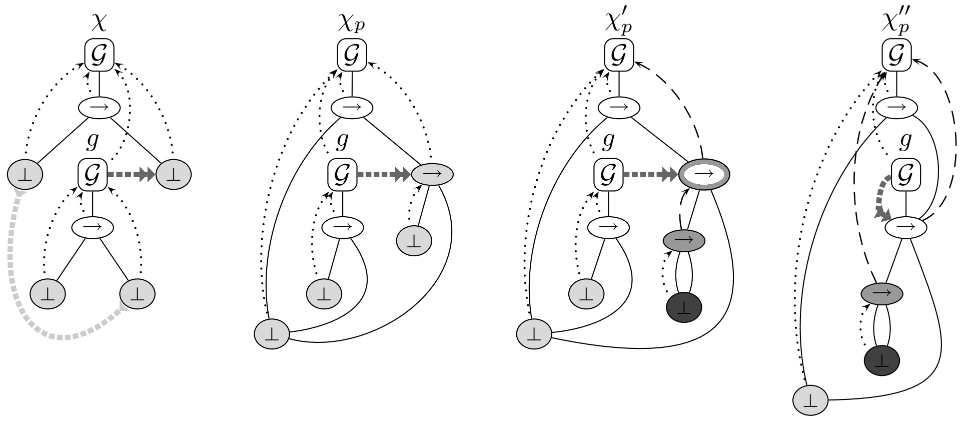


Figure 15.1.1 – Some presolutions of $\lambda(x)\ \lambda(y)\ x$

Let us consider the term K defined as $\lambda(x)\ \lambda(y)\ x$. The corresponding typing constraint is χ. Notice that we have used rule Var-Abs in order to obtain a simpler constraint, and simpler examples; this is discussed in more detail in §15.4.2.

▶ **A first presolution** Let us consider χ_p, which is a presolution of χ. In this presolution, K itself has type

$$\forall\,(\alpha \geqslant \bot)\ \forall\,(\beta \geqslant \forall\,(\delta \geqslant \bot)\ \delta \to \alpha)\ \alpha \to \beta$$

while the subterm $\lambda(y)\ x$ has type $\forall\,(\gamma)\ \gamma \to \alpha$. We thus introduce three type abstractions, for α, β and γ respectively. Moreover, the first two are introduced before the λ-abstraction $\lambda(x)$ _ , while the last one is added before the abstraction $\lambda(y)\ x$. This results in the xML$^{\mathsf{F}}$ term

$$\Lambda(\alpha \geqslant \bot)\ \Lambda(\beta \geqslant \forall\,(\delta \geqslant \bot)\ \delta \to \alpha)\ \lambda(x : \alpha)\ \Lambda(\gamma \geqslant \bot)\ \lambda(y : \gamma)\ x$$

While this term is well-typed, it has type

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

which is not quite correct yet: the codomain of the toplevel arrow is erroneous. This can however be fixed by abstracting the type of the abstraction $\lambda(y)\ x$ under the name β, which can be done using computation $\beta\triangleleft$. This results in the term below, which has the first type given above.

$$\Lambda(\alpha \geqslant \bot)\ \Lambda(\beta \geqslant \forall\,(\delta \geqslant \bot)\ \delta \to \alpha)\ \lambda(x : \alpha)\ (\Lambda(\gamma \geqslant \bot)\ \lambda(y : \gamma)\ x)[\beta\triangleleft]$$

▶ **Another presolution** The constraint χ_p is the principal presolution for the term K. Another presolution is χ'_p, in which we have instantiated the node $\langle 12 \rangle$. Interestingly, the term $\lambda(y)\ x$ itself is typed identically in both χ_p and χ'_p; the difference lies in the way this term is used as the codomain of the abstraction $\lambda(x)\ \_$. On the elaborated xML$^{\mathsf{F}}$ term representing χ'_p (which is given below), this is reflected by the fact that the λ-term for $\lambda(y)\ x$ does not change: only the computation does.

Notice also that χ'_p uses rigid quantification. However, since this form of quantification is only useful for type inference purposes (and is not present in xML$^{\mathsf{F}}$ anyway), we inline all the rigidly quantified types when translating presolutions into xML$^{\mathsf{F}}$. Thus the abstraction on β in the previous example disappears, as this bound is inlined.

Finally, let us describe the computation needed to instantiate the subterm $\lambda(x)\ y$. As in χ_p, this subterm has type $\forall\,(\gamma \geqslant \bot)\ \gamma \to \alpha$. However, it is used with type $(\forall\,(\epsilon \geqslant \bot)\ \epsilon \to \epsilon) \to \alpha$. Hence, it suffices to instantiate γ by the type $\forall\,(\epsilon \geqslant \bot)\ \epsilon \to \epsilon$. This results in the term

$$\Lambda(\alpha \geqslant \bot)\ \lambda(x:\alpha)\ (\Lambda(\gamma \geqslant \bot)\ \lambda(y:\gamma)\ x)[\forall\,(\epsilon \geqslant \bot)\ \epsilon \to \epsilon]$$

which has type $\forall\,(\alpha \geqslant \bot)\ \alpha \to (\forall\,(\epsilon \geqslant \bot)\ \epsilon \to \epsilon) \to \alpha$.

▶ **A third possibility** Finally, while preserving the type of K in our second example, we could have instantiated the subconstraint corresponding to $\lambda(y)\ x$; this results in the constraint χ''_p. This time, the λ-term for $\lambda(y)\ x$ changes. Moreover, since the corresponding gen node is degenerate, its only instance is itself. Hence there is no computation to add, and χ''_p elaborates into the λ-term

$$\Lambda(\alpha \geqslant \bot)\ \lambda(x:\alpha)\ \lambda(y:\forall\,(\epsilon \geqslant \bot)\ \epsilon \to \epsilon)\ x$$

Interestingly, this term is also the result of fully reducing the term returned by the elaboration of χ'_p. (Which also shows that both terms have the same type, by preservation of typing.)

15.2 Translatable presolutions

15.2.1 Pitfalls of the translation

Let us detail the translation a little more. We consider a λ-term a, χ its corresponding typing constraint, and χ_p a presolution of χ. Elaborating χ_p into an xML$^{\mathsf{F}}$ term is essentially a two step process:

1. Given a subterm a' of a, let g be the gen node corresponding to a' in χ. In the elaborated term for a, we add type abstractions for the type nodes bound on g in χ_p. Those abstractions are added in front of a'.

2. Consider an instantiation edge e from a gen node g to a node d in χ_p. By construction, e is solved in χ_p, and the type under d is an instance of the type scheme represented by $\langle g1 \rangle$. A (graphic) witness of this fact is an instance derivation of $\chi_p^e \sqsubseteq \chi_p$. We translate this derivation into a computation, and insert it as a type application at the appropriate place in the elaboration of a.

While seemingly simple, this approach actually contains several subtle points:

- When translating a derivation $\chi_p^e \sqsubseteq \chi_p$ into a computation, some parts of this derivation are not related to instantiating the type scheme into the type of the constrained node. Those subparts must not be translated.
- Not all operations inside $\chi_p^e \sqsubseteq \chi_p$ can be reflected in xML$^\mathsf{F}$, as the type instance relations of xML$^\mathsf{F}$ and gML$^\mathsf{F}$ do not exactly coincide. Thus we must rule out some of those derivations (and in fact some entire presolutions).
- Unrelated quantifiers such as $\forall(\alpha \geqslant \sigma)\ \forall(\beta \geqslant \sigma)$ are not ordered in graphic types, while they are ordered in xML$^\mathsf{F}$. When translating graphic operations into xML$^\mathsf{F}$ computations we must be careful to correctly order the type abstractions.
- Some nodes of χ_p, which include the nodes corresponding to type schemes and some existential nodes, must not result in the addition of type abstractions in the elaborated term for a.
- The typing constraint we use for let constructs is slightly unusual w.r.t. scopes, making the translation unnecessarily difficult. We introduce an alternative possibility.

We detail all these points in the next five sections, and summarize the various hypotheses we make on presolutions and instance derivations in §15.2.7. Once this is done, §15.3 will present the algorithm for translating a gML$^\mathsf{F}$ presolution into an xML$^\mathsf{F}$ term.

15.2.2 Identifying which operations to translate

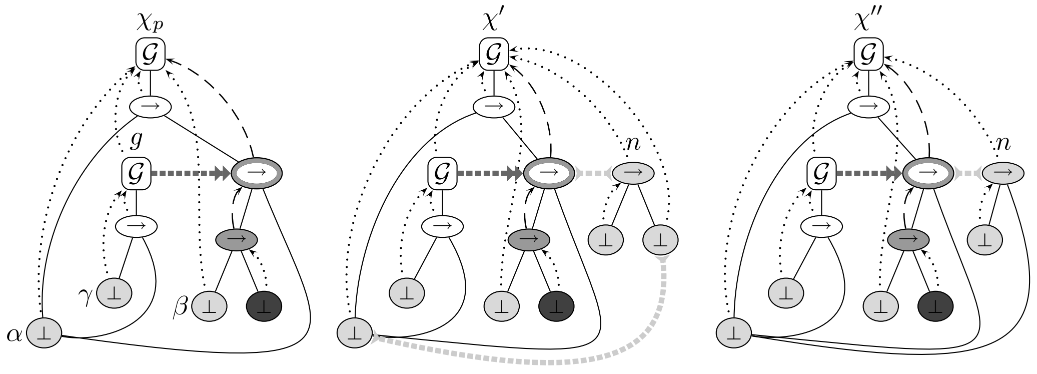


Figure 15.2.1 – Propagation witnesses

When translating an instance derivation $\chi_p^e \sqsubseteq \chi_p$ into a computation, not all the derivation is useful. In fact, some parts of the derivation have no equivalent in xML$^\mathsf{F}$ and must *not* be considered. We characterize them below, starting by an example.

▶ **Example** Consider Figure 15.2.1. It features a presolution χ_p of the constraint χ of Figure 15.1.1. The constraint χ' is χ_p^e, *i.e.* the result of propagating the instantiation edge e from g to $\langle 12 \rangle$. Notice that $\mathcal{F}^s(g)$ is not empty, as it contains $\langle g12 \rangle$. Consequently, in χ', there is an unification edge between this node and the corresponding one in the expansion, namely $\langle n2 \rangle$. The steps solving this unification edge are not useful to us, as they are not really related to the transformation of the type $\forall\,(\gamma)\ \gamma \to \alpha$ of g into the type under the constrained node $\langle 12 \rangle$.

As we mentioned in §10.1, the unification edges resulting from frontier nodes are there mostly for technical reasons: directly reusing nodes of the frontier in the expansion could result in ill-dominated constraints. (Interestingly, this is no longer the case on presolutions, as nodes have always been raised enough.) Still, we are *not* interested in the instantiation steps solving this kind of unification edges, as they have no meaning in $x\mathsf{ML}^\mathsf{F}$.

Interestingly, we have already proven in Lemma 11.5.3 that a derivation $\chi_p^e \sqsubseteq \chi_p$ can always be unambiguously decomposed into two derivation I_u and I, the first one solving the frontier unification edges. Hence, when translating $\chi_p^e \sqsubseteq \chi_p$ into a computation, we only consider I.

Convention In fact, in the remainder of this chapter we are never interested in I_u. Since we always reason about presolutions, we slightly change the meaning of «expansion», «propagation» and of the notation χ^e, and assume that, in an expansion, all the frontier unification edges are solved by unification.

▶ **Example** In Figure 15.2.1, χ_p^e no longer means the second constraint χ', but χ'', the third—which is the result of solving the lowermost unification edge of χ'.

With this convention, Lemma 11.5.3 implies that all instance derivations $\chi_p^e \sqsubseteq \chi_p$ can be assumed to be normalized, with the weakenings being delayed. In the following we only consider this kind of derivation, which we call *propagation witness.*

Definition 15.2.1 (*Propagation witness)* A *propagation witness* for an instantiation edge e of a presolution χ_p is a normalized derivation of $\chi_p^e \sqsubseteq \chi_p$ in which weakenings are delayed. ■

15.2.3 Removing operations on inert-locked nodes

Not all presolutions (and propagation witnesses) are suitable for translation. Indeed, there are graphic operations which cannot be reflected in $x\mathsf{ML}^\mathsf{F}$, namely those on an inert node which is not transitively flexibly bound to the root. We call such a node *inert-locked.*

Definition 15.2.2 (*Inert-locked nodes)* Given a constraint χ, a node n of χ is said to be *inert-locked* if $\overset{\diamond}{\chi}(n) = (\geqslant)$, $\overline{\diamond}_\chi(n)$ contains $(=)$ and n is inert in χ. ■

▶ **Example** Consider Figure 15.2.2. The node $\langle 11 \rangle$ is inert-locked. The graphic type τ translates, after inlining of rigid bounds, into the $x\mathsf{ML}^\mathsf{F}$ type

$$\sigma \quad \triangleq \quad (\forall\,(\alpha \geqslant \bot \to \bot)\ \alpha \to \alpha) \to (\forall\,(\alpha \geqslant \bot \to \bot)\ \alpha \to \alpha)$$

Conversely, the graphic type τ' which is obtained by weakening $\langle 11 \rangle$ in τ, translates into

$$\sigma' \quad \triangleq \quad ((\bot \to \bot) \to (\bot \to \bot)) \to ((\bot \to \bot) \to (\bot \to \bot))$$

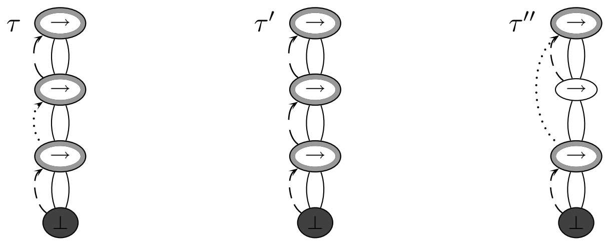


Figure 15.2.2 – Weakening inert-locked nodes

However σ' is not in instance relation with σ in xML^F, as we cannot transform bounds under the root arrow constructor.

Alternatively, we could have raised $\langle 11\rangle$ in τ. This results in τ'', whose translation in xML^F is

$$\sigma'' \quad \triangleq \quad \forall\,(\alpha \geqslant \bot \to \bot)\;(\alpha \to \alpha) \to (\alpha \to \alpha)$$

Again, σ'' is not an instance of σ in xML^F.

At first, this could seem problematic, as propagation witnesses involving an operation on an inert-locked node cannot be translated into xML^F. Our solution to this problem is however simple: we rule out any presolution containing such a node, by selectively weakening some nodes.

15.2.3.1 Inert-equivalent presolutions

We are going to be a bit more general than what is needed for this section: indeed, §15.2.5 will also rule out some presolutions as unsuitable for translation.

Definition 15.2.3 (*Inert-equivalent presolutions)* Two presolutions χ_p and χ_p' of a typing constraint are inert-equivalent if $\check{\chi_p} = \check{\chi_p}'$, $\hat{\chi_p} = \hat{\chi_p}'$ and for any gen node g, the nodes created by expanding $\langle g1\rangle$ at g in both constraints only differ by the binding flags of some inert nodes. ■

The idea behind this definition is the following: we identify presolutions in which subterms have the same types up to the weakening of inert nodes. Thus the differences between the presolutions are only superficial. Expanding $\langle g1\rangle$ at g is a simple technical solution to obtain an instance of $\langle g1\rangle$ in a “safe” way (*i.e.* without raising the nodes of the structural frontier of g).

Simply requiring $\overset{\diamond}{\chi_p}$ and $\overset{\diamond}{\chi_p}'$ to be equal except on inert nodes would not have been sufficient. Indeed, we need the possibility to weaken some non-inert nodes that do not really belong to the type part of the constraint. This includes for example existential nodes, or the binder of a non-degenerate type scheme. Some examples will be given in §15.2.5.

15.2.3.2 Removing inert-locked nodes

We are going to show that the inert-locked nodes of a presolution can be weakened. In such a weakened presolution, propagation witnesses never transform inert-locked nodes. This is the property we strive for, as such propagation witnesses can be translated into $x\mathsf{ML}^{\mathsf{F}}$ computations.

First, we show that the constraint obtained by weakening an inert node of a presolution can be transformed further, so as to re-obtain a presolution; moreover this can be done merely by weakening some other inert nodes.

Lemma 15.2.4 *Let χ_p be a presolution, n an inert flexibly bound node of χ_p. There exists an instance χ'_p of χ_p that is a presolution inert equivalent to χ_p, and in which n is rigidly bound.* □

We give a constructive proof of the existence of χ'_p.

Proof: Let N be a set of nodes, and $\mathcal{N}(N)$ be

$$\left\{ n' \mid \exists n \in N, \exists e = g \dashrightarrow\!\!\!\!\!\rightarrow d,\ \wedge \left\{ \begin{array}{l} n \in \mathcal{I}^s(g) \\ n \neq \langle g1 \rangle \\ n^c \text{ is merged with } n' \text{ when } \chi_p^e \sqsubseteq \chi_p \text{ is solved} \end{array} \right. \right\}$$

The operator $\mathcal{N}$ is used to find the nodes that must be weakened to obtain χ'_p. That is, if N is a set of nodes we want to be rigid, then $\mathcal{N}(N)$ finds the nodes with which the (copies of the) nodes of N are merged after a propagation. By monotony of instance on binding flags, this means that the nodes of $\mathcal{N}(N)$ must also be rigid. Notice the special case if n is of the form $\langle g1 \rangle$. In this case the flag of n is reset during expansion, and the weakening of n does not force the weakening of the nodes with which the copies of n are merged.

Thus, let $N_0 = N'_0 = \{n\}$ and $N'_{i+1} = N_i + \mathcal{N}(N_i)$ for all $i > 0$. Necessarily, there exists k such that $N_{k+1} = N_k$, as there are finitely many nodes in χ_p. Notice that, given N, $\mathcal{N}(N)$ is effectively computable: there are finitely many instantiation edges, and the propagation steps can be solved by unification. Thus, k is also computable, and so is N'_k.

Let us write $\mathsf{Weaken}(N)$ the weakening of all the nodes in N not already rigid. We call χ'_p the constraint $\mathsf{Weaken}(N'_k)(\chi_p)$. Let us show it is of the desired form; as a side result, we show that $\chi_p \sqsubseteq^W \chi'_p$.

▷ $\chi_p \sqsubseteq^W \chi'_p$: it suffices to prove that all the nodes of N'_k are inert in χ_p. Indeed, as the weakening of inert nodes preserves permissions (Lemma 5.4.1), we can weaken one (inert) node, and all the nodes remaining to weaken are still inert. Moreover, by definition of N_k, and since n is inert, it also suffices to show that if all the nodes of N are inert, the nodes of $\mathcal{N}(N)$ are inert.
For this last result, let n be an inert node of χ_p, and let n' be a of $\mathcal{N}(\{n\})$. Let e be an edge verifying the hypotheses of the definition of $\mathcal{N}$ for n and n'. By hypothesis, n is not of the form $\langle g1 \rangle$. Thus the subgraphs under n and n^c are identical in χ_p^e. Since n is inert, this ensures that n^c is inert. Since instance preserves inert nodes, n' is inert, which is the desired result.

▷ χ'_p is a presolution: it suffices to show that $\chi_p'^e \sqsubseteq \chi'_p$ holds (**1**) for any e. By hypothesis and the point above, we have $\chi_p^e \sqsubseteq \chi_p \sqsubseteq \chi'_p$ (**2**). Let $N_k'^c$ be the copies of the nodes of N'_k in the expansion of e in $\chi_p'^e$, except for the root of the expansion (whose flag is reset during expansion). It is immediate that $\chi_p'^e = (\mathsf{Weaken}(N'_k)\ ;\mathsf{Weaken}(N_k'^c))(\chi_p)$. We also have $\chi_p^e \sqsubseteq^W \chi_p'^e$ (**3**) by the equality above, and the fact that all the nodes of

$N'_k \cup {N'_k}^c$ are inert. Moreover, by definition of χ'_p, the nodes with which the nodes of ${N'_k}^c$ are merged when $\chi_p^e \sqsubseteq \chi_p$ is solved are rigid in χ'_p (**4**). Thus (1) holds by (2), (3), (4) and repeated application of Lemma 6.6.4.

▷ χ_p and χ'_p are inert equivalent: since $\chi_p \sqsubseteq^W \chi'_p$, all nodes bound differently in χ_p and χ'_p are inert. The only inert nodes can that become non-inert in an expansion are those of the form $\langle g1\rangle$, but their flag are reset by the expansion. Hence the expansions of a scheme $\langle g1\rangle$ in χ_p and χ'_p only differ by some inert nodes.

The algorithm used in this proof to find χ'_p weakens as few nodes as possible. However, let N be the set of inert nodes of χ_p. Reusing the notations above, we have $\mathcal{N}(N) \subseteq N$, as all nodes of $\mathcal{N}(N)$ are inert. An immediate adaptation of the proof shows that the constraint χ''_p obtained by weakening all the nodes of N (*i.e.* all the inert nodes of χ_p) is also a presolution instance of χ_p. This way, we can avoid the (costly) computation of N'_k, at the expense of potentially weakening more nodes.

As an immediate corollary, we can weaken all the inert-locked nodes in a presolution, and apply the result above to obtain another inert-equivalent presolution.

Corollary 15.2.5 *Given a presolution χ_p of a constraint χ, there exists a presolution χ'_p of χ inert-equivalent to χ_p and which does not contain inert-locked nodes.* □

Finally, given a presolution that does not contain inert-locked nodes, we can show that none of its propagation witnesses transform inert-locked nodes—a consequence of the fact that weakenings are delayed.

Lemma 15.2.6 *Let χ_p be a presolution that does not contain inert-locked nodes. For any instantiation edge e of χ_p, a propagation witness of $\chi_p^e \sqsubseteq \chi_p$ does not transform inert-locked nodes.* □

Proof: Let $o_1 ; \ldots ; o_k$ be the propagation witness $\chi_p^e \sqsubseteq \chi_p$. We show by induction on i that no constraint $(o_1 ; \ldots ; o_i)(\chi_p^e)$ contains inert-locked node—which implies the result.

▷ *Case $i = 0$*: expansion does not create inert nodes, but only copies existing ones. The conclusion is thus immediate, as χ_p does not contain inert-locked nodes.

▷ *Case $i = j + 1$*: let χ_j be $(o_1 ; \ldots ; o_j)(\chi_p^e)$, $\chi_i = o_i(\sigma_j)$. By induction hypothesis, no node of χ_j is inert-locked (**1**). We prove by case disjunction on o_i that it is still the case in χ_i. It suffices to consider the nodes which become inert in χ_i (**2**), or whose binding path become of the form $\geqslant(\geqslant|=)^* = (\geqslant|=)^*$ in χ_i (**3**), and show that they are not inert-locked in χ_i; for all the other nodes the result is by (1).

