

Recasting MLF

Didier Le Botlan, Didier Rémy

▶ **To cite this version:**

Didier Le Botlan, Didier Rémy. Recasting MLF. [Research Report] RR-6228, 2007, pp.60. inria-00156628v3

HAL Id: inria-00156628
https://inria.hal.science/inria-00156628v3

Submitted on 10 Jul 2007 (v3), last revised 3 Dec 2008 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Recasting ML^{F}

Didier Le Botlan and Didier Rémy

N° 6228

June 2007

Thème SYM



ISSN 0249-6399 ISRN INRIA/RR--6228--FR+ENG

INRIA
ROCQUENCOURT

Recasting ML^F

Didier Le Botlan and Didier Rémy

Thème SYM — Systèmes symboliques
Projet Gallium

Rapport de recherche n° 6228 — June 2007 — 59 pages

Abstract: The language ML^F has been proposed as an alternative to System F that permits partial type inference *a la* ML. It differs from System F by its types and type-instance relation. Unfortunately, the definition of type instance is only syntactic, and not underpined by some underlying semantics. It has so far only been justified a posteriori by the type soundness result.

In this work, we revisit ML^F following a more progressive approach stepping on System F. We argue that System F is not a well-suited language for ML-style type inference because it fails to factorize some closely related typing derivations. We solve this problem in Curry's style ML^F by enriching types with a new form of quantification that may represent a whole collection of System F types. This permits a natural interpretation of ML^F types as sets of System-F types and pulling back the instance relation from set inclusion on the interpretations. We also give an equivalent syntactic definition of the type-instance, presented as a set of inference rules.

We derive a Church's style version of ML^F by further refining types so as to distinguish between user-provided and inferred type information. The resulting language is more canonical than the one originally proposed. We show an embedding of ML in ML^F and an encoding of System F into ML^F. Besides, as ML^F is more expressive than System F, an encoding of ML^F is given towards an *extension* of System F with local binders.

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

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

Recasting ML^F

Résumé : Le langage ML^F a été conçu comme une alternative au Système F permettant l'inférence partielle de type à la ML. Il diffère du Système F par la nature de ses types et par sa relation d'instance sur les types. Malheureusement, la définition de l'instance de type est purement syntaxique et n'est pas soutenue par une sémantique sous-jacente. Jusqu'à présent elle n'a été justifiée qu'a posteriori par le résultat de correction.

Dans ce travail nous réexaminons ML^F en suivant une approche plus progressive en s'appuyant sur le Système F. Nous soutenons que le Système F n'est pas bien adapté pour l'inférence de type à la ML parce qu'il ne réussit pas à factoriser des dérivations de typage étroitement reliées. Nous résolvons ce problème dans la présentation de ML^F à la Curry en enrichissant les types avec une nouvelle forme de quantification qui permet de représenter une collection tout entière de types du système F. Cela conduit à une interprétation naturelle des types de ML^F comme des ensembles de types du système F et à en déduire la relation d'instance par inclusion des interprétations. Nous donnons également une définition syntaxique équivalence de l'instance de type sous la forme d'un ensemble de règles d'inférences.

Nous en dérivons une présentation à la Church de ML^F en raffinant encore les types de façon à distinguer l'information de type inférée de celle fournie par l'utilisateur. Le langage résultant est plus canonique que celui initialement proposé. Nous montrons un plongement de ML dans ML^F ainsi qu'un codage du Système F dans ML^F. De plus, comme ML^F est plus expressif que le Système F, un codage de ML^F est fourni vers une *extension* du Système F avec des liaisons locales.

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

Contents

1 Introduction **3**

2 An intuitive introduction to ML^F **8**
2.1 A Generic Curry's style second-order type system 8
2.2 Curry's style ML^F. 12
2.3 Church's style ML^F. 13
2.4 F^{let}, the closure of System F with let-contraction 16

3 IML^F, Curry's style ML^F **17**
3.1 Types and Prefixes 18
3.2 Interpretation of types and prefixes 18
3.3 Syntactic versions of instance and equivalence 19
3.4 Typing rules 22
3.5 System F as a subset of (Simple) IML^F 23
3.6 Type soundness, by viewing IML^F as a subset of F^{let} 23
3.7 Expressiveness and modularity 24

4 XML^F, Church's style ML^F **26**
4.1 Types, prefixes and relations under prefix 26
4.2 Typing rules and type soundness 30
4.3 Translating System F into XML^F 32
4.4 Embedding ML into XML^F 37
4.5 Programs that we intendedly reject 38

5 Related works **38**
5.1 Type inference for System F 39
5.2 Embedding first-class polymorphism in ML 41
5.3 Comparison with (other presentations of) ML^F 43

6 Conclusion and future work **43**

A Proofs **44**

1 Introduction

The design of programming languages is often an area of difficult compromises. In the end, programming languages must help programmers write correct, maintainable and reusable programs quickly. This implies, in particular, that programming languages must be *expressive*, so as to write algorithms concisely and directly, avoiding code duplication and acrobatic programming patterns, *modular*, so as to ease code reuse and be robust to small code changes, and obviously *typed*. Types are indeed a key towards program correctness, as they ensure, with very little overhead and without relying on the programmer's skill, that a certain class of errors will never occur at runtime; moreover most common and stupid programming mistakes may so be trapped as type errors.

Simple types, and in particular ground types, are still in use in many languages such as Pascal, C, Java, *etc.* to categorize values between basic values such as integers, strings, *etc.*, named structured values, and functions. Most basic values and primitive function have a unique ground type. However, higher-order functions, which receive among their arguments another function typically used to transform other arguments, are usually polymorphic. That is, they work uniformly for a whole collection of ground types. Unfortunately, this property cannot be described using simple types. As a result, higher-order functions must often be artificially specialized to one or, worse, several instances, introducing duplication, worsening maintainability, and preventing code reuse.

A well-known solution is of course *parametric polymorphism*, which extends simple types with type variables that may be universally quantified. The simplest form of parametric polymorphism is known as *ML-style* polymorphism [Mil78]: quantifiers are limited to the outermost position of types and the use of polymorphism is limited to definitions, as opposed to parameters of functions. For instance, the application—the function that takes two arguments and apply the first one to the second one—may be given type $\forall\,(\alpha,\beta)\ (\alpha \to \beta) \to (\alpha \to \beta)$.

A whole collection of similar types obtained by instantiating universal variables by arbitrary types can thus be captured as a single polymorphic type. In this view, types can be thought of as the set of all their instances, which makes them about as easy to understand as simple types. This also eases type inference, as each well-typed program may be characterized by a *principal type*, *i.e.* one of which all others are instances. ML-style polymorphism has been extremely successful for several decades and is still at the core of the most widely used implementations of statically typed functional languages, such as OCaml or Haskell. The advantages of parametric polymorphism over simple types are now widely recognized, as shown by its introduction in the Java language—version 5—and C#.

However, polymorphic higher-order functions soon or later need themselves to be passed as values to other functions. This requires *first-class polymorphism*, that is, to allow quantifiers to appear at any position within types and, more generally, to treat polymorphic types as any other types. System F, also called the second-order polymorphic λ-calculus is the reference for first-class polymorphism [Gir72, Rey74].

In Curry's view, terms of System F are unannotated and types are left implicit, as in ML. However, as opposed to ML, type inference for System F is undecidable [Wel99] and does not have principal types. Thus, in practice, one rather uses Church's view, where source programs contain explicit type information, so as to make type checking decidable and straightforward. More precisely, function arguments come with explicit *type annotations* and polymorphism is both explicitly introduced by *type abstractions* and explicitly eliminated by *type applications*.

For example, the following function maps its argument—a list of pairs—to the list obtained by applying the first projection of each pair to its second projection. We assume given listmap whose type is $\forall\,(\alpha\beta)\ (\alpha \to \beta) \to list(\alpha) \to list(\beta)$ as well as the projections fst and snd from pairs with respective types $\forall\,(\alpha\beta)\ (\alpha \times \beta) \to \alpha$ and $\forall\,(\alpha\beta)\ (\alpha \times \beta) \to \beta$.

```
Λ(α) Λ(β) λ(pairlist : list((α → β) × α))
  listmap [(α → β) × α] [β]
    (λ(p : (α → β) × α) (fst [α → β] [α] p) (snd [α → β] [α] p))
    pairlist
```

As illustrated by this example, explicit type information may be quite intrusive sometimes obfuscating and often a pain to write and read, while most of this information is rather obvious from context. By contrast, the very same example, written in ML, looks as follows:

```
λ( pairlist ) listmap (λ(p) (fst p) (snd p)) pairlist
```

Annotations in ML are unnecessary because the language enjoys full type inference. Polymorphism is implicitly introduced and eliminated. This is made possible in ML because type inference does not have to look for first-class polymorphic types, hence it never needs to *guess* polymorphism. On the opposite, type inference in System F should also consider solutions where the argument of a function, *e.g.* p in the example above, may have a polymorphic type, such as $(\forall\,(\alpha)\ (\alpha \times \alpha) \to list(\alpha)) \times (\forall\,(\beta)\ \tau \times \tau)$, where τ is a type that may mention β. Finding all such solutions is undecidable in the general case. Hence, in System F, one must also annotate programs that are typable in ML if only to say not to look for other types. While System F is attractive for its expressiveness, its poor treatment of the common simple case has limited its use as the core of a programming language, which indirectly benefited to the long life of ML dialects.

Searching for the grail. In the last two decades, considerable research has been carried out to reduce the gap between ML and System F. Unfortunately, all solutions that have been proposed so far are unsatisfactory, from both a theoretical and—worse—practical point of view. The problem has naturally been tackled from two opposite directions.

Starting with System F, one may allow some type annotations to be omitted and attempt to rebuild them, hopefully accepting more ML programs. One theoretically very attractive solution of this kind is to reduce type inference to second-order unification [Pfe88]. This approach does not need type annotations at all, but still requires placeholders for type abstractions and type applications, which unfortunately, are not very convenient to write. Furthermore, type inference remains undecidable, as it then amounts to second-order unification. Other less ambitious approaches rely on *local* type information to rebuild omitted type information [PT98, OZZ01], by contrast with unification which is based on *global* computational effects. However, places where the user must provide type annotations are not always so obvious [HP99a]. Worse, these approaches appear to be fragile with respect to small program transformations; moreover, all of them fail to type a significant subset of useful ML programs.

Conversely, starting with ML, polymorphic types can be embedded into first-class monomorphic types via explicit injection and projection functions. This technique, known as *boxed polymorphism* [Rém94, OL96]

may be improved in several ways and has been implemented in both Haskell and ML. While it is useful as a default solution, and acceptable when polymorphic types are used occasionally, the solution does not scale up to intensive use of polymorphism.

In fact, all approaches have somehow assumed that the solution was to be found between ML and System F. Indeed, in Church's view, hence in theory, ML is a subset of System F. However, in practice, ML is *implicitly* typed while System F is *explicitly* typed. This makes the previous comparison misleading—if not meaningless. Maybe, the solution lies outside of System F, as a more expressive type-system that combines implicit let-polymorphism with explicit second-order types.

Our main contribution is a proposal for a new type system, called ML$^\mathsf{F}$, that supersedes both ML and System F, allows for simple, efficient, predictable, and complete type inference for partially annotated terms. The language ML$^\mathsf{F}$ has been introduced in previous work [LBR03, LB04], which we shall below refer to as Full ML$^\mathsf{F}$ to avoid ambiguity. In this work, we focus on a simplified version, here called ML$^\mathsf{F}$ for conciseness—or Plain ML$^\mathsf{F}$ when there is ambiguity[1]. While Plain ML$^\mathsf{F}$ is less expressive than Full ML$^\mathsf{F}$, it is still more expressive than both ML and System F, it retains interesting theoretical properties and practical applications, and it has a significantly less technical and more intuitive presentation.

For another simplification, we first consider a Curry's style version of ML$^\mathsf{F}$, called[2] IML$^\mathsf{F}$. This implicitly typed version requires an extension of System F types with only *flexible quantification*, written $\forall\,(\alpha \geq \sigma)\ \sigma'$. Remarkably, we may interpret types of IML$^\mathsf{F}$ as sets of System-F types. Roughly, $\forall\,(\alpha \geq \sigma)\ \sigma'$ may be seen as the collection of all System-F types $\tau'[\tau/\alpha]$ where τ' and τ range in the interpretation of σ' and σ, respectively. This interpretation induces an instance relation on types. It can also be used to exhibit a translation of expressions into a small extension of System F with local bindings. This shows that the additional expressive power of ML$^\mathsf{F}$ is theoretically small, although practically important as it modularity in an essential way. For sake of comparison, one may also consider a weaker version, called[3] Simple ML$^\mathsf{F}$, that has exactly the same expressiveness as System F. Although it retains most theoretical properties of ML$^\mathsf{F}$, it is no longer an extension of ML and, as a result, has little practical interest.

Unsurprisingly, full type inference for IML$^\mathsf{F}$ is undecidable. We thus also devise a Church's style version of ML$^\mathsf{F}$, called[4] XML$^\mathsf{F}$, with optional type annotations. For the purpose of type inference we enrich types with *rigid quantification* $\forall\,(\alpha \Rightarrow \sigma)\ \sigma'$, which, in contrast with flexible bindings, indicates that the polymorphic type σ cannot be instantiated. Rigid quantification may at first be viewed as a notation for representing inner polymorphism $\sigma'[\sigma/\alpha]$ within a form of type schemes $\forall\,(\alpha \Rightarrow \sigma)\ \sigma'$. More importantly, it keeps track of sharing by distinguishing between $\forall\,(\alpha{\Rightarrow}\sigma, \alpha'{\Rightarrow}\sigma)\ \sigma'$ and $\forall\,(\alpha{\Rightarrow}\sigma)\ \sigma'[\alpha/\alpha']$. This distinction is essential for the purpose of type inference, as guessed polymorphism must be shared while explicit polymorphism need not be shared. For instance, the identity function $\lambda(x)\ x$ may be typed with $\forall\,(\alpha{\Rightarrow}\sigma)\ \alpha \to \alpha$ but not with $\forall\,(\alpha{\Rightarrow}\sigma, \alpha{\Rightarrow}\sigma')\ \alpha \to \alpha$, while $\lambda(x : \sigma)\ x$, with an explicit type annotation σ on the parameter, may be assigned both types—the latter being more general.

The problem of type inference with *partial* type annotations is to find, for a program given with some type annotations, the set of all types of the program that respect its type annotations. In ML$^\mathsf{F}$, solvable type inference problems have principal solutions, which depends on the program type annotations, indeed—as illustrated by the two versions of the identity function just above.

The richer types and let-polymorphism of ML$^\mathsf{F}$ make it significantly superior to System F as a programming language: programs admit more general types and require fewer type annotations. More precisely, removing all type abstractions and type applications from a term of System F, leaving only type annotations on function parameters produces a term that is well-typed in ML$^\mathsf{F}$ and with a more general type than its original type in System F—modulo a straightforward translation of types. Moreover, annotations on function parameters may often be omitted. Precisely, only those that are *used polymorphically* need an annotation. In particular, ML programs do not need type annotations at all.

Although type inference is not addressed in this paper, it has been shown in some previous work that it can be reduced to a simple first-order unification algorithm for (a form of) second-order types, combined with let-polymorphism *a la* ML [LBR03, LB04]. While worst-case complexity is at least as hard as in ML, *i.e.* exponential-time complete, it seems to be quite tractable.

[1] Plain ML$^\mathsf{F}$ was called Shallow ML$^\mathsf{F}$ in [LBR03, LB04].
[2] The I stands for "implicit".
[3] Simple ML$^\mathsf{F}$ was called Restricted ML$^\mathsf{F}$ in [LBR03, LB04].
[4] The X stands for "explicit".

The full picture. Of course, something must have been lost while going from Full ML^F to ML^F. That is, local bindings cannot (in general) be split apart in two different pieces of code—a well-known limitation of ML. Technically, ML^F polymorphism is second-order, yet not first-class. As a consequence, the typing of local bindings cannot be derived from the one of immediate applications and must be primitive.