- *Case* $\mathsf{Graft}(\tau, n)$: For the nodes freshly grafted: by definition of propagation witnesses, τ is a constructor type, which does not contain rigidly bound nodes. Since the grafting occurs at a green node, the grafted nodes cannot be inert-locked.
  For the nodes already in χ_j: binding paths do not change; for (2), permissions change only for the nodes on which n is transitively bound (Lemma 5.4.1), which are green in σ_j, hence flexibly bound in χ_j and χ_i, hence not inert-locked in χ_i.
- *Case* $\mathsf{Merge}(n_1, n_2)$: binding paths and permissions do not change.
- *Case* $\mathsf{Raise}(n)$: for (2), only the permissions of $\hat{n}$ can change (Lemma 5.4.1). In order for $\hat{n}$ to become inert, it must be green in χ_j (Lemma 5.4.1), hence flexibly bound in χ_i and χ_j, hence not inert-locked in χ_i.
  For (3): no binding path of the required form appears, as no rigid edge is introduced.

- *Case* Weaken(n): for (2), only the nodes on which n are strictly transitively bound can become inert, but this implies that they were green (Lemma 5.4.1), hence flexibly bound in χ_j and χ_i, hence not inert-locked in χ_i.

  For (3), suppose there exists n' inert in χ_j with $n' \xrightarrow{\geqslant^+} \langle\epsilon\rangle$ and $n' \xrightarrow{+} n$. By monotony of instance, n' is in χ_p; it is also flexibly bound in χ_p, as the weakening are delayed in propagation witnesses and we have just transformed n which is strictly above n'. Lemma 5.4.1 shows that inert nodes are stable by instance. Hence n' is inert-locked in χ_p, which is a contradiction.

In all cases, no node is inert-locked in σ_i, and the conclusion is by induction hypothesis.

15.2.4 Ordering the nodes

While graphic types do not impose an order on the nodes bound on another node, the syntactic presentations of MLF, including *x*MLF, do: commuting two binders must be done explicitly, for example through the rule of Eq-Comm in the original presentation of MLF (Le Botlan and Rémy 2003). Thus, when we translate a presolution into an *x*MLF term, we must take this ordering into account.

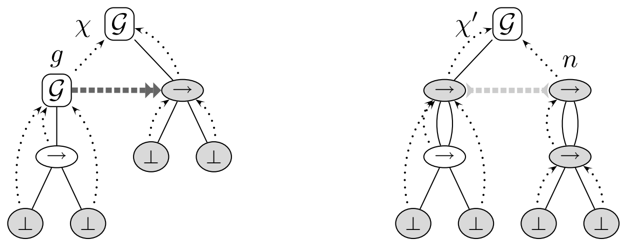


Figure 15.2.3 – Ordering nodes in syntactic types

▶ **Example** Consider the constraint χ of Figure 15.2.3; it is not the instance of a typing constraint, but this is unimportant here. This constraint is solved, as the instantiation edge e leaving g is trivially solved.

From a syntactic standpoint, suppose that the translation of the graphic type for g is

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

If the translation of the type under $\langle 1\rangle$ is also $\forall(\alpha \geqslant \bot)\,\forall(\beta \geqslant \bot)\ \alpha \to \beta$, the computation to insert for e is simply $\gamma \triangleleft$, where γ is the name of the bound for $\langle 1\rangle$. Things are however more complicated if one of the two types (but not both) is translated as

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

This time, applying $\gamma \triangleleft$ would be ill-typed.

To remedy this problem, one possibility is to coerce one type into the other. In general, in $x\mathsf{ML}^\mathsf{F}$, transforming $\forall\,(\alpha \geqslant \sigma)\ \forall\,(\beta \geqslant \sigma')\ \sigma''$ into $\forall\,(\beta \geqslant \sigma')\ \forall\,(\alpha \geqslant \sigma)\ \sigma''$ (when $\alpha \notin \mathsf{ftv}(\sigma')$ and $\beta \notin \mathsf{ftv}(\sigma)$) can be done by the computation

$$\parr; \forall\,(\geqslant \triangleright \sigma'); \forall\,(\beta \geqslant)\ (\parr; \forall\,(\geqslant \triangleright \sigma); \forall\,(\alpha \geqslant)\ (\forall\,(\geqslant \alpha \triangleleft); \&; \forall\,(\geqslant \beta \triangleleft); \&))$$

This computation first introduces two fresh bounds β and α (in this order), ranging over σ' and σ respectively. Then it abstracts the two preexisting bounds over the proper name, and substitutes them.

Still, this approach is inelegant. First, it introduces very complicated (and big) computations—something clearly undesirable if $x\mathsf{ML}^\mathsf{F}$ is used as an internal language. Also, since permuting two binders is not reflected in graphic instance derivations, there is no clear guide as to when we should insert a computation such as the one above. Thus, and unlike what we did in the previous chapters (for example in §8.2.1), we choose to *totally* order the nodes in the graphic types. We use this order in the translation to syntactic types, when we have to choose the first bound to translate.

Of course, this order cannot be arbitrary, as it must at least respect the syntactic scope of variables. We present a suitable order below.

Definition 15.2.7 (*Leftmost-lowermost order*) Given two paths π and π' we write $<_\mathsf{P}$ the lexicographic order on paths inductively defined by

$$\begin{array}{l}\forall \pi,\ \pi <_\mathsf{P} \epsilon \\ \forall i_1,\ i_2,\ \pi_1,\ \pi_2,\ i_1 \cdot \pi_1 <_\mathsf{P} i_2 \cdot \pi_2 \iff i_1 < i_2 \vee (i_1 = i_2 \wedge \pi_1 <_\mathsf{P} \pi_2)\end{array}$$

We extend this path to nodes by

$$n_1 <_\mathsf{P} n_2 \iff \min_{<_\mathsf{P}}\{\pi_1 \in n_1\} <_\mathsf{P} \min_{<_\mathsf{P}}\{\pi_2 \in n_2\} \qquad \blacksquare$$

Notice that this order is slightly unusual: $\pi <_\mathsf{P} \epsilon$ holds, while the converse usually does. Indeed, we want lower nodes to appear (hence to be bound) first—otherwise some variables would be out of scope.

Let us show that this order extends the two ones imposed by structure and binding edges.

Property 15.2.8 *Given a graphic type τ, if $n \multimap n'$ or $n \leftarrow\!\!\triangleleft\, n'$ holds, then $n' <_\mathsf{P} n$.* $\square$

Proof: ▷ *For $n \multimap n'$*: let Π'' be $\{\pi'' \mid \exists \pi \in n, \exists \pi' \in n', \pi' = \pi \cdot \pi''\}$. By congruence, $\pi' = \{\pi\pi'' \mid \pi \in n, \pi'' \in \Pi''\}$ **(1)**. Let π'' be $\min_{<_\mathsf{P}} \Pi''$. By (1), we have $\min_{<_\mathsf{P}}\{\pi' \in n'\} = \min_{<_\mathsf{P}}\{\pi \in n\} \cdot \pi''$, which implies the result by definition of $<_\mathsf{P}$.

▷ *For $n \leftarrow\!\!\triangleleft\, n'$*: on graphic types, this relation implies $n \overset{+}{\multimap} n'$ and the result is immediate by transitivity of $<_\mathsf{P}$ and the previous result.

The result above is restricted to graphic types: on a gen node of a graphic constraint, we would also need to impose an order on existential nodes. This is however not needed for the use of $<_\mathsf{P}$ we have in mind, as we will always use it on type nodes of graphic constraints.

Since the algorithm $\mathcal{S}$ of §8.2.1 orders bounds according to $\overset{+}{\multimap}$ and lowermost nodes first, $<_\mathsf{P}$ is a suitable order for the translation.

Corollary 15.2.9 *Ordering the bounds according to $<_{\mathsf{P}}$ in the algorithm $\mathcal{S}$ is correct.* □

Notice also that, since $<_{\mathsf{P}}$ is a total order, the translation is now deterministic; this was the desired goal.

15.2.4.1 Preserving the order on nodes through instance operations

Importantly, it is not sufficient to translate the bounds in the correct order: we must also preserve this invariant when we build the computations. Consider as an example the constraint χ' of Figure 15.2.3. As for χ, it is not a typing constraint, but this is also unimportant here. Since we have $\langle 111\rangle <_{\mathsf{P}} \langle 112\rangle <_{\mathsf{P}} \langle 11\rangle$ and $\langle n11\rangle <_{\mathsf{P}} \langle n12\rangle$, the translations of the types under the nodes $\langle 1\rangle$ and n are respectively

$$\begin{aligned} \sigma_{\langle 1\rangle} &\triangleq \forall\,(\alpha \geqslant \bot)\ \forall\,(\beta \geqslant \bot)\ \forall\,(\gamma \geqslant \alpha \to \beta)\ \gamma \to \gamma \\ \sigma_n &\triangleq \forall\,(\gamma \geqslant \forall\,(\alpha \geqslant \bot)\ \forall\,(\beta \geqslant \bot)\ \alpha \to \beta)\ \gamma \to \gamma \end{aligned}$$

A normalized instance derivation merging n and $\langle 1\rangle$ is for example

$$\mathsf{Raise}(\langle n12\rangle)\ ;\ \mathsf{Raise}(\langle n11\rangle)\ ;\ \mathsf{Merge}(n, \langle 1\rangle)$$

Applying the first raising to σ_n is non-ambiguous, and results in

$$\sigma'_n \quad \triangleq \quad \forall\,(\beta \geqslant \bot)\ \forall\,(\gamma \geqslant \forall\,(\alpha \geqslant \bot)\ \alpha \to \beta)\ \gamma \to \gamma$$

However, for the second raising we can choose to bind α before or after β. In order to respect $<_{\mathsf{P}}$, only the former possibility is applicable. Thus a computation that reflects this raising in σ'_n must take this invariant into account.

15.2.5 Adding xML$^{\mathsf{F}}$ type abstractions

As we mentioned in the introduction of this section, we translate the nodes bound on gen nodes into xML$^{\mathsf{F}}$ type abstraction. However, we do not do so for all nodes.

- Rigid quantification is always inlined; thus we do not introduce type abstractions for rigidly bound nodes.
- Given a term a whose corresponding gen node is g, we are interested in the type of a in the presolution, *i.e.* the type of $\langle g1\rangle$. This type corresponds to the nodes of the structural interior of g, except for $\langle g1\rangle$ itself. With the revised definition of expansion (given in §15.2.2), this type is also exactly the type obtained by expanding $\langle g1\rangle$.

Thus, given a gen node g, we should introduce type abstractions for the nodes of the structural interior of g that are flexibly bound, except for $\langle g1\rangle$ itself. In particular we should not translate the existential nodes bound on g.

We detail these points through examples below. There are additional subtleties with the typing of applications and abstractions, which are dealt with in the last part of this section. In the examples, we use a term f of type $\forall\,(\alpha \geqslant \tau)\ \forall\,(\beta \geqslant \tau)\ \alpha \to \beta$, and another term v of type τ; we suppose that τ is polymorphic. We consider various presolutions of the typing constraint for $f\ v$, which is given at the left of Figure 15.2.4 (we have removed the subconstraint for v for brevity). Finally, we assume that σ is the xML$^{\mathsf{F}}$ translation of τ.

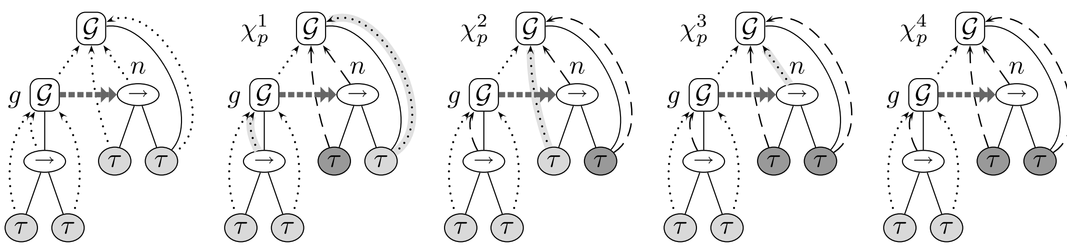


Figure 15.2.4 – Typing constraint for $f\ v$

15.2.5.1 Inlining scheme nodes

Consider a non-degenerate type scheme $\langle g1\rangle$. Whether it is flexibly or rigidly bound should be indifferent for the translation, as the binding flag is reset during expansion. As it happens however, allowing $\langle g1\rangle$ to be flexible in presolutions needlessly complicates the translation, as we show below.

▶ **Example** Consider the presolution χ_p^1 of Figure 15.2.4. If we introduce type abstractions for the nodes $\langle g1\rangle$ and $\langle 1\rangle$, f and $f\ v$ receive respectively the types

$$\forall\,(\alpha \geqslant \sigma)\ \forall\,(\beta \geqslant \sigma)\ \forall\,(\gamma \geqslant \alpha \to \beta)\ \gamma \qquad \text{and} \qquad \forall\,(\delta \geqslant \sigma)\ \delta$$

Of course, the quantifications for γ and δ are useless. If we entirely elaborate the term, we obtain

$$\Lambda(\delta \geqslant \sigma)\ f[\&;\forall\,(\geqslant \delta\,\triangleleft);\&;\&]\ v[\delta\,\triangleleft]$$

This is needlessly complicated: the quantification for γ is eliminated by the third computation &, and the quantification $\Lambda(\delta \geqslant \sigma)$ requires to abstract β and the type of v under the name δ. Worse, when $f\ v$ will be «used», we will likely start by eliminating the dummy quantification on δ.

A much better solution is to rigidify the nodes $\langle g1\rangle$ and $\langle 1\rangle$. The resulting constraint χ_p^4 is still a presolution, as the flag of those nodes was reset during expansion anyway. Afterwards, we obtain the simpler term

$$f[\&;\&]\ v$$

This approach is fully general. Hence, we suppose that non-degenerate schemes are rigid in presolutions.

15.2.5.2 Naming the domain of a decorrelated application

There is a subtlety in the typing of an application, related to the domain of the argument of the application. If we introduce type abstractions only for the nodes in the structural interior of gen nodes (which are the only nodes appearing in the expansion of the corresponding type schemes), some problems occur for an application $a_1\ a_2$, when the domain and the codomain of the type of a_1 are not correlated. Indeed, we might not have a name to refer to the (existentially introduced) domain of the arrow.

▶ **Example** Consider the presolution χ_p^2 of Figure 15.2.4. To build the computation for the instantiation edge, we must instantiate the type of f into the type of the node n. However, by lack of a name for $\langle n1\rangle$, this last type is actually undefined!

As in the previous section, a possible solution would be to introduce a dummy type quantification in front of $f\ v$. This would result in the (well-typed) xMLF term

$$a_2 \quad \triangleq \quad \Lambda(\alpha \geqslant \sigma)\ f[\forall\,(\geqslant \alpha \triangleleft); \&; \&]\ v[\alpha \triangleleft]$$

However, this term has type $\forall\,(\alpha \geqslant \sigma)\ \sigma$, not σ. As a workaround, we could instead consider $a_2[\&]$ instead of $\&$, which indeed has type σ (and which reduces to $f[\&; \&]\ v$). However, this is not really satisfactory: the term for a_2 is needlessly complicated, and we need to insert a computation $\&$ everywhere it is used.

Thankfully, there exists a much cleaner solution, which consists in rigidifying $\langle n2\rangle$. The resulting constraint is again χ_p^4. This time, we are assured that the resulting constraint is still a presolution by Lemma 11.6.1, since $\langle n1\rangle$ is not in the structural interior of the root gen node.

15.2.5.3 Existential application node

There is a potential problem with the arrow existentially introduced in the typing constraint for an application, orthogonal to the issues above. In xMLF, a term a_1 in an application $a_1\ a_2$ must have an arrow type $\sigma_1 \to \sigma_2$. Hence, the elaborated λ-term must *not* abstract the type of the arrow node as a variable of the typing environment.

▶ **A first example** Consider the constraint χ_p^3 of Figure 15.2.4. If we introduce a type abstraction $\Lambda(\alpha \geqslant \tau \to \tau)$ for the node n, f would have type α in this presolution. There is no way to cast this type into an arrow type in xMLF, and the elaboration would necessarily be ill-typed.

In this example, we may remedy this problem by rigidifying n; we then obtain χ_p^4 yet again. Notice that n is existential, which ensures that we can rigidify it. However, this is not always the case, as we illustrate below.

▶ **A more involved example** Figure 15.2.5 shows a slightly simplified typing constraint for the term $\lambda(x)\ x\ 1$, in which we have solved and removed the subconstraint for the integer 1. The constraint χ_p is a presolution of χ. Observe that the existential node n introduced for the argument of the application in χ becomes the node $\langle 11\rangle$ in χ_p, and that this node is in the structural interior of the root gen node. If we introduce a type quantification for this node at the toplevel of the elaborated λ-term, the resulting term will be of the form

$$\Lambda(\alpha \geqslant \bot)\ \Lambda(\beta \geqslant \mathsf{int} \to \alpha)\ \lambda(x : \beta)\ x[\varphi]\ 1$$

However this term is ill-typed for any φ, as we cannot give to x (which has type β) an arrow type.

Thankfully, the node n is inert in χ_p, and also in χ. We can weaken it, resulting in the presolution χ'_p in which the bound of n will be inlined. Indeed, χ'_p, translates into the well-typed term

$$\Lambda(\alpha \geqslant \bot)\ \lambda(x : \mathsf{int} \to \alpha)\ x\ 1$$

which has type $\forall\,(\alpha \geqslant \bot)\ (\mathsf{int} \to \alpha) \to \alpha$

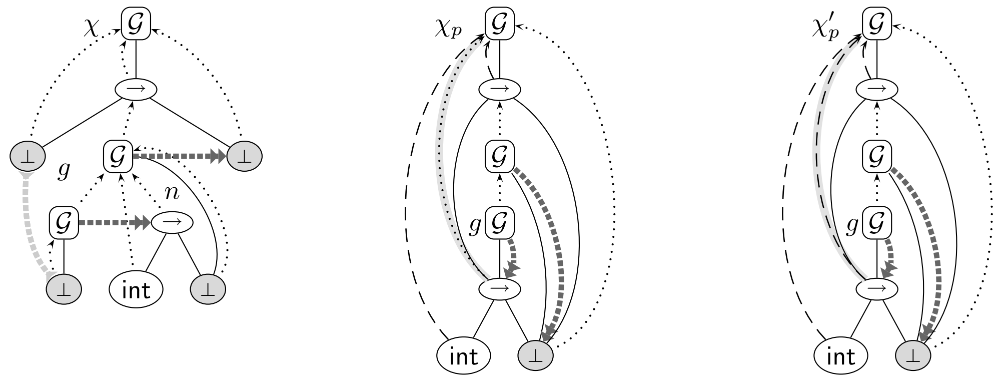


Figure 15.2.5 – Typing constraint for $\lambda(x)\ x\ 1$

We us call *application arrow node* the arrow node introduced in the typing constraint for an application. The reasoning done for this example is fully general: an application arrow node is inert in a typing constraint, and inert nodes are stable by instance. Thus by Lemma 15.2.4 we can suppose that those nodes are rigid in all the presolutions we consider.

15.2.5.4 Arrow node in abstractions

We call *abstraction arrow node* the node $\langle g1 \rangle$ of the typing constraint for an abstraction. Those nodes suffer from the same problem as application arrow nodes. (One can for example consider the constraint χ''_p of Figure 15.1.1, and imagine that the node $\langle 11 \rangle$ is flexibly bound.) We want an abstraction to have an arrow type, not a variable of the typing environment. Thus, as for application arrow nodes, we suppose that abstraction arrow nodes are inert in presolutions.

15.2.6 Scopes in a let construct

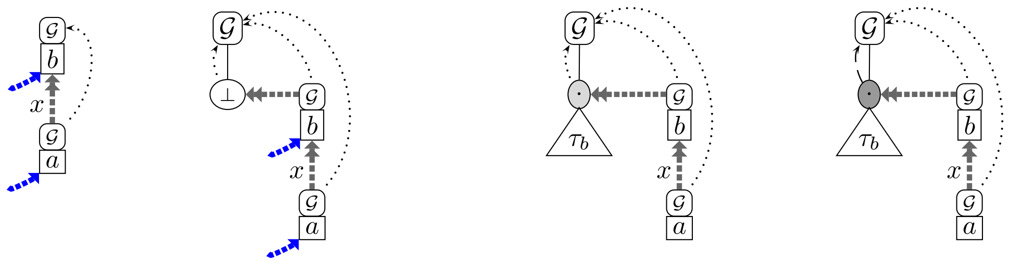


Figure 15.2.6 – Alternative typing for a let construct

Consider a term let $x = a$ in b. The basic typing constraint for this construction, which we have reproduced in the left of Figure 15.2.6, does not introduce a new gen node for the entire let expression. Instead, it piggybacks the one introduced for the subterm b. Another possibility would have been the second constraint of the figure, in which we have added a trivial type scheme for the let expression.

From a type inference point of view, the leftmost constraint is the good one, as it is simpler. Indeed, the root gen node in the second constraint can only pick instances of the one of b. In the principal presolution of this constraint, both would even have exactly the same type. Hence the root gen node is redundant.

From the point of view of the the translation into xML$^{\mathsf{F}}$, things are not so clear. In the second constraint, we have a supplementary instantiation edge; hence a priori another computation to insert. On the other hand, things are simpler w.r.t. scopes. Indeed, in the leftmost constraint, the scope of the let expression, hence of b itself, is visible by a. This is quite unusual, and severely complicates the translation. Moreover, as we justify below, the extra edge translates into the identity computation if the presolution being translated is the principal presolution of the constraint. Hence we choose to translate presolutions built using the second form of constraint.

15.2.6.1 Computations for trivial type schemes

Let us consider the trivial type schemes (corresponding to the type $\forall\alpha.\ \alpha$) introduced in the typing for let-bound variables, or in the revised typing for a let expression. The rightmost part of Figure 15.2.6 shows the principal typing for an expression let $x = a$ in b. The root (trivial) type scheme has exactly the type of b in the presolution, as the bottom node $\langle 1 \rangle$ is not otherwise constrained. Following the convention we have established in §15.2.5.1, we in fact translate the last constraint of the figure. This means that all the type abstractions introduced on the root node would also have been present with the "regular" typing for a let construct (on the left of the figure). Moreover, the type computations corresponding to the instantiation edge for the trivial scheme will simply be the identity computation, since there is no difference between the type of b and the type of the root gen node.

The reasoning is exactly the same for the type scheme for a let bound variable. More generally, we can safely insert such trivial type schemes inside constraints: they will not significantly complicate the term resulting from the elaboration of the principal presolution of a constraint—which is the one we are interested when xML$^{\mathsf{F}}$ is used as an internal language.

15.2.7 Translatable presolutions

Let us summarize:

Definition 15.2.10 (*Translatable presolutions)* Consider a typing constraint χ, in which let constructs are typed using the rightmost constraint of Figure 15.2.6. A presolution χ_p of χ is said to be *translatable* if the following conditions hold:

1. χ_p does not contain inert-locked nodes;
2. non-degenerate type schemes nodes are rigidly bound;
3. application and abstraction arrow nodes are rigidly bound;

4. a node bound on a gen node but which is not in its structural interior is rigidly bound. ■

Theorem 15.2.11 *Consider a presolution χ_p of a typing constraint χ created with the alternative typing for* let. *There exists a translatable presolution of χ which is inert-equivalent to χ_p.* □

Proof: We consider χ, and we weaken all its inert-locked and application or abstraction arrow nodes. Inert locked nodes are inert by definition, and application or abstraction arrow nodes are inert by monotony of instance (as they are inert in χ). Then we use Lemma 15.2.4 to obtain a presolution χ'_p instance of χ_p and inert-equivalent to χ_p. Finally, in χ'_p, we weaken all the non-degenerate type scheme nodes, and all the nodes bound on gen nodes that are not on a structural interior and are flexibly bound. Those nodes are bound on a gen node, hence are not red. The resulting constraint χ''_p is thus an instance of χ'_p, and a presolution by flag reset and Lemma 11.6.1. Moreover, by definition of the nodes weakened, the expansions are unchanged between χ'_p and χ''_p, and those two constraints are inert-equivalent.

Without loss of generality, we also suppose that an operation $\mathsf{Merge}(n, n')$ appearing in a propagation witness verifies $n' <_{\mathsf{P}} n$. This makes the translation of a propagation witness into a computations a bit simpler.

15.2.8 Using xML$^\mathsf{F}$ as an internal language

The proof of Theorem 15.2.11 has been designed to weaken as few inert nodes as possible. However, if xML$^\mathsf{F}$ is used as an internal language, the translation must be modular. To do so, we will be slightly less fined-grained in regard to inert nodes. Indeed, we weaken *all* inert nodes in a presolution; Corollary 15.2.5 ensures that the resulting constraint is still a presolution, which is moreover inert-equivalent to the initial one.

Let us explain why this is sufficient to make the translation modular. Given a type τ, we write $\mathsf{W}(\tau)$ the type obtained by weakening all the inert nodes of τ, and use the same notation for a constraint. Consider a term a that has (graphic) type τ. In xML$^\mathsf{F}$, a will have type $\mathsf{W}(\tau)$. Suppose now that a is used inside a term a'. In the typing constraint χ for a', a has type τ. In a presolution χ_p of χ, it has type τ', with $\tau \sqsubseteq \tau'$.

Let now χ' be the typing constraint for a', except that in this constraint a has type $\mathsf{W}(\tau)$ instead of τ. In order to be modular, it must be possible to deduce from χ_p a presolution of χ'. By construction of W we have $\chi \sqsubseteq^W \chi'$ and $\chi_p \sqsubseteq^W \mathsf{W}(\chi_p)$. Since χ_p is a presolution of χ we also have $\chi \sqsubseteq \chi_p$. Finally, since inert nodes are preserved by instance, all the nodes weakened between χ and χ' are weakened between χ and $\mathsf{W}(\chi_p)$. Thus Lemma 6.6.4 ensures that $\chi' \sqsubseteq \mathsf{W}(\chi_p)$, and $\mathsf{W}(\chi_p)$ is a presolution of χ'.

Hence the translation is modular: type inference and W essentially commute. Type inference is done as usual, using gML$^\mathsf{F}$ types. Internally, those types are normalized using W into xML$^\mathsf{F}$ ones. The gML$^\mathsf{F}$ presolutions are also normalized using W, and the resulting constraints are presolutions in which the subterms have their xML$^\mathsf{F}$ type.

15.3 Translating presolutions into *x*ML^F

We now describe how to translate a presolution of *g*ML^F into an *x*ML^F term in a systematic way. In the remainder of the section, we consider a *g*ML^F term a, χ_t its corresponding typing constraint, and χ_p a translatable *g*ML^F presolution of χ_t. Without loss of generality, we suppose that all term variables of a are distinct, which avoids α-conversion problems. Given a subterm a' of a, we write $g_{a'}$ the gen node that corresponds to a' in χ_t and χ_p. In order to simplify some notations, we often identify a' and $g_{a'}$

▶ **(Running) example** In this section, we will use as our main example the typing of $\lambda(x)\ \lambda(y)\ x$. Unlike in Figure 15.2.1, we do not use VAR-ABS. The resulting typing constraint χ_t and the translatable presolution χ_p on which we will work are given in Figure 15.3.1.

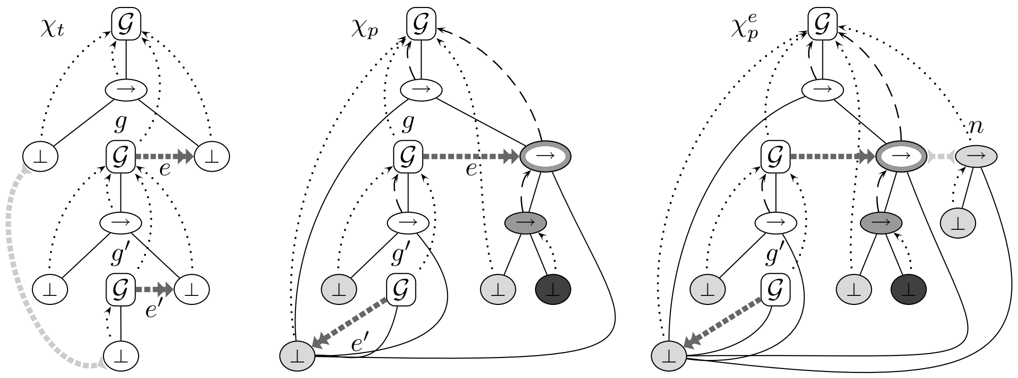


Figure 15.3.1 – An example translatable presolution

15.3.1 Obtaining syntactic types

The first step in translating χ_p into an *x*ML^F λ-term consists in naming all the type nodes which will be translated into type quantifications, so that we can refer to them. By the various restrictions on translatable presolutions, this is actually the set of type nodes flexibly bound on gen nodes. We call them *named nodes*. In order not to burden ourselves with α-conversion issues, we suppose that each named node n is associated to a variable α_n. (To simplify the examples, if π is such that $\langle\pi\rangle = n$, we often write α_π instead of $\alpha_{\langle\pi\rangle}$.)

▶ **Example** In our example, the named nodes of χ_p are the nodes $\langle 11\rangle$, $\langle 1211\rangle$ and $\langle g11\rangle$.

Figure 15.3.2 presents an algorithm to translate a subpart of χ_p into a syntactic type—we have simply specialized the algorithm of Figure 8.3.3 to the inlining of rigid nodes. We have also defined a variant $\mathcal{S}'_{\chi_p}$, which differs from $\mathcal{S}_{\chi_p}$ only on named nodes. In general, we will use $\mathcal{S}'_{\chi_p}$; however $\mathcal{S}_{\chi_p}$ is needed to compute the bounds of the named nodes themselves.

$$
\begin{array}{rcl}
\mathcal{S}_{\chi_p}(n) & = & \mathsf{B}(n_1)\cdots\mathsf{B}(n_k)\ \chi_p(n)\Big(\mathsf{V}(n\cdot 1),\ldots,\mathsf{V}(n\cdot\mathsf{arity}(\chi_p(n)))\Big) \\
& & \text{where } \{n_1,\ldots,n_k\} = (\xrightarrow{\geqslant} n) \text{ and } n_1 <_{\mathsf{P}} \ldots <_{\mathsf{P}} n_k \\
& & \text{and } \mathsf{B}(n_i) \ \triangleq\ \forall\,(\alpha_{\langle n_i\rangle} \geqslant \mathcal{S}_{\chi_p}(n_i)) \\
& & \text{and } \mathsf{V}(n\cdot i) \ \triangleq\ \begin{cases} \mathcal{S}_{\chi_p}(\langle n\cdot i\rangle) & \text{if } \langle n\cdot i\rangle \text{ is rigidly bound} \\ \alpha_{\langle n\cdot i\rangle} & \text{otherwise} \end{cases} \\
\mathcal{S}'_{\chi_p}(n) & = & \begin{cases} \alpha_n & \text{if } n \text{ is named} \\ \mathcal{S}_{\chi_p}(n) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 15.3.2 – Translation of a subpart of a constraint

Let us mention that the two algorithms are deterministic when called on any type node of χ_p. Indeed, they will only reach nodes accessible by $\overset{+}{\multimap}$, which are thus totally ordered by $<_{\mathsf{P}}$. We also often translate the type created by an expansion of a scheme of χ, or even an instance of this expansion. This translation is also well-defined: the flexible nodes translated are either in the interior of the root of the expansion (and then structurally reachable from this root), or merged with some existing structure of χ_p, which is necessarily a named node.

15.3.2 Types and environments of subterms

15.3.2.1 Gen nodes abstractions

Once we know how to translate graphic types into syntactic ones, we can build the type abstractions that will be added in front of terms in the $x\mathsf{ML}^{\mathsf{F}}$ elaborations of χ_p.

Definition 15.3.1 (*Abstractions for a subterm*) Let a' be a subterm of a, and g the gen node $g_{a'}$. Let also $n_0, \ldots, n_k$ be the set of named nodes bound on $g_{a'}$, ordered according to $<_{\mathsf{P}}$. We write $\Gamma^{a'}$ (or Γ^g) the type environment $\alpha_{n_0} \geqslant \mathcal{S}_{\chi_p}(n_0), \ldots, \alpha_{n_k} \geqslant \mathcal{S}_{\chi_p}(n_k)$, called the *abstractions for* a'. ■

By construction of translatable presolutions, the n_i are in the structural interior of g; hence they are indeed ordered by $<_{\mathsf{P}}$ and the definition of $\Gamma^{a'}$ is unambiguous. The abstractions for a' are exactly the type abstractions that will be introduced in front of a' (and in this order).

Convention Given an environment Γ equal to $\alpha_1 \geqslant \sigma_1, \ldots, \alpha_k \geqslant \sigma_k$, we write $\Lambda(\Gamma)$ for $\Lambda(\alpha_1 \geqslant \sigma_1)\ \ldots\ \Lambda(\alpha_k \geqslant \sigma_k)$ and $\forall\,(\Gamma)$ for $\forall\,(\alpha_1 \geqslant \sigma_1)\ldots\forall\,(\alpha_k \geqslant \sigma_k)$. If Γ contains some term bindings of the form $x : \sigma$, they are ignored.

15.3.2.2 Type of a subterm

Our next step is to define the type a subterm a' of a will have in the elaboration of χ_p.

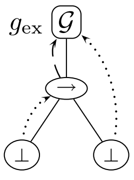
Definition 15.3.2 (*Type of a subterm*) The type $\mathsf{Typ}(a')$ (or $\mathsf{Typ}(g_{a'})$) of a subterm a' of a is the type $\forall\,(\Gamma^{a'})\ \mathcal{S}'_{\chi_p}(\langle g_{a'}\cdot 1\rangle)$ ■

Notice that we use $\mathcal{S}'$ and not $\mathcal{S}$: if $\langle g_{a'}1\rangle$ is named, we must use that name and not inline its bound.

▶ **Example** In χ_p, we have

$$\mathsf{Typ}(g) = \forall\,(\alpha_{g11} \geqslant \bot)\ \alpha_{g11} \to \alpha_{11} \qquad\qquad \mathsf{Typ}(g') = \alpha_{11}$$

$$\mathsf{Typ}(\langle\epsilon\rangle) = \forall\,(\alpha_{11} \geqslant \bot)\ \forall\,(\alpha_{1211} \geqslant \bot)\ \alpha_{11} \to ((\forall\,(\alpha_{1212} \geqslant \bot)\ \alpha_{1211} \to \alpha_{1212}) \to \alpha_{11})$$



Unfortunately, even though we will carefully maintain the bounds of syntactic types correctly ordered w.r.t. $<_{\mathsf{P}}$, there is still one case where the $g\mathsf{ML^F}$ representation of types will clash with the $x\mathsf{ML^F}$ one. This happens when some nodes are bound on a node g, and some others on $\langle g1\rangle$. For example, if we consider the gen node on the right, we have $\mathsf{Typ}(\langle\epsilon\rangle) = \forall\,(\beta \geqslant \bot)\ \forall\,(\alpha \geqslant \bot)\ \alpha \to \beta$, while the translation of an expansion of g is $\forall\,(\alpha \geqslant \bot)\ \forall\,(\beta \geqslant \bot)\ \alpha \to \beta$: the ordering between the type quantifications differ. This is of course problematic, as propagation witnesses "reason" on the second type, and $x\mathsf{ML^F}$ on the first.

There is unfortunately no way to entirely avoid this issue. It is not always possible to raise the nodes bound on a scheme node s so that they become bound on the corresponding g instead (in the example above, $\langle 11\rangle$ is locked). Thus we define the type of a subterm in an expansion, and create computations that convert the type of a subterm to this type.

Definition 15.3.3 (*Type of a subterm in an expansion)* Let a' be a subterm of a, g the gen node $g_{a'}$ and s be $\langle g1\rangle$. Let χ be the constraint resulting from expanding s in the context of g at g itself; let s^c be the root of the expansion in this constraint. The type of a' in an expansion, written $\mathsf{Typ}^{\exp}(a')$ (or $\mathsf{Typ}^{\exp}(g_{a'})$), is the type $\mathcal{S}'_\chi(s^c)$. ■

Notice that if s is degenerate, the expansion is empty and the root of the expansion is s itself. In this case, $\mathcal{S}'_\chi(s^c)$ returns exactly $\mathcal{S}'_\chi(s)$, which is coherent with the fact that a degenerate scheme has not other instance than itself.

▶ **Example** In our example presolution χ_p, Typ and $\mathsf{Typ}^{\exp}$ coincide for all three gen nodes. For the gen node g_{ex} above, we have $\mathsf{Typ}(g_{\text{ex}}) = \forall\,(\alpha_{12} \geqslant \bot)\ \forall\,(\alpha_{11} \geqslant \bot)\ \alpha_{11} \to \alpha_{12}$ and $\mathsf{Typ}^{\exp}(g_{\text{ex}}) = \forall\,(\alpha_{11} \geqslant \bot)\ \forall\,(\alpha_{12} \geqslant \bot)\ \alpha_{11} \to \alpha_{12}$.

The only possible difference between Typ and $\mathsf{Typ}^{\exp}$ is in the ordering between quantifiers, and they are thus closely related. We can define a computation coercing one into the other.

Definition 15.3.4 (*Quantifiers reordering)* Let a' be a subterm of a, g the gen node $g_{a'}$. Let S_1 (*resp.* S_2) be the named nodes bound on g (*resp.* $\langle g1\rangle$), and S_0 be $S_1 \cup S_2$. Let $n^l_1, \dots, n^l_{k_l}$ be the nodes of S_l sorted according to $<_{\mathsf{P}}$. The reordering $\varphi_{\mathsf{R}}(a')$ (or $\varphi_{\mathsf{R}}(g_{a'})$) for a' is the computation

$$\otimes; \forall\,(\geqslant \triangleright \mathcal{S}(n^0_1)); \forall\,(\alpha_{n^0_1} \geqslant)\ \Big(\ldots \otimes; \forall\,(\geqslant \triangleright \mathcal{S}(n^0_{k_0})); \forall\,(\alpha_{n^0_{k_0}} \geqslant)\ \Big(\overline{\forall\,(\geqslant \alpha_{n^1_i} \triangleleft); \&}; \overline{\forall\,(\geqslant \alpha_{n^2_j} \triangleleft); \&}\Big)\Big)$$

if $\mathsf{Typ}(a')$ and $\mathsf{Typ}^{\exp}(a')$ are distinct, or the identity computation otherwise. ■

In the non-trivial case, the reordering for a' introduces fresh quantifications for all the named nodes that are quantified in $\mathsf{Typ}^{\exp}(a')$, and abstracts all the quantification present in $\mathsf{Typ}(a')$ under the proper names. By construction of $<_{\mathsf{P}}$, those quantification start by the nodes bound on g, and then those bound on $\langle g1\rangle$.

▶ **Example** In our main example, the different reordering are always the identity computation. For the gen node g_{ex} of the example above, we have

$$\varphi_{\mathsf{R}}(g_{\mathrm{ex}}) = \otimes; \forall\,(\geqslant \triangleright \bot); \forall\,(\alpha_{11} \geqslant)\ (\otimes; \forall\,(\geqslant \triangleright \bot); \forall\,(\alpha_{12} \geqslant)\ (\forall\,(\geqslant \alpha_{12} \triangleleft); \&; \forall\,(\geqslant \alpha_{11} \triangleleft); \&))$$

Lemma 15.3.5 *Let a' be a subterm of a. Let Γ be an environment that binds all the free type variables of $\mathsf{Typ}(a')$. Then $\Gamma \vdash \varphi_{\mathsf{R}}(a') : \mathsf{Typ}(a') \leq \mathsf{Typ}^{exp}(a')$.* □

Proof: Let g be $g_{a'}$, s be $\langle g1 \rangle$. Let χ be the expansion of s in the context of g, at g in χ_p. By definition we have $\mathsf{Typ}^{\mathrm{exp}}(a') = \mathcal{S}'_{\chi}(s^c)$.

If s is degenerate, there is no named node bound on g at all and $\Gamma^{a'}$ is empty. The expansion of s is degenerate, and s^c is s. Hence the two translations are equal.

If s is not degenerate, by definition of translatable presolutions, it is rigidly bound. Then the quantifications in front of $\mathsf{Typ}(a')$ are the nodes flexibly bound on g (which thus do not include s), plus the nodes flexibly bound on s. In χ, the set of nodes bound on s^c is the same by binding reset. Hence we introduce the same quantifications on both sides.

15.3.3 Typing environments

We can now deduce the environment under which a term of a will be typed in the elaboration of χ_p.

Definition 15.3.6 (*Typing environment*) Let a' be a subterm of a. The typing environment $\Gamma_\star^{a'}$ for a' is inductively defined as $\Gamma_\star^{a} = \varnothing$ if a' is a, or by case analysis on the subterm a'' of a' that immediately encloses a' otherwise.

- *If $a'' = \lambda(x)\ a'$:* $\Gamma_\star^{a'} = \Gamma_\star^{a''}, \Gamma^{a''}, x : \mathcal{S}'_{\chi_p}(\langle g''_a 11 \rangle)$
- *If $a'' = \mathsf{let}\ x = b\ \mathsf{in}\ a'$, for some term b:* $\Gamma_\star^{a'} = \Gamma_\star^{a''}, \Gamma^{a''}, x : \mathsf{Typ}(b)$
- *In all the other cases:* $\Gamma_\star^{a'} = \Gamma_\star^{a''}, \Gamma^{a''}$ ■

In essence, we concatenate the environments for the gen nodes strictly above the one for a'. The only two exceptions are for the expressions that bind a term variable. In this case we also enrich the environment with this binder. In the case of a λ-abstraction, we extract the type of the argument of the abstraction. For a let, we simply read the scheme of the let-bound variable.

Property 15.3.7 *Typing environments are well-formed xMLF environments.* □

Proof: The different type variables have distinct names given our convention, while the term variables are supposed to be distinct. Hence it suffices to show that there is no free variable in the bounds. This result is immediate by well-domination and the fact that bounds in $\Gamma^{a''}$ are introduced according to $<_{\mathsf{P}}$ (which ensures well-scopedness).

As a last definition, we introduce a notion of environment for an instantiation edge. Consider for example a subterm let $x = b$ in b' of a, and an instantiation edge e resulting from an occurrence of x in b'. The environment for e is the typing environment for this occurrence of x.

Definition 15.3.8 (*Environment for an edge)* Let $g \dashrightarrow d$ be an instantiation edge e of χ_p. Let d' be the unique node of χ constrained by e in χ_t, and let g' be $\hat{\chi}_t(d')$. The environment for e, written $\Gamma^e_\sharp$, is the environment $\Gamma^{g'}_\star, \Gamma^{g'}$. ■

(There is a unique edge corresponding to e in χ_t: since gen nodes cannot be merged, instantiation edges cannot be merged either. This ensures the unicity of d'.)

▶ **Example** In our example, the typing environments for $\lambda(x)\ \lambda(y)\ x$, $\lambda(y)\ x$ and x are respectively first $\emptyset$, then $\alpha_{\langle 11\rangle} \geqslant \bot$, $\alpha_{\langle 1211\rangle} \geqslant \bot$, $x : \alpha_{\langle 11\rangle}$ and finally $\alpha_{\langle 11\rangle} \geqslant \bot$, $\alpha_{\langle 1211\rangle} \geqslant \bot, x : \alpha_{\langle 11\rangle}$, $\alpha_{\langle g11\rangle} \geqslant \bot, y : \alpha_{\langle g11\rangle}$. The environment for e is $\alpha_{\langle 11\rangle} \geqslant \bot$, $\alpha_{\langle 1211\rangle} \geqslant \bot$.

Property 15.3.9 *Given an instantiation edge e, $\Gamma^e_\sharp$ is a well-formed xMLF environment.*□

Proof: Similar to the proof of Property 15.3.7.

15.3.4 Computation contexts

Before proceeding further on, we need to introduce the notion of computation contexts, which are used to relate graphic nodes and xMLF bounds.

Consider indeed a graphic type τ, and σ its translation in xMLF. Consider also an operation o that transforms a node n of τ, and suppose that we want to reflect this transformation in σ. In order to do so, we must find the correct alternation of computations $\forall\,(\geqslant \varphi)$ and $\forall\,(\alpha \geqslant)\ \varphi$ that descends into σ and positions us in front of the type corresponding to n in σ. Such sequences, which we call computation contexts, are given by the grammar

$$\mathcal{C} ::= \{\cdot\} \mid \forall\,(\geqslant \mathcal{C}) \mid \forall\,(\alpha \geqslant)\ \mathcal{C}$$

As evaluation contexts, computation contexts contain a single hole $\{\cdot\}$, in which we can substitute a computation or another context using the syntax $\mathcal{C}(\varphi)$ or $\mathcal{C}(\mathcal{C}')$.

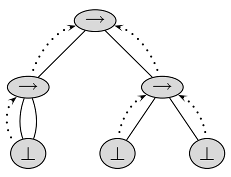


Figure 15.3.3 – Using computation contexts

▶ **Example** Consider the graphic type τ of Figure 15.3.3. This type translates in xML^F as

$$\forall\,(\alpha \geqslant \forall\,(\beta \geqslant \bot)\;\beta \to \beta)\;\forall\,(\gamma \geqslant \forall\,(\delta \geqslant \bot)\;\forall\,(\epsilon \geqslant \bot)\;\delta \to \epsilon)\;\alpha \to \beta$$

The grafting of a type τ' at $\langle 21 \rangle$ in τ can be translated in xML^F as the computation $\forall\,(\alpha \geqslant)$ $\forall\,(\geqslant \forall\,(\geqslant \triangleright \sigma'))$, where σ' is the translation of τ'. This computation can for example be decomposed into $\mathcal{C}\,(\varphi')$, where $\mathcal{C}$ and φ' are respectively $\forall\,(\alpha \geqslant)\;\forall\,(\geqslant \{\cdot\})$ and $\forall\,(\geqslant \triangleright \sigma')$

Any operation on a node transitively flexibly bound to the root of a type can be expressed using a computation context. Conversely, the operations on rigidly bound or inert-locked nodes cannot. This is unimportant in our case, as witness propagations of translatable presolutions only transform nodes transitively flexibly bound to the root.