To see this limitation, we shall proceed by comparison with ML. Consider the following program in ML:

$$\mathsf{let\ f} = \lambda(\mathsf{x})\ \mathsf{x\ in\ (f\ 42,\ f}\ \textit{"foo"}\mathsf{)}$$

The polymorphic type $\forall(\alpha)\ \alpha \to \alpha$ is inferred for the identity function and bound to the identifier f, which may then be used with at different type instances, namely $\mathit{int} \to \mathit{int}$ and $\mathit{string} \to \mathit{string}$. On may consider replacing the let-binding may also be replaced by an abstraction followed by an immediate application:

$$(\lambda\ (\mathsf{f})\ (\mathsf{f\ 42,\ f}\ \textit{"foo"}))\ (\lambda(\mathsf{x})\ \mathsf{x})$$

This expression is not typable in ML, as it would require the function parameter f to be assigned a polymorphic type. In contrast, this program is typable in ML^F, which features second order polymorphism. For that purpose, one need only add an explicit type-annotation $\forall(\alpha)\ \alpha \to \alpha$, which we write σ_{id}, on the parameter f. However, this is only a simplistic example. For instance, consider a small variant of the previous program:

$$\mathsf{let\ f = choose\ id\ in\ (f\ succ,\ auto\ (f\ id))}$$

The expression succ stands for the successor function of type $\mathit{int} \to \mathit{int}$. The expression choose stands for a function that takes two arguments and return either one, which could be defined as $\lambda\ (\mathsf{x})\ \lambda\ (\mathsf{y})\ \mathsf{if\ true\ then\ x\ else\ y}$, of polymorphic type $\forall(\alpha)\ \alpha \to \alpha \to \alpha$. The expression auto stands for a function that requires its argument to be of type σ_{id}, which could be defined as $\lambda(x)\ \mathsf{let}\ y = 1 + x\ 1\ \mathsf{in}\ x\ x$. In the program above, the call to auto forces f id to have type σ_{id}. Hence, the two uses of f altogether require f to have both type $\sigma_1 \to \sigma_{\mathsf{id}}$ and type $(\mathit{int} \to \mathit{int}) \to \sigma_2$ for some σ_1 and σ_2 while the definition of f can only have a type of the form $\sigma_3 \to \sigma_4$ where σ_4 is an instance of both σ_3 and σ_{id}. Unfortunately, the combination of all these constraints has no solution. Hence, the program is not typable in System F.

Interestingly, it is typable in ML^F, using the additional expressiveness of flexible quantification. Precisely, the expression choose id can be typed with $\forall(\alpha \geq \sigma_{\mathsf{id}})\ \alpha \to \alpha$, which intuitively stands for all types $\sigma \to \sigma$ where σ is any instance of σ_{id}. The parameter f can then be used with two peculiar instances, namely $(\mathit{int} \to \mathit{int}) \to (\mathit{int} \to \mathit{int})$ and $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$.

Unfortunately, replacing the local-binding by an abstraction followed by an immediate application leads to the program

$$(\lambda\ (\mathsf{f})\ (\mathsf{f\ succ,\ auto\ (f\ id)}))\ (\mathsf{choose\ id})$$

which is not typable in ML^F. The problem is that the λ-bound variable f cannot be assigned the required flexible type $\forall(\alpha \geq \sigma_{\mathsf{id}})\ \alpha \to \alpha$, as only let-bound variables may be assigned flexible types in (Plain) ML^F. This restriction is relaxed in Full ML^F, which precisely allows flexible quantification at arbitrary positions in expressions and types. Hence, the previous example is typable in Full ML^F.

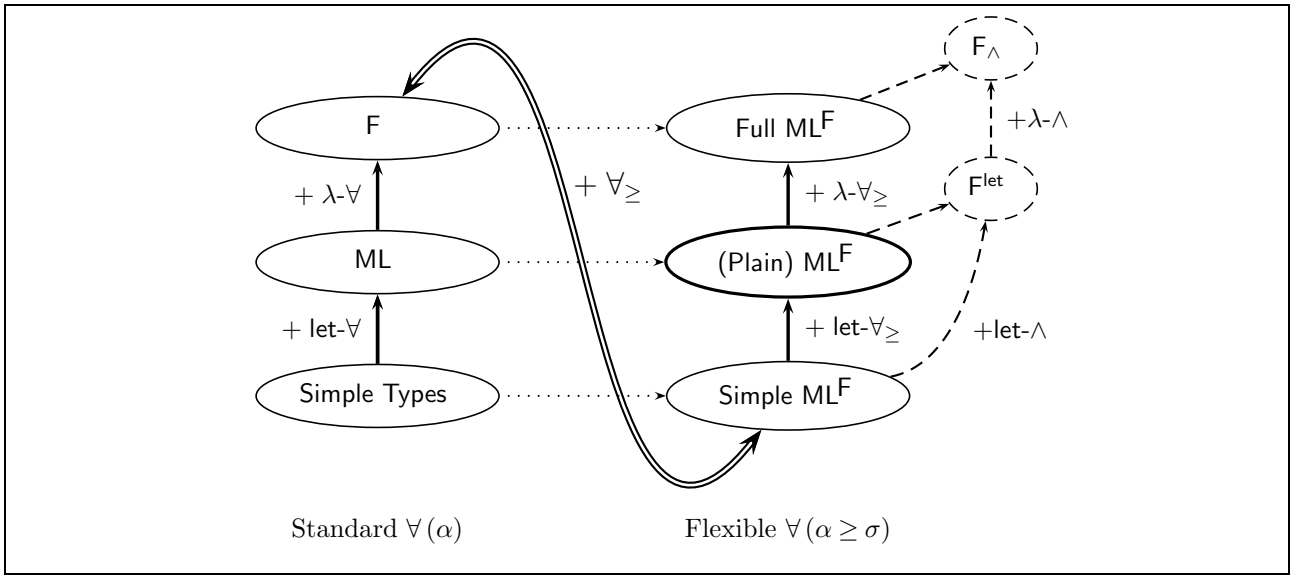
The main outcomes of staying within ML^F are a more comprehensive presentation of the language and the connections drawn with existing systems, using the semantics of types as a tool not only to vehicle strong intuitions but also to recover the syntactic instance relation as a pull-back of set inclusion on semantic types.

The meta-theoretical study of ML^F presented hereafter steps on this semantic support and is thus significantly simpler than and mostly independent from the previous study of the full version [LBR03, LB04].

The remaining gap between (Plain) ML^F and Full ML^F is a small step for the user but another big step for the theoretician: the typing rules are exactly the same in both languages except that we unlock the restriction that is imposed on the occurrences of flexible quantification in the plain version. Hopefully, the intuitions built for the plain version should carry over to the full version. Unfortunately, our semantics of types cannot be easily extended to cope with the full version.

The different versions of the language are summarized in Figure 1. On the left-hand side are well-known languages with increasing polymorphism based on standard $\forall$-quantification. On the right-hand side are several versions of ML^F. As mentioned earlier, Simple ML^F is exactly as expressive as System F and is obtained by adding flexible quantification to System F, yet without any construction to exploit it. In this respect, it is to (Plain) ML^F what simply typed λ-calculus is to ML. Restated in the other direction, ML^F is to Simple ML^F what ML is to simply typed λ-calculus, as both enable the underlying polymorphism on local-bindings in the very same manner. Pursuing the analogy, Full ML^F is to ML^F what System F is to ML—it enables flexible polymorphism on functions parameters and, more generally, to appear at arbitrary position in types: $\forall$-quantification is first-class—but of different power—in both Full ML^F and System F. By contrast, it can only be used at local bindings in both (Plain) ML^F and ML.

Figure 1: The small hierarchy of ML$^{\mathsf{F}}$ versions.



The Systems $\mathsf{F}_\wedge$ (right of Figure 1) is System F with intersection types [Pie91], while $\mathsf{F}^{\mathsf{let}}$ below is its restriction to rank-1 intersection types (see §2.4). Equivalently, $\mathsf{F}^{\mathsf{let}}$ is the closure of Simple ML$^{\mathsf{F}}$ by let-expansion. The arrows between $\mathsf{F}^{\mathsf{let}}$ and $\mathsf{F}_\wedge$ and Simple ML$^{\mathsf{F}}$ and $\mathsf{F}^{\mathsf{let}}$ are materializing these inclusions. The inclusion of ML$^{\mathsf{F}}$ into $\mathsf{F}^{\mathsf{let}}$, which is proved below (Section §**??**), implies the correctness of ML$^{\mathsf{F}}$. The inclusion between Full ML$^{\mathsf{F}}$ and $\mathsf{F}_\wedge$ also holds but is not shown in this paper.

For completeness, two small remarks can also be made. First, ML$^{\mathsf{F}}$ and $\mathsf{F}^{\mathsf{let}}$ are two extensions of Simple ML$^{\mathsf{F}}$ with let-bindings that differ significantly in the way local-bindings are typed: In ML$^{\mathsf{F}}$, they can be typed with Simple ML$^{\mathsf{F}}$ types and generalized afterward, while in $\mathsf{F}^{\mathsf{let}}$, they must either use intersection types or be typed after performing let-reduction. Second, the difference between let-$\forall$ extension and let-$\wedge$ vanishes when replacing Simple ML$^{\mathsf{F}}$ with Simple Types; that is, ML is both the let-$\forall$ extension and let-$\wedge$ extension of Simple Types.

We believe that a programming language should be based on the full rather than the plain version of ML$^{\mathsf{F}}$. However, other extensions such as higher-order types may be easier to explore in the simple version. Hence, the plain version is not only a pedagogical restriction, but also an interesting and solid point in the design space, from which further investigations may be started.

Type inference is not addressed here, as it is a technically orthogonal issue and is not significantly easier for Plain ML$^{\mathsf{F}}$ than for Full ML$^{\mathsf{F}}$. The reader is referred to [LBR03, LB04] or independent study in subsequent work [RY07]. As ML$^{\mathsf{F}}$ was designed with first-order type inference and let-polymorphism in mind, polymorphism need never be guessed but only picked at local bindings and user-provided type annotations and propagated by first-order unification. The difficulty, and in fact the whole design of ML$^{\mathsf{F}}$, lies in the specification of its type system, and in particular, how every use of polymorphism that would imply guessing has been ruled out. So, although we do not develop type inference here, all the key ingredients can already be found in the Church's style version XML$^{\mathsf{F}}$.

Outline of the paper. While previous studies focused on Full XML$^{\mathsf{F}}$, this work is limited to the study of Plain ML$^{\mathsf{F}}$ (Simple ML$^{\mathsf{F}}$ is only introduced as a tool).

The paper is organized as follows. A gentle introduction to ML$^{\mathsf{F}}$ exposing successively its Curry's and Church's views, can be found in §2. The Curry's view is explored in details, including discussions of type-soundness and of expressiveness in §3. The Church's view is studied formally in §4: although the Church's view has been designed especially for type inference, we focus on its fundamental properties here and leave out type inference for reasons explained above. We also address expressiveness of the Church's view by showing that it subsumes both ML and System F. Related works are discussed in §5. Concluding remarks can be found in §6. For clarity of exposition, all proofs have been moved to appendices.

Figure 2: Encodings

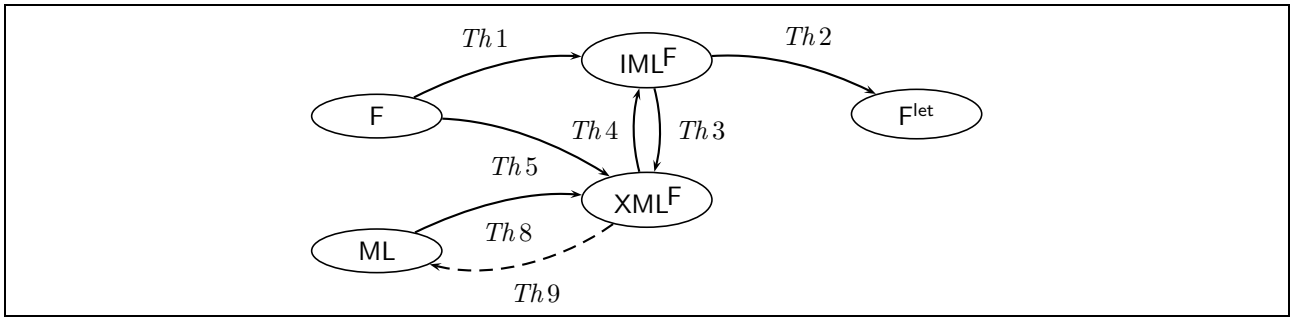


Figure 2 summarizes the seven encodings presented in sections §3 and §4. Encoding are represented by edges and labelled with the corresponding theorem. (The interest of the direct encoding of System F to XML^F is to introduce fewer annotations than the composition of encodings via IML^F, and in particular, fewer type annotations than present on the original term.) The encoding from XML^F to ML is partial, defined on the subset of XML^F programs that do not contain any annotations.

Notations. We write $A \# B$ to mean that the two sets A and B are disjoint. We write $\bar{e}$ for a sequence of elements $e_1, \dots e_n$. We use standard notions of variables, terms, binders, and free variables. The simultaneous capture-avoiding substitution of a sequence of variables $\bar{u}$ by a sequence of objects $\bar{e}'$ in an object e is written $e[\bar{e}'/\bar{u}]$.

We use numerical labels in bold face such as (**1**) as a binding annotation in text or formulas and normal font as (1) to refer to the corresponding binding occurrence. The scope of such labels is the current proof, paragraph, or inner section, and is left implicit.

2 An intuitive introduction to ML^F

This section is primarily an informal introduction to ML^F. The only prerequisite is a good knowledge of ML and some knowledge of System F. We first remind Curry's style System F. However, we use a generic presentation G so as to emphasize the strong relations between all type systems described here. In particular, we present both Curry's style and Church's style versions of ML^F as instances of G. We provide intuitions on the flexible and rigid quantification that are at the heart of ML^F, by means of examples. We also discuss—still informally—some of the advantages of ML^F compared to System F, besides type inference. This section may also be read back after some technical knowledge of ML^F has been acquired to deepen one's own understanding.

2.1 A Generic Curry's style second-order type system

Expressions are the pure λ-terms with optional local definitions. There BNF grammar is:

$$a ::= x \mid \lambda(x)\ a \mid a_1\ a_2 \mid^?\ \mathsf{let}\ x = a\ \mathsf{in}\ a' \qquad \text{Terms}$$

That is, terms are variables x, functions $\lambda(x)\ a$ where the parameter x is bound in a, applications $a_1\ a_2$, and optionally, local definitions $\mathsf{let}\ x = a\ \mathsf{in}\ a'$ where the variable x is bound (to a) in a'. Terms are always taken up to α-equivalence, that is, up to (capture avoiding) renaming of bound variables. Local definitions $\mathsf{let}\ x = a\ \mathsf{in}\ a'$ can always be seen as a way of marking immediate applications $(\lambda(x)\ a')\ a$. The intention is to type them in a special way, much as in ML, which is often easier and more general than typing the function and the application independently. In some cases, we may however exclude local definitions in order to either simplify the language, when local definitions do not actually increase expressiveness, or intentionally restrict the language. In either case, we may still use local-bindings in examples but only see them as syntactic sugar for immediate applications.

Types

Throughout this paper, we use several related but different notions of second-order types. For simplicity, we use the same countable set of type variables ϑ for all notions of types and letters α, β, or γ to range over type variables.

The generic presentation, summarized in the following BNF grammar, abstracts over the exact sets of first-class types $\mathcal{T}$ and type schemes $\mathcal{S}$, which are only partially specified.

$$\begin{array}{rr}
\tau \in \mathcal{T} ::= \alpha \mid \tau \to \tau \mid \ldots & \text{Types} \\
\mid^? \ \forall\,(q)\ \tau & \text{Polymorphic types (optionally)} \\
\sigma \in \mathcal{S} ::= \tau \mid \forall\,(q)\ \sigma \mid \ldots & \text{Type schemes} \\
q \in \mathcal{Q} ::= \alpha :: k & \text{Bindings}
\end{array}$$

Types should at least contain type variables α and arrow types $\tau \to \tau$. Depending on the particular instance, types may also contain polymorphic types $\forall\,(q)\ \tau$ where q is a binding of the form $(\alpha :: \kappa)$. For sake of generality, quantified type variables are constrained by kinds $k \in K$ where the set of kinds K is left unspecified for the moment. We write $\mathsf{dom}(q)$ to refer to α. Type schemes σ extend types with outermost quantification, as in ML. Still as in ML, types are first-class while type schemes will appear only in typing environments and typing judgments.

In simple cases, such as ML or System F, K is a singleton $\{\star\}$. We may then abbreviate $\alpha :: \star$ as α. The level of generality allows both for multi-kinded type expressions, *e.g.* taking for K a set of atomic kinds and for more complex forms of quantification, such as subtyping. Note, that K may contain type schemes as subexpressions (see, for instance, types of $\mathsf{F}_{<:}$ described below). We write $\mathsf{ftv}(k)$ for the free type variables of types expressions that appear in k. In $\forall\,(\alpha :: k)\ \sigma$, the $\forall$ quantifier binds α in σ but not in k. Free type variables are defined as usual:

$$\mathsf{ftv}(\alpha) = \{\alpha\} \qquad \mathsf{ftv}(\tau \to \tau') = \mathsf{ftv}(\tau) \cup \mathsf{ftv}(\tau') \qquad \mathsf{ftv}(\forall\,(\alpha :: k)\ \tau) = \mathsf{ftv}(k) \cup (\mathsf{ftv}(\tau) \setminus \{\alpha\})$$

Indeed, this definition must be adjusted to the exact definition of $\mathcal{T}$. We always consider types up to α-conversion. The scope of $\forall$-quantification extends to the right as far as possible and $\to$ is right associative. That is, $\forall\,(q)\ \tau \to \tau \to \tau$ means $\forall\,(q)\ (\tau \to (\tau \to \tau))$. We may write $\forall\,(qq')\ \sigma$ for $\forall\,(q)\ \forall\,(q')\ \sigma$.