15.3.4.1 Operating on a given node

Consider a type τ and its xML^F translation σ. Let also n be a node of τ transitively flexibly bound to the root (but different from the root). There exists a unique computation context $\mathcal{C}^{\langle\epsilon\rangle \to n}$ that can be used to descend in front of the quantification corresponding to n in σ.

▶ **Example** Consider the type τ of Figure 15.3.3, whose xML^F translation has been given in the previous section. We have

$$\mathcal{C}^{\langle\epsilon\rangle \to \langle 1 \rangle} = \{\cdot\} \qquad \mathcal{C}^{\langle\epsilon\rangle \to \langle 11 \rangle} = \forall\,(\geqslant \{\cdot\}) \qquad \mathcal{C}^{\langle\epsilon\rangle \to \langle 2 \rangle} = \forall\,(\alpha \geqslant)\;\{\cdot\}$$

$$\mathcal{C}^{\langle\epsilon\rangle \to \langle 21 \rangle} = \forall\,(\alpha \geqslant)\;\forall\,(\geqslant \{\cdot\}) \qquad \mathcal{C}^{\langle\epsilon\rangle \to \langle 22 \rangle} = \forall\,(\alpha \geqslant)\;\forall\,(\geqslant \forall\,(\delta \geqslant)\;\{\cdot\})$$

Notice that we descend *in front* of the quantification, but not on the bound itself. (Otherwise, we would have $\mathcal{C}^{\langle\epsilon\rangle \to \langle 1 \rangle} = \forall\,(\geqslant \{\cdot\})$.) Indeed, if for example we want to weaken $\langle 1 \rangle$, we must act on the quantification $\forall\,(\alpha \geqslant \_\,)$ for $\langle 1 \rangle$, not on the bound $\forall\,(\beta \geqslant \bot)\;\beta \to \beta$ of this node.

Interestingly, we can generalize this notation to nodes other than the root. Then we have for example

$$\mathcal{C}^{n \to n''} = \mathcal{C}^{n \to n'}(\mathcal{C}^{n' \to n''})$$

In order not to burden ourselves with α-conversion related issues, we suppose that computation contexts are built using variables whose names are based on the nodes they traverse, *e.g.* α_n for a node n.

▶ **Example** With this convention, we have

$$\mathcal{C}^{\langle\epsilon\rangle \to \langle 22 \rangle} = \forall\,(\alpha_{\langle 1 \rangle} \geqslant)\;\forall\,(\geqslant \forall\,(\alpha_{\langle 21 \rangle} \geqslant)\;\{\cdot\})$$

15.3.5 Translating normalized derivations into computations

In this section we consider an instantiation edge e of χ_p equal to $g \dashrightarrow\!\!\!\!\twoheadrightarrow d$. Our ultimate goal is to build a computation that witnesses $\chi_p^e \sqsubseteq \chi_p$. In order to do so, we define a more general translation function that acts on an instance χ of χ_p^e.

Translating a sequence of operations:

$$\begin{aligned} \mathcal{T}_r^\chi() &= \varepsilon \\ \mathcal{T}_r^\chi(o; I) &= \mathcal{T}_r^\chi(o); \mathcal{T}_r^{o(\chi)}(I) \end{aligned}$$

Translating an operation on a rigid node:

$$\left.\begin{aligned} \mathcal{T}_r^\chi(\mathsf{Raise}(n)) \\ \mathcal{T}_r^\chi(\mathsf{Merge}(n, n')) \\ \mathcal{T}_r^\chi(\mathsf{RaiseMerge}(n, n')) \end{aligned}\right\} = \varepsilon \quad \text{if } \overset{\diamond}{\chi}(n) = (=)$$

Translating an operation on the (flexibly bound) root of the expansion:

$$\begin{aligned} \mathcal{T}_r^\chi(\mathsf{Graft}(\tau, r)) &\;=\; \triangleright \mathcal{S}(\tau) \\ \mathcal{T}_r^\chi(\mathsf{RaiseMerge}(r, n')) &\;=\; \alpha_{n'} \triangleleft \\ \mathcal{T}_r^\chi(\mathsf{Weaken}(r)) &\;=\; \varepsilon \end{aligned}$$

Translating an operation on a flexible node different from the root:

$$\begin{aligned} \mathcal{T}_r^\chi(\mathsf{Graft}(\tau, n)) &\;=\; \mathcal{C}^{r \to n}(\forall\,(\geqslant \triangleright \mathcal{S}(\tau))) \\ \mathcal{T}_r^\chi(\mathsf{RaiseMerge}(n, n')) &\;=\; \mathcal{C}^{r \to n}(\forall\,(\geqslant \alpha_{n'} \triangleleft); \&) \\ \mathcal{T}_r^\chi(\mathsf{Merge}(n, n')) &\;=\; \mathcal{C}^{r \to n}(\forall\,(\geqslant \alpha_{n'} \triangleleft); \&) \\ \mathcal{T}_r^\chi(\mathsf{Weaken}(n)) &\;=\; \mathcal{C}^{r \to n}(\&) \\ \mathcal{T}_r^\chi(\mathsf{Raise}(n)) &\;=\; \mathcal{C}^{r \to n'}(\parr; \forall\,(\geqslant \triangleright \mathcal{S}_\chi(n)); \forall\,(\beta_n \geqslant)\; (\mathcal{C}^{n' \to n}(\forall\,(\geqslant \beta_n \triangleleft); \&)))) \\ &\qquad \text{where } n' = \min_{<_{\mathsf{P}}}\{n' \in (\longrightarrow \hat{\chi}(\hat{\chi}(n))) \mid n <_{\mathsf{P}} n'\} \end{aligned}$$

Figure 15.3.4 – Translating normalized instance operations

We let r be the root of the expansion of e in χ. Let also I be a normalized instance derivation that transforms the expansion in χ; we suppose that I does not transform inert-locked nodes. We are going to build an $\mathsf{xML^F}$ computation that mirrors the operations of I. Thus, if σ is the translation of r in χ, and supposing that our computation is correctly built, it will be applicable to σ, and the result will be the type of r in $I(\chi)$. We write $\mathcal{T}_r^\chi(I)$ this translation function, with I, χ and r defined as above. The definition of $\mathcal{T}_r^\chi$ is given in Figure 15.3.4, and is explained below. The function $\mathcal{T}_r^\chi$ is overloaded to act on both a sequence of operations or a single operation.

Translating a sequence of operations We start by explaining the translation of an entire instance derivation, which is easier:

- if the derivation is empty, the type of r does not change. We simply return the identity computation, which is correct by construction.

- if the derivation is not empty, we translate its first operation o into a computation φ, which transforms the type of r in χ into the type of r in $o(\chi)$. Then we recursively call the translation on the remainder of the derivation. However, the context has changed: the constraint to consider is now $o(\chi)$.

Translating a single operation The interesting part is thus the translation of a single operation. Operations on rigid nodes are easy: since rigid bounds are inlined, the operation does not change the translation of the expansion in xML^F, and we simply return the identity computation. (Of course, only raising and merging are possible on rigid nodes.)

Next, we consider an operation on the root r of the expansion, provided this node is flexibly bound.

- The grafting of a type τ is simply translated as a computation $\triangleright\sigma$—after translation of τ into xML^F. By definition of grafting, τ is closed so we only need to inline rigid nodes; there will be no free variables in the result. (In particular the translation is not linked to χ at all.)
- A raising-merging of r with a node n' of its exterior is the last operation of the derivation, as no node remains in the expanded part afterwards. Necessarily, the bound for n' is in the typing environment, and we abstract the type of r under this name.
- The weakening of r is the next-to-last operation in the derivation (as weakenings are delayed), before the merging of r with a rigidly bound node of its exterior. There is actually nothing to reflect in xML^F, as the type of r itself is not changed—only its binding flag in the expansion. Hence we simply translate the operation as the identity computation.

The most involved cases are the operations on a flexibly bound node n which is not r. Since the derivation does not transform inert-locked nodes, and since (by subcase hypothesis) we are not transforming a rigidly bound node, n is transitively bound to r, and there exists a computation context $\mathcal{C}^{r\to n}$ for the bound of n in σ.

- The grafting of a type τ at n is translated as a computation $\triangleright\mathcal{S}(\tau)$ instantiating the bound $\bot$ of n in χ into $\mathcal{S}(\tau)$.
- For the merging of n with a node n', we first abstract the bound of n under the name of the bound of n', and then immediately substitute the bound of n. This is correct w.r.t. scopes: by the hypothesis on the merging operations in propagation witnesses, we have $n' <_{\mathsf{P}} n$ and n' is thus quantified before n in $\mathcal{S}_\chi(r)$.
- The computation for a raising-merging of n with a node n' of the exterior is the same as for a merging. This time however, n' is not quantified in $\mathcal{S}_\chi(r)$ but in the typing environment.
- The weakening of n is simply translated as the computation $\&$, inserted at the proper location in $\mathcal{S}_\chi(r)$.

- The most involved case is the one for raising. As a first step, we insert inside $\mathcal{S}_\chi(r)$ a fresh quantification for a copy of the type of n. The difficulty consists in finding *where* to insert this quantification, as it is important to respect the ordering between bounds. Notice that n' exists: the set $\{n' \in (\longrightarrow\!\!\triangleright\ \hat{\chi}(\hat{\chi}(n))) \mid n <_{\mathsf{P}} n'\}$ is not empty, as it contains at least $\hat{n}$.

  Next we use an outer quantification to abstract the new bound, under a name that will not create a clash. Then we find the current bound of n, abstract it under the name of the new quantification, and substitute this modified bound. This effectively raises n, as it is now quantified one level higher.

▶ **Example** Since g' is degenerate in χ_p, the propagation witness for e' is the empty derivation, which thus translates into ε. More interestingly, a propagation witness for $\chi_p^e \sqsubseteq \chi_p$ and the corresponding translation is given below.

Graphic operation	Computation
$\mathsf{Graft}(\sigma_{\bot\to\bot}, \langle n1\rangle)$	$\forall\,(\geqslant \triangleright \forall\,(\alpha \geqslant \bot)\ \forall\,(\beta \geqslant \bot)\ \alpha \to \beta)$
$\mathsf{Raise}(\langle n11\rangle)$	$⅋; \forall\,(\geqslant \triangleright \bot); \forall\,(\beta_{\langle n11\rangle} \geqslant)\ \forall\,(\geqslant \forall\,(\geqslant \beta_{\langle n11\rangle}); \&)$
$\mathsf{RaiseMerge}(\langle n11\rangle, \langle 1211\rangle)$	$\forall\,(\geqslant \alpha_{\langle 1211\rangle} \triangleleft); \&$
$\mathsf{Weaken}(\langle n1\rangle)$	$\forall\,(\geqslant \&)$
$\mathsf{Weaken}(n)$	ε
$\mathsf{Merge}(n, \langle 12\rangle)$	ε

where $\sigma_{\bot\to\bot}$ is the graphic type corresponding to $\forall\,(\alpha \geqslant \bot)\ \forall\,(\beta \geqslant \bot)\ \alpha \to \beta$. We call φ_e the sequence of all the computations above.

15.3.5.1 Soundness of the translation

We make the same hypothesis as in the previous section regarding I. The correctness of our translation is stated under $\Gamma_\sharp^e$.

Lemma 15.3.10 *Suppose that I contains an operation* $\mathsf{RaiseMerge}(n, n')$*, with n flexibly bound. Then $\alpha_{n'} \in \mathsf{dom}(\Gamma_\sharp^e)$.* □

Proof: Let g' be the gen node on which d is bound in χ_t, χ'' be $\hat{\chi_p}(d)$. By monotony of instance and the fact that gen nodes are not raised, we have $g' \xrightarrow{*}\!\!\triangleright g''$. In χ, which is an instance of χ_p, we have $\hat{r} = g''$: normalized derivations do not allow raising r step by step, but use an atomic raise-merge operation. By definition, an operation $\mathsf{RaiseMerge}(n, n')$ of I merges n (which is in $\mathcal{I}^s(r)$) with a node n' bound on a gen node g''' such that $g'' \xrightarrow{*}\!\!\triangleright g'''$. By construction $\Gamma_\sharp^e$ contains all the nodes flexibly bound on a gen node equal to or above g', thus in particular n'. This is the desired result.

Lemma 15.3.11 *Suppose σ is $\mathcal{S}'_\chi(r)$ and σ' is $\mathcal{S}'_{I(\chi)}(r)$. Then $\Gamma_\sharp^e \vdash \mathcal{T}_r^\chi(I) : \sigma \leq \sigma'$.* □

Proof: It suffices to show the result for a single instance operation o, as the result immediately follows by induction for the general case. The cases for an operation on a rigid bound are immediate, as those bounds are inlined during the translation.

For an operation on the root:

- ▷ *Case* $o = \mathsf{Graft}(\tau, r)$: necessarily σ is $\perp$, as otherwise r could not be grafted. The result is then immediate.
- ▷ *Case* $o = \mathsf{RaiseMerge}(n, n')$: by Lemma 15.3.10, $\alpha_{n'}$ is in $\Gamma^e_\sharp$. The bounds of n and n' are syntactically equal, since nodes are always translated according to $<_\mathsf{P}$.
- ▷ *Case* $o = \mathsf{Weaken}(r)$: the translation of r before and after the weakening is the same, as r is not a named node.

For an operation on a flexible node different from the root: the computations contexts we use exist, as by hypothesis I does not transform inert-locked nodes. Notice that for any σ and σ', we have $(\forall\,(\alpha \geqslant \sigma)\ \sigma')[\forall\,(\geqslant \beta \triangleleft); \&] = \sigma'\{\alpha \leftarrow \beta\}$. This ensures the correctness of the operations $\mathsf{Merge}(n, n')$ and $\mathsf{RaiseMerge}(n, n')$ (using also Lemma 15.3.10 in the second case). The correctness of the translations for a grafting and a weakening are immediate.

The difficult point is the correctness of an operation $\mathsf{Raise}(n)$, as we must ensure that the free variables of $\mathcal{S}_\chi(n)$ are in scope at the place where we insert the new bound. Thus we must justify that for a node n'' in $\mathcal{F}^s(n)$, we have $n'' <_\mathsf{P} n'$. By definition of n'', we have $n \xrightarrow{+} n''$ and $\hat{\chi}(n) \xrightarrow{+} \hat{\chi}(n'')$. We cannot have $\hat{\chi}(n) = \hat{\chi}(n'')$, as n would not be raisable. Hence n'' is bound on $\hat{\chi}(\hat{\chi}(n))$ or above. If it is bound strictly above the result holds, as $\hat{\chi}(n') = \hat{\chi}(\hat{\chi}(n))$. Otherwise, if both n' and n'' are bound on $\hat{\chi}(\hat{\chi}(n))$, the result holds because $n <_\mathsf{P} n'$ and n'' is under n.

As a consequence, the translation of an entire propagation witness is correct.

Definition 15.3.12 (*Translation of an instantiation edge*) Let e be an instantiation edge $g \twoheadrightarrow d$ of χ_p. Let I be a propagation witness for $\chi^e_p \sqsubseteq \chi_p$. We write $\mathcal{T}(e)$ the computation $\mathcal{T}^{\chi^e_p}_r(I)$, where r is the root of the expansion in χ^e_p. ■

Since there can exists more than one propagation witness, this definition it not completely deterministic. However this only reflects the fact that different computations can be used to transform one type into another. Thus we simply pick any propagation witness.

Lemma 15.3.13 *Let a' be a subterm of a, e an instantiation edge $g_{a'} \twoheadrightarrow d$ of χ_p. Let σ and σ' be $\mathsf{Typ}^{exp}(a')$ and $\mathcal{S}'_{\chi_p}(d)$ respectively. Then $\Gamma^e_\sharp \vdash \mathcal{T}(e) : \sigma \leq \sigma'$.* □

Proof: Let r be the root of the expansion in χ^e_p. Let I be the propagation witness used to obtain $\mathcal{T}(e)$. By definition of a propagation witness we have $(I(\chi^e_p))(r) = d$. By definition of $\mathsf{Typ}^{\exp}$, we have $\sigma = \mathcal{S}'_{\chi_p}(r)$. By Lemma 15.2.6, I does not transform inert-locked nodes. The result holds by the points above and Lemma 15.3.11.

15.3.6 Elaborating a translatable presolution

We are now ready to finish the elaboration of χ_p. The translation of a into an xML^F term is defined inductively on the shape of a, and given in Figure 15.3.5. We sometimes need to name some nodes and instantiation edges of χ_p, and it is actually simpler to name those constructs on the corresponding nodes of χ_t; thus we have reproduced at the bottom of the figure the basic typing constraint for the four interesting cases.

Let us give some details, assuming that we are translating a subterm a' of a. If a' is a λ-bound variable x, the corresponding type scheme in χ_p is degenerate. Thus there is

$$\begin{aligned}
\mathsf{T}(x) &= \begin{cases} x & \text{if } x \text{ is } \lambda\text{-bound} \\ \Lambda(\Gamma^g)\ x[\varphi_{\mathsf{R}}(x); \mathcal{T}(e)] & \text{if } x \text{ is } \mathsf{let}\text{-bound} \end{cases} && \textbf{(1)} \\
\mathsf{T}(\lambda(x)\ a) &= \Lambda(\Gamma^g)\ \lambda(x : \mathcal{S}'_{\chi_p}(n))\ (\mathsf{T}(a))[\varphi_{\mathsf{R}}(a); \mathcal{T}(e)] && \textbf{(2)} \\
\mathsf{T}(a_1\ a_2) &= \Lambda(\Gamma^g)\ (\mathsf{T}(a_1))[\varphi_{\mathsf{R}}(a_1); \mathcal{T}(e_1)]\ (\mathsf{T}(a_2))[\varphi_{\mathsf{R}}(a_2); \mathcal{T}(e_2)] && \textbf{(3)} \\
\mathsf{T}(\mathsf{let}\ x = a\ \mathsf{in}\ b) &= \Lambda(\Gamma^g)\ \mathsf{let}\ x = \mathsf{T}(a)\ \mathsf{in}\ (\mathsf{T}(b))[\varphi_{\mathsf{R}}(b); \mathcal{T}(e)] && \textbf{(4)}
\end{aligned}$$

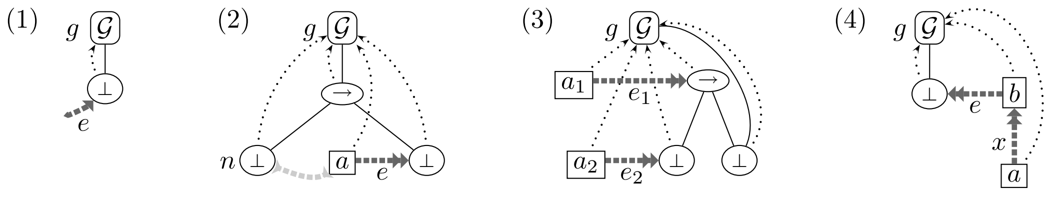


Figure 15.3.5 – Elaboration of a λ-term

no quantification to insert, and no computation either: the elaboration of x is x itself. In all the other cases the gen node for a' is a priori not degenerate, and we insert a type quantification for the root node of the typing constraint for a' in front of a'. Moreover:

- For a let-bound variable, we instantiate the type of the variable according to the incoming instantiation edge.
- For an abstraction $\lambda(x)\ a$, we annotate x according to its type in χ_p, and instantiate the translation of a in χ_p according to the instantiation edge linking a and the return type of the abstraction.
- For an application $a_1\ a_2$, we simply instantiate the translations of a_1 and a_2 according to the corresponding instantiation edges.
- The translation of let $x = a$ in b, with the revised typing constraint for let, is also quite simple. Both a and b are recursively translated, and b is coerced to the type of the root scheme.

Importantly, each time we translate an instantiation edge, we also convert the xMLF view of the type scheme into gMLF vision using a quantifier reordering.

▶ **Example** The elaboration of χ_p is

$$\Lambda(\alpha_{\langle 11\rangle} \geqslant \perp)\ \Lambda(\alpha_{\langle 1211\rangle} \geqslant \perp)\ \lambda(x : \alpha_{\langle 11\rangle})\ (\Lambda(\alpha_{\langle g11\rangle} \geqslant \perp)\ \lambda(y : \alpha_{\langle g11\rangle})\ x[\varepsilon; \varepsilon])[\varepsilon; \varphi_e]$$

In both computations, the frontmost identity computations are for the reorderings; since Typ and $\mathsf{Typ}^{\mathrm{exp}}$ agree on the gen nodes of our example, they are not actually needed.

Of course, our translation respects the shape of terms.

Property 15.3.14 *The type-erasure $\lceil \mathcal{T}(a) \rceil$ of $\mathcal{T}(a)$ is a.* □

Proof: Immediate induction on the shape of a.

15.3.7 Correctness of the translation

We can finally prove the correctness of our translation.

Theorem 15.3.15 *Let a' be a subterm of a. Then $\Gamma_\star^{a'} \vdash \mathcal{T}(a') : \mathsf{Typ}(a')$.* □

Proof: The proof is by induction on the shape of a. We use the same notations for nodes and instantiation edges as in Figure 15.3.4. In the following we omit χ_p when $\mathcal{S}'$ is called on it; we also omit « $\langle\cdot\rangle$ » around compound nodes. Thus $\mathcal{S}'(g1)$ is $\mathcal{S}'_{\chi_p}(\langle g \cdot 1\rangle)$.