For example, ML types and type schemes are defined as follows:

$$\begin{array}{rr}
\tau \in \mathcal{T}_{\mathsf{ML}} ::= \alpha \mid \tau \to \tau & \mathsf{ML}\text{ types} \\
\sigma \in \mathcal{S}_{\mathsf{ML}} ::= \tau \mid \forall\,(q)\ \sigma & \mathsf{ML}\text{ type schemes} \\
q \in \mathcal{Q}_{\mathsf{ML}} ::= \alpha :: \star & \mathsf{ML}\text{ bindings}
\end{array}$$

This is indeed the simplest notion of generic types, as all options in the definition have been turned off.

Types of System F, which we call F-types, may also be described by an instance of the generic grammar:

$$\begin{array}{rr}
t \in \mathcal{T}_{\mathsf{F}} ::= \alpha \mid t \to t \mid \forall\,(q)\ t & \mathsf{F}\text{ types} \\
\sigma \in \mathcal{S}_{\mathsf{F}} ::= t & \mathsf{F}\text{ type schemes} \\
q \in \mathcal{Q}_{\mathsf{F}} ::= \alpha :: \star & \mathsf{F}\text{ bindings}
\end{array}$$

Quantifiers may appear in types directly and thus at arbitrary positions in F-types—as expected. In this case, types are closed by outermost quantifications and coincide with type schemes. We use letter t rather than τ to range over F-types, so as to distinguish them from any form of types.

Types of $\mathsf{F}_{<:}$ are yet another example, that uses kinds in a more interesting way:

$$\begin{array}{rr}
\tau \in \mathcal{T}_{\mathsf{F}_{<:}} ::= \alpha \mid \tau \to \tau \mid \forall\,(q)\ \tau \mid \top & \mathsf{F}_{<:}\text{ types} \\
\sigma \in \mathcal{S}_{\mathsf{F}_{<:}} ::= \tau & \mathsf{F}_{<:}\text{ type schemes} \\
q \in \mathcal{Q}_{\mathsf{F}_{<:}} ::= \alpha <: \tau & \mathsf{F}_{<:}\text{ bindings}
\end{array}$$

Types and type schemes still coincide, but contain an extra element $\top$ and kinds are now types. Bindings are written $\alpha <: \tau$ rather than $\alpha :: \tau$.

Prefixes

As type variables come with bounds, most operations on types will be defined under a prefix that assigns bounds to their free type variables. A *prefix* Q is a sequence of bindings $q_i^{i \in 1..n}$. Its domain, written $\mathsf{dom}(Q)$ is the set $\{\alpha_1, .., \alpha_n\}$. We write $\emptyset$ for the empty prefix. A prefix Q binds all type variables of $\mathsf{dom}(Q)$ (defined as the union of the pointwise domains). However, they may themselves have free type variables, written $\mathsf{ftv}(Q)$ and defined recursively as $\mathsf{ftv}(\emptyset) = \emptyset$ and $\mathsf{ftv}(Q, q) = \mathsf{ftv}(Q) \cup (\mathsf{ftv}(q) \setminus \mathsf{dom}(Q))$. Hence, for a closed prefix $q_i^{i \in 1..n}$, $\mathsf{dom}(q_i)$ may only intersect $\mathsf{ftv}(q_j)$ for $j > i$.

We assume that the following well-formedness condition holds for all prefixes $q_i^{i \in 1..n}$: $\mathsf{dom}(q_i)$'s are pointwise disjoint and $\mathsf{dom}(q_i)$ may only intersect $\mathsf{ftv}(q_j)$ for $j > i$.

Type instance

The type-instance relation is meant to capture the idea that some types are better than others, in the sense that some types can be automatically deduced from those of which they are instances. This may be specified directly using a specific typing rule. For example, an expression of System F that has a polymorphic type $\forall(\alpha)\ \sigma$ may be applied to any type τ resulting in an expression of type $\sigma[\tau/\alpha]$. That instantiation may also be left implicit, *i.e.* without markers such as type abstraction and type application in expressions, as in Curry's style type systems.

The instance relation for Curry's style System F, written $\leq_{\mathsf{F}}$, is the binary relation composed of exactly all pairs of the form $\forall(\bar{\alpha})\ \sigma \leq_{\mathsf{F}} \forall(\bar{\beta})\ \sigma[\bar{\tau}/\bar{\alpha}])$ such that none of the variables $\bar{\beta}$ is free in $\forall(\bar{\alpha})\ \sigma$. The quantification $\forall(\bar{\beta})$ is used to generalize some of the type variables that might have been introduced in $\bar{\tau}$.

The type instance relation $\leq_{\mathsf{ML}}$ for ML may be defined similarly except that it applies to weaker sets of types and type schemes. In fact, it is exactly the restriction of $\leq_{\mathsf{F}}$ to ML type schemes.

We may abstract over the precise definition of the instance relation $\leqslant$. For the sake of generality, we assume that the instance relation is taken under some prefix Q. That is, $\leqslant$ is a ternary relation $(Q)\ \sigma \leqslant \sigma'$ between a prefix Q and two type schemes σ and σ'. We say that $\leqslant$ is a relation under prefix or also that $\leqslant$ is a prefixed relation. We use letter $\mathcal{R}$ to range over such relations. We may view (ternary) prefixed relations as binary relations by treating the relation as a family of relations $\mathcal{R}_Q$ indexed by a prefix Q. That is $\mathcal{R}$ is reflexive (respectively symmetric, transitive, *etc.*) if relations $\mathcal{R}_Q$ are reflexive (respectively symmetric, transitive, *etc.*) for all prefixes Q. The symmetric of a prefixed relation $\mathcal{R}$ is the prefixed relation $\mathcal{R}^{-1}$ defined by taking $(\mathcal{R}_Q)^{-1}$ for $(\mathcal{R}^{-1})_Q$. (We often write $\succ$ for $(\prec)^{-1}$ when $\mathcal{R}$ is a symbol $\prec$.)

The System-F type-instance relation $\leq_{\mathsf{F}}$ happens to be a particular case where the relation is actually independent of the prefix (hence, treated as a binary relation on types).

Another type instance relation that generalizes $\leq_{\mathsf{F}}$ in an interesting way is *type-containment* $\leq_\eta$, introduced by Mitchell in the late 80's [Mit88]. As for $\leq_{\mathsf{F}}$, type containment is independent of prefixes and can thus also be treated as a binary relation. It is congruent but propagates contra-variantly on the left-hand side of arrow types (and covariantly everywhere else) and distributes $\forall$-quantifiers over arrows (see [Mit88]). Type containment allows to capture deep instantiations, *e.g.* $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}} \leq_\eta \sigma_{\mathsf{id}} \to (\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}})$, as well as extrusion of quantifiers, *e.g.* $\forall(\alpha')\ \alpha' \to (\forall(\alpha)\ \alpha \to \alpha) \leq_\eta \forall(\alpha', \alpha)\ \alpha' \to \alpha \to \alpha$.

A truly ternary type-instance relation is the subtyping relation $<:$ used in $\mathsf{F}_{<:}$. The prefix is used to assign upper bounds to free type variables of the types being compared. As for type containment, this relation propagates contra-variantly on the left-hand side of arrows.

Type equivalence

An instance relation $\leqslant$ induces an equivalence under prefix, defined as the kernel of $\leqslant$, *i.e.* $\leqslant \cap \geqslant$.

In the case of F-types where the equivalence does not depend on prefixes, we actually treat types up to equivalence, *i.e.* up to commutation of adjacent binders and removal of redundant binders (in addition to α-conversion). That is, $\forall(\alpha\alpha')\ \alpha \to \alpha'$, $\forall(\alpha\alpha')\ \alpha' \to \alpha$, and $\forall(\alpha\alpha'\alpha'')\ \alpha' \to \alpha$ are thus considered as equal in System F.

The generic Curry's style second-order type system

The generic Curry's style second-order *type system*, written $\mathsf{G}(\mathcal{T}, \mathcal{S}, \mathcal{Q}, \leqslant)$, is parameterized by a set of types $\mathcal{T}$, type schemes $\mathcal{S}$, bindings $\mathcal{Q}$, and an instance relation $\leqslant$ over type schemes.

Typing contexts are partial mappings from program variables to type schemes with finite domains. The free type-variables of a typing context are the union of the free type variables of its codomain. We write $\emptyset$ the mapping defined nowhere and $\Gamma, x : t$ the mapping that sends x to t and behaves as Γ everywhere else.

Typing judgments are of the form $(Q)\ \Gamma \vdash a : \sigma$ where Γ is a typing context and Q a closed prefix that binds all type variables that appear free in σ or Γ. Hence, we must have $(\mathsf{ftv}(\sigma) \cup \mathsf{ftv}(\Gamma)) \subseteq \mathsf{dom}(Q)$ (and $\mathsf{ftv}(Q)$ empty).

Typing rules are given in Figure 3. Rules for variables, abstractions, and applications are standard, modulo the explicit mention of the prefix. As terms are unannotated, instantiation and generalization are left implicit, as in ML. Hence, rules Inst and Gen are not syntax-directed: in each case, the expression a appears identically in the premise and the conclusion. Rule Gen is standard and introduces polymorphism by discharging type abstraction from the judgment hypothesis into the type of the expression. Rule Inst is a type-containment rule that generalizes the more traditional polymorphism single-elimination step. This approach is preferable

Figure 3: Typing rules for $\mathsf{G}(\mathcal{T}, \mathcal{S}, \mathcal{Q}, \leqslant)$.

$$\textsc{Var}\ \frac{x : \sigma \in \Gamma}{(Q)\ \Gamma \vdash x : \sigma} \qquad \textsc{Fun}\ \frac{(Q)\ \Gamma, x : \tau \vdash a : \tau'}{(Q)\ \Gamma \vdash \lambda(x)\ a : \tau \to \tau'} \qquad \textsc{App}\ \frac{(Q)\ \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad (Q)\ \Gamma \vdash a_2 : \tau_2}{(Q)\ \Gamma \vdash a_1\ a_2 : \tau_1}$$

$$\textsc{Inst}\ \frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \leqslant \sigma'}{(Q)\ \Gamma \vdash a : \sigma'} \qquad \textsc{Gen}\ \frac{(Q, q)\ \Gamma \vdash a : \sigma \qquad \mathsf{dom}(q) \notin \mathsf{ftv}(\Gamma)}{(Q)\ \Gamma \vdash a : \forall (q)\ \sigma}$$

$$\textsc{Let}\ \frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \Gamma, x : \sigma \vdash a' : \sigma'}{(Q)\ \Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \sigma'}$$

in a Curry's style type system as it moves instantiation from the typing derivation into a type-instance sub-derivation.

Rule Let is used for typechecking local definitions in a special way. This rule is indeed inspired from ML and reproduces the very same mechanism for typing local-derivations within System G. This improves over the default rule that would consist in typechecking $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ as the immediate application $(\lambda(x)\ a_2)\ a_1$. In cases where types and type schemes coincide, *e.g.* as in System F, we could simply view local definition as syntactic sugar for immediate applications. In other cases, Let is actually a key rule that truly empowers System G.

If G_1 is an instance of System G, we write $G_1 :: J$ to mean without ambiguity that judgment J refers to the system G_1. However, we usually leave the underlying type system implicit from context.

Hypotheses

In order for the type system to have sane properties, the instance relation $\leqslant$ is assumed to satisfy some conditions.

Let *extension* over well-formed prefixes be the smallest order $\supseteq$ that contains all pairs $QqQ' \supseteq QQ'$ for any prefixes Q and Q'. We say that a prefix Q_2 *extends* a prefix Q_1 whenever $Q_2 \supseteq Q_1$ holds. Intuitively, Q_2 just contains more bindings than Q_1.

In the rest of the paper, we only consider instance relations that satisfy the following two axioms:

$$\textsc{Renaming}\ \frac{(Q)\ \sigma_1 \leqslant \sigma_2 \qquad \phi\ \text{renaming}}{(\phi(Q))\ \phi(\sigma_1) \leqslant \phi(\sigma_2)} \qquad \textsc{Extra-bindings}\ \frac{Q \supseteq Q' \qquad (Q')\ \sigma_1 \leqslant \sigma_2}{(Q)\ \sigma_1 \leqslant \sigma_2}$$

Then, we can prove that typing judgments can be renamed and prefixes can be extended:

Lemma 2.1.1

i) Renaming of Typing Derivations: *If $(Q)\ \Gamma \vdash a : \sigma$ holds and ϕ is a renaming, then $(\phi(Q))\ \phi(\Gamma) \vdash a : \phi(\sigma)$ holds.*

ii) Prefix extension: *If $(Q)\ \Gamma \vdash a : \sigma$ holds and $Q' \supseteq Q$, then $(Q')\ \Gamma \vdash a : \sigma$ holds.*

Both proofs are by induction on the derivation and indeed relies on both axioms.

Particular instances of System G.

Curry's style System F and ML are two by-design immediate instances of System G, namely $\mathsf{G}(\mathcal{T}_\mathsf{F}, \mathcal{T}_\mathsf{F}, \mathcal{Q}_\mathsf{F}, \leq_\mathsf{F})$ and $\mathsf{G}(\mathcal{T}_\mathsf{ML}, \mathcal{S}_\mathsf{ML}, \mathcal{Q}_\mathsf{ML}, \leq_\mathsf{ML})$. Notice that we slightly depart from the tradition to view ML as a subcase of System F and instead view both as special cases of System G.

An interesting extension of System F, introduced by Mitchell and called F^η is the closure of System F by η-contraction [Mit88]. It may be concisely described as $\mathsf{G}(\mathcal{T}_\mathsf{F}, \mathcal{T}_\mathsf{F}, \mathcal{Q}_\mathsf{F}, \leq_\eta)$. As noticed by Mitchell, F^η allows more terms to have principal types. For instance, $\forall (\alpha)\ \alpha \to \alpha$, say σ_id, is a principal type for the identity function. Other correct types $\sigma_\mathsf{id} \to \sigma_\mathsf{id}$ or $\sigma_\mathsf{id} \to \sigma_\mathsf{id} \to \sigma_\mathsf{id}$ are $\leq_\eta$-instance of σ_id. In fact, this is also the case for any possible type of the identity. Hence, σ_id captures all types of the identity up to $\leq_\eta$-instantiation. For

that reason Mitchell has suggested that F^η could be a better candidate than System F for type inference. Still, many expressions do not have principal types in F^η. Somehow, F^η is simultaneously too expressive (we do not really need contra-variance of type instance) and too weak for our needs (it lacks simultaneous instantiation constraints).

The language $\mathsf{F}_{<:}$ [Car93] can also be defined as the generic type system $\mathsf{G}(\mathcal{T}_{\mathsf{F}_{<:}}, \mathcal{S}_{\mathsf{F}_{<:}}, \mathcal{Q}_{\mathsf{F}_{<:}}, <:)$. Note however, that this is a Curry's style presentation while $\mathsf{F}_{<:}$ is usually presented in Church's style.

2.2 Curry's style ML^F.

Returning to our goal, we seek for a language that has at least the expressiveness of System F while enabling global first-order type inference for partially annotated terms. Based and improving on previous experiences, we wish to manipulate second-order types transparently, so as to avoid inelegant and verbose boxing and unboxing operations and avoid annotations for all ML programs. Indeed, we seek for the fusion of ML-style *implicit* polymorphism with *explicit* F-style polymorphism, rather than their juxtaposition, so that the best of each approach is really transmitted to the other one.

The inadequacy of F-types

Our goal implies that type instantiation must be left implicit, as in ML. Implicit instantiation is easy and rather natural in ML. The main reason is that polymorphism is not first-class. That is, only type schemes can be polymorphic. Types which may appear on the left of arrows cannot be polymorphic. Therefore, polymorphic values must always be instantiated before being passed as arguments to functions. This is no more true in System F—or any other language with first-class polymorphism, where a polymorphic value may also be passed as argument. Moreover, *implicit* polymorphism, as in Curry's style, brings an additional difficulty: the application of a polymorphic function to a polymorphic value may become ambiguous, as a result of permitting any polymorphic expression e of type τ to be treated as an expression of any type τ' that is an instance of τ.

For example, consider a value v of type τ and a function choose of type $\forall(\alpha)\ \alpha \to \alpha \to \alpha$—which for example could either be a polymorphic comparison returning the greatest of two arguments or just a function randomly returning one of two arguments. What should be the type of $\mathsf{choose}\ v$ in System F? Should v be kept as polymorphic as τ or instantiated to some type τ'—but which one? Indeed, any type $\tau' \to \tau'$ is a correct one for $\mathsf{choose}\ v$ as long as τ' is an instance of τ. In other words, the correct types for $\mathsf{either}\ e$ form a set $\{\tau' \to \tau', \tau \leq \tau'\}$[5]. Unfortunately, this set does not have a greatest lower bound that could be used as a principal type to represent all others.

This very simple example raises a crucial issue whose solution is really the key to understanding ML^F from both an intuitive and formal perspective.

Using infinite intersection types $\bigwedge\{\tau' \to \tau', \tau \leq \tau'\}$, as suggested by Leivant in another context [Lei90] is a temptation. However, this is pernicious from a logical point of view. Moreover, this would ignore the underlying structure of such sets, which are always instantiation upward closed.