- ▷ *Case $a' = x$, with x λ-bound*: let g be the gen node corresponding to the λ-abstraction introducing x in a. By definition of solved typing constraints and unification, we have $\langle g_{a'}1\rangle = \langle g11\rangle$ in χ_p (**1**). In particular, $g_{a'}$ is degenerate and $\mathsf{Typ}(a')$ is $\mathcal{S}'(g_{a'}1)$ (**2**). By definition of typing environments, x is bound in $\Gamma_\star^{a'}$ and its bound is $\mathcal{S}'(g11)$. Thus the result holds by rule Var, (1), (2) and the fact that $\mathsf{T}(a') = x$.
- ▷ *Case $a' = x$, with x let-bound*: then $\mathsf{T}(a') = \Lambda(\Gamma^{a'})\ x[\varphi_{\mathsf{R}}(x); \mathcal{T}(e)]$. Let $\mathsf{let}\ x = a\ \mathsf{in}\ b$ be the subterm of a introducing x. The edge e is $g_a \dashrightarrow \langle g_{a'}1\rangle$. Let Γ' be $\Gamma_\star^{a'}, \Gamma^{a'}$. By Lemma 15.3.13 we have $\Gamma' \vdash \mathcal{T}(e) : \mathsf{Typ}^{\exp}(a) \leq \mathcal{S}'(g_{a'}1)$. By Lemma 15.3.5 and rule Inst-Trans, $\Gamma' \vdash (\varphi_{\mathsf{R}}(x); \mathcal{T}(e)) : \mathsf{Typ}(a) \leq \mathcal{S}'(g_{a'}1)$. By definition of typing environment, x is in $\mathsf{dom}(\Gamma_\star^{a'})$ and its bound to $\mathsf{Typ}(a)$. Thus by Var and TApp we obtain $\Gamma' \vdash x[\varphi_{\mathsf{R}}(x); \mathcal{T}(e)] : \mathcal{S}'(g_{a'}1)$. By applying TAbs repeatedly we obtain $\Gamma_\star^{a'} \vdash \Lambda(\Gamma^{a'})\ x[\varphi_{\mathsf{R}}(x); \mathcal{T}(e)] : \forall(\Gamma^{a'})\ \mathcal{S}'(g_{a'}1)$. This last type is indeed $\mathsf{Typ}(a')$.
- ▷ *Case $a' = a_1\ a_2$*: By induction hypothesis, for $1 \leq i \leq 2$, we have $\Gamma_\star^{a_i} \vdash \mathsf{T}(a_i) : \mathsf{Typ}(a_i)$ (**3**). Since $a' = a_1\ a_2$, we have $\Gamma_\star^{a_1} = \Gamma_\star^{a_2} = \Gamma_\star^{a'}, \Gamma^{a'}$ (**4**). We call this environment Γ'. Let n be the arrow node in the typing constraint for $a_1\ a_2$. By Lemma 15.3.13, Lemma 15.3.5, rule TApp, rule Inst-Trans, (3) and (4) we have $\Gamma' \vdash (\mathsf{T}(a_1))[\varphi_{\mathsf{R}}(a_1); \mathcal{T}(e_1)] : \mathcal{S}'(n)$ and $\Gamma' \vdash (\mathsf{T}(a_2))[\varphi_{\mathsf{R}}(a_2); \mathcal{T}(e_2)] : \mathcal{S}'(n1)$ (**5**). Since χ_p is translatable, n is rigidly bound; moreover there is no node bound on it in χ_p, as there was no node bound on it in χ_t. Thus $\mathcal{S}'(n) = \mathcal{S}'(n1) \to \mathcal{S}'(n2)$ (**6**). By App, (5) and (6), $\Gamma' \vdash (\mathsf{T}(a_1))[\varphi_{\mathsf{R}}(a_1); \mathcal{T}(e_1)]\ (\mathsf{T}(a_2))[\mathcal{T}(\varphi_{\mathsf{R}}(a_2); e_2)] : \mathcal{S}'(n2)$. By repeated applications of TAbs we obtain $\Gamma_\star^{a'} \vdash \Lambda(\Gamma^{a'})\ (\mathsf{T}(a_1))[\varphi_{\mathsf{R}}(a_1); \mathcal{T}(e_1)]\ (\mathsf{T}(a_2))[\varphi_{\mathsf{R}}(a_2); \mathcal{T}(e_2)] : \forall(\Gamma^{a'})\ \mathcal{S}'(n2)$ (**7**). By definition of the typing constraint for application, we have $\langle n2\rangle = \langle g_{a'}1\rangle$. Thus $\forall(\Gamma^{a'})\ \mathcal{S}'(n2)$ is $\mathsf{Typ}(a')$; together with (7) this is the desired result.
- ▷ *Case $a' = \mathsf{let}\ x = a\ \mathsf{in}\ b$*: by induction hypothesis we have $\Gamma_\star^{a} \vdash \mathsf{T}(a) : \mathsf{Typ}(a)$ and $\Gamma_\star^{b} \vdash \mathsf{T}(b) : \mathsf{Typ}(b)$ (**1**). Moreover we have $\Gamma_\star^{a} = \Gamma_\star^{a'}, \Gamma^{a'}$ and $\Gamma_\star^{b} = \Gamma_\star^{a'}, \Gamma^{a'}, x : \mathsf{Typ}(a)$ (**2**). By Lemmas 15.3.13 and 15.3.5, rules TApp and Inst-Trans, (1) and (2) we have $\Gamma_\star^{b} \vdash (\mathsf{T}(b))[\varphi_{\mathsf{R}}(b); \mathcal{T}(e)] : \mathcal{S}'(g_{a'}1)$. Thus, by Let and (1) we have $\Gamma_\star^{a} \vdash \mathsf{let}\ x = \mathsf{T}(a)\ \mathsf{in}\ (\mathsf{T}(b))[\varphi_{\mathsf{R}}(b); \mathcal{T}(e)] : \mathcal{S}'(g_{a'}1)$. By iterating TAbs, we have $\Gamma_\star^{a'} \vdash \Lambda(\Gamma^{g})\ \mathsf{let}\ x = \mathsf{T}(a)\ \mathsf{in}\ (\mathsf{T}(b))[\varphi_{\mathsf{R}}(b); \mathcal{T}(e)] : \forall(\Gamma^{a'})\ \mathcal{S}'(g_{a'}1)$. This last type is $\mathsf{Typ}(a')$.
- ▷ *Case $a' = \lambda(x)\ a$*: by induction hypothesis we have $\Gamma_\star^{a} \vdash \mathsf{T}(a) : \mathsf{Typ}(a)$ (**1**). We also have $\Gamma_\star^{a} = \Gamma_\star^{a'}, \Gamma^{a'}, x : \mathcal{S}'(g_{a'}11)$ (**2**). By Lemma 15.3.13, Lemma 15.3.5, rule TApp, rule Inst-Trans, (1) and (2) we have $\Gamma_\star^{a} \vdash (\mathsf{T}(a))[\varphi_{\mathsf{R}}(a); \mathcal{T}(e)] : \mathcal{S}'(g_{a'}12)$. By Abs we

obtain $\Gamma_\star^{a'}, \Gamma^{a'} \vdash \lambda(x : \mathcal{S}'(g_{a'}11))\ (\mathsf{T}(a))[\varphi_\mathsf{R}(a); \mathcal{T}(e)] : \mathcal{S}'(g_{a'}11) \to \mathcal{S}'(g_{a'}12)$ (**3**). By definition of translatable presolutions, $\langle g_{a'}1 \rangle$ is rigid. Moreover there is no node bound on this node in χ_t, hence in χ_p. Thus $= \mathcal{S}'(g_{a'}1) = \mathcal{S}'(g_{a'}11) \to \mathcal{S}'(g_{a'}12)$ (**4**). By applying TApp to (3), and by (4), we have $\Gamma_\star^{a'} \vdash \Lambda(\Gamma^{a'})\ \lambda(x : \mathcal{S}'(g_{a'}11))\ (\mathsf{T}(a))[\varphi_\mathsf{R}(a); \mathcal{T}(e)] : \forall\,(\Gamma^{a'})\ \mathcal{S}'(g_{a'}1)$. This last type is $\mathsf{Typ}(a')$, hence the result.

Since the typing environment for a itself is the empty environment, the elaboration of χ_p is a valid *x*MLF term.

Corollary 15.3.16 (Translated terms are typable) $\varnothing \vdash \mathit{T}(a) : \mathit{Typ}(a)$. □

15.3.8 Translating type annotations

Let us consider the term ω equal to $\lambda(x : \sigma_{\mathsf{id}})\ x\ x$. It desugares into

$$\lambda(x)\ \mathsf{let}\ y = c_{\sigma_{\mathsf{id}}}\ x\ \mathsf{in}\ y\ y$$

In order to translate this term into *x*MLF, we need to find a term for the coercion $c_{\sigma_{\mathsf{id}}}$. Interestingly, in *x*MLF we do not need to add coercion functions to the initial environment, as they are one of the possible typings for the identity function. For example, the coercion $c_{\sigma_{\mathsf{id}}}$ may be defined as

$$\Lambda(\gamma \geqslant \sigma_{\mathsf{id}})\ \lambda(z : \sigma_{\mathsf{id}})\ z[\gamma] \quad : \quad \forall\,(\gamma \geqslant \sigma_{\mathsf{id}})\ \sigma_{\mathsf{id}} \to \gamma$$

Let us finish our example. We do not show the principal presolution for ω, as the resulting constraint is quite big. However, up to identity computations, ω elaborates into the following *x*MLF term:

$$\Lambda(\alpha \geqslant \sigma_{\mathsf{id}})\ \lambda(x : \sigma_{\mathsf{id}})\ \mathsf{let}\ y = c_{\sigma_{\mathsf{id}}}[\&]\ x\ \mathsf{in}\ (y[\sigma_{\mathsf{id}}]\ y)[\alpha]$$

which has type $\forall\,(\alpha \geqslant \sigma_{\mathsf{id}})\ \sigma_{\mathsf{id}} \to \alpha$.

Notice that we can reduce the term $\mathsf{let}\ y = c_{\sigma_{\mathsf{id}}}[\&]\ x\ \mathsf{in}\ \_$ by using reduction under abstractions, resulting in the term:

$$\Lambda(\alpha \geqslant \sigma_{\mathsf{id}})\ \lambda(x : \sigma_{\mathsf{id}})\ (x[\sigma_{\mathsf{id}}]\ x)[\alpha]$$

On the principal presolution of a constraint, this is actually always possible: the type inference algorithm does not instantiate the type of the coercion (as it not constrained), and in the typing constraint for $c_\kappa\ v$, the (flexibly bound) domain of the arrow is rigidified, as it is the scheme node for that constraint. Hence the application can be reduced by β-reduction. Thus coercion functions are introduced for type inference purposes, but can always be removed in *x*MLF.

15.3.9 Soundness of *g*MLF

Putting all the results obtained previously together, we can finally state the soundness of *g*MLF.

Theorem 15.3.17 *gMLF is sound, for both call-by-value or call-by-name semantics.* □

Proof: Let a be a term typable in xML^F, χ_p one of its presolutions. By Theorem 15.2.11, there exists a translatable presolution χ'_p of the typing constraint for a. Let a' be the elaboration of this presolution in xML^F. Property 15.3.14 ensures that a and a' have the same type-erasure. Corollary 15.3.16 ensures that a' is typable in xML^F. The conclusion is by Theorems 14.4.5, 14.4.9 and 14.4.14.

Theorem 13.2.4 shows that ay term typable in eML^F is also typable in gML^F. Thus we have also proven the soundness of this system.

Theorem 15.3.18 *eML^F is sound.* □

Section 15.5 discusses how to translate eML^F and iML^F presolutions into xML^F, giving another (more direct) proof of the result above.

15.3.10 Obtaining instance derivations

So far, we have only explained how to instrument instance derivations in order to obtain type computations. The instance derivations themselves are obtained from the (constructive) proof of Lemma 11.5.3. Another possibility would be to instrument the type inference algorithm so that it also returns a witness. However, there are some difficulties with this approach, which we explain below.

A first step would be to instrument the unification algorithm so that it returns an instance derivation I showing that its result is an instance of its argument. This seemingly simple part is not immediate. Indeed, Unif computes the structure of the unifier using first-order unification, but the operations Merge in I must be found by taking into account the binding tree of the unifier (so as to merge only locally congruent nodes). Moreover, the unification algorithm is top-down for efficiency reasons, while normalized derivations are bottom-up, at least for weakenings. Finally, we would need to change Unif so that it returns an unifier in which inert-locked nodes are weakened. This is also difficult to do top-down, as the fact that a node is inert depends on the nodes bound on it. Thus generating I on the fly during unification is not simple, and a post-treatment phase is probable needed.

Instrumenting the type inference algorithm is not immediate either. Indeed, the exterior of gen nodes can change during inference, which might require changing computations already built. For example, suppose that at some point we generate a computation for an operation RaiseMerge(n, n'). Later during inference, n' might be instantiated, for example by the grafting of a type τ. Then we need to change the computation for the operation already built, by adding a grafting on n before the translation of RaiseMerge(n, n').

The points above show that building the elaborated terms on-the-fly during type inference is tricky. Our approach of translating presolutions is comparatively much simpler. The only potential downside could be the fact that presolutions can be huge, compared to the solutions themselves. However the elaboration of a presolutions is at least as big as the presolution itself: entirely building the presolution during type inference is thus not really a concern.

15.4 Obtaining simpler elaborated terms

There are many ways to simplify the $x\mathsf{ML^F}$ terms obtained by translating a presolution, as the type computations we generate are not especially optimized. This can be useful if $x\mathsf{ML^F}$ is used as a core internal language, where smaller terms are likely to lead to better performances during the compilation process. We present the most obvious solutions (that can be readily implemented) below. We discuss a more involved approach in §15.4.1 while §15.4.2 shows that the simplifications rules we have presented in §12.4.1 do not really help in obtaining smaller $x\mathsf{ML^F}$ terms.

Removing identity computations: an immediate optimization consists in removing the computations ε that are sometimes generated during translation.

Optimizing the introduction of fresh bounds the computations for a raising or a reordering introduce fresh bounds using a computation of the form $⅋; \forall\,(\geqslant \triangleright \mathcal{S}(n))$. However, if $\mathcal{S}(n)$ is $\bot$, the second part of the computation is entirely superfluous, and we can simply insert $⅋$.

Optimizing reordering: the computations we generate for reordering quantifications are simple and fully general, but partially redundant when $\mathsf{Typ}(a)$ and $\mathsf{Typ}^{\exp}(a')$ have some similarities. For example, in order to convert $\forall\,(\alpha \geqslant \sigma_\alpha)\ \forall\,(\beta \geqslant \sigma_\beta)\ \forall\,(\gamma \geqslant \sigma_\gamma)\ \forall\,(\delta \geqslant \sigma_\delta)\ \sigma$ into $\forall\,(\alpha \geqslant \sigma_\alpha)\ \forall\,(\gamma \geqslant \sigma_\gamma)\ \forall\,(\beta \geqslant \sigma_\beta)\ \forall\,(\delta \geqslant \sigma_\delta)\ \sigma$, there is no need to reintroduce quantifications for α and δ, as is currently done. A simpler computation is $\forall\,(\alpha \geqslant)\ \varphi_{\beta\gamma}$, where $\varphi_{\beta\gamma}$ commutes the bounds of β and γ.

Sharing computation contexts: when translating an entire propagation witness, two consecutive operations on two nodes n and n' will result in two computations $\mathcal{C}^{r\to n}(\varphi)$ and $\mathcal{C}^{r\to n'}(\varphi')$. If n and n' have a common binder n_0, we can write the compositions of those two computations as $\mathcal{C}^{r\to n''}(\mathcal{C}^{n_0\to n}(\varphi); \mathcal{C}^{n_0\to n'}(\varphi'))$. This avoids the introduction of some inner or outer computations, which are shared in $\mathcal{C}^{r\to n''}$.

We can improve the efficiency of this optimization by generating particular propagation witnesses. This is discussed in the next section.

Also, even though we took great care to maintain sharing during $g\mathsf{ML^F}$ type inference (by considering $\sqsubseteq$ and not $\sqsubseteq^{\approx}$), the sharing on rigid bounds is lost during the translation into $x\mathsf{ML^F}$. Hence the complexity of the translation may not be optimal. A possible solution may be to use hash-consing. Alternatively, rigid bounds could be reincarnated in $x\mathsf{ML^F}$ as type abbreviations, and used to express this sharing internally.

15.4.1 Creating optimized propagation witnesses

The instance relations of $g\mathsf{ML^F}$ and $x\mathsf{ML^F}$ are "syntactically" very different. As a consequence, translating one $g\mathsf{ML^F}$ instance operation after the other can result in unnecessarily big computations. Instead, it is much more interesting to group some operations together, and to translate them atomically. We give some examples below.

- a sequence of k raisings on the same node n is currently translated by introducing k times the bound for n, interleaved with k computations $\alpha \triangleleft$. A much simpler solution is to insert only the final bound, then doing a single abstraction.

- a raising on a node n followed by a weakening of this node can be reversed; *i.e.* the weakening is performed first, then the raising. As a consequence, the raising needs not be translated at all (its effect is lost once the bound of n is inlined). Likewise, if n is merged with a node n' before being weakened, it might be simpler to weaken both n and n' first, and not to translate the merging.

- a sequence of operation on a node n starting by a grafting, followed by some operations on the grafted nodes, and ending by an operation $\mathsf{Merge}(n, n')$ or $\mathsf{RaiseMerge}(n, n')$ can, in $x\mathsf{ML^F}$, be entirely replaced by this last operation. This reflects the fact that $x\mathsf{ML^F}$ allows using a type σ containing free variables in a computation $\triangleright\,\sigma$, while $g\mathsf{ML^F}$ only allows grafting closed types.

Crucially, the efficiency of those optimizations depend on the way the propagation witness is built, as it needs to group operations on the same nodes together. As we have already discussed at the beginning of §15.4, it is also interesting to group operations sharing the same computation context.

15.4.2 Using the simplifications rules on constraints

We have presented in §12.4.1 two simplifications rules for the typing of variables, Var-Let and Var-Abs. We detail in this section how their use change the elaborated $x\mathsf{ML^F}$ λ-terms.

15.4.2.1 Let-bound variables

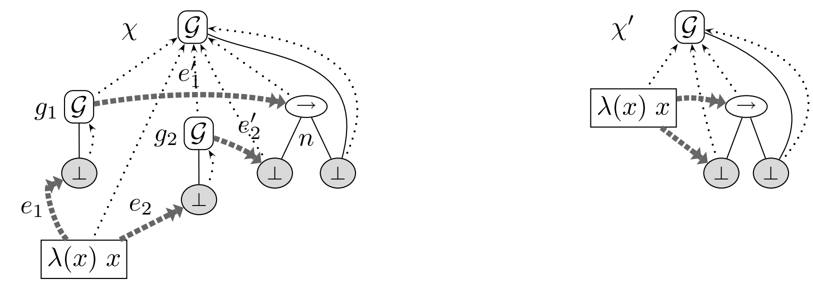


Figure 15.4.1 – Typing constraint for $\mathsf{let}\ y = \lambda(x)\ x\ \mathsf{in}\ y\ y$ with and without Var-Let

Let us consider the typing constraint for the term $\mathsf{let}\ y = \lambda(x)\ x\ \mathsf{in}\ y\ y$, which is presented as χ in Figure 15.4.1. Using Var-Let, we obtain instead the constraint χ'. The interesting part is the typing of the two occurrences of y. In χ, there are two different opportunities for instantiating each occurrence of y, *i.e.* one by instantiation edge e_i and e'_i. For example, if the term $\lambda(x)\ x$ is typed as $\forall\,(\alpha \geqslant \bot)\ \alpha \rightarrow \alpha$, we could first instantiate the gen node g_1 with type

$$\forall\,(\beta \geqslant \forall\,(\gamma \geqslant \bot)\ \gamma \rightarrow \gamma)\ \beta \rightarrow \beta$$

then the node n with type

$$(\mathsf{int} \rightarrow \mathsf{int}) \rightarrow (\mathsf{int} \rightarrow \mathsf{int})$$

This would result on a term of the form

$$\mathsf{let}\ y = \Lambda(\alpha \geqslant \bot)\ \lambda(x : \alpha)\ x\ \mathsf{in}\ y[\forall\,(\geqslant \triangleright (\forall\,(\gamma \geqslant \bot)\ \gamma \rightarrow \gamma))][\forall\,(\geqslant \mathsf{int}); \&]\ y[\varphi_2][\varphi_2']$$

where φ_2 and φ_2' are two computations giving to y the type $\mathsf{int} \rightarrow \mathsf{int}$.

Conversely, things are simpler for χ', in which each occurrence of y can be instantiated only once. Notice however that we have the same expressivity in χ' and χ, as the two type applications $y[\varphi_i][\varphi_i']$ in χ can simply be recombined as $y[\varphi_i; \varphi_i']$ in χ'. This is in fact exactly what the proof of correctness of VAR-LET (Lemma 12.4.1) shows on graphic constraints: the possibility to instantiate the type of the let-bound expression twice does not increase the expressiveness of the constraint.

Nevertheless, we have seen in §15.2.6.1 that trivial type schemes, such as the ones for let-bound variables, result in identity computation in the elaboration of the principal presolution of a constraint. More precisely, this is the case as long as the trivial type scheme does not have a type different from the one of the variable itself (which is possible in non-principal presolutions). Thus, using VAR-LET to simplify the elaborated *x*ML$^{\mathsf{F}}$ term is only useful on presolutions that do not verify this property.

15.4.2.2 Lambda-bound variables

At least from an elaboration point of view, rule VAR-ABS is even less useful. Indeed, in *g*ML$^{\mathsf{F}}$, the gen node g for a λ-bound variable is always degenerate: the node $\langle g1 \rangle$ is unified with the node corresponding to the domain of the arrow for the abstraction, which is bound higher in the constraint. Thus, using VAR-ABS will only avoid the generation of a trivial computation ε for the variable, and this computation can trivially be removed anyway.

15.5 Translating presolutions of *e*ML$^{\mathsf{F}}$ and *i*ML$^{\mathsf{F}}$

From a practical standpoint, there is little interest in translating *e*ML$^{\mathsf{F}}$ or *i*ML$^{\mathsf{F}}$ presolutions, as type inference is done in *g*ML$^{\mathsf{F}}$. However, while *e*ML$^{\mathsf{F}}$ and *i*ML$^{\mathsf{F}}$ are quite different from *g*ML$^{\mathsf{F}}$, this is not reflected in the translation of their presolutions: there will be very few changes compared to the presentation done for *g*ML$^{\mathsf{F}}$. Moreover, the existence of such a translation can be used to prove the soundness of *e*ML$^{\mathsf{F}}$ and *i*ML$^{\mathsf{F}}$.

In this section, we use the same conventions as in §13. In particular, $\sqsubset$ *ranges over* $\sqsubseteq^{rmw}$ *and* $\sqsubseteq$*.*

15.5.1 Preliminary results

The instance relations $\sqsubseteq^{\approx}$ and $\sqsubseteq^{\boxminus}$ of *e*ML$^{\mathsf{F}}$ and *i*ML$^{\mathsf{F}}$ are larger than $\sqsubseteq$. In particular, while in *g*ML$^{\mathsf{F}}$ the only problematic operations of $\sqsubseteq$ were the ones on inert-locked nodes, in *e*ML$^{\mathsf{F}}$ and *i*ML$^{\mathsf{F}}$ we can also create flexible nodes by strengthening a rigid $\sqsubset$ node. We cannot reflect this strengthening in *x*ML$^{\mathsf{F}}$. However this is not a problem: by weakening enough nodes, we can ensure never needing a strengthening operation.

The definitions below generalize the notions of translatable presolutions and normalized derivations to eMLF and iMLF. In a second time we show that translatable presolutions lead to suitable propagation witnesses, and that normalized ones can be translated—exactly as in gMLF.

Definition 15.5.1 (***Translatable eMLF and iMLF presolutions)*** A presolution χ_p is translatable in eMLF (*resp.* iMLF) if it is translatable according to Definition 15.2.10, and if all the monomorphic (*resp.* inert or orange) nodes of χ_p are rigidly bound. ■

Definition 15.5.2 (***Normalized instance derivations in eMLF and iMLF)*** Let χ_p be an eMLF (*resp.* iMLF) presolution, e an instantiation edge of χ_p. A derivation of $\chi_p^e \sqsubseteq \chi_p$ is normalized if it is of the form $\chi_p^e \sqsubseteq \chi' \sqsupset \chi_p$ with the derivation $\chi_p^e \sqsubseteq \chi'$ normalized as per Definition 11.5.1, except that it can also contain arbitrary operations of $\sqsubseteq^{rmw}$ (*resp.* $\sqsubseteq$).■

The idea behind this definition is the following: instance operations inside a derivation $\chi_p^e \sqsubseteq \chi_p$ are not necessarily restricted to the nodes created by the expansion. Indeed, we can freely share and unshare the remainder of the constraint, but only along $\sqsubset$: any other instance operation could not possibly be reversed by $\sqsupset$.

Using those definitions, we can show that translatable presolutions exist, and are suitable for translation.

Lemma 15.5.3 *Given a translatable eMLF (*resp. *iMLF) presolution χ_p, and an instantiation edge e of χ_p, there exists a normalized instance derivation of $\chi_p^e \sqsubseteq \chi_p$ in which weakenings are delayed.* □

Proof: The decomposition of $\chi_p^e \sqsubseteq \chi_p$ into $\chi_p^e \sqsubseteq \chi' \sqsupset \chi_p$ is by Property 13.1.3. Inside the derivation $\chi_p^e \sqsubseteq \chi'$, the operations involving the nodes of χ_p (*i.e.* those that do not have been created by the expansion) can only be in $\sqsubset$. Indeed, $\sqsubset$ permissions are stable by instance, and an operation of $\sqsubseteq \setminus \sqsubset$ cannot be cancelled by $\sqsupset$ later in the derivation. The remainder of the result is proven exactly as Lemma 11.5.3.

Lemma 15.5.4 *Consider an $\sqsubset$-presolution χ, and n a node of χ flexibly bound on a gen node g that is not in $\mathcal{I}^s(g)$. Then* Weaken$(n)(\chi)$ *is an $\sqsubset$-presolution.* □

Proof: Let χ' be Weaken$(n)(\chi)$. If n has $\sqsubset$ permissions, the result is by Lemma 13.1.6. In the other case, consider an instantiation edge e of χ'. By hypothesis, it is $\sqsubset$-solved in χ. Consider a derivation of $\chi^e \sqsubseteq^{\sqsubset} \chi$. In this derivation, the only operations that can involve n or a $\sqsubset$-node transitively bound to n are of the form Merge(n, n'), with n' created by the expansion (**1**). Indeed

- ▷ an operation that is not a merging would alter n (or the node below) in a way that cannot be cancelled later, as the nodes are not $\sqsubset$ by hypothesis;
- ▷ a merging Merge(n', n'') with $n' \xrightarrow{+} n$ (and n' not $\sqsubset$) would mean that n'' has not been created by the expansion. Indeed, necessarily n'' is not $\sqsubset$ either (as n' and n'' are merged), and the root of the expansion is bound on a gen node; hence non-$\sqsubset$ nodes of the expansion can only "communicate" with the exterior of the expansion through a gen node. Thus, such a merging would involve two nodes not created by the expansion, and cannot be cancelled either.

By (1) it is easy to rewrite the derivation $\chi^e \sqsubseteq^{\Box} \chi$ into a derivation of $\chi'^e \sqsubseteq^{\Box} \chi'$: we simply replace all operations $\mathsf{Merge}(n, n')$ by $\mathsf{Weaken}(n')$; $\mathsf{Merge}(n, n')$. This shows that e is solved in χ', hence that χ' is a presolution.

Lemma 15.5.5 *Given an eMLF (*resp. *iMLF) presolution of a constraint χ, there exists a presolution χ'_p inert-equivalent to χ_p and translatable in eMLF (*resp. *iMLF).* □

Proof: Weakening the $\sqsubset$ nodes of χ_p results in a presolution by Lemma 13.1.6. Notice that application and abstraction arrow nodes are $\sqsubset$, as they were monomorphic in the typing constraint corresponding to χ_p, and $\sqsubset$ nodes are preserved by $\sqsubseteq^{\Box}$. For all the other nodes (*i.e.* those that may not be $\sqsubset$), the result is by Lemma 15.5.4.

Lemma 15.5.6 *Given a translatable eMLF (*resp. *iMLF) presolution χ_p, and an instantiation edge e of χ_p, let I be a normalized propagation witness for e in which weakenings are delayed. Then I does not transform inert-locked nodes, and does not transform a rigid node into a flexible one.* □

Proof: The reasoning for inert-locked is exactly the same as for Lemma 15.2.6. For the creation of flexible nodes: those nodes can be created only by an operation $\sqsupseteq^{w}$ on an $\sqsubset$ node, which occurs only at the end of a normalized derivation. It can only be followed by operations of $\sqsupseteq^{rmw}$, which preserve flexible $\sqsubset$ nodes. Hence there would be such a node in χ_p. Contradiction with the fact that χ_p is translatable.

15.5.2 Translating an *e*MLF or *i*MLF presolution

The elaboration of a translatable presolution is now immediate. The shape of the λ-term itself is exactly the same. The only difference lies in the translation of the propagation witnesses. But, even there, the differences are minimal: we can simply dismiss the operations that were not present in *g*MLF, as they occur on rigid nodes.

Definition 15.5.7 (*Translation of an eMLF or iMLF presolution*) The translation of a translatable *e*MLF or *i*MLF presolution χ_p is done as if χ_p was a *g*MLF presolution, except that, in propagations witnesses, the operations in $\sqsupseteq$ or on nodes of χ_p are translated by the identity computation. ■

Theorem 15.5.8 *The systems eMLF and iMLF are sound.* □

Proof: Immediate generalization of the results of §15.3, using the results of the previous section. The operations not translated only involve nodes that are rigidly bound, hence inlined during the translation into syntactic types. This ensures that our translation remains correct.

15.6 Translating the syntactic presentations of ML^F into *x*ML^F

In this section, iML^F and eML^F refer to the new syntactic presentation of ML^F (Le Botlan and Rémy 2007).

At a cursory glance, it would seem that translating either *i*ML^F or *e*ML^F into *x*ML^F is immediate, as their presentations are syntactic and very similar. However, this is not the case, for two unrelated reasons detailed below.

15.6.1 Type equivalence under bounds

Translating the type instance relation of *i*ML^F and *e*ML^F into *x*ML^F is not really difficult (except for the point developed in §15.6.2), as the rules in all three systems are quite similar. In fact, those relations are closer than the syntactic and graphic instance relations are, and we expect the translation to be slightly easier than the one we have presented here. This is however not the case for the type *equivalence* relation. Indeed, this relation is much smaller in *x*ML^F than in (syntactic) *e*ML^F and *i*ML^F—in *x*ML^F, up-to α-conversion, it is exactly the reflexive relation! In comparison, the relation $\sqsubseteq^{rmw}$ of graphic types is simpler to translate.

As an example, types such as $\forall\,(\alpha \geqslant \sigma)\ \alpha$ and σ, or types differing only by the commutation of two binders, are equivalent in *i*ML^F and *e*ML^F. That is, one type can be freely used instead of the other in *i*ML^F. By contrast, the transformation of one type into the other must be explicitly witnessed by a type computation in *x*ML^F. For most equivalence steps, we can find a witness computation. However, this is not the case for the following rules:

- Inert types, which are roughly all the types generated by the grammar $\sigma ::= \alpha \mid \sigma \to \sigma$ in the syntactic presentations, can be inlined inside types through the following rule

$$\text{iEq-Inert}\ \frac{\alpha \geqslant \sigma \in \Gamma \qquad \sigma \text{ is inert}}{\Gamma \vdash \sigma' \equiv \sigma'\{\alpha \leftarrow \sigma\}}$$

  This rule has no equivalent in *x*ML^F, and is not derivable.

- Type equivalence is congruent under type constructors, as per the following rule

$$\text{iEq-Con-Arrow}\ \frac{\Gamma \vdash \sigma_1 \equiv \sigma_1' \qquad \Gamma \vdash \sigma_2 \equiv \sigma_2'}{\Gamma \vdash \sigma_1 \to \sigma_1 \equiv \sigma_1' \to \sigma_2'}$$

  In *x*ML^F, this rule is not derivable either: transforming a type is possible only by changing bounds or inlining them, and we cannot modify a type under a type constructor such as an arrow (except by a computation $\&$, but applying such a computation is almost never reversible).

- In *e*ML^F, type equivalence can be performed under rigid bounds. This is problematic to deal with if rigid bounds are inlined during the translation, as we may again have to perform some transformations under type constructors.

Thus we need to show—and actually exhibit an effective translation on derivations—that, once types are normalized in a certain way, a term typable in *i*ML^F or *e*ML^F is also typable without using equivalence under rigid bounds or under type constructors, and without using rule iEq-Inert[1]. In essence, this is similar to considering only translatable presolutions on graphic types; on non-translatable presolutions, we weaken inert-locked nodes, so as to remove the need to translate an instance operation on them. However, we believe this normalization result will be more difficult to establish for the syntactic presentations, since the equivalence relation is larger.

We also expect that some of the other difficulties we encountered when translating graphic types will show up when translating the syntactic versions of $\mathsf{ML^F}$. Leijen (2007) has presented a translation of $\mathsf{ML^F}$ into System F, and there are many similarities between his work and ours. For example, he introduces a notion of canonical instance and abstraction, which prevents the commutation of some binders. Leijen also normalizes all the types into their normal syntactic form during the translation, in order to remove spurious occurrences of type equivalence.

15.6.2 Expressivity of alias bounds

Let us call *alias bound* a bound of the form $\forall\,(\beta \geqslant \alpha)\ \sigma$. Perhaps surprisingly, the types of $x\mathsf{ML^F}$ on the one hand, and of either $i\mathsf{ML^F}$ or $e\mathsf{ML^F}$ on the other hand, cannot be interpreted in exactly the same way, as the interpretation of alias bounds in those systems differ.

In both (Le Botlan 2004) and (Le Botlan and Rémy 2007), alias bounds can be inlined. For example, let us call σ_0 the type

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

It is equivalent in those two presentations to $\forall\,(\alpha \geqslant \sigma)\ \alpha \to \alpha$. In fact, the set of ground types into which σ_0 can be instantiated into is exactly

$$\{\sigma' \to \sigma' \mid \sigma \leq \sigma'\}$$

Alternatively, (in $i\mathsf{ML^F}$) we can write the type $\forall\,(\alpha \geqslant \sigma)\ \sigma \to \alpha$. However, the set of its instances is only

$$\{\sigma \to \sigma' \mid \sigma \leq \sigma'\}$$

In $x\mathsf{ML^F}$, the set of instances of σ_0 is larger, and at least a superset of

$$\{\sigma' \to \sigma'' \mid \sigma \leq \sigma' \leq \sigma''\}$$

as witnessed by the computations $\forall\,(\geqslant \varphi); \&; \forall\,(\geqslant \varphi'); \&$ (where φ and φ' verify $\vdash \varphi : \sigma \leq \sigma'$ and $\vdash \varphi' : \sigma' \leq \sigma''$).

This level of generality can be expressed neither in the syntactic nor in the graphic presentation of $\mathsf{ML^F}$.[2] In fact, it seems at first to endanger type soundness. In both (syntactic) $i\mathsf{ML^F}$ and $e\mathsf{ML^F}$, the identity function can be assigned the type $\forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha$ (which we call as usual σ_{id}) as well as the equivalent type

$$\sigma_1 \ \triangleq\ \forall\,(\alpha \geqslant \bot)\ \forall\,(\beta \geqslant \alpha)\ \forall\,(\gamma \geqslant \alpha)\ \beta \to \gamma$$

[1]At least in the right-to-left direction, as the left-to-right one can be partially simulated by the rule Inst-Quant-Elim of $x\mathsf{ML^F}$.

[2]In the graphic presentation, one could think to use an instantiation edge from the codomain of the arrow to the domain. However such an edge would be ill-sorted.

In xML^F, σ_1 can be instantiated into $\sigma' \to \sigma''$, where σ' and σ'' are arbitrary instances of $\bot$, which is certainly not a sound type for the identity function. Thankfully, the identity function cannot receive the type σ_1 in xML^F. We can explain the differences as follows.

- Proving that σ_1 and σ_{id} are equivalent in iML^F and eML^F requires the equivalence rule iEq-Inert. As we mentioned in the previous section, this rule is not present—and not derivable—in xML^F. Type soundness is safe!

- Conversely, the instance rule Inst-Quant-Elim of xML^F is more general than the corresponding rule iIns-Subst of iML^F: the latter is restricted to the inlining of variables that are not *exposed*, which exactly prevents deriving $\vdash \sigma_1 \leq \forall (\alpha \geqslant \sigma)\ \forall (\beta \geqslant \sigma)\ \alpha \to \beta$ (Le Botlan and Rémy 2007).

Thus, by restricting iEq-Inert and extending iIns-Subst, we have reached another point in the design space for the syntactic presentations of ML^F. It is not yet entirely clear if one presentation is superior to the other.

In summary, special care must be taken when translating syntactic iML^F and eML^F into xML^F. A simple solution is to entirely inline all inert bounds (which include alias ones). Another possibility would be to forbid alias bounds entirely. This is the direction followed by our graphic presentation of ML^F, where they cannot be expressed at all. In parallel, a preliminary version of the HML system (Leijen 2009) also inlined inert bounds. However, this approach does not seem so natural in the Church-style version of ML^F, and it would have severely complicated the presentation of xML^F. Hence, we chose to stick to the current system.

Part

IV

Conclusions

16

Related works

Abstract

We summarize related works. §16.1 discusses other proposals aiming at performing type inference in presence of second-order polymorphism. §16.2 summarizes various efficient approaches for ML type inference, including the use of constraints, and link them to our $\mathsf{ML^F}$ type inference algorithm. §16.3 briefly compares $x\mathsf{ML^F}$ with some other fully-explicit languages.

16.1 Type inference and second-order polymorphism

$\mathsf{ML^F}$ in general, and this work in particular, continues a long line of research aimed at enabling partial type inference in languages with second-order polymorphism. A very thorough comparison between those works and $\mathsf{ML^F}$ has been given by Le Botlan and Rémy (2007), and also applies to the version of $\mathsf{ML^F}$ presented in this document. We summarize the must relevant works next. A comparison between the various flavours of $\mathsf{ML^F}$ is given in Appendix A.

Type containment. We have briefly discussed the system $\mathsf{F}\eta$ of Mitchell (1988) in §3.4. Mitchell had noticed that System F might not be well suited for studying type inference, and proposed to extend the instance relation of System F so as to have more terms with principal types. While there are some similarities between $\mathsf{F}\eta$ and $\mathsf{ML^F}$, they are only superficial. In particular, the ability to instantiate a subtype in $\mathsf{F}\eta$ is entirely driven by the structure of the type, according to the variance of the arrow constructor, *i.e.* covariantly on the right of an arrow, and contravariantly on the left. This is unfortunately not sufficient to obtain principal types for all terms. Conversely, in $\mathsf{ML^F}$, type instantiation is *explicitly* enabled, through the use of flexible quantification. However, since $\mathsf{ML^F}$ does not allow instantiating variables in contravariant positions along $\sqsupseteq$ (unlike $\mathsf{F}\eta$), both systems are incomparable. Still ML^F is much better suited to perform type inference.

Type inference based on higher-order unification. Higher-order unification has been used to explore the practical effectiveness of type inference for System F by Pfenning (1988). Although known to be undecidable, higher-order unification is often tractable in practice.

In ML^F we have chosen the opposite approach: we strictly keep a first-order unification mechanism and never infer polymorphic types—we just propagate them. Interestingly, one proposal seems to require annotations exactly where the other can skip them: Pfenning's system requires placeholders (without type information) for type abstractions and type applications, but never needs type information on arguments of functions. Conversely, ML^F requires type information on (some) arguments of functions, but no information for type abstractions or applications.

Another important difference is the fact that Pfenning's work extends seamlessly to systems with higher-order polymorphism (such as $\mathsf{F}\omega$), while this question remains to be studied for ML^F.

Decidable fragments of System F. Several authors have studied fragments of System F for which type inference is decidable, in particular rank-2 polymorphic types (Kfoury and Wells 1994), called Λ_2, and rank-2 intersection types (Jim 1995), called I_2. (For $n \geq 3$, type inference in Λ_n or I_n is undecidable.) Interestingly, I_2 and Λ_2 type exactly the same programs. However, a severe limitations of both systems is that they are not compositional: because of the rank limitation, one may not abstract over arbitrary values of the language. As first-class polymorphism is precisely needed to introduce a higher level of abstraction, we think this is a fundamental limitation.

As presented by Kfoury and Wells (1994), type inference in Λ_2 is also quite involved. Indeed, it requires rewriting programs according to some non-intuitive set of reduction rules into acyclic semi-unification problems. Hence, no simple specification of well-typedness is provided to the user, and it is almost impossible to translate back an error during type inference in terms of the original typing problem. Worse, Λ_2 does not have principal types, and type inference can thus only be performed on full programs. An improved type inference algorithm for Λ_2 has recently been proposed by Lushman (2007). Lushman attempts to solve the non-modularity inherent to the lack of principal types by using types of F_3, the third-order λ-calculus. Unfortunately, his algorithm is incomplete. Noticeably, I_2 has better properties than Λ_2, including principal typings.

Intersection types and System E. Carlier *et al.* (2004) have proposed a type system, called System E, that generalizes intersection types with expansion variables. Their goal is quite different from ours, as they aim at subsuming all previous systems based on intersection types. Since systems with intersection types of arbitrary rank can type all strongly normalizing terms, their type inference algorithm is intrinsically incomplete.

There is at least one connection to be made before their work and ours. While we share several possible typing derivations for a (sub)term through the use of flexible quantification, they achieve a somewhat similar same goal via expansion variables. Nevertheless, there exists deep differences between the two mechanisms, as the two underlying systems are very different: expansion variables introduce intersection types, while flexible quantification is based on universal (bounded) quantification.

Local type inference. Local type inference (Pierce and Turner 2000) uses typing constraints to propagate type information locally—as opposed to the global propagation resulting from unification, as used in ML. More precisely, type information is propagated only between adjacent nodes in the syntax tree. Another key feature of local type inference is the fact that type information is propagated in a bidirectional fashion. That is, the type of an expression can be either *inferred* or *checked*. As an example, when an anonymous function appears as an argument to another function, the expected domain type is used as the expected type for the anonymous abstraction, allowing the type annotations on its parameters to be omitted. Local type inference has later been refined into *colored* local type inference (Odersky *et al.* 2001), in which partial type information may be propagated.

Local type inference is quite successful at leaving implicit many (but not all) eliminations of both subtyping and universal polymorphism. However, one drawback is that it is somewhat fragile, and does not support some simple program transformations. As an example, if $f\ x$ is typable, the application $apply\ f\ x$ may be untypable when f is polymorphic. Moreover, while many type annotations can be removed, a few of them remained necessary and sometimes in rather unpredictable ways (Hosoya and Pierce 1999). Moreover, local type inference is not a conservative extension of ML: some valid ML programs are not typable using only local type inference.

It should be noted that local type inference, colored or not, is natively able to handle subtyping, unlike most other proposal for type inference in presence of second-order polymorphism. Extending ML^{F} with subtyping remains to be explored.

Boxed first-class polymorphism refers to the encapsulation of first-class polymorphic values into ML (monomorphic) ones, through the use of injection and projection constructions. In their most basic version, injections and projections are explicit, even though, in practice, they can be attached to datatype constructors (Läufer and Odersky 1994; Rémy 1994). This however requires preliminary type definitions to be made for all polymorphic types that need to appear inside programs. Moreover, one must explicitly distinguish between boxed and unboxed values, which may be quite annoying in practice.

As an improvement, Garrigue and Rémy (1999) proposed Poly-ML. In this system, the projection from polytypes to monotypes only needs a placeholder: the actual type of the projection needs not be specified. In parallel, the injection of polymorphic values into monomorphic ones uses one unified construct for all the injections. This alleviates the need for prior type definitions.

While representing a significant progress, Poly-ML is not entirely satisfactory. First, the programmer still needs to distinguish between monomorphic and polymorphic values explicitly. Moreover, the type annotation which is needed when a polymorphic value is created is utterly redundant: the programmer has to write $\mathsf{let}\, v = e : \sigma$, while the compiler only *checks* that the type it infers for e is indeed σ.

(Partial) type inference for the predicative fragment of System F. Odersky and Läufer (1996) have extended boxed polymorphism to implicit predicative instantiation of rank-2 polymorphism, which was later improved to arbitrary-rank types by Peyton Jones *et al.*

(2007). Technically, this approach mixes local type inference with ML-style, unification-based type inference. However, the restriction to predicative polymorphism is quite strong, and does not allow reaching all the expressivity of System F.

In parallel, Rémy (2005) studied a form of stratified type inference for System F, allowing explicit impredicative type-instantiation, and implicit predicative instantiation along the restriction of the containment relation of Mitchell (1988) to its predicative fragment. Since the instance relation $\sqsubseteq$ of ML^F does not subsume $\leqslant_{\mathsf{F}\eta}$, Remy's system and ML^F are incomparable w.r.t. to their expressivity. However, as noted by Rémy (2005), the fragment of $\leqslant_{\mathsf{F}\eta}$ not present in $\sqsubseteq$ (which concerns the instantiation of subtypes in contravariant positions according to $\geqslant_{\mathsf{F}\eta}$) is not really needed in practice.

16.1.1 More recent proposals

The last few years have seen the introduction of an important number of systems aiming at extending ML with the expressivity of System F polymorphism. We first briefly present those systems, then compare them with ML^F at the end of this section.

Boxy types have been proposed by Vytiniotis *et al.* (2006) as an extension to both boxed polymorphism and local type inference. Types include boxes, and the identity function can receive either the usual type $\forall\,(\alpha)\ \alpha \to \alpha$, or the boxed type $\boxed{\forall\,(\alpha)\ \alpha \to \alpha}$. Boxes essentially encode the inferred/checked duality of local type inference. That is, a judgment of the form

$$\Gamma \vdash t : \boxed{\mathsf{bool}} \to \mathsf{int}$$

checks that t has a type of the form *something* $\to$ int, and infers that *something* is bool. Seen in another way, boxy types go one step further than Poly-ML, by removing the projection from polytypes to monotypes of Poly-ML from the level of expressions, and retaining it only at the level of types.

Unfortunately, there is an obvious competition between the boxed form and the unboxed one. The typing rules of the system are presented in an algorithmic fashion, and resolve the competing cases in favor of one or the other view. The type system has principal types, but only with respect to this algorithmic specification. Also, as noted by their authors (Vytiniotis *et al.* 2008), boxy types often require programs to unbox the content of a box too early. As a result, many type annotations are required. To avoid them, boxy types have to be extended with ad-hoc heuristics such as N-ary applications. This complicates the system, and reduces its predictability.

Rigid ML^F is a restriction of ML^F that restricts the typechecking of let-bindings to use only F types (Leijen 2007). By also restricting type annotations to F types, all expressions can now be typed with ML^F types containing only rigid bindings (except for type variables themselves), *i.e.* F types.

One of the advantages of Rigid ML^F is the fact that a typable term can be elaborated very easily into System F—in fact, the system was introduced for this very reason. Unfortunately, Rigid ML^F looses principal types in the usual sense. (Technically it recovers them by using an *ad hoc*, non logical side condition in typing rules to rule out some otherwise correct typing derivations.) Moreover, while Rigid ML^F uses only F-types, its instance relation is based on the ML^F one, and imports all its complexity.