Flexible quantification—the Key

Since intersection types are too powerful for our purposes, we introduce a new form of type scheme $\forall(\alpha \geq \sigma)\ \alpha \to \alpha$ to describe the set of all types $\tau \to \tau$ such that τ is an instance of σ. We may indeed interpret such type schemes as sets of System-F types (§3.1 (page 18)). The instance relation $\leq$ between type schemes is then defined as set inclusion on their interpretations. This makes $\forall(\alpha \geq \sigma')\ \alpha \to \alpha$ an instance of $\forall(\alpha \geq \sigma)\ \alpha \to \alpha$ whenever σ' is an instance of σ and thus really makes $\forall(\alpha \geq \sigma)\ \alpha \to \alpha$ a principal type for the expression $\mathsf{choose}\ v$ where v has principal type scheme σ. Type schemes contain types, but all type schemes are not types. Types may still be polymorphic. For instance, σ_{id} shall remain a type in IML^F.

The binding $(\alpha \geq \sigma)$ is called a *flexible binding*, as the bound σ may be soundly replaced by a type scheme σ' or a type τ that are instances of σ, producing an instance of the whole type. The occurrence of a σ in $(\alpha \geq \sigma)$ is also called a *flexible* occurrence.

By contrast, we call *rigid* occurrences of a type below an arrow, as those of σ_{id} in $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$. Rigid occurrences may not be instantiated, as this could be unsound. For example, the function $\lambda(x)\ x\ x$ has type $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$, where σ_{id} is $\forall(\alpha)\ \alpha \to \alpha$, but does not have type $\tau' \to \tau$ nor even type $\tau' \to \tau'$ for arbitrary instances τ' of σ_{id}. In particular, it does not have type $\forall(\alpha)\ (\alpha \to \alpha) \to (\alpha \to \alpha)$. Although it would always be safe to

[5] We use "$\leq$" rather than $\leq_\mathsf{F}$ here, as we are about to extend the set of types and enlarge the type-instance relation accordingly.

instantiate types on the right occurrence of arrow types, we do not do so. We may always use a type scheme $\forall\,(\alpha \geq \tau)\ \tau' \to \alpha$ instead of of $\tau' \to \tau$ to *explicitly* allow instances of τ to be taken for α.

For the sake of uniformity, we introduce a special *trivial bound* $\bot$ (read *bottom*) to mean *any* type. We may then see $\forall\,(\alpha)\ \alpha \to \alpha$ as syntactic sugar for $\forall\,(\alpha \geq \bot)\ \alpha \to \alpha$. Intuitively, $\bot$ could itself be seen as representing the set of all types and, indeed, $\bot$ is equivalent to $\forall\,(\alpha \geq \bot)\ \alpha$ in our setting. In this view, $\forall\,(\alpha \geq \sigma_{\mathsf{id}})\ \alpha \to \alpha$ is an instance of $\forall\,(\alpha)\ \alpha \to \alpha$: since the bound σ_{id} of the former is an instance of the bound $\bot$ of the latter, the interpretation of the former contains the interpretation of the latter.

One may wonder what is the meaning of a type such as $(\forall\,(\alpha \geq \tau)\ \alpha \to \alpha) \to \tau'$ when a flexible type appears under an arrow type. We just forbid such types—in Plain ML^F. This is achieved by restricting the use of flexible quantification to type schemes and only allow quantification with trivial bounds in types. More precisely, types and type schemes for Curry's style ML^F (that is, IML^F) are defined as follows:

$$\tau \in \mathcal{T}_{\mathsf{I}} ::= \alpha \mid \tau \to \tau \mid \forall\,(\alpha \geq \bot)\ \tau \qquad \text{IML}^F \text{ types}$$
$$\sigma \in \mathcal{S}_{\mathsf{I}} ::= \tau \mid \forall\,(\alpha \geq \sigma)\ \sigma \mid \bot \qquad \text{IML}^F \text{ types schemes}$$
$$q \in \mathcal{Q}_{\mathsf{I}} ::= \alpha \geq \sigma \qquad \text{IML}^F \text{ bindings}$$

Type schemes that are not types are called *proper* type schemes; they may not appear under arrows. A consequence of this stratification is that proper type schemes cannot be assigned to parameters of function. Therefore, local definitions $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ play a key role, exactly as in ML. Indeed, one may first assign a type scheme σ to a_1 and use σ as the type for the parameter x while typechecking a_2. By contrast, this assignment of type σ to variable x would be forbidden in the immediate application $(\lambda(x)\ a_2)\ a_1$ whenever σ is a proper type scheme, as x would be λ-bound and its type would have to appear under the arrow type of the function (see Rule FUN).

Our stepping stone is the instance $\mathsf{G}(\mathcal{T}_{\mathsf{I}}, \mathcal{S}_{\mathsf{I}}, \mathcal{Q}_{\mathsf{I}}, \leq)$ of Generic Curry's style System F with flexible quantification and local definitions. This intermediate language is in Curry's style and all type information is still left implicit. For this reason, we also call it IML^F (read *implicit* ML^F). Remarkably, the power of IML^F only lies in its type, type scheme, and type-instance definitions and not in its typing rules, as it is just an instance of System G.

2.3 Church's style ML^F.

In IML^F more expressions have principal types and also many expressions have more general types than in System F. However, IML^F does not allow for type inference yet, even though it was designed with type inference in mind, as all type information is still left implicit. We now devise XML^F—a Church's style version of ML^F that enables type inference.

First-order inference of second-order types...

Our goal is to perform some first-order only type inference but in a language with second-order types, with the two additional constraints to leave all ML programs unannotated and to reach all of System F programs via suitable type annotations.

This has immediate consequences in terms of examples that we should or *should not* type. For example, we should infer a type for the identity function id defined as $\lambda(z)\ z$, or the expression $\mathsf{let}\ x = \mathsf{id}\ \mathsf{in}\ x\ x$, as both are already typable in ML. Conversely, we should not type the auto-application $\lambda(x)\ x\ x$ $(\mathbf{1})$, unless we explicitly annotate the parameter as in the expression $\lambda(x : \sigma_{\mathsf{id}})\ x\ x$ which we further refer to as auto.

We claim to never guess second-order polymorphism. But what does guessing exactly mean? How can we combine ML style *implicit* polymorphism with second-order *explicit* polymorphism? The difficulty may be seen by comparing the expressions a_1 defined as $(\lambda(z)\ z)$ auto and a_2 defined as $(\lambda(x)\ x\ x)$ id. Should we reject both, as their function parameters carry values of polymorphic types—$\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$ for z in a_1 and σ_{id} for x in a_2? Indeed, a_1 may be typed as $(\lambda(z : \sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}})\ z)$ auto. Here, it seems that $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$ must be guessed as the type of the parameter z. However, one may also type $\lambda(z)\ z$ in a_1 as $\alpha \to \alpha$, generalize the resulting type to $\forall\,(\alpha)\ \alpha \to \alpha$, and finally instantiate it to $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$, which may be more concisely summarized by its fully annotated form $(\Lambda\alpha.\ \lambda(z : \alpha)\ \alpha)\ [\sigma_{\mathsf{id}}]$ auto in Church's style System F. In this expression, $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$ need not be guessed as the type of the parameter z, but as the type for specializing the universal variable α to the polymorphic type σ_{id} of $\lambda(z)\ z$. Fortunately, we may—and must, as argued in §2.2—devise ML^F to fully infer type abstractions and type applications (and never *guess* polymorphic types for function parameters). Thus we

accept a_1 (**2**). Conversely, we reject a_2 for the very same reason we rejected $\lambda(x)\ x\ x$ (1). Actually, our system will be compositional, hence any closed subterm of a well-typed term must also be well-typed. (Thus a_2, which contains the ill-typed closed subterm $\lambda(x)\ x\ x$ will also be ill-typed.)

Of course, we distinguish a function parameter whose type is inferred from one whose type is given. We do so much as in ML, by distinguishing between types τ (also called monotypes), which do not contain any quantifier and can be inferred, and proper type schemes σ, also called polytypes for emphasis. The parameters z and x of a_1 and a_2 may only be assigned monotypes. This justifies the rejection of a_2, as $\lambda(x)\ x\ x$ may not be typed when x is a monotype. Conversely, a_1 may be typed by assigning z a monotype, as explained above (2). By contrast with a_2, the parameter x of $\lambda(x : \sigma_{\mathsf{id}})\ x\ x$ may be assigned the polytype σ_{id}, since it is explicitly annotated.

Unfortunately, this distinction is not sufficiently permissive. Consider the expression a_3 defined as $\lambda(z)$ $(z\ \mathsf{auto})$, which somehow lies between a_1 and a_2. The parameter z of a_3 must have a polymorphic type while typechecking the body of the function, exactly as in the expression a_2. However, this polymorphism is not used in the body of a_3 but only carried through. Wishfully, it should thus also be accepted. As another hint, remark that a_3 is the β-reduction of $(\lambda(y)\ \lambda(z)\ z\ y)\ \mathsf{auto}$, which we refer to as a_4. Arguing as for a_1, it is clear than a_4 must be typable. As the β-expanded form is typable, we may expect the β-reduced form to also be—subject reduction will hold for terms without type-annotations. As a cross-checking final example, should the expression a_5 defined as $\lambda(z)\ \mathsf{auto}\ \ z$ be accepted? We may reason by analogy with the previous example and either check that z is not used polymorphically in the body of the function or check that its β-expansion $(\lambda(x)\ \lambda(y)\ x\ y)\ \mathsf{auto}$ is typable.

It is actually a remarkable and *essential* property of ML^F that whenever $a_1\ a_2$ is typable, then $\mathsf{apply}\ a_1\ a_2$ also is, with the same type, where indeed apply stands for the expression $\lambda(x)\ \lambda(y)\ x\ y$—with no type annotation on its parameters. As a remarkable and important corollary, if a function a is typable with some type σ, then so is its η-expanded form $\lambda(z)\ a\ z$. In practice, these too properties will ensures that well-typed programs will be stable by some common small program transformations.

Abstracting second-order polymorphism into first-order types

To solve this last series of examples, our solution is very much inspired by boxed polymorphism, which allows second-order polymorphism to hang under monotypes [GR99]. We retain the very same idea of boxing polymorphism, but make boxes virtual, by abstracting (instead of boxing) second-order polymorphism as a first-order type variable. For instance, abstracting σ_{id} as α let the polymorphic type $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$ be represented by the monotype $\alpha \to \alpha$.

Technically, we keep abstractions in the prefix Q that appears in front of typing judgments $(Q)\ \Gamma \vdash a : \sigma$, using a new form of bindings $(\alpha \Rightarrow \sigma)$, which should be read "$\alpha$ abstracts σ." Resuming with the typechecking of a_5, we may write $(\alpha \Rightarrow \sigma_{\mathsf{id}})\ z : \alpha \vdash \mathsf{auto}\ \ z : \alpha$ (**3**). The hypothesis that α abstracts σ_{id} allows to abstract the type $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$ of auto as $\alpha \to \alpha$. We may then assign type $\alpha \to \alpha$ to auto and type α to the application of $\mathsf{auto}\ \ z$ when z is assumed of type α. Note that abstraction is an asymmetric relation and it is not the case that σ_{id} abstracts α. In particular, $(\alpha \Rightarrow \sigma)\ z : \alpha \vdash z : \sigma_{\mathsf{id}}$ does not hold. This would reveal hidden information and it is not allowed implicitly, but only explicitly via a type annotation. Discharging the assumption on z in the judgment (3) (like in Rule Fun), leads to $(\alpha \Rightarrow \sigma_{\mathsf{id}})\ \vdash a_4 : \alpha \to \alpha$. Finally discharging the prefix (like in Rule Gen), we may conclude $\vdash a_4 : \forall\,(\alpha \Rightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha$. This can be read as "$a_4$ has type $\alpha \to \alpha$ where α is σ_{id}." Notice both the analogy and the difference with flexible bounds. Here, the bound of α means exactly σ_{id} and cannot be instantiated. We call $(\alpha \Rightarrow \sigma_{\mathsf{id}})$ a *rigid binding* and the position of σ_{id} a *rigid occurrence*.

Although σ_{id} is the only possible meaning for α, it cannot be substituted inside $\alpha \to \alpha$. That is, σ_1 defined as $\forall\,(\alpha \Rightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha$ is not equivalent to $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$, nor to $\forall\,(\alpha \Rightarrow \sigma_{\mathsf{id}}, \alpha' \Rightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha'$, which we refer to as σ_2. Maybe surprisingly, σ_1 is more abstract than σ_2, which we write $\sigma_2 \sqsubseteq \sigma_1$. To see this, one may read the latter as $\forall\,(\alpha \Rightarrow \sigma_{\mathsf{id}})\ (\forall\,(\alpha' \Rightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha')$ and abstract σ_{id} as α in the binding $(\alpha' \Rightarrow \sigma_{\mathsf{id}})$, leading to $\forall\,(\alpha \Rightarrow \sigma_{\mathsf{id}})\ (\forall\,(\alpha' \Rightarrow \alpha)\ \alpha \to \alpha')$, which is equivalent to $\forall\,(\alpha \Rightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha$. The later step holds because monotype bounds, which have no other instances but themselves, are "transparent" and can always be substituted for the variable they bound.

Polytype bounds may also intuitively be expanded. For instance, $\forall\,(\alpha \Rightarrow \sigma_{\mathsf{id}}, \alpha' \Rightarrow \sigma_{\mathsf{id}})\ \beta \to \alpha'$ intuitively stands for $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$. However, this is not quite correct as in general the position of quantifiers would be ambiguous. For example $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$ could be read either as $\forall\,(\alpha \Rightarrow \sigma_{\mathsf{id}}, \alpha' \Rightarrow \sigma_{\mathsf{id}}, \alpha'' \Rightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha' \to \alpha''$ or $\forall\,(\alpha \Rightarrow \sigma_{\mathsf{id}}, \alpha' \Rightarrow \sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}})\ \alpha \to \alpha'$ where $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$ need in turn to be expanded. However, both are not considered equivalent in IML^F. Thus, we simply forbid polytypes to appear under arrow types, and instead force

them to be abstracted as variables in auxiliary bindings. Consistently, and by contrast with $\mathsf{IML}^{\mathsf{F}}$, we restrict types to monotypes and force all polytypes to be type schemes.

Type annotations are used to reveal abstractions. For instance, $\lambda(x)\ (x : \sigma_{\mathsf{id}})\ x$, is typed as follows: the annotation $(x : \sigma_{\mathsf{id}})$ requires that x has some type α where α is an abstraction of σ_{id} and reveals σ_{id} as the type of the annotated expression. We thus have $(\alpha \Rightarrow \sigma_{\mathsf{id}})\ x : \alpha \vdash (x : \sigma_{\mathsf{id}}) : \sigma_{\mathsf{id}}$. Once the type σ_{id} has been revealed, it may be instantiated, *e.g.* into $\alpha \to \alpha$. Therefore, we have $(\alpha \Rightarrow \sigma_{\mathsf{id}})\ x : \alpha \vdash (x : \sigma_{\mathsf{id}})\ x : \alpha$ and, finally, $\vdash \lambda(x)\ (x : \sigma_{\mathsf{id}})\ x : \forall\,(\alpha \Rightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha$. There is still one subtlety when typechecking the simpler program $\lambda(x)\ (x : \sigma_{\mathsf{id}})$. From the intermediate step $(\alpha \Rightarrow \sigma_{\mathsf{id}})\ x : \alpha \vdash (x : \sigma_{\mathsf{id}}) : \sigma_{\mathsf{id}}$ we may not conclude $(\alpha \Rightarrow \sigma_{\mathsf{id}}) \ \vdash (x : \sigma_{\mathsf{id}}) : \alpha \to \sigma_{\mathsf{id}}$ as $\alpha \to \sigma_{\mathsf{id}}$ would be an ill-formed type scheme. Moreover, this would just be one solution among many others, as σ_{id} could also have been instantiated. The solution is to use a flexible binding $(\alpha' \geq \sigma_{\mathsf{id}})$ to represent any type of x through the type variable α' leading to the judgment $(\alpha \Rightarrow \sigma_{\mathsf{id}}, \alpha' \geq \sigma_{\mathsf{id}}) \ \vdash (x : \sigma_{\mathsf{id}}) : \alpha'$. We may then discharge both the context and the prefix and conclude that $\lambda(x : \sigma_{\mathsf{id}})\ x$ has type $\forall\,(\alpha \Rightarrow \sigma_{\mathsf{id}}, \alpha' \geq \sigma_{\mathsf{id}})\ \alpha \to \alpha'$. This captures all possible types—given the annotation. Retrospectively, we may see the annotation $(a : \sigma_{\mathsf{id}})$ as the application $(_ : \sigma_{\mathsf{id}})\ a$, where the notation $(_ : \sigma)$ stands for the expression $\lambda(x)\ (x : \sigma)$ and may be provided as a (collection of) primitive(s) with type scheme $\forall\,(\alpha \Rightarrow \sigma, \alpha' \geq \sigma)\ \alpha \to \alpha'$.

Fitting it together into XML$^{\mathsf{F}}$

The type system $\mathsf{XML}^{\mathsf{F}}$ we have devised so far does not fit directly into the Curry's style System G that does not permit *any* annotations on source terms. As type abstractions and type instantiation remain implicit in $\mathsf{XML}^{\mathsf{F}}$, the only new construction is, for the moment, a new form of abstraction $\lambda(x : \sigma)\ a$ where the parameter x is annotated with a type scheme σ. We shall see below how this construction can be explained in terms of a more atomic simple term annotation.

In fact, we restrict the bounds of rigid bindings to a subset of type schemes, ranged over by letter ρ, that correspond to System-F types as defined by the following grammar.

$$\begin{array}{ll}
\tau \in \mathcal{T}_{\mathsf{X}} ::= \alpha \mid \tau \to \tau & \mathsf{XML}^{\mathsf{F}} \text{ types} \\
\sigma \in \mathcal{S}_{\mathsf{X}} ::= \tau \mid \forall\,(q)\ \sigma \mid \bot & \mathsf{XML}^{\mathsf{F}} \text{ type schemes} \\
q \in \mathcal{Q}_{\mathsf{X}} ::= \alpha \geq \sigma \mid \alpha \Rightarrow \rho & \mathsf{XML}^{\mathsf{F}} \text{ bindings} \\
\rho \in \mathcal{R}_{\mathsf{X}} ::= \tau \mid \forall\,(\alpha \geq \bot)\ \rho \mid \forall\,(\alpha \Rightarrow \rho)\ \rho & \mathsf{F}\text{-like type schemes}
\end{array}$$

As a consequence, non-trivial flexible bounds may not appear under a rigid bound. This is only to keep $\mathsf{XML}^{\mathsf{F}}$ in exact correspondence with $\mathsf{IML}^{\mathsf{F}}$.

Of course, we must adapt the instance relation $\leq$ of $\mathsf{IML}^{\mathsf{F}}$ to an instance relation $\sqsubseteq$ on $\mathsf{XML}^{\mathsf{F}}$ type schemes. In fact, $\sqsubseteq$ is recursively defined together with a subrelation $\sqsubseteq$ that captures the notion of *type abstraction* mentioned above, which is the essential difference between $\mathsf{IML}^{\mathsf{F}}$ and $\mathsf{XML}^{\mathsf{F}}$. There is some degree of liberty in the definition of these two relations, which is discussed at the end of this section, while the precise definition can be found in §4.

From a typing point of view, we may hide type annotations into a collection of primitives $(_ : \rho)$ as suggested above and see $\lambda(x : \rho)\ a$ as syntactic sugar for $\lambda(x)\ \mathsf{let}\ x = (x : \rho)\ \mathsf{in}\ a$. This encoding will be explained in detail below. In short, it works as follows. On the one hand, the annotation on $(_ : \rho)$ requests its argument x to have type α where α abstracts the type scheme ρ. Thus the λ-bound variable x has type α, which is a monotype as requested. On the other hand, the annotation returns a value of (concrete) type σ as opposed to the abstract type α—we may say that it reveals the concrete type σ of α. Hence, the let-bound variable x has type σ and may be used within a with different instances. Of course, we may derive the following typing rule for annotated abstractions:

$$\textsc{Fun'}\ \frac{(Q)\ \Gamma, x : \rho \vdash a : \tau \qquad \alpha \notin \mathsf{ftv}(\tau)}{(Q)\ \Gamma \vdash \lambda(x : \rho)\ a : \forall\,(\alpha \Rightarrow \rho)\ \alpha \to \tau}$$

Following this approach, $\mathsf{XML}^{\mathsf{F}}$ remains an instance of System G—at least from a typechecking viewpoint.

However, this solution requires ρ to appear at a rigid occurrence in the type $\forall\,(\alpha \Rightarrow \rho)\ \forall\,(\alpha' \geq \rho)\ \alpha \to \alpha'$, which presents ρ to be an arbitrary type schemes σ. While this restriction is not a problem in practice, it does not allow to reach all of $\mathsf{IML}^{\mathsf{F}}$ programs.

Hence, we rather give a direct account of type annotations, moving slightly out of System G, and introduce the following typing rule:

$$\textsc{Annot}\ \frac{(Q)\ \Gamma \vdash a : \sigma' \qquad (Q)\ \sigma' \ni \sigma}{(Q)\ \Gamma \vdash (a : \sigma) : \sigma}$$

This allows all type schemes to be *explicitly* coerced along the inverse of type abstraction. There is still no surprise in the typing rules of XML^F: the power of XML^F lies in its types, the enforcement of a clear separation between types scheme and types, and the decomposition of the instance relation into the reversible, explicit abstraction relation and another irreversible, implicit subrelation.

The semantics of XML^F is given by translation into IML^F both dropping type annotations and inlining rigid bindings.

Design space

While $\leq$ is determined by the encoding of its types into sets of System-F types (given in §3.2), the relation $\sqsubseteq$ is only a subrelation of $\leq$. In fact, the difference cannot be explained without introducing the abstraction relation $\sqsubseteq$, which is itself a subrelation of $\sqsubseteq$. Abstraction is *implicitly* used to abstract (forget) type information, *i.e.* to replace a concrete type scheme σ by a type variable α that abstracts σ. Conversely, type annotations are *explicitly* used to reveal (actually, recover) the concrete type scheme that a variable abstracts over. Hence, we may also walk along the relation $\ni$ (the symmetric of $\sqsubseteq$) in XML^F but via explicit annotations.

In order for XML^F and IML^F to have the same expressiveness, the relations $\sqsubseteq$ and $\ni$ must be closely related to $\leq$. Precisely, $\leq$ and $(\sqsubseteq \cup \ni)^*$ must be equal. There remains some apparent degree of liberty in the choice of $\sqsubseteq$. However, the larger $\sqsubseteq$ is, the smaller $\ni$ need to be, *i.e.* the fewer explicitly annotations are requested, but simultaneously the harder type inference is. Indeed, type inference would be undecidable if $\sqsubseteq$ were $\leq$.

So, there is actually little choice for the definition of $\sqsubseteq$, except changing the relation $\leq$ itself—or considering arbitrary special cases. Interestingly, while $\mathsf{Full\ ML}^\mathsf{F}$ uses a richer set of types, the restriction of its type instance relation to ML^F types is exactly the type instance relation of ML^F. Hence, the increased confidence that we bring to the definition of $\sqsubseteq$ and $\sqsubseteq$ for $\mathsf{Plain\ ML}^\mathsf{F}$ also increases our confidence in their definition for $\mathsf{Full\ ML}^\mathsf{F}$.

2.4 $\mathsf{F}^{\mathsf{let}}$, the closure of System F with let-contraction

In this section, we introduce a new type system called $\mathsf{F}^{\mathsf{let}}$ that extends System F with let-bindings *a la ML*. This language lies between System F and ML^F and is used below as a mediator to relate them.

ML extends simple types with type schemes, which can be used to factor out all the simple types of an expression, and a specific rule for typechecking local bindings that takes advantage of type schemes. Namely, while typechecking $\mathsf{let}\ x = a\ \mathsf{in}\ a'$, the locally bound variables x may be assigned a type scheme $\forall\,(\alpha)\ \tau$ rather than a simple type τ, which amounts to assigning x the whole collection of simple types $\tau[\tau'/\alpha]$ where τ' ranges over all types. This is the essence of the ML type system. Its simplicity lies in the fact that type schemes may not be assigned to function parameters, hence, ML retains nearly the simplicity of simple types. Actually, it is well-known that an expression is typable in ML if (and only if) it is typable with simple types after reduction of all its local-bindings, *i.e.* the replacement of $\mathsf{let}\ x = a\ \mathsf{in}\ a'$ by $a'[a/x]$ in any context. This reduction always terminates but the size of the resulting expression may be exponentially larger than its source. Hence, this operational view of ML is inefficient. It is not very modular either. Thus, it is never used in practice. However, it provides ML with a very simple specification: ML is the closure of simple types by let-expansion[6].

Unfortunately, the empowering effect of the Let-Gen typing rule for local bindings becomes inoperative in (the generic presentation of) System F. That is, it does not allow more programs to be well-typed than by seeing local-bindings $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ an immediate applications $(\lambda(x)\ a_2)\ a_1$. This is not a weakness of System F. It simply follows from the fact that types are first-class (or, in our generic setting, that type schemes and types are identical).

One may then consider the alternative definition of ML—the closure of simple types by let-expansion—and apply it to System F. More precisely, we define $\mathsf{F}^{\mathsf{let}}$ as the smallest superset of System F that contains all terms $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ such that both a_1 and $a_2[a_1/x]$ are in $\mathsf{F}^{\mathsf{let}}$. The requirement that a_1 also be in $\mathsf{F}^{\mathsf{let}}$ is to reject terms such as $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ where x would not appear in a_2 and a_1 could be *any* expression.

[6]Formally, this is only true if we restrict local bindings $\mathsf{let}\ x = a\ \mathsf{in}\ a'$ to cases where x appears at most once in a', or if we define the closure more precisely, as done for the language $\mathsf{F}^{\mathsf{let}}$ below.

This definition is equivalent to adding the following typing rule to System F:

$$\textsc{Let-Expand}\quad \frac{(Q)\ \Gamma \vdash a_1 : \sigma_1 \qquad (Q)\ \Gamma \vdash a_2[a_1/x] : \sigma}{(Q)\ \Gamma \vdash \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 : \sigma}$$

As let-reduction may duplicate let-redexes but not create new ones, it must terminate, as implied by the Levy's finite development theorem for the λ calculus [Bar84]. That is, a term of $\mathsf{F}^{\mathsf{let}}$ may always be let-reduced to a term of System F.

As for ML, this operational specification is not quite satisfactory. Fortunately, there is also a more direct specification based on a very restricted use of intersection types, where intersections are only allowed in types schemes. Consider the following instance of generic types:

$$\begin{array}{ll} \tau \in \mathcal{T}_{\mathsf{F}^{\mathsf{let}}} ::= \alpha \mid \tau \to \tau \mid \forall\,(q)\ \tau & \mathsf{F}^{\mathsf{let}} \text{ types} \\ \sigma \in \mathcal{S}_{\mathsf{F}^{\mathsf{let}}} ::= \tau \mid \sigma \wedge \sigma & \mathsf{F}^{\mathsf{let}} \text{ type schemes} \\ q \in \mathcal{Q}_{\mathsf{F}^{\mathsf{let}}} ::= \alpha :: \star & \mathsf{F}^{\mathsf{let}} \text{ bindings} \end{array}$$

and the instance relation $\leqslant_{\mathsf{F}^{\mathsf{let}}}$ defined as the smallest transitive relation that treats $\wedge$ as associative, commutative, contains $\leq_{\mathsf{F}}$ and all pairs $\sigma \wedge \sigma' \leqslant_{\mathsf{F}^{\mathsf{let}}} \sigma$. Then, $\mathsf{F}^{\mathsf{let}}$ may be equivalently defined by the generic system $\mathsf{G}(\mathcal{T}_{\mathsf{F}^{\mathsf{let}}}, \mathcal{S}_{\mathsf{F}^{\mathsf{let}}}, \mathcal{Q}_{\mathsf{F}^{\mathsf{let}}}, \leqslant)$ extended with the following typing rule:

$$\textsc{Inter}\quad \frac{(Q)\ \Gamma \vdash a : \sigma_1 \qquad (Q)\ \Gamma \vdash a : \sigma_2}{(Q)\ \Gamma \vdash a : \sigma_1 \wedge \sigma_2}$$

This system is a particular case of the extension of System F with intersection types studied by Pierce [Pie91][7], which we refer to as $\mathsf{F}_\wedge$. The language $\mathsf{F}^{\mathsf{let}}$ is significantly weaker—but simpler—than $\mathsf{F}_\wedge$. However, to the best of our knowledge it has not be considered on its own.

Our main interest in $\mathsf{F}^{\mathsf{let}}$ is that although it has a very simple and intuitive specification and is only a small extension to System F, it is already a superset of ML^{F} as we shall see in the next section.

Type soundness of $\mathsf{F}^{\mathsf{let}}$

Type soundness relates the static semantics of programs, *i.e.* well-typedness, to their dynamic semantics, *i.e.* evaluation. In pure lambda-calculus where all values are functions, evaluation may never go wrong except by looping. Type soundness of System F ensures that well-typed programs are strongly normalizable.

However, most real languages allow loops or arbitrary recursion and contain interesting programs that may not terminate. In this setting, well-typedness cannot ensure termination any longer. Simultaneously, real languages also introduce non-functional values, and therefore other sources of errors such as applying a non-functional value to some argument. Well-typedness must then prevent such errors from happening.

In this paper we do not define the dynamic semantics of expressions. We do not address type soundness directly, but only indirectly by showing that well-typed expressions are also well-typed in $\mathsf{F}^{\mathsf{let}}$.

Type soundness of $\mathsf{F}^{\mathsf{let}}$ follows from type soundness of $\mathsf{F}_\wedge$. To the best of our knowledge, the type soundness of $\mathsf{F}_\wedge$, which is folklore knowledge has never been published. While proving type soundness for $\mathsf{F}^{\mathsf{let}}$ directly should not raise any difficulty, this is, by lack of space, out of the scope of this paper.

3 $\mathsf{IML}^{\mathsf{F}}$, Curry's style ML^{F}

In this section we study $\mathsf{IML}^{\mathsf{F}}$, that is $\mathsf{G}(\mathcal{T}_{\mathsf{I}}, \mathcal{S}_{\mathsf{I}}, \mathcal{Q}_{\mathsf{I}}, \leq)$ and, in particular, the type-instance relation $\leq$ introduced in §2.2. For that purpose, we define an interpretation of $\mathsf{IML}^{\mathsf{F}}$ types as sets of F types that induces a semantic definition of the type instance (§3.2). An equivalent but syntactic definition of type instance is given next (§3.3). We also provide an encoding of $\mathsf{IML}^{\mathsf{F}}$ terms into terms of $\mathsf{F}^{\mathsf{let}}$ (§3.6 and §3.5), which reduces type safety of $\mathsf{IML}^{\mathsf{F}}$ to that of $\mathsf{F}^{\mathsf{let}}$. The expressiveness and modularity of $\mathsf{IML}^{\mathsf{F}}$ are discussed in §3.7.

[7] The presentation of [Pie91] is in Church's style, but this is irrelevant here.

3.1 Types and Prefixes

Flexible bindings are the main novelty of IML^F. So as to be self-contained, we remind their definition here:

$$\tau \in \mathcal{T}_\mathsf{I} ::= \alpha \mid \tau \to \tau \mid \forall\,(\alpha \geq \bot)\ \tau \qquad \text{Types}$$
$$\sigma \in \mathcal{S}_\mathsf{I} ::= \tau \mid \forall\,(\alpha \geq \sigma)\ \sigma \mid \bot \qquad \text{Type Schemes}$$
$$q \in \mathcal{Q}_\mathsf{I} ::= \alpha \geq \sigma \qquad \text{Bindings}$$

The arrow is a type constructor. For simplification, it is the only type constructor but there is no difficulty in generalizing types with other type constructors. The types τ_1 and τ_2 in $\tau_1 \to \tau_2$ are called type arguments.

Type schemes are used as bounds for variables, which limits the way those variables may be instantiated. The special type scheme $\bot$ (read bottom) is the most general bound, also called the trivial bound. A variable with a trivial bound is said to be *unconstrained*. We define $\forall\,(\alpha)\ \sigma$ as syntactic sugar for $\forall\,(\alpha \geq \bot)\ \sigma$.

We recall that free (type) variables are defined in Section 2.1 (page 9).

Monotypes. A type is *monomorphic* if and only if it is a type variable or an arrow type (that is, of the form $\tau_1 \to \tau_2$ for some types τ_1 and τ_2). The set of monomorphic types is written $\mathcal{M}$. Intuitively, monomorphic types have no other instance but themselves (up to equivalence), *i.e.*, if τ is in $\mathcal{M}$ and τ is an instance of τ' then τ' is an instance of τ—and so equivalent to τ.

Polytypes. The set $\mathcal{T}_\mathsf{I}$ of IML^F types contains only types with trivial bounds $\bot$. Notice that $\bot$ is not a type but a type scheme. However, $\forall\,(\alpha)\ \alpha$, which as we shall see shortly, is equivalent to $\bot$, is a type.

Types of $\mathcal{T}_\mathsf{I}$ can be mapped to $\mathcal{T}_\mathsf{F}$ in a trivial way, just by exchanging the trivial bound with the unique kind $\star$. If τ is in $\mathcal{T}_\mathsf{I}$, we write $\lceil\tau\rceil$ for the counter-part of τ in $\mathcal{T}_\mathsf{F}$.

F-substitutions. We call *F-substitutions* and write θ for *idempotent* substitutions mapping type variables to F-types.

Renamings. A *renaming* is a finite bijective mapping from type variables to type variables. As usual, $\mathsf{dom}(\phi)$ is $\{\alpha \mid \phi(\alpha) \neq \alpha\}$. Note that if ϕ is a renaming, then $\mathsf{dom}(\phi)$ and $\mathsf{codom}(\phi)$ are equal and ϕ is a permutation of its domain. We extend renamings to bindings, taking $(\phi(\alpha) \geq \phi(\sigma))$ for $\phi(\alpha \geq \sigma)$ and to prefixes, taking $(\phi(q_i))^{\,i\in I}$ for $\phi(q_i{}^{\,i\in I})$.