HML Leijen (2009) has identified a quite interesting variant of ML^F, called HML. His system does not use rigid bindings at all, only flexible ones. But, crucially, it still permits type inference. His proposal, inspired by the syntactic presentation of $i\mathsf{ML}^\mathsf{F}$ (Le Botlan and Rémy 2007), can be seen as another possible syntactic $e\mathsf{ML}^\mathsf{F}$.

Of course, dropping rigid quantification has a cost. Indeed, it sometimes becomes necessary to introduce type annotations during reduction. As an example, let us define ω as $\lambda(x : \sigma_{\mathsf{id}})\ x\ x$. While $(\lambda(x)\ \lambda(y)\ x\ y)\ \omega$ is typable in HML, its reduct $\lambda(y)\ \omega\ y$ is not. This example illustrates the loss of stability by both η-expansion and β-reduction. Nevertheless, HML remains a very interesting variant of ML^F: it preserves principal types and the logical flavor of typing rules, as well as interesting program transformation such as let-conversion. Moreover, compared to ML^F, the removal of rigid quantification is a real simplification.

HMF is a moderate extension of ML proposed by Leijen (2008), which still possesses the full expressivity of System F. Its attractiveness stems from its simple type inference algorithm, which is only a small extension of the ML one, and from its relatively simple specification. At the heart of HMF is an application rule which performs a form of local matching, which is used to determine impredicative instantiations. In order to ensure good properties to the system, this matching procedure imposes certain "minimality" conditions. In particular, the allowed impredicative instantiations are those that minimize the amount of polymorphism introduced. In its simplest form, HMF is quite sensitive to the order of arguments in applications. In order to remove some type annotations, and to make the system more robust to program transformations, the matching procedure for applications can be extended to an N-ary application rule.

Unfortunately, without flexible quantification, true principal types are lost. As in Rigid ML^F, they are technically recovered in HMF through non-logical side-conditions added to the typing rules, which favor ML-style polymorphism. This means that the system is not stable by, for example, the sharing of common subexpressions through let-bindings.

FPH is another proposal by Vytiniotis *et al.* (2008), that enriches Systems F types with boxes. However, the boxes of FPH are quite different from the one used in boxy types, as they are used in boxy types to keep track of impredicative instantiations. Boxes are removed when types are compared with user-supplied type annotations, or during the typing of applications. Conversely, they are prevented from entering the typing environment: let-bound expressions must have box-free types. Thus the system privileges ML types for those expressions, while the richer typings available in System F can be obtained through the use of type annotations.

The specification of FPH is quite simple, and leads to the following rule for the placement of type annotations: they are only needed on let-bindings and λ-abstractions with second-order types. Interestingly, type inference in FPH uses ML^F types, and represents boxes by flexible quantification. However, this is entirely hidden from the programmer, which only "sees" the high-level specification. (Unfortunately, it is not clear that type error messages encountered during inference can be explained using only boxed types.)

Finally, while FPH has a modular type inference algorithm, in the sense that typechecking of let-bound expressions can be done without looking at the various uses of the bound expression, it does not enjoy principal types. Indeed, even though flexible quantification is used for type inference, this is only done internally: flexible types cannot appear in the result of type inference. We think that the lack of principal types is a severe limitation in practice (although the ad-hoc form of principal types found in FPH and Rigid $\mathsf{ML^F}$ is not really satisfactory either). Moreover, this implies that FPH does not enjoy many of the modularity properties of $\mathsf{ML^F}$, as explained below.

Interestingly, in spite of their limitations when compared with $\mathsf{ML^F}$, FPH and HML show that rigid quantification is not strictly needed to perform $\mathsf{ML^F}$-style type inference. Rigid bindings however allow for better (more expressive) type inference, and for more robustness to program transformations.

A comparison between all these systems Let us conclude this section by a comparison between $\mathsf{ML^F}$ and the five systems presented above. None of those systems is superior to all the others on all points, and we compare them according to five criterion we believe are important in practice.

Simplicity of the specification

The specification of $\mathsf{ML^F}$ is complex. While mastering the system in its entirety is not needed to use it (for example the programmer does not really need to know about the similarity relation, or even the abstraction relation), $\mathsf{ML^F}$ remains significantly more complex than *e.g.* ML. Moreover, the form of bounded quantification used inside $\mathsf{ML^F}$ types is unusual. Using our syntactic sugar bounded quantification is often inlined. Nevertheless, it appears sometimes. We believe that the programmer needs to be aware of its existence to fully use the power of $\mathsf{ML^F}$, and to really understand the system.

In comparison, FPH and Rigid $\mathsf{ML^F}$ are much easier to present: they use the more familiar System F types, and their instance relations and typing rules are relatively simple. In fact, they have been designed from the ground-up to have a simple specification. Still, we believe that the large gap between the specification of FPH and its type inference algorithm will make very hard the generation of good error messages, as it will be difficult to explain them in terms of the high-level specification.

HML is also simpler than $\mathsf{ML^F}$, thanks to the removal of rigid quantification. It however remains more complicated than FPH or Rigid $\mathsf{ML^F}$.

Conversely, Rigid $\mathsf{ML^F}$ is actually (slightly) *more* complicated than $\mathsf{ML^F}$. Indeed, its typing rules are those of $\mathsf{ML^F}$, except the one for the let construct which requires the principal $\mathsf{ML^F}$ type of the bound expression to be extruded into a System F type.

Finally, the full specification of boxy types is also quite involved, as it needs many extensions to avoid the introduction of too many type annotations. The typing rules also have a strongly algorithmic flavor. One advantage over $\mathsf{ML^F}$ (w.r.t. simplicity) is the fact that boxy types are essentially F types.

Type annotations needed to reach all of System F

The various systems differ by the amount of type annotations required to make a System F program typable. Leaving aside boxy types temporarily, $\mathsf{ML^F}$ requires the least type annotations. In fact, it only needs type annotations on arguments of abstraction that are used polymorphically.[1]

Next probably comes HML: compared to ML^F, the removal of rigid quantification means that all abstractions using arguments with a second-order type must be annotated. This was not the case in ML^F, where $\lambda(x)\ f\ x$ is typable as soon as f itself is typable. HML does however not require annotations elsewhere.

FPH requires annotations on abstractions with second-order types, *and* on let-bindings with a second-order type. This specification is however conservative, and some of those annotations might be redundant. Rigid ML^F requires annotations on the same locations.

In comparison, HMF requires significantly more type annotations than the systems above. In fact, when translating a System F term into an HMF one, annotations are potentially required on the arguments and on the body of abstractions, and on the argument of applications.

Because of their algorithmic specification, it is difficult to compare boxy types with ML^F precisely. As they privilege propagation of type information from the function type to the argument type, they can type examples where ML^F would require an annotation. Conversely, there are many examples that ML^F can type and that boxy types cannot. We believe that besides the few biases of the algorithmic propagation, boxy types require significantly more annotations than ML^F.

Expressivity of the system with type annotations

An interesting point of comparison is to consider the set of terms typable in each system, using as much type annotations as needed. Through the use of flexible quantification, ML^F is actually (significantly) *more* expressive than System F. For example, the following ML^F program is not typable in System F, as it is not possible to give a correct type annotation for l

$$\lambda(l : \forall\,(\alpha \geqslant \sigma_{\mathsf{id}})\ \alpha\ \mathsf{list})\ (\mathsf{map}\ (\lambda(x : \sigma_{\mathsf{id}})\ (x\ 1, x\ 'c'))\ l, (\lambda(x)\ x+1) :: l)$$

Indeed, the first component of the pair requires l to have type σ_{id} list, while the second one forces it to have int list. As a consequence, the systems that have exactly the expressivity of System F (which we believe is the case of all the systems above but ML^F and HML) are strictly less expressive than ML^F.

Interestingly, even though it uses ML^F-style bounded quantification, HML is also less expressive than ML^F. This time however, this is because HML is based on the *shallow* version of ML^F, which is strictly less expressive than the full one. HML is however more expressive than the four other systems.

Complexity of the implementation

Type inference is significantly more complex to implement for ML^F than for ML, mainly because of the increased complexity in the type unification algorithm (though graphs

[1]This is different from saying that the argument is used with two different types. Consider indeed the expression $\lambda(x : \sigma_{\mathsf{id}})\ (\lambda(y : \forall\,(\alpha)\ \mathsf{unit} \to \alpha \to \alpha)\ (y\ ())\ (y\ ()))\ (\lambda()\ x)$; while x is only used once, the expression is not typable without the annotation on x.

somewhat reduce this complexity gap). Some of the systems discussed above incorporate $\mathsf{ML^F}$, either in their specification (Rigid $\mathsf{ML^F}$ and HML), or in the specification of their inference algorithm (FPH). For those systems, type inference is as involved as in $\mathsf{ML^F}$.[2]

Type inference for boxy types is also quite involved, and it is not clear it is really simpler than type inference in $\mathsf{ML^F}$, especially if graphs are used. However, since boxy types are essentially F types (up to boxes), replacing ML types by boxy ones inside an existing implementation is likely to be less invasive than replacing them by $\mathsf{ML^F}$ types.

Finally, compared to all the other systems, type inference in HMF is significantly simpler. This is unsurprising, as it was one of the design goals of this system.

Resilience to program transformation

$\mathsf{ML^F}$ is remarkably robust to small program transformations. In particular, programs are stable by η-expansion, η-reduction of unannotated abstractions, let-contraction, and β-contraction (with possibly an annotation on the argument of the abstraction which is introduced). Moreover, application can be redefined: if $f\ x$ is typable, so is *apply* $f\ x$ and *revapply* $x\ f$, where *apply* and *revapply* are respectively $\lambda(f)\ \lambda(x)\ f\ x$ and $\lambda(x)\ \lambda(f)\ f\ x$.

None of the systems discussed at the beginning of this section support all the transformations above. Except for boxy types, we believe that they are all stable by the redefinition of application through the use of *apply.* However, this is not the case for *revapply.* Indeed, this term can receive two incompatible types in System F, $\forall\alpha.\ \forall\beta.\ \alpha \to (\alpha \to \beta) \to \beta$ or $\forall\alpha.\ \alpha \to \forall\beta.\ (\alpha \to \beta) \to \beta$. Thus the systems that are based on F types must privilege one form or the other (making the system not stable by the use of *revapply*), or use some form of local matching for applications. Unfortunately, in this second case, the system is in general not stable by let-contraction: if $f\ x$ is typable, it is not always the case that let g = *revapply* x in $g\ f$ is.

let-contraction is actually a very good example, as it requires "true" principal types to hold. Thus we believe that this transformation is only valid in $\mathsf{ML^F}$ and HML, since all the other systems sometimes require type annotations on let-bound variables that have second-order types. Similarly, we believe that η-expansion preserves typability only in $\mathsf{ML^F}$. Indeed, when the function which is η-expanded has a polymorphic argument, it requires an annotation in all the systems but $\mathsf{ML^F}$. This is in particular the case in HML, where the lack of rigid quantification makes the annotation necessary.

Finally, we also believe that β-contraction is only possible in $\mathsf{ML^F}$. Indeed, it also requires "true" principal types. But it does not hold in HML, as this system is stratified: even though an expression can have a type σ, it is not always possible to write a function of type $\sigma \to \sigma'$.

The comparison above highlights the strengths of $\mathsf{ML^F}$: its expressiveness, the low amount of type annotations needed to type all of System F, and its remarkable robustness to program transformations. In fact, we believe that the only real drawback of the system is its complex

[2]In particular, much of the complexity in $\mathsf{ML^F}$ unification lies in maintaining proper scopes (through adequate raisings) for the bounded quantification used by $\mathsf{ML^F}$ types. Thus, the removal in HML of rigid quantification should not result in a major simplification of the type inference algorithm.

specification—the only other issue, namely the complexity of the type inference algorithm, is shared by essentially all the other systems.

Moreover, we strongly believe that flexible quantification is a key ingredient to perform type inference in presence of second-order polymorphism while retaining good theoretical properties (as shown by the discussion above). Thus it seems difficult to significantly reduce the complexity of the specification below the one of HML.

16.2 Type inference for $\mathsf{ML^F}$

16.2.1 Efficient type inference for ML

Efficient type inference algorithms for ML have many similarities with our graphic type inference algorithm. Of course, they all use a graph-based unification algorithm and reduce type schemes in an inner-outer fashion. They also use a notion of ranks (or frames) to keep track of generalization levels and perform generalization more efficiently (Rémy 1992; McAllester 2003; Pottier and Rémy 2005; Kuan and MacQueen 2007). Merging two multi-equations in (Rémy 1992) requires them to have the same rank, hence lowering their rank to the smallest of the two beforehand. Similarly, merging two nodes in graphic types requires them to have the same bound, hence raising them to their lowest common binder. Raising binding edges has also strong similarities with Rule S-Let-All of (Pottier and Rémy 2005).

16.2.2 Type inference using typing constraints

To the best of our knowledge Henglein has first expressed type inference as the satisfaction of type-inference constraints, which led him to semi-unification problems (Henglein 1993). Hence, the obvious similarity between our constraints and his. However, his constraints are interpreted over first-order types while ours are interpreted over graphic types, that are second-order. Our constraints are therefore more expressive. His constraints avoid the explicit representation of gen nodes, and instead read types as type schemes according to the context. We cannot make this simplification in $\mathsf{ML^F}$, because $\mathsf{ML^F}$ types are second-order, and expansion is more complicated in $\mathsf{ML^F}$ than in ML.

Typing constraints for ML have been explored in detail (Pottier and Rémy 2005). There are many similarities between this work and ours. Typing constraints are introduced first, independently of the underlying language; then a set of sound and complete transformations on typing constraints are introduced; the type inference algorithm is finally obtained by imposing a strategy on applications of constraint transformations. Moreover, some important steps of both frameworks can be put in correspondence (solving unification constraints, expansion of type-schemes, *etc.*). Our constraints are however more concise, for two reasons. Firstly, the graphic representation of types is more canonical; for instance, we need no rule for the commutation of adjacent binders. Secondly, the underlying binding structure of graphic types is reused for describing the binding constructs of graphic constraints. Hence, the representation of constraints requires fewer extension to the representation of types, as the latter is already richer.

Semi-unification As shown by Henglein (Henglein 1993), type inference for ML reduces to semi-unification problems that are trivially acyclic by construction—in the absence of polymorphic recursion. Hence, we could possibly see our constraints as encoding a form of acyclic graphic-type semi-unification problems.

16.3 Explicit languages

Extending $\mathsf{ML^F}$ with qualified types Leijen and Löh (2005) have studied the extension of $\mathsf{ML^F}$ with qualified types and, as a subcase, the translation of $\mathsf{ML^F}$ without qualified types into System F. However, since System F types cannot capture the full richness of $\mathsf{ML^F}$ types, an $\mathsf{ML^F}$ term of type $\forall\,(\alpha \diamond \sigma')\ \sigma$ has to be elaborated into a function of type

$$\forall\,(\alpha)\ (\sigma'_\star \to \alpha) \to \sigma_\star$$

when σ' is not $\bot$. In the type above, $\sigma_\star$ is a runtime representation of σ, and the first argument of the function is a *runtime coercion.* The translation to System F has been improved by Leijen (2007). In this last work, rigid bindings are inlined and do not give rise to coercion functions.

Regarding the translation into an explicitly-typed language, the most important difference between those works and ours (in the context of $x\mathsf{ML^F}$) lies in the fact that their coercions are at the level of terms, while our instantiations are at the level of types. In particular, although coercion functions should not change the semantics, this has not been proved so far. In our settings the type-erasure semantics comes for free by construction. The incidence of coercion functions in a call-by-value language with side effects is also unclear. We have not yet explored the use of $x\mathsf{ML^F}$ to accommodate qualified types, but expect no real difficulty in doing so. In fact, we believe the use of $x\mathsf{ML^F}$ type instantiations will permit a cleaner separation between evidence functions and type instantiations resulting from impredicative instantiations.

While Leijen (2007) and we translate $\mathsf{ML^F}$ partially annotated terms to quite different languages—System F and $x\mathsf{ML^F}$ respectively, our translation shares some ideas with his. For example, Leijen normalizes the $\mathsf{ML^F}$ typing derivations, so as to prevent unwanted commutation of binders which would create ill-typed terms. This is similar to what we did when translating presolutions, where we order the nodes of the presolutions according to $<_{\mathsf{P}}$. Likewise, Leijen normalizes all the types to their syntactic normal forms during the translation, to avoid translating some spurious (and useless) instances; this is reminiscent of our normalization of presolutions into translatable ones. Interestingly, although we identified the same issues when defining the translation, we chose quite different approaches to solve them.

Extending Church-style System F Sulzmann *et al.* (2007) have extended System F with *type equality coercions*, to accommodate some of the advanced features of the type system of the GHC compiler, in particular generalized algebraic datatypes (Xi *et al.* 2003; Jones *et al.* 2006). Type equality coercions are used to express that two types are (locally) equal, and to explicitly coerce one type into the other. The coercions are made explicit through witnesses that have some similarities with the computations we introduced in $x\mathsf{ML^F}$. System FC has also been designed to be a compiler intermediate language, which is one of the objectives we have pursued with $x\mathsf{ML^F}$.

17

Conclusion

17.1 Our work in the context of $\mathsf{ML^F}$

The metatheory of $\mathsf{ML^F}$ We have given a formal meaning to the graphs informally used in the original presentation of $\mathsf{ML^F}$ (Le Botlan 2004). Our definition of the instance relation in terms of graphic types is both simpler and more intuitive than the syntactic based presentations. Moreover, we entirely sidestep some artifacts of the syntactic definitions, such as the rule EQ-VAR of the equivalence relation, or the protected type abstraction relation of (Le Botlan and Rémy 2007).

We have found the graphic approach to be quite suitable to modifications; in fact, we have refined the instance relation several times during the course of this work. Our understanding of the design space is also much improved.

The system presented is this document is the full version of $\mathsf{ML^F}$ and is, to this date, more general than all the other versions. Compared to the original presentation (Le Botlan and Rémy 2003), our better understanding of the design space has allowed us to significantly extend the instance relation. Compared to the «recast» presentation of $\mathsf{ML^F}$ (Le Botlan and Rémy 2007), the instance relations coincide: the improvements to the instance relation we have found on graphic types have been transferred back to the syntactic presentation. However, this is only the case on the set of types common to both systems, and our types are strictly more general than the ones of that version. In particular, the recast version is stratified, and abstraction is not possible over all values: if σ is a valid type, it is not necessarily the case that $\sigma \to \sigma'$ is. No such restriction exists in our system.

Graphic constraints The graphic constraints we have presented are simpler than the syntactic ones that have been developed for ML; in particular they sidestep tedious issues such as α-renaming or commutations of binders. They are also simpler because the operation of extruding polymorphism, which is needed in ML constraints, is already present in graphic types as the raising operation. Hence $\mathsf{ML^F}$ constraints need fewer new constructs compared to ML constraints, as $\mathsf{ML^F}$ types are already richer than ML ones.

We have obtained a new, fully graphical presentation of ML^F, where both the specification and the type inference algorithm are done graphically. This presentation highlights the very strong ties between ML and ML^F. Our constraint-based approach also offers more freedom than a simple type inference algorithm, particularly in the resolution strategies.

We have also formally established the complexity bound for ML^F type inference. Our results are very similar to the ones obtained for ML. In particular, under reasonable assumptions, we have shown that type inference for ML^F has linear-time complexity.

Displaying ML^F types ML^F types generalize the types of System F, and the form of bounded quantification they feature can be a little intimidating to the user. We have proposed a simple form of syntactic sugar which, we believe, is quite efficient. In our experience the principal type of most terms can be displayed as a System F type.

An internal language for ML^F We have introduced a fully Church-style version of ML^F, suitable for use as an internal language in a typed compilation chain. This completes the Curry-style $i\mathsf{ML}^\mathsf{F}$ and the type-inference version $e\mathsf{ML}^\mathsf{F}$ (that requires partial type annotations but does not tell how to track them during reduction). Interestingly, $x\mathsf{ML}^\mathsf{F}$ is quite simple. It is also a strict extension of System F, and could be used as a drop-in replacement for an internal language based on that system.

We have also described a translation of well-typed $e\mathsf{ML}^\mathsf{F}$ programs into well-typed $x\mathsf{ML}^\mathsf{F}$ ones, that moreover preserves the type erasure of terms. This ensures in particular the type soundness of $e\mathsf{ML}^\mathsf{F}$, with either call-by-value or call-by-name semantics. This is the first time ML^F has been proven sound for call-by-name.

17.2 Applications beyond ML^F

Our work has some applications that go beyond ML^F, which we discuss below.

Graphic types Our representation of types using graphs can be used for other type systems than ML^F. In fact, we have already discussed in §3 how System F types could be represented. Since ML^F uses bounded quantification, our work extends to systems that have the same type of syntactic construction, such as $\mathsf{F}_\leq$ (Cardelli *et al.* 1994). Second-order types have a notion of well-scopedness whether they use bounded quantification or not. Our characterization of correct types as well-dominated graphs (Definition 4.3.3) should a priori be the same in all forms of graphic types.

A "low-level" instance relation Our decomposition of the instance modulo similarity relation $\sqsubseteq^\approx$ into an oriented relation $\sqsubseteq$ and a reversible relation $\sqsupseteq^{rmw}$ has many advantages. The relation $\sqsubseteq$ is quite simpler than $\sqsubseteq^\approx$ (in particular it is noetherian), making proofs more lightweight. Moreover, it brings theory and implementation closer. Indeed, we have studied and implemented a type inference algorithm for $g\mathsf{ML}^\mathsf{F}$, and formally shown that it is not more general than an $e\mathsf{ML}^\mathsf{F}$ one. This contrasts with the informal approach usually followed for ML or its variants: the system studied is «$e\mathsf{ML}$» (*i.e.* a syntactic representation of types, which works up to similarity), but the implementation uses term-graphs for efficiency reasons (hence is done for «gML»). Our approach shows that a fully-formal development is possible.

A framework for type inference It is easy to extend our constraints framework beyond ML and ML^F types. Provided unification on the types of the system under consideration is principal, it suffices to define an appropriate expansion operation, and to verify that eager propagation preserves presolutions (Lemma 11.4.2). Good candidates for this are the systems FPH (Vytiniotis *et al.* 2008) and HML (Leijen 2009), which are both partially based on ML^F.

17.3 Perspectives

There are several ways to extend our work, which we did not explore by lack of time. We comment on some of them below.

Beyond second-order types By simplifying ML^F and increasing our understanding of it, the graphic presentation permits exploring further extensions of ML^F with richer type structures, which can be already be found in some ML-like languages. This includes generalized algebraic datatypes (Xi *et al.* 2003; Jones *et al.* 2006; Pottier and Régis-Gianas 2006), recursive, higher-order or primitive existential types (Läufer and Odersky 1994), subtyping *etc.*

The combination of recursive types and second-order polymorphism alone is already tricky (Gauthier and Pottier 2004), and we expect the addition of recursive types to ML^F to be challenging. Allowing cyclic term-graphs in our types should be possible (even though we did not try to do so). The main difficulties likely lie in the treatment of recursion in the binding structure.