Notice that if $\mathsf{dom}(\phi)$ is disjoint from $\mathsf{dom}(Q)$, then $\mathsf{dom}(\phi(Q))$ is equal to $\mathsf{dom}(Q)$ but $\phi(Q)$ is not, in general, equal to Q.

3.2 Interpretation of types and prefixes

Intuitively, the type scheme $\forall\,(\alpha \geq \sigma)\ \sigma'$ is meant to represent all types σ' where α is any instance of σ. We formalize this intuition by giving a formal interpretation of types and type schemes as sets of F-types. If S is a set of F-types, we write $\forall\,(\alpha)\ S$ for the set $\{\forall\,(\alpha)\ t \mid t \in S\}$.

Definition 3.2.1 (Semantics of types) The semantics of a type τ, written $\{\!\{\tau\}\!\}$ is the instance closure of its translation in System F, *i.e.* $\{t \in \mathcal{T}_\mathsf{F} \mid \lceil\tau\rceil \leq_\mathsf{F} t\}$. The semantics of type schemes, written $\{\!\{\sigma\}\!\}$, is recursively defined by $t \in \{\!\{\sigma\}\!\}$ if and only if σ is $\bot$ or of the form $\forall\,(\alpha \geq \sigma')\ \sigma''$ and t is of the form $\forall\,(\bar\beta)\ t''[t'/\alpha]$ with $\bar\beta \# \mathsf{ftv}(\sigma)$, $t' \in \{\!\{\sigma'\}\!\}$, and $t'' \in \{\!\{\sigma''\}\!\}$. □

Note that the semantics of $\bot$ and $\forall\,(\alpha)\ \alpha$ are both equal to $\mathcal{T}_\mathsf{F}$, as suggested earlier, although the former is only a type scheme and not a type. A type of the form $\forall\,(\alpha)\ \tau$ can be seen both as a type and as a type scheme. In the following lemma, we check that both views lead to the same interpretation.

Lemma 3.2.2 (Consistency) *For any type τ, the semantics of τ seen as a type and the semantics of τ seen as a type scheme are equal.*

Example 3.1 The interpretation of the polymorphic type σ_id, defined as $\forall\,(\alpha)\ \alpha \to \alpha$, is the set composed of all types of the form $\forall\,(\bar\alpha)\ t \to t$. In turn, the interpretation of $\forall\,(\alpha \geq \sigma_\mathsf{id})\ \alpha \to \alpha$ is the set composed of all types of the form $\forall\,(\bar\beta)\ (\forall\,(\bar\alpha)\ t \to t) \to (\forall\,(\bar\alpha)\ t \to t)$. Although both sides of the arrow may vary, they must do so in sync and always remain equal. Note also that $\forall\,(\bar\alpha)\ t \to t$ is not necessarily closed, hence the quantification over variables $\bar\beta$ in front.

Figure 4: Congruence.

IMLF-ALL-LEFT
$$\frac{(Q)\ \sigma_1\ \mathcal{R}\ \sigma_2}{(Q)\ \forall(\alpha \geq \sigma_1)\ \sigma\ \mathcal{R}\ \forall(\alpha \geq \sigma_2)\ \sigma}$$

IMLF-ALL-RIGHT
$$\frac{(Q, \alpha \geq \sigma)\ \sigma_1\ \mathcal{R}\ \sigma_2}{(Q)\ \forall(\alpha \geq \sigma)\ \sigma_1\ \mathcal{R}\ \forall(\alpha \geq \sigma)\ \sigma_2}$$

IMLF-ARROW
$$\frac{(Q)\ \tau_1\ \mathcal{R}\ \tau_2 \qquad (Q)\ \tau_1'\ \mathcal{R}\ \tau_2'}{(Q)\ \tau_1 \to \tau_1'\ \mathcal{R}\ \tau_2 \to \tau_2'}$$

The instance relation of System F can be pulled back into an instance relation in IML^F. However, as type instance is defined under prefixes, we must first give a meaning to prefixes.

In a typing judgment, a prefix is meant to capture the possible types that may be substituted for the variables in the domain of the prefix. Thus, the interpretation of a prefix is a set of substitutions. As usual, the composition operator is written $\circ$.

Definition 3.2.3 (Semantics of prefixes) The semantics of a prefix Q of the form $(\alpha_i \geq \sigma_i)^{i \in 1..n}$, written $\{\!\{Q\}\!\}$ is the set of all F-substitutions of the form $\circ_{i \in 1..n}(\alpha_i \mapsto t_i)$ where $t_i \in \{\!\{\sigma_i\}\!\}$ for i in $1..n$. As a particular case, $\{\!\{\emptyset\}\!\}$ is the singleton composed of the identity function. □

In fact, we may restrict to certain decompositions that are *canonical*.

Definition 3.2.4 A composition $\circ_{i \in 1..n}\theta_i$ of an idempotent substitution is *canonical* if all θ_i's are idempotent and have disjoint domains. □

Notice, that given the idempotence of individual substitutions and disjointness of their domains, the idempotence of the composition is then equivalent to the property $\mathsf{codom}(\theta_i) \# \mathsf{dom}(\theta_j)$ for all $i < j$.

Lemma 3.2.5 *Any member of a prefix of the form $\{\!\{Q_i^{i \in 1..n}\}\!\}$ has a canonical decomposition $\circ_{i \in 1..n}\theta_i$ with $\theta_i \in \{\!\{Q_i\}\!\}$.*

Canonical decompositions are interesting because any grouping (by associativity) is also canonical. Moreover, they enjoy the following property:

Lemma 3.2.6 *If $\theta_1 \circ \theta_2$ is a canonical decomposition of an idempotent substitution θ, then $\theta_2 \circ \theta = \theta \circ \theta_1 = \theta$.*

Definition 3.2.7 (Type Instance) The instance relation $\leq$ in IML^F is defined by $(Q)\ \sigma_1 \leq \sigma_2$ if for all $\theta \in \{\!\{Q\}\!\}$, we have $\theta\{\!\{\sigma_1\}\!\} \supseteq \theta\{\!\{\sigma_2\}\!\}$. The equivalence relation $\boxminus$ is the kernel of $\leq$. □

Notice that type instance is transitive, and as a result, $\boxminus$ is simply $(\leq) \cap (\geq)$. Expanding the definition, we have $(Q)\ \sigma_1 \boxminus \sigma_2$ if and only if for all $\theta \in \{\!\{Q\}\!\}$, we have $\theta\{\!\{\sigma_1\}\!\} = \theta\{\!\{\sigma_2\}\!\}$. We simply write $\sigma_1 \boxminus \sigma_2$, when $(\emptyset)\ \sigma_1 \boxminus \sigma_2$.

If $\lceil\tau_1\rceil$ and $\lceil\tau_2\rceil$ are equal, their semantics $\{\!\{\tau_1\}\!\}$ and $\{\!\{\tau_2\}\!\}$ are also equal, hence $\tau_1 \boxminus \tau_2$. (The only reason why τ_1 and τ_2 may not coincide exactly is that F-types are taken modulo equivalence.) We may thus invert the definition $\lceil\cdot\rceil$ into a function from $\mathcal{T}_\mathsf{F}$ to $\mathcal{T}_\mathsf{I}$. For any type t in $\mathcal{T}_\mathsf{F}$, we write $\lfloor t \rfloor$ the type τ of $\mathcal{T}_\mathsf{I}$ defined up to equivalence such that t is $\lceil\tau\rceil$. We extend the $\lfloor\cdot\rfloor$-mapping point-wise to F-substitutions, writing $\lfloor\theta\rfloor$ for the substitution $\alpha \mapsto \lfloor\theta(\alpha)\rfloor$.

Type instance may also be defined on prefixes as follows.

Definition 3.2.8 (Prefix instance) We say that Q_2 is an instance of Q_1 and we write $Q_1 \leq Q_2$ if $\{\!\{Q_1\}\!\} \supseteq \{\!\{Q_2\}\!\}$. □

3.3 Syntactic versions of instance and equivalence

The semantic definition of type instance is not constructive, as it involves quantification over infinite sets. A first step towards an algorithm for checking type instance is to provide an equivalent but syntactic definition of type instance.

While semantic type equivalence is defined after type instance, as its kernel, it is simple (and more intuitive) to define syntactic type equivalence first, and syntactic type instance as a larger relation containing type equivalence.

A judgment for a prefixed relation $\mathcal{R}$ is a triple written $(Q)\ \sigma_1\ \mathcal{R}\ \sigma_2$. It is *closed* if free type variables of σ_1 and σ_2 are included in $\mathsf{dom}(Q)$ and Q is closed. We also say that the prefix Q is *suitable* for the judgment.

Figure 5: Rules for syntactic type equivalence $\boxminus$.

FE-Comm
$$\frac{\alpha_1 \notin \mathsf{ftv}(\sigma_2) \qquad \alpha_2 \notin \mathsf{ftv}(\sigma_1)}{(Q)\ \forall\,(\alpha_1 \geq \sigma_1)\ \forall\,(\alpha_2 \geq \sigma_2)\ \sigma \boxminus \forall\,(\alpha_2 \geq \sigma_2)\ \forall\,(\alpha_1 \geq \sigma_1)\ \sigma}$$

FE-Free
$$\frac{\alpha \notin \mathsf{ftv}(\sigma)}{(Q)\ \forall\,(\alpha \geq \sigma')\ \sigma \boxminus \sigma}$$

FE-Mono
$$\frac{(\alpha \geq \tau) \in Q \qquad \tau \in \mathcal{M}}{(Q)\ \sigma \boxminus \sigma[\tau/\alpha]}$$

FE-Var
$$(Q)\ \forall\,(\alpha \geq \sigma)\ \alpha \boxminus \sigma$$

Definition 3.3.1 (Congruence) A relation $\mathcal{R}$ is $\geq$-*congruent* when it satisfies both Imlf-All-Left and Imlf-All-Right (Figure 4). It is *congruent* if it is $\geq$-congruent and moreover satisfies Imlf-Arrow (Figure 4). □

Lemma 3.3.2 *Type instance is $\geq$-congruent. Type equivalence is congruent.*

Definition 3.3.3 (Syntactic type equivalence) Syntactic type equivalence is the smallest reflexive, transitive, symmetric, and congruent relation on closed judgments satisfying the rules of Figure 5. □

Rule FE-Comm let independent binders commute; Rule FE-Free allows for removal of useless bindings; Rule FE-Mono allows for inlining of monomorphic bindings (monomorphic types are defined on page 18). Rule FE-Var identifies $\forall\,(\alpha \geq \sigma)\ \alpha$ and σ, *i.e.* it states that a (possibly polymorphic) type σ stands for all of its instances.

Interestingly, Rule FE-Mono allows for reification of a substitution $[\tau/\alpha]$ as a prefix $(\alpha \geq \tau)$. Actually, any substitution can be represented as a prefix. This is a key technical point that is used in the presentation of type inference and unification, where the solution of a unification problem is not a substitution but rather a prefix, which is more general (see [LBR03] or [LB04, Chapter 3] for more details).

Rules FE-Mono, FE-Free and FE-Var may be oriented from left to right and used as rewriting rules to transform every type scheme σ into a normal form, up to commutation of binders (see [LBR03] or [LB04, Chapter 1] for details).

Syntactic and semantic definitions of type equivalence coincide, as shown by Theorem 3.3.9 and Conjecture 3.3.13 below. Although these results will follow from similar results for type instance, they are easier to prove in the case of type equivalence, hence we prove them independently. We first establish a few lemmas. We remind the notation $\forall\,(\alpha)\ S$ introduced at the beginning of §3.2.

Lemma 3.3.4 *If $\alpha \notin \mathsf{ftv}(\sigma)$, then $\forall\,(\alpha)\ \{\!\{\sigma\}\!\} \subseteq \{\!\{\sigma\}\!\}$ holds.*

Lemma 3.3.5 *For any F-substitution θ and type scheme σ, we have $\theta(\{\!\{\sigma\}\!\}) \subseteq \{\!\{\lfloor\theta\rfloor(\sigma)\}\!\}$.*

The converse inclusion only holds under some hypotheses on the occurrences of free type variables of σ that are in $\mathsf{dom}(\theta)$, which must not be *exposed*.

Definition 3.3.6 (Exposed type variables) Exposed type variables in a type scheme σ are free type variables $\mathsf{etv}(\sigma)$ that are reachable from the root of σ without crossing an arrow, recursively defined as follows:

$$\mathsf{etv}(\alpha) = \alpha \qquad \mathsf{etv}(\tau \to \tau) = \emptyset \qquad \mathsf{etv}(\bot) = \emptyset \qquad \mathsf{etv}(\forall\,(\alpha \geq \sigma)\ \sigma') = (\mathsf{etv}(\sigma') \setminus \{\alpha\}) \cup \mathsf{etv}(\sigma)$$

□

For example α is exposed in $\forall\,(\beta \geq \alpha)\ \sigma$, but not in $\alpha \to \forall\,(\beta)\ \alpha$.

Lemma 3.3.7 *If α is not exposed in σ, then $\mathsf{etv}(\sigma) = \mathsf{etv}(\sigma[\tau/\alpha])$ holds for any τ.*

Our interest is more in type variables that are *not* exposed, as substituting them is sound.

Lemma 3.3.8 *Let σ be a type scheme and θ a substitution $[t/\alpha]$ such that either t is monomorphic or α is not exposed in σ. If $t' \in \{\!\{\lfloor\theta\rfloor(\sigma)\}\!\}$ and $\alpha \notin \mathsf{ftv}(t')$, then $t' \in \theta(\{\!\{\sigma\}\!\})$.*

Figure 6: Rules for type instance $\leq$.

$$\text{FI-EQUIV}\ \frac{(Q)\ \sigma_1 \boxminus \sigma_2}{(Q)\ \sigma_1 \leq \sigma_2} \qquad \text{FI-BOT}\ (Q)\ \bot \leq \sigma \qquad \text{FI-HYP}\ \frac{(\alpha \geq \sigma) \in Q}{(Q)\ \sigma \leq \alpha} \qquad \text{FI-SUBST}\ \frac{\alpha \notin \mathsf{etv}(\sigma)}{(Q)\ \forall\,(\alpha \geq \tau)\ \sigma \leq \sigma[\tau/\alpha]}$$

This lemma is stated in the particular case of a singleton substitution. This is only to simplify its presentation. Similarly, the condition $\alpha \notin \mathsf{ftv}(t')$ may always be satisfied by appropriate renaming of σ and θ. On the opposite, the two conditions on θ are more important and prevent cases where the lemma would not hold.

(Proof p. 44)

Lemma 3.3.9 (Soundness of syntactic equivalence) *The type equivalence relation satisfies all rules of Figure 5.*

(Proof p. 45)

Definition 3.3.10 (Syntactic Type instance) The syntactic instance relation is thjue smallest transitive and $\geq$-congruent relation on closed judgments satisfying rules of Figure 6. □

Rule FI-EQUIV ensures that type instance contains type equivalence. Rule FI-BOT states that $\bot$ is the most general type. Rule FI-HYP is obvious as a logical rule, as it just uses an hypothesis. Its effect is to replace a type scheme by a variable that stands for an instance of that type scheme. As it can be used repeatedly, its effect is often to join two flexible bindings that have the same bound (when used in combination with flexible-congruence rules). FI-SUBST inlines a flexible binding $(\alpha \geq \tau)$, whose bound τ is a type, thus settling the choice made for τ. Type scheme bounds must be instantiated to types using flexible congruence before they can be inlined. The side condition requires that variable α be not exposed in σ. This is to prevent cases where α would appear in σ as a flexible bound. For example, without the side condition, one could derive $(Q)\ \forall\,(\alpha \geq \tau)\ \forall\,(\beta \geq \alpha)\ \forall\,(\gamma \geq \alpha)\ \beta \to \gamma \leq \forall\,(\beta \geq \tau)\ \forall\,(\gamma \geq \tau)\ \beta \to \gamma$, which implies $(Q)\ \forall\,(\alpha \geq \tau)\ \alpha \to \alpha \leq \forall\,(\beta \geq \tau)\ \forall\,(\gamma \geq \tau)\ \beta \to \gamma$ and is certainly false: the semantics ensures that β and γ are substituted by the same instance of τ on the left-hand side, but not on the right-hand side. Remark that this condition seems to also prevent the case where σ is itself α and prevent the type instance $(Q)\ \forall\,(\alpha \geq \tau)\ \alpha \leq \tau$. However, this case is also a particular case of equivalence, and thus provable using rule FI-EQUIV.

Instance derivations can be renamed, prefixes can be extended, and substitutions by equivalent types preserve equivalence, as stated by the following lemma.

Lemma 3.3.11 *Assume $\mathcal{R}$ is $\boxminus$ or $\leq$.*

i) *The relation $\mathcal{R}$ satisfies both axioms* RENAMING *and* EXTRA-BINDINGS *of page 11.*

ii) *If $(Q)\ \tau_1 \boxminus \tau_2$ holds, $(Q)\ \sigma[\tau_1/\alpha] \boxminus \sigma[\tau_2/\alpha]$ holds.*

(Proof p. 45)

Lemma 3.3.12 (Soundness of syntactic instance) *The type instance relation satisfies all rules of Figure 6.*

Proof: Each rule is considered separately. Rule FI-EQUIV is by Lemma 3.3.9. Rule FI-BOT is by definition. For Rule FI-SUBST, it suffices to show $\{\!\{\sigma[\tau/\alpha]\}\!\} \subseteq \{\!\{\forall\,(\alpha \geq \tau)\ \sigma\}\!\}$. Let t be in $\{\!\{\sigma[\tau/\alpha]\}\!\}$. We may assume that $\alpha \notin \mathsf{ftv}(t)$, *w.l.o.g.* Besides, by hypothesis, α is not exposed in σ. Thus, by Lemma 3.3.8, we have $t \in \{\!\{\sigma\}\!\}[\lceil\tau\rceil/\alpha]$. This implies $t \in \{\!\{\forall\,(\alpha \geq \tau)\ \sigma\}\!\}$ by Definition 3.2.1, which is as expected. For FI-HYP, we assume $(\alpha \geq \sigma) \in Q$, *i.e.* Q of the form $(Q_1, \alpha \geq \sigma, Q_2)$. Assume $\theta \in \{\!\{Q\}\!\}$. By Definition 3.2.3, θ is of the form $\theta_1 \circ [t/\alpha] \circ \theta_2$. with $t \in \{\!\{\sigma\}\!\}$ and the decomposition is canonical. By Lemma 3.2.6, $[t/\alpha] \circ \theta_2$ is equal to $[t/\alpha] \circ \theta_2 \circ [t/\alpha]$. Composing by θ_1 on the left, we get that θ is equal to $\theta \circ [t/\alpha]$. Therefore $\theta(\alpha)$ is $\theta(t)$, which implies $\theta(\alpha) \in \theta(\{\!\{\sigma\}\!\})$. Thus $(Q)\ \sigma \leq \alpha$ holds, as expected.

An obvious question is whether the syntactic definitions of type equivalence and type instance are complete for the corresponding semantic definitions.

Conjecture 3.3.13 (Completeness of syntactic relations) *Any type equivalence relation can be derived with rules of Figure 5. Any type instance relation can be derived with rules of Figure 6.*

A proof of this conjecture has only been sketched, using an intermediate semantics of types based on a graphic representation to factor all of their instances as a single and simple object—see discussion in §5. While showing the equivalence of the two semantics should not be difficult, a description of the graph representation of types is beyond the scope of this paper (See [RY07]) and a direct proof using the semantics of this paper without referring to graphs would be too long and tedious.

This conjecture combined with Lemma 3.3.9 justifies our semantics of types. In the rest of the paper we do not distinguish between syntactic and semantic relations and only say *type equivalence* or *type instance*. However, we do not actually rely on this conjecture: we only use the soundness of the syntactic definitions with respect to their semantics definitions, not their completeness. Therefore, reading *type equivalence* and *type instance* as the syntactic versions hereafter is always technically correct and never relies on the conjecture.

3.4 Typing rules

As IML^F is an instance of System G, its typing rules are as described in Figure 3. Some examples of typings have been introduced informally in §1. Other examples will be presented with more details in Section 3.7.

There is an interesting admissible rule in IML^F that helps typing abstractions:

$$\textsc{Imlf-Fun}^\star \quad \frac{(Q)\ \Gamma, x : \tau \vdash a : \sigma \qquad \alpha \notin \mathsf{ftv}(\tau)}{(Q)\ \Gamma \vdash \lambda(x)\ a : \forall\,(\alpha \geq \sigma)\ \tau \to \alpha}$$

To see this, assume $(Q)\ \Gamma, x : \tau \vdash a : \sigma$. Let α' be a variable that does not appear free in (Q). We have $(Q, \alpha' \geq \sigma)\ \Gamma, x : \tau \vdash a : \sigma$ (**1**) by lemma 2.1.1.ii. We have $(Q, \alpha' \geq \sigma)\ \sigma \leq \alpha$ (**2**) by FI-Hyp. By Rule Inst with (1) and (2), we get $(Q, \alpha' \geq \sigma)\ \Gamma, x : \tau \vdash a : \alpha$. We conclude by Rule Fun followed by Rule Gen and renaming.

Generalized application

Rule Gen generalizes a binding by moving a binding from the prefix to the right-hand side. The converse rule is in fact admissible.

$$\textsc{UnGen}^\star \quad \frac{(Q)\ \Gamma \vdash a : \forall\,(\alpha \geq \sigma)\ \sigma' \qquad \alpha \notin \mathsf{dom}(Q)}{(Q, \alpha \geq \sigma)\ \Gamma \vdash a : \sigma'}$$

Indeed, assume $(Q)\ \Gamma \vdash a : \forall\,(\alpha \geq \sigma)\ \sigma'$ holds. Then, by Lemma 2.1.1.ii, we get $(Q, \alpha \geq \sigma)\ \Gamma \vdash a : \forall\,(\alpha \geq \sigma)\ \sigma'$. We conclude by Rule Inst and $(Q, \alpha \geq \sigma)\ \forall\,(\alpha \geq \sigma)\ \sigma' \leq \sigma'$, as shown below:

$$\begin{array}{llll} (Q, \alpha \geq \sigma) \quad \forall\,(\alpha \geq \sigma)\ \sigma' & = & \forall\,(\beta \geq \sigma)\ \sigma'[\beta/\alpha] & \text{by } \alpha\text{-conversion} \\ & \leq & \forall\,(\beta \geq \alpha)\ \sigma'[\beta/\alpha] & \text{by FI-Hyp} \\ & & & \text{and Imlf-All-Left} \\ & \boxminus & \sigma' & \text{by FE-Mono} \end{array}$$

More interestingly, the following generalized application rule is also admissible—it is actually derivable with repeated applications of admissible Rule UnGen* on both premises, and an application of rule App followed by an application of rule Gen:

$$\textsc{App}^\star \quad \frac{(Q)\ \Gamma \vdash a_1 : \forall\,(Q')\ \tau_2 \to \tau_1 \qquad (Q)\ \Gamma \vdash a_2 : \forall\,(Q')\ \tau_2}{(Q)\ \Gamma \vdash a_1\ a_2 : \forall\,(Q')\ \tau_1}$$

We refer to typing rules extended with App* as generalized typing rules. We presented the system with Rule App, rather than Rule App* for economy of the formalization as well as for emphasizing the generic presentation of the type system. However, the generalized typing rules, while defining the same judgments allow for more derivations and so have actually more interesting modularity properties. In particular, we use the generalized presentation in §3.7.

Figure 7: Translating IML^F to F^let.

IMLF-VAR-LET
$$\frac{x : \sigma \in \Gamma}{(Q \ni \theta)\ \Gamma \vdash x : \sigma \ni t \Rightarrow (x : t)}$$

IMLF-VAR-FUN
$$\frac{y : \tau \in \Gamma}{(Q \ni \theta)\ \Gamma \vdash y : \tau \ni t \Rightarrow \emptyset}$$

IMLF-FUN
$$\frac{(Q \ni \theta)\ \Gamma, y : \tau \vdash a : \tau' \ni \lceil \tau' \rceil \Rightarrow \Delta}{(Q \ni \theta)\ \Gamma \vdash \lambda(y)\ a : \tau \to \tau' \ni \lceil \tau \to \tau' \rceil \Rightarrow \Delta}$$

IMLF-APP
$$\frac{(Q \ni \theta)\ \Gamma \vdash a_1 : \tau_2 \to \tau_1 \ni \lceil \tau_2 \to \tau_1 \rceil \Rightarrow \Delta_1 \qquad (Q \ni \theta)\ \Gamma \vdash a_2 : \tau_2 \ni \lceil \tau_2 \rceil \Rightarrow \Delta_2}{(Q \ni \theta)\ \Gamma \vdash a_1\ a_2 : \tau_1 \ni \lceil \tau_1 \rceil \Rightarrow \Delta_1 \wedge \Delta_2}$$

IMLF-INST
$$\frac{(Q \ni \theta)\ \Gamma \vdash a : \sigma \ni t \Rightarrow \Delta \qquad (Q)\ \sigma \le \sigma'}{(Q \ni \theta)\ \Gamma \vdash a : \sigma' \ni t \Rightarrow \Delta}$$

IMLF-GEN
$$\frac{\alpha \notin \mathsf{ftv}(\Gamma) \qquad \bar\beta \# \mathsf{dom}(Q) \qquad (Q, \alpha \ge \sigma \ni \theta \circ [t/\alpha])\ \Gamma \vdash a : \sigma' \ni t' \Rightarrow \Delta}{(Q \ni \theta)\ \Gamma \vdash a : \forall(\alpha \ge \sigma)\ \sigma' \ni \forall(\bar\beta)\ t'[t/\alpha] \Rightarrow \Delta}$$

IMLF-LET-0
$$\frac{x \notin \mathsf{ftv}(a') \qquad (Q \ni \theta)\ \Gamma \vdash a : \sigma \ni t \Rightarrow \Delta \qquad (Q \ni \theta)\ \Gamma, x : \sigma \vdash a' : \sigma' \ni t' \Rightarrow \Delta'}{(Q \ni \theta)\ \Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \sigma' \ni t' \Rightarrow \Delta \wedge \Delta'}$$

IMLF-LET
$$\frac{((Q \ni \theta)\ \Gamma \vdash a : \sigma \ni t_i \Rightarrow \Delta_i)^{i \in I} \qquad I \neq \emptyset \qquad (Q \ni \theta)\ \Gamma, x : \sigma \vdash a' : \sigma' \ni t' \Rightarrow \Delta, x : \wedge(t_i)^{i \in I}}{(Q \ni \theta)\ \Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \sigma' \ni t' \Rightarrow \Delta \wedge (\wedge {\Delta_i}^{i \in I})}$$

3.5 System F as a subset of (Simple) IML^F

We recall that Simple ML^F is IML^F without terms with local-bindings. Before showing the inclusion of System F in Simple ML^F, we show the inclusion of their instance relations.

Lemma 3.5.1 *Assume $t \le_F t'$ holds. Then, $(Q)\ \lfloor t \rfloor \le \lfloor t' \rfloor$ holds under any suitable prefix Q.*

(Proof p. 46)

The next lemma states the inclusion of System F into Simple ML^F.

The translation $\lfloor A \rfloor$ of a typing context A into one of Simple ML^F, is the pointwise translation of types, *i.e.* $\lfloor x : t \rfloor$ is $x : \lfloor t \rfloor$.

Theorem 1 *Assume $F :: \Gamma \vdash a : t$ holds. Then, the judgment $\mathit{IML}^F :: (Q)\ \lfloor \Gamma \rfloor \vdash a : \lfloor t \rfloor$ holds under any prefix Q binding the free variables of Γ and t.*

(Proof p. 46)

3.6 Type soundness, by viewing IML^F as a subset of F^{let}

In this section, we show that IML^F is a subset of and thus as sound as F^{let} by translating typing derivations of IML^F into typing derivations of F^{let}. For that purpose, we instrument typing judgments of IML^F $(Q)\ \Gamma \vdash a : \sigma$ into judgments of the form $(Q \ni \theta)\ \Gamma \vdash a : \sigma \ni t \Rightarrow \Delta$ to mean "given an F-substitution θ in $\{\!\{Q\}\!\}$, and a type t in $\{\!\{\sigma\}\!\}$, the judgment $(Q)\ \Gamma \vdash a : \sigma$ requires a context Δ". These judgments may also be read as an algorithm that takes θ, t, and a typing derivation of a (regular) typing judgment IML$^F :: (Q)\ a \vdash \sigma$ and returns a context Δ. These judgments are defined by typing rules of Figure 7.

In the translation, we distinguish let-bound variables, written x, from λ-bound variables, written y. Hence, the two rules IMLF-VAR-LET and IMLF-VAR-FUN corresponding to the original rule IMLF-VAR. Notice that only the IMLF-VAR-LET inserts a binding in Δ. The context Δ maps let-bound variables to intersection types, written $\wedge t_i^{i \in I}$. We write $\Delta_1 \wedge \Delta_2$ for the environment that maps x to $\Delta_1(x) \wedge \Delta_2(x)$ when x is in both $\mathsf{dom}(\Delta_1)$ and $\mathsf{dom}(\Delta_2)$ or as Δ_1 or Δ_2 when x is in either $\mathsf{dom}(\Delta_1)$ or $\mathsf{dom}(\Delta_2)$. There are two rules for local-bindings. Rule IMLF-LET assumes that variable x appears free in a'. The bound expression is typechecked as many times as there are occurrences of x in a', which enables each occurrence to pick a different instance t of σ via rule IMLF-VAR-LET. Rule IMLF-LET-0 is for the degenerate case where x does not appear free in a'. We must still typecheck the premise once, so as to be ensured that a is well-typed—since in a call-by-value a is evaluated even if its result is to be discarded. Other rules are straightforward. By convention, all judgments $(Q \ni \theta)\ \Gamma \vdash a : \sigma \ni t \Rightarrow \Delta$ carry the implicit side-conditions $\theta \in \{\!\{Q\}\!\}$ and $\theta(t) \in \theta(\{\!\{\sigma\}\!\})$.

The following lemma justifies our suggestion to read these judgments as an algorithm.

Lemma 3.6.1 *If the judgment* $(Q)\ \Gamma \vdash a : \sigma$ *holds, then for any* θ *in* $\{\!\{Q\}\!\}$ *and* t *in* $\{\!\{\sigma\}\!\}$*, there exists a context* Δ *such that* $(Q \ni \theta)\ \Gamma \vdash a : \sigma \ni t \Rightarrow \Delta$ *holds.*

Notice that, by construction, neither $\{\!\{Q\}\!\}$ nor $\{\!\{\sigma\}\!\}$ may be empty.

(Proof p. 46)

The next two lemmas show the soundness of IML^F by translation into F^let, which is itself sound. We define $\{\!\{\Gamma\}\!\}$ as $\{y : \lceil\tau\rceil \mid y : \tau \in \Gamma\}$. Notice that bindings with true type schemes are not in $\{\!\{\Gamma\}\!\}$; they will be replaced by bindings with conjunctive types in some additional environment Δ.

Lemma 3.6.2 *If the judgment* $(Q \ni \theta)\ \Gamma \vdash a : \sigma \ni t \Rightarrow \Delta$ *holds, then* $F^{let} :: \theta(\{\!\{\Gamma\}\!\}), \Delta \vdash a : \theta(t)$ *holds.*

(Proof p. 46)

Theorem 2 *IML^F is a subset of F^{let}. More precisely, if the judgment $\mathit{IML}^\mathsf{F} :: (Q)\ \Gamma \vdash a : \sigma$ holds, then for any θ in $\{\!\{Q\}\!\}$ and τ in $\{\!\{\sigma\}\!\}$, there exists a context Δ such that the judgment $F^{let} :: \theta(\{\!\{\Gamma\}\!\}), \Delta \vdash a : \theta(t)$ holds.*

(Proof p. 46)

Type Soundness Type soundness is a corollary of Theorem 2, as it ensures that IML^F is as safe as F^let, which is itself as safe as $\mathsf{F}_\wedge$, which is safe.

Discussion As a particular case of the previous lemma, Simple ML^F is a subset of System F, since terms of F^let without local bindings are also in F. The converse is also true, as shown in the previous section. In particular, Simple ML^F and F coincide—regarding the sets of typable terms. We may summarize this section and the previous one with the inclusions

$$\text{Simple } \mathsf{IML}^\mathsf{F} \subseteq \mathsf{F} \subseteq \text{Simple } \mathsf{IML}^\mathsf{F} \subset \mathsf{IML}^\mathsf{F} \subset \mathsf{F}^\mathsf{let}.$$

We already know that at least one of the two last inclusions is strict, as the term a^{auto}_{IK} defined as $\mathsf{let}\ y = \lambda(x)\ x\ x\ \mathsf{in}\ K\ (y\ I)\ (y\ K)$ is in F^let (as it let-reduces to a term in F) but not in F [GR88]. (Notice that a^{auto}_{IK} is in any higher-order extension F^n of F for $n > 2$, hence also in F^ω.)

In fact, the two inclusions are strict. We shall exhibit an example that is typable in ML^F but not in System F in the following subsection. We now argue (informally) that a is not in ML^F.

Intuitively, Rule Inter is much more powerful than Imlf-Let since a conjunctive type may be an arbitrary (finite) set of types in Inter, whereas Imlf-Let only allows to form conjunctions between types that belong to a common type scheme. To see that a is not in ML^F, we may reproduce the argument used for F [GR88] by analyzing all possible derivations of a in ML^F. In fact, the parameter y will be assigned a type σ that must be a type for $\lambda(x)\ x\ x$. In turn, as x is used polymorphically, it must be assigned an exact type, hence σ is of the form $\forall\,(\alpha_1 \Rightarrow \sigma_1)\ \forall\,(\alpha_2 \diamond \sigma_2)\ \alpha_1 \to \alpha_2$ where σ_1 must be a System-F type $\lfloor t \rfloor$. Reproducing the same reasoning as in [GR88] (see also [Pie02, §23] or [Wel99]), t must be of form $\forall\,(\alpha)\ .(\ldots(\alpha \to t_n) \to \ldots t_1) \to t_0$. However, no type of this form can be simultaneously a type for I and K, as required by the two uses of y. In fact, the term a is not typable in Full ML^F either.