Probably harder, but also quite useful, would be to extend the mechanism of ML^F to higher-order types. The interaction of β-reduction at the level of types with a first-order type inference *à la ML^F* seems non-trivial. A possible solution would be to make all higher-order quantifications fully explicit.

The encoding of existential types into universal types behaves rather well in ML^F: the unpacking of existential types does not require type information but only the position of unpackings. It is thus 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 ML^F.

Strong normalization We expect terms typable in ML^F to be strongly normalizing; however this is currently only known for the shallow restriction of ML^F. A possible approach would be to prove this result first for xML^F. However, since type computations $\triangleright \sigma$ or $\alpha \triangleleft$ block reduction, such a result would only be preliminary. Alternatively, we could extend the semantics of ML^F types introduced by Le Botlan and Rémy (2007) for shallow types, and invoke the fact that System F is strongly normalizing.

Improving xML^F An obvious application of xML^F would be to implement qualified types (Jones 1994)—the demand for an internal language for ML^F was first made by some of the implementers of Haskell. This question remains to be investigated.

The translation of $e\mathsf{ML}^\mathsf{F}$ presolutions into $x\mathsf{ML}^\mathsf{F}$ terms is also involved. Creating the type computations coercing a type scheme into one of its instances on the fly during type inferences raises some difficulties. Moreover, the type computations we currently generate are complex, because of the differences between the instance relation $\leq$ of $x\mathsf{ML}^\mathsf{F}$ and $\sqsubseteq$ of $g\mathsf{ML}^\mathsf{F}$. Generating better computations is be possible, but the difficulty consists in doing so with a good complexity.

Also, while graphical type inference has been designed to keep maximal sharing of types during inference (so as to have good practical complexity) our elaboration implementation reads back dags as trees and undoes all the sharing carefully maintained during inference. A poor man's solution to overcome this would be to use hash-consing. Alternatively, a graphic presentation of $x\mathsf{ML}^\mathsf{F}$ could be devised.

Expressivity of the ML^F variants It was somewhat of a surprise to realize that $x\mathsf{ML}^\mathsf{F}$ types are actually more expressive than $i\mathsf{ML}^\mathsf{F}$ ones, because of a different interpretation of alias bounds (§15.6.2). While the interpretation of $x\mathsf{ML}^\mathsf{F}$ seems quite natural in an explicitly typed context, and is in fact similar to the interpretation of subtype bounds in $\mathsf{F}_{\leq}$, the interpretation of alias bounds in $i\mathsf{ML}^\mathsf{F}$ also seemed the obvious choice in the context of type inference. We have left for future work the exploration of the additional power brought by the $x\mathsf{ML}^\mathsf{F}$ interpretation, as well as the question of whether this additional power could be returned back to $e\mathsf{ML}^\mathsf{F}$ while retaining type inference.

Interestingly, this is linked to one possible improvement of our constraints framework. We cannot currently add an instantiation edge between two arbitrary type nodes, for various technical reasons. Lifting this restriction would allow accessing the expressivity currently only available in $x\mathsf{ML}^\mathsf{F}$.

Part V

Appendix

A

The flavours of $\mathsf{ML^F}$

A.1 The $\mathsf{ML^F}$ cube

Except for $x\mathsf{ML^F}$, presented in §14 in this document, the versions of $\mathsf{ML^F}$ that have been proposed so far are essentially Curry-style: terms are either entirely unannotated, or contain partial type annotations. All those systems share strong similarities. More generally, partially Curry-style $\mathsf{ML^F}$-based systems can be defined along three entirely orthogonal axis:

$e\mathsf{ML^F}$/$i\mathsf{ML^F}$: the systems called $e\mathsf{ML^F}$ form (by convention) the explicit versions of $\mathsf{ML^F}$: they require partial type annotations, but type inference is possible. Conversely, $i\mathsf{ML^F}$ is the implicit presentation of $\mathsf{ML^F}$, in which type annotations are unnecessary; as a counterpart, type inference is undecidable. $i\mathsf{ML^F}$ and $e\mathsf{ML^F}$ differ by their instance relation, the one of $i\mathsf{ML^F}$ being more general than the one of $e\mathsf{ML^F}$. In this document, we also introduce $g\mathsf{ML^F}$, whose instance relation is even more restricted than the one of $e\mathsf{ML^F}$ (but $e\mathsf{ML^F}$ and $g\mathsf{ML^F}$ have similar expressivity).

Full/Shallow: full $\mathsf{ML^F}$ systems use the entire set of $\mathsf{ML^F}$ types. Conversely, in shallow systems, types are stratified. More precisely, non-trivial flexible quantification cannot appear under rigid bounds. The shallow variants are slightly simpler to present, but have some drawbacks. In particular, polymorphism is second-order, but not first-class: given a term of type σ, it is not always possible to write a function of type $\sigma \to \sigma'$, because of the stratification in types.

Syntactic/graphic: the graphic version of $\mathsf{ML^F}$ is at the heart of this document. The syntactic and graphic versions of $\mathsf{ML^F}$ are quite different, but only in the way they are presented—at the core they describe the same systems, which are defined by the two points above.

On top of this, we can add two less important directions:

Original/improved instance relation: the original instance relation of $\mathsf{ML^F}$ was less general than the one presented in this document or in (Le Botlan and Rémy 2007).

Types with or without rigid quantification: when considering $i\mathsf{ML^F}$, the presentation can either include rigid quantification or inline rigid bounds. As for the choice between the syntactic or the graphic presentations, this is mainly a matter of style. However, inlining rigid edges usually results in a simpler presentation.

A.2 Existing variants

Not all the possible combinations above have been studied—there would be little point in doing so anyway. We describe below the existing variants.

The original presentation (Le Botlan 2004; Le Botlan and Rémy 2003)

This presentation is entirely syntactic, and introduced $\mathsf{ML^F}$. The instance relation is the original (weaker) one. Both $e\mathsf{ML^F}$ and $i\mathsf{ML^F}$, in full and shallow versions, are described; $i\mathsf{ML^F}$ is presented with rigid binders.

The «recast» presentation (Le Botlan and Rémy 2007)

This is an improved syntactic presentation of $\mathsf{ML^F}$, in which types are given a semantics in terms of sets of System F types. Both $i\mathsf{ML^F}$ (with inlined rigid quantification) and $e\mathsf{ML^F}$ are described, but only in their shallow versions—extending the semantics proposed in this document to non-shallow types remains to be done.

HML (Leijen 2009)

This is an alternative version of $e\mathsf{ML^F}$, slightly less general than the one of the «recast» presentation. As a counterpart to this loss of expressivity (HML requires more type annotations than $\mathsf{ML^F}$), HML does not use rigid quantification at all, which simplifies the presentation of the system. The types of HML are shallow.

The graphic presentation (this work)

This work describes both $e\mathsf{ML^F}$ and $i\mathsf{ML^F}$ with the new (improved) instance relation, and using graphs. Both systems use «full» types; hence this document presents the most expressive version of $\mathsf{ML^F}$ to date.[1] $i\mathsf{ML^F}$ is presented with rigid bounds; presenting graphic constraints with unbound nodes raises some challenges, hence our choice to leave rigid edges.

[1]In fact, no syntactic presentation corresponding to our work currently exists.

B
Syntactic $\mathsf{ML^F}$ relations

The equivalence, abstraction and instance relations of the original syntactic presentation of $\mathsf{ML^F}$ are presented in Figures B.1, B.2 B.3 respectively. Figure B.4 presents two important derived rules. We use the original notations of $\mathsf{ML^F}$ for the various relations.

Eq-Refl
$$(Q)\ \sigma \equiv \sigma$$

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

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

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

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

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

Eq-Var
$$(Q)\ \forall\,(\alpha \diamond \sigma)\ \alpha \equiv \sigma$$

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

Figure B.1 – Type Equivalence

$$\textsc{A-Equiv}\quad \frac{(Q)\ \sigma_1 \equiv \sigma_2}{(Q)\ \sigma_1 \sqsubseteq \sigma_2} \qquad \textsc{A-Hyp}\quad \frac{\alpha_1 = \sigma_1 \in Q}{(Q)\ \sigma_1 \sqsubseteq \alpha_1} \qquad \textsc{A-Trans}\quad \frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2 \qquad (Q)\ \sigma_2 \sqsubseteq \sigma_3}{(Q)\ \sigma_1 \sqsubseteq \sigma_3}$$

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

Figure B.2 – Type Abstraction

$$\textsc{I-Abstract}\quad \frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \sigma_1 \sqsubseteq \sigma_2} \qquad \textsc{I-Hyp}\quad \frac{\alpha_1 \geqslant \sigma_1 \in Q}{(Q)\ \sigma_1 \sqsubseteq \alpha_1} \qquad \textsc{I-Trans}\quad \frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2 \qquad (Q)\ \sigma_2 \sqsubseteq \sigma_3}{(Q)\ \sigma_1 \sqsubseteq \sigma_3}$$

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

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

Figure B.3 – Type Instance

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

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

Figure B.4 – Derived rules

Bibliography

BRIAN AYDEMIR, ARTHUR CHARGUÉRAUD, BENJAMIN C. PIERCE, RANDY POLLACK AND STEPHANIE WEIRICH. Engineering formal metatheory. In POPL'08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pages 3–15. ACM, New York, NY, USA, 2008a. ISBN 978-1-59593-689-9. DOI: `10.1145/1328438.1328443`.

BRIAN AYDEMIR, STEPHANIE WEIRICH AND STEVE ZDANCEWIC. Abstracting syntax, 2008b. Draft available from `http://www.cis.upenn.edu/~baydemir/`.

HENK P. BARENDREGT. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984. ISBN 0-444-86748-1.

LUCA CARDELLI, SIMONE MARTINI, JOHN C. MITCHELL AND ANDRE SCEDROV. An extension of system F with subtyping. *Information and Computation*, 109(1-2):4–56, 1994. ISSN 0890-5401. DOI: `10.1006/inco.1994.1013`.

SÉBASTIEN CARLIER, JEFF POLAKOW, J.B. WELLS AND ASSAF J. KFOURY. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In DAVID A. SCHMIDT, editor, Prooceedings of the 13th European Symposium on Programming., volume 2986 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2004. ISBN 978-3-540-21313-0. DOI: `10.1007/b96702`.
`http://www.macs.hw.ac.uk/~sebc/SystemE-short.pdf`

RICHARD COLE AND RAMESH HARIHARAN. Dynamic LCA queries on trees. *SIAM Journal on Computing*, 34(4):894–923, 2005. ISSN 0097-5397. DOI: `10.1137/S0097539700370539`.

THE COQ DEVELOPMENT TEAM. The Coq proof assistant reference manual, version 8.1. February 2007.
`http://coq.inria.fr/doc/`

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. DOI: `10.1145/582153.582176`.

Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. *Journal of Functional Programming*, 155:134–169, 1999. A preliminary version appeared in TACS'97.
`ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/iandc.ps.gz`

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. doi: `10.1145/1016850.1016872`.
`http://cristal.inria.fr/~fpottier/publis/gauthier-fpottier-icfp04.pdf`

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

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

Haruo Hosoya and Benjamin C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.
`http://repository.upenn.edu/cis_reports/180/`

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

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

Trevor Jim. Rank-2 type systems and recursive definitions. Technical Report MIT-LCS-TM-531, Massachusetts Institute of Technology, Laboratory for Computer Science, November 1995.
`http://www.research.att.com/~trevor/papers/ranktwo.ps.gz`

Mark P. Jones. A theory of qualified types. *Sci. Comput. Program.*, 22(3):231–256, 1994. doi: `10.1007/3-540-55253-7_17`.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich and Geoffrey Washburn. Simple unification-based type inference for GADTs. In ICFP'06: Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming, pages 50–61. ACM, New York, NY, USA, 2006. ISBN 1-59593-309-3. doi: `10.1145/1159803.1159811`.
`http://research.microsoft.com/~simonpj/papers/gadt/gadt-rigid-contexts.pdf`

Paris C. Kanellakis, Harry G. Mairson and John C. Mitchell. Unification and ML type reconstruction. In J.-L. Lassez and G. D. Plotkin, editors, Computational Logic: Essays in Honor of Alan Robinson, pages 444–478. MIT Press, 1991.
`http://www.cs.brandeis.edu/~mairson/Papers/KMM.ps.gz`

A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In Proceedings of the ACM Conference on Lisp and functional programming, pages 196–207. June 1994. doi: `10.1145/182590.182456`.

George Kuan and David MacQueen. Efficient type inference using ranked type variables. In ML'07: Proceedings of the 2007 workshop on ML, pages 3–14. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-676-9. doi: `10.1145/1292535.1292538`.

Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994. doi: `10.1145/186025.186031`.
`http://doi.acm.org/10.1145/186025.186031`

Didier Le Botlan. *MLF : An extension of ML with second-order polymorphism and implicit instantiation.* Ph.D. thesis, École Polytechnique, June 2004. English version.
`http://wwwdgeinew.insa-toulouse.fr/~lebotlan/`

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. doi: `10.1145/944705.944709`.
`http://gallium.inria.fr/~remy/work/mlf/icfp.pdf`

Didier Le Botlan and Didier Rémy. Recasting MLF. Research Report 6228, INRIA, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, June 2007.
`https://hal.inria.fr/inria-00156628`

Daan Leijen. A type directed translation of MLF to System F. In In Proceedings of the 2007 International Conference on Functional Programming (ICFP'07), ACM Press, October 2007. doi: `10.1145/1291220.1291169`.
`http://research.microsoft.com/users/daan/download/papers/mlftof.pdf`

Daan Leijen. HMF: Simple type inference for first-class polymorphism. In (ICFP'08). doi: `10.1145/1411203.1411245`. Extended version available as Microsoft Research technical report MSR-TR-2007-118, Sep 2007.

Daan Leijen. Flexible types: robust type inference for first-class polymorphism. In Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL'09), Savannah, Georgia, USA, January 2009. To appear.

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. ACM Press, New York, NY, USA, September 2005. ISBN 1-59593-064-7. doi: `10.1145/1090189.1086385`.
`http://murl.microsoft.com/users/daan/download/papers/qmlf.pdf`

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy and Jérôme Vouillon. The Objective Caml system, documentation and user's manual - release 3.10. Technical report, INRIA, May 2007. Documentation distributed with the Objective Caml system.
`http://caml.inria.fr/ocaml/htmlman/`

Bradley Lushman. *Direct and Expressive Type Inference for the Rank 2 Fragment of System F.* Ph.D. thesis, University of Waterloo, 2007.
`http://uwspace.uwaterloo.ca/handle/10012/3267`

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

David McAllester. A logical algorithm for ML type inference. In Rewriting Techniques and Applications, 14th International Conference (RTA 2003), volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer-Verlag, Valencia, Spain, June 2003.
`http://www.springerlink.com/content/auehenre84tcp3gb/`

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.
`http://www.diku.dk/undervisning/2006-2007/2006-2007_b2_246/milner78theory.pdf`

John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 2/3(76):211–249, 1988. doi: 10.1016/0890-5401(88)90009-0.

Martin Odersky and Konstantin Läufer. Putting type annotations to work. In Proceedings of the 23rd ACM Conference on Principles of Programming Languages, pages 54–67. January 1996. doi: 10.1145/237721.237729.
`http://lamp.epfl.ch/~odersky/papers/popl96.ps.gz`

Martin Odersky, Christoph Zenger and Matthias Zenger. Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41–53, March 2001. doi: 10.1145/373243.360207.
`http://lamp.epfl.ch/papers/clti-colored.ps.gz`

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

Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003. ISBN 0521826144.
`http://www.haskell.org/onlinereport/`

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, 2007. ISSN 0956-7968. doi: 10.1017/S0956796806006034.

Frank Pfenning. Partial polymorphic type inference and higher-order unification. In Proceedings of the ACM Conference on Lisp and Functional Programming, pages 153–163. ACM Press, July 1988. doi: 10.1145/62678.62697.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Massachusetts Institute of Technology Cambridge, Massachusetts 02142, 2002. ISBN 0-262-16209-1.
`http://www.cis.upenn.edu/~bcpierce/tapl/`

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000. doi: 10.1145/345099.345100.
`http://www.cis.upenn.edu/~bcpierce/papers/lti-toplas.pdf`

François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, Advanced Topics in Types and Programming Languages, chapter 10, pages 389–489. MIT Press, 2005.
`http://cristal.inria.fr/attapl/`

François Pottier. *Types et contraintes.* Mémoire d'habilitation à diriger des recherches, Université Paris 7, December 2004.
`http://cristal.inria.fr/~fpottier/publis/fpottier-hdr.pdf`

François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL'06), pages 232–244. Charleston, South Carolina, January 2006. doi: 10.1145/1111037.1111058.

Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992.
`ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/eq-theory-on-types.ps.gz`

Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, Theoretical Aspects of Computer Software, volume 789 of *Lecture Notes in Computer Science*, pages 321–346. Springer-Verlag, April 1994.
`ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/tacs94.ps.gz`

Didier Rémy. Simple, partial type-inference for System F based on type-containment. In Proceedings of the tenth International Conference on Functional Programming, pages 130–143. Tallinn, Estonia, September 2005. doi: 10.1145/1090189.1086383.
`http://gallium.inria.fr/~remy/work/fml/fml-icfp.pdf`

Didier Rémy and Boris Yakobowski. A graphical presentation of MLF types with a linear-time unification algorithm. In Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI'07), pages 27–38. ACM Press, Nice, France, January 2007. ISBN 1-59593-393-X. doi: 10.1145/1190315.1190321.
`http://www.yakobowski.org/tldi07.html`

Didier Rémy and Boris Yakobowski. From ML to MLF: Graphic type constraints with efficient type inference. In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08), Victoria, British Columbia, Canada, pages 63–74. ACM Press, September 2008. doi: 10.1145/1411203.1411216.
`http://www.yakobowski.org/icfp08.html`

John C. Reynolds. Towards a theory of type structure. In Proc. Colloque sur la Programmation, pages 408–425. Springer-Verlag LNCS 19, New York, 1974.
`http://www.springerlink.com/index/p5801737k78207p7.pdf`

Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones and Kevin Donnelly. System F with type equality coercions. In Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007, pages 53–66. ACM, 2007. ISBN 1-59593-393-X. doi: 10.1145/1190315.1190324.

Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008. ISSN 0168-7433. doi: 10.1007/s10817-008-9097-2.

Dimitrios Vytiniotis, Stephanie Weirich and Simon Peyton Jones. FPH: First-class Polymorphism for Haskell. In (ICFP'08). doi: `10.1145/1411203.1411246`.
`http://www.seas.upenn.edu/~sweirich/papers/icfp08.pdf`

Dimitrios Vytiniotis, Stephanie Weirich and Simon Peyton Jones. Boxy types: inference for higher-rank types and impredicativity. In ICFP'06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming, pages 251–262. ACM Press, New York, NY, USA, 2006. ISBN 1-59593-309-3. doi: `10.1145/1160074.1159838`.
`http://www.cis.upenn.edu/~dimitriv/boxy/boxy.pdf`

Joe B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS), pages 176–185. 1994.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 1994. doi: `10.1006/inco.1994.1093`.
`http://www.cs.rice.edu/CS/PLT/Publications/Scheme/ic94-wf.ps.gz`

Hongwei Xi, Chiyan Chen and Gang Chen. Guarded recursive datatype constructors. *SIGPLAN Not.*, 38(1):224–235, 2003. ISSN 0362-1340. doi: `10.1145/640128.604150`.

Boris Yakobowski. Le caractère ‘ à la rescousse - factorisation et réutilisation de code grâce aux variants polymorphes. In JFLA 2008 - Dix-neuvièmes Journées Francophones des Langages Applicatifs, pages 63–77. INRIA, Étretat, January 2008. ISBN 2-7261-1295-11.
`http://www.yakobowski.org/jfla08.html`

Index of definitions

Résumé

$\mathsf{ML^F}$ est un système de types combinant le polymorphisme implicite de seconde classe de ML avec le polymorphisme de première classe mais explicite du Système F. Nous proposons une représentation des types de $\mathsf{ML^F}$ qui superpose un graphe acyclique orienté du premier ordre (encodant la structure du type avec partage) et un arbre inversé (encodant la structure de lieurs du type). Cela permet une définition simple et directe de l'instance sur les types, qui se décompose en une instance sur la structure du type, des opérations simples sur l'arbre de lieurs, et un contrôle acceptant ou rejetant ces opérations. En utilisant cette représentation, nous présentons un algorithme d'unification sur les types de $\mathsf{ML^F}$ ayant une complexité linéaire.

Nous étendons ensuite les types graphiques en un système de contraintes graphiques permettant l'inférence de types à la fois pour ML et $\mathsf{ML^F}$. Nous proposons quelques transformations préservant la sémantique de ces contraintes, et donnons une stratégie pour utiliser ces transformations afin de résoudre les contraintes de typage. Nous montrons que l'algorithme résultant a une complexité optimale pour l'inférence de types dans $\mathsf{ML^F}$, et que, comme pour ML, cette complexité est linéaire sous des hypothèses raisonnables.

Enfin, nous présentons une version à la Church de $\mathsf{ML^F}$, appelée $x\mathsf{ML^F}$, dans laquelle tous les paramètres de fonctions, toutes les abstractions de type et toutes les instantiations de types sont explicites. Nous donnons des règles de réduction pour réduire les instantiations de types. Le système obtenu est confluent lorsque la réduction forte est autorisée, et vérifie la propriété de réduction du sujet. Nous montrons aussi le lemme de progression pour des stratégies faibles de réduction, dont l'appel par nom et l'appel par valeur en restreignant ou non le polymorphisme aux valeurs. Nous proposons un encodage de $\mathsf{ML^F}$ dans $x\mathsf{ML^F}$ qui préserve les types, ce qui assure la sureté de $\mathsf{ML^F}$.

Abstract

$\mathsf{ML^F}$ is a type system that seamlessly merges ML-style implicit but second-class polymorphism with System-F explicit first-class polymorphism. We propose a dag representation of $\mathsf{ML^F}$ types that superimposes a first-order term-dag, encoding the underlying term structure (with sharing), and a binding tree encoding the binding structure. This permits a simple and direct definition of type instance, that combines type instance on 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 $\mathsf{ML^F}$ types.

We then extend graphic types into a system of graphic constraints that can be used to perform type inference in both ML or $\mathsf{ML^F}$. We give a few semantic preserving transformations on constraints, and propose a strategy for applying those transformations to solve typing constraints. We show that the resulting algorithm has optimal complexity for $\mathsf{ML^F}$ type inference, and that, as for ML, this complexity is linear under reasonable assumptions.

We finally present a church-style version $x\mathsf{ML^F}$ of $\mathsf{ML^F}$, in which all parameters of functions, all type abstractions, and all type instantiations are explicit. We give a set of reduction rules for simplifying type instantiations. The resulting system is confluent when strong reduction is allowed, and enjoys the subject reduction property. We also show progress for weak-reduction strategies, including call-by-name and call-by-value, with or without the value restriction. We exhibit a type-preserving encoding of $\mathsf{ML^F}$ into $x\mathsf{ML^F}$, ensuring the type soundness of $\mathsf{ML^F}$.