Notice that the translation only introduces intersection types $\wedge t^I_i$ such that there exists an ML^F type scheme σ of which all t_i's are instances. For example, an intersection type of the form $(\forall\,(\alpha)\ \alpha \to \alpha) \wedge (\forall\,(\alpha)\ \forall\,(\beta)\ \alpha \to \beta \to \alpha)$ will never be used.

Subject reduction The subject reduction property holds in Simple IML^F as a consequence of the two-directional encoding between System F and Simple IML^F.

We expect subject reduction to hold in IML^F, since it holds in Full ML^F [LB04]. However, we did not check this result, as type soundness could easily be established by encoding IML^F into F^let.

3.7 Expressiveness and modularity

As typing derivations of System F can be mapped directly to typing derivations of IML^F, the language IML^F performs at least as well as System F with regard to typechecking. We claim that IML^F is strictly more expressive than System F in a rather unusual and weak but practically meaningful sense, as it is more modular than System F. For that purpose, we exhibit an unannotated expression a (for sake of conciseness, we use

constants in expressions) that is typable in System F, hence also in ML^F. However, we argue that a single, small and local change in a induces many changes in its typing derivation in System F (in fact, changes are proportional to the size of a), while changes needed in its typing derivation in ML^F remain small and local.

We actually consider the generalized presentation of ML^F, using Rule App* rather than App. We could also argue in the original system as well, but with a more careful definition of modularity. For fairness of comparison, we consider the implicit version of System F, indeed. The result can only be (significantly) worsened in explicit System F, as not only typing derivations will have to be changed, but the type abstractions and type applications in the source programs as well.

Although our statement is based on a particular example—we do not actually prove that changes in the derivation *must* be non local but argue informally—it is also seconded by formal results in Le Botlan's thesis [LB04] where it is shown how a single type of ML^F captures all type abstractions and type applications of a given expression in System F. However, this formal result uses the principal type property of ML^F, which we do not show here, as we do not address type inference—hence, our informal, but simpler explanation.

The following example emphasizes once again that only type annotations on functions parameters need to be kept in ML^F programs and all other type information can be reconstructed in a principal manner.

In fact, we exhibit a sequence of expressions $(a^n)_{n\in N}$ of increasing size, defined inductively. So as to ease the presentation, we assume that the core language is extended with a ground type i (such as int) and that the initial environment Γ_0 contains the functions id, eq, auto and comp that satisfy the following signature in ML^F (their signature in System F follows by translation).

$$\begin{array}{rcll} \mathsf{id} & : & \forall\,(\alpha)\ \alpha \to \alpha & \triangleq \sigma_{\mathsf{id}} \\ \mathsf{eq} & : & \forall\,(\alpha)\ \alpha \to \alpha \to \alpha & \\ \mathsf{auto} & : & \sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}} & \triangleq \sigma_{\mathsf{auto}} \\ \mathsf{comp} & : & (\mathsf{i} \to \mathsf{i}) \to (\mathsf{i} \to \mathsf{i}) & \triangleq \sigma_{\mathsf{comp}} \end{array}$$

For instance, auto and comp could be the expressions $\lambda(x:\sigma_{\mathsf{id}})\ x\ x$ and $\lambda(g)\ (\lambda(x)\ g\ (\mathsf{succ}\ x))$ where succ is the successor function for integers.

We now define a sequence of expressions a^n and a sequence of types σ^n parameterized by an initial expression a for a^0 and an initial closed type σ for σ^0.

$$a^0 \triangleq a \qquad a^{n+1} \triangleq \mathsf{eq}\ a^n \qquad \sigma^0 \triangleq \sigma \qquad \sigma^{n+1} \triangleq \forall\,(\alpha \geq \sigma^n)\ \alpha \to \alpha$$

We have the following derivation in ML^F :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(\alpha \geq \sigma^n)\ \Gamma_0 \vdash \mathsf{eq} : \alpha \to \alpha \to \alpha}\ \textsc{Inst} \qquad \dfrac{\dfrac{\boxed{\Gamma_0 \vdash a^n : \sigma^n}}{(\alpha \geq \sigma^n)\ \Gamma_0 \vdash a^n : \alpha}\ \textsc{Context} \qquad (\alpha \geq \sigma^n)\ \sigma^n \leq \alpha}{(\alpha \geq \sigma^n)\ \Gamma_0 \vdash a^n : \alpha}\ \textsc{Inst}}{(\alpha \geq \sigma^n)\ \Gamma_0 \vdash \mathsf{eq}\ a^n : \alpha \to \alpha}\ \textsc{App}}{\boxed{\Gamma_0 \vdash a^{n+1} : \sigma^{n+1}}}\ \textsc{Gen}$$

Hence, by induction, we have $\Gamma_0 \vdash a^n : \sigma^n$ for all n whenever $\Gamma_0 \vdash a : \sigma$. Observe that, by construction, we have $\Gamma_0 \vdash \mathsf{id} : \sigma_{\mathsf{id}}$ (**1**), $\Gamma_0 \vdash \mathsf{comp} : \sigma_{\mathsf{comp}}$ (**2**), and $\Gamma_0 \vdash \mathsf{auto}$: σ_{auto} (**3**).

Assume moreover that $\sigma_{\mathsf{id}} \leq \sigma$. Using Rule Imlf-All-Left repeatedly, we may show that $\sigma_{\mathsf{id}}^n \leq \sigma^n$. In particular, $\sigma_{\mathsf{id}}^{n+1} \leq \forall\,(\alpha \geq \sigma^n)\ \alpha \to \alpha$ (**4**). Let b be id^{n+1}. We have $\Gamma \vdash b : \sigma_{\mathsf{id}}^{n+1}$, which gives $\Gamma \vdash b : \forall\,(\alpha \geq \sigma^n)\ \alpha \to \alpha$ by Rule Inst and (4). Using generalized Rule App* we have $\Gamma_0 \vdash b\ a^n : \sigma^n$ (**5**). As both σ_{auto} and σ_{comp} are instances of σ_{id}, we may thus conclude that both applications $b\ \mathsf{auto}^n$ and $b\ \mathsf{comp}^n$ are typable in ML^F. More importantly, the typing derivations of b are the same for both terms—only the typing of the arguments and final application differs. The key point here is that the instantiation of the type of b may be delayed as much as possible. This is possible only because of the expressiveness of types and of the instance relation of ML^F.

In System F, both applications are typable as well, but unlike in ML^F, the typing derivations of b are significantly different. In particular, each node of the typing derivation tree differs up to the leaves, *i.e.* up to the typing of the expression id. Indeed, the type applications required at each application node are always different in both derivations. We see that a single difference in the unannotated term occurring at an arbitrary depth in the argument of the application $\mathsf{id}^{n+1}\ \mathsf{auto}^n$ (compared to $\mathsf{id}^{n+1}\ \mathsf{comp}^n$) induces changes in the typing of the body of the function a_{n+1} up to depth n.

It follows that expressions a^n_{let} defined as $\mathsf{let}\ x = \mathsf{id}^{n+1}\ \mathsf{in}\ \mathsf{let}\ z = x\ \mathsf{auto}^n\ \mathsf{in}\ x\ \mathsf{comp}^n$ and $\mathsf{let}\ x = \mathsf{eq}\ \mathsf{id}\ \mathsf{in}\ \mathsf{let}\ z = x\ \mathsf{auto}\ \mathsf{in}\ x\ \mathsf{comp}$ as a particular case are not typable in System F. There are all typable in IML^F, indeed.

Notice that although $\mathsf{F}^{\mathsf{let}}$ is larger than IML^F, it is not necessarily better: while a^n_{let} is also typable in $\mathsf{F}^{\mathsf{let}}$ its typing derivation in $\mathsf{F}^{\mathsf{let}}$ is still problematic as the typing derivation for id contains two similar sub-derivations specialized for auto and comp, respectively, and joined with $\wedge$ rather than a principal typing derivation independent of further applications, as could be done IML^F. The analysis of the open world modular problem described by a_n when n increases is more informative about modularity than that of the closed expression a^n_{let}.

Remark also that a^n_λ defined as $\mathsf{app}\ \ (\lambda(x)\ \mathsf{let}\ z = x\ \mathsf{auto}^n\ \mathsf{in}\ x\ \mathsf{comp}^n)\ \mathsf{id}^{n+1}$ does not typecheck in IML^F, as the argument id^{n+1} must be assigned a type scheme and not a type. However, this example typechecks in $\mathsf{Full\ ML}^\mathsf{F}$.

In summary, the implicitly typed system IML^F that typechecks more programs than System F and may typecheck them more modularly. The main benefit of IML^F over System F is that its types are more principal, so that typing derivations of IML^F are more modular than typing derivations in System F. This is a key for the design of XML^F that permits simple type inference.

4 XML^F, Church's style ML^F

In this section, we step on modular typechecking properties of IML^F to design a version with optional type annotations, called XML^F, that has a clear and intuitive specification of *where* and *when* to put type annotations. After a presentation of XML^F types (§4.1) and relations between them, we introduce typing rules and show type safety (§4.2) by translation of well-typed programs into IML^F. We also exhibit a translation of System F into XML^F, which shows its expressiveness, as well as its economy of explicit type information (§4.3). We show that XML^F is a conservative extension to ML (§4.4). Finally, we argue on an example that XML^F does not infer polymorphism (§4.5), as claimed earlier.

Specifying where and when to put type annotations

In IML^F, no type annotation is ever given. As a consequence, type inference is undecidable, just like in implicit System F. In order to make type inference decidable, we need some annotations to be mandatory. Our guideline is:

> *Only function parameters that are used polymorphically need an annotation.*

This implies that types of annotated arguments be distinguishable from those of unannotated arguments. The solution is to have two different ways of representing a given type: one for explicit type information and another one for inferred type information. Unlike previous works, no explicit coercion is however needed to cast the former into the latter. Types that are explicitly introduced with type annotations are represented directly with a type scheme σ, as usual. On the contrary, types that have been inferred are represented indirectly via a variable α that is rigidly bound to a type scheme σ in the prefix. This means that α stands for the type σ, but α may not be freely replaced by σ or an instance of σ. Still, values of type α can be merged with other values of type σ, by "weakening them to the abstract type α"—and not conversely. This operation, called *abstraction* and written $\sqsubseteq$, plays a crucial role with respect to type inference. The converse relation, implicitly recasting an abstract variable α to its bound σ is not allowed, as it would allow—and hence force—type inference to guess polymorphic types and, as a result, make it undecidable. However, as this operation is always sound, it may be performed explicitly via a type annotation.

4.1 Types, prefixes and relations under prefix

We remind the definition of types and prefixes below, so as to make this section self-contained:

$$\tau \in \mathcal{T}_\mathsf{X} ::= \alpha \mid \tau \to \tau \qquad \mathsf{XML}^\mathsf{F}\text{ types}$$
$$\sigma \in \mathcal{S}_\mathsf{X} ::= \tau \mid \forall\,(q)\ \sigma \mid \bot \qquad \mathsf{XML}^\mathsf{F}\text{ type schemes}$$
$$q \in \mathcal{Q}_\mathsf{X} ::= \alpha \geq \sigma \mid \alpha \Rightarrow \rho \qquad \mathsf{XML}^\mathsf{F}\text{ bindings}$$
$$\rho \in \mathcal{R}_\mathsf{X} ::= \tau \mid \forall\,(\alpha \geq \bot)\ \rho \mid \forall\,(\alpha \Rightarrow \rho)\ \rho \qquad \mathsf{F}\text{-like type schemes}$$

Figure 8: Congruence in XML^F.

$$\textsc{Xmlf-Flex-Left}\ \frac{(Q)\ \sigma_1\ \mathcal{R}\ \sigma_2}{(Q)\ \forall\,(\alpha \geq \sigma_1)\ \sigma\ \mathcal{R}\ \forall\,(\alpha \geq \sigma_2)\ \sigma} \qquad \textsc{Xmlf-All-Left}\ \frac{(Q)\ \sigma_1\ \mathcal{R}\ \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma_1)\ \sigma\ \mathcal{R}\ \forall\,(\alpha \diamond \sigma_2)\ \sigma} \qquad \textsc{Xmlf-All-Right}\ \frac{(Q, \alpha \diamond \sigma)\ \sigma_1\ \mathcal{R}\ \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1\ \mathcal{R}\ \forall\,(\alpha \diamond \sigma)\ \sigma_2}$$

Figure 9: Equivalence in XML^F.

$$\textsc{Eq-Comm}\ \frac{\alpha_1 \notin \mathsf{ftv}(\sigma_2) \qquad \alpha_2 \notin \mathsf{ftv}(\sigma_1)}{(Q)\ \forall\,(\alpha_1 \diamond_1 \sigma_1)\ \forall\,(\alpha_2 \diamond_2 \sigma_2)\ \sigma \equiv \forall\,(\alpha_2 \diamond_2 \sigma_2)\ \forall\,(\alpha_1 \diamond_1 \sigma_1)\ \sigma} \qquad \textsc{Eq-Free}\ \frac{\alpha \notin \mathsf{ftv}(\sigma)}{(Q)\ \forall\,(\alpha \diamond \sigma')\ \sigma \equiv \sigma} \qquad \textsc{Eq-Mono}\ \frac{(\alpha \diamond \tau) \in Q}{(Q)\ \sigma \equiv \sigma[\tau/\alpha]}$$

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

As in ML, monotypes are simple types, for the purpose of type inference. This contrasts with System F or IML^F, for which type inference is not considered.

We use the symbol $\diamond$ as a meta-variable that denotes either $\geq$ or $=$. For example, $(\alpha \diamond \sigma)$ stands for either $(\alpha \Rightarrow \sigma)$ or $(\alpha \geq \sigma)$, which are called *rigid* (resp. flexible) *bindings*. We also say that α is *rigidly* (resp. *flexibly*) bound. A prefix that contains only rigid bindings is called *rigid*. The *rigid domain* of a prefix Q, written $\mathsf{dom}_=(Q)$, is the set of α such that α is rigidly bound in Q.

Notice that XML^F types do not form a superset of IML^F types, since for type inference purposes, they cannot have quantifiers. For instance, the IML^F type $(\forall\,(\alpha_1)\ \tau_1) \to (\forall\,(\alpha_2)\ \tau_2)$ is not an XML^F type. However, it can be represented as the XML^F type $\forall\,(\beta_1 \Rightarrow \sigma_1, \beta_2 \Rightarrow \sigma_2)\ \beta_1 \to \beta_2$ via extra rigid bindings.

The equivalence and instance relations in IML^F are adapted to XML^F to deal with rigid bindings. In fact, type equivalence in IML^F is too large to permit type inference. We split type equivalence into two inverse relations $\sqsubseteq$, called abstraction, and $\sqsupseteq$, called revelation. More precisely, type equivalence $\boxminus$ in IML^F corresponds to the transitive closure of $\sqsubseteq \cup \sqsupseteq$, while type equivalence in XML^F is $\sqsubseteq \cap \sqsupseteq$. Moreover, $\sqsubseteq$ is a subrelation of type instance, and may be left implicit in programs. Conversely, uses of $\sqsupseteq$ must be made explicit, via type annotations.

We now present the equivalence, abstraction, and instance relations formally and in this order, from the smaller to the larger relations, as their definitions depend on these inclusions.

Definition 4.1.1 (Congruence) A relation $\mathcal{R}$ is $\geq$-*congruent* if it satisfies both Xmlf-Flex-Left and Xmlf-All-Right (Figure 8). It is *congruent* if it satisfies both Xmlf-All-Left and Xmlf-All-Right. □

Definition 4.1.2 (Equivalence) The equivalence relation, written $\equiv$, is the smallest congruent equivalence relation satisfying the rules of Figure 9. □

Rules of Figure 9 are straightforward adaptations from those of Figure 5. The only difference is the replacement of single $\geq$ bound in IML^F by the two possible bounds $\geq$ or $\Rightarrow$ in XML^F. We write $\mathcal{V}$ for the set of type schemes that are equivalent to a variable under the empty prefix. In fact, it is convenient to have a notation for the top-most structure of a type scheme.

Definition 4.1.3 (Head) The head of a type scheme σ is the symbol or variable, written $\mathsf{head}(\sigma)$, defined inductively as follows:

$$\mathsf{head}(\alpha) \triangleq \alpha \qquad \mathsf{head}(\tau_1 \to \tau_2) \triangleq \to \qquad \mathsf{head}(\bot) \triangleq \bot \qquad \mathsf{head}(\forall\,(\alpha \diamond \sigma_1)\ \sigma_2) \triangleq \begin{cases} \mathsf{head}(\sigma_1), & \text{if } \mathsf{head}(\sigma_2) = \alpha \\ \mathsf{head}(\sigma_2), & \text{otherwise} \end{cases}$$

□

The head of a type scheme is preserved by equivalence under an empty prefix. Hence, the head of all elements of $\mathcal{V}$ are variables. We can show the converse—a type scheme σ whose head is a variable is in $\mathcal{V}$—by an easy structural induction on σ. Hence, $\mathcal{V}$ is also the set of type schemes whose head are type variables.

Figure 10: Abstraction in XML^F.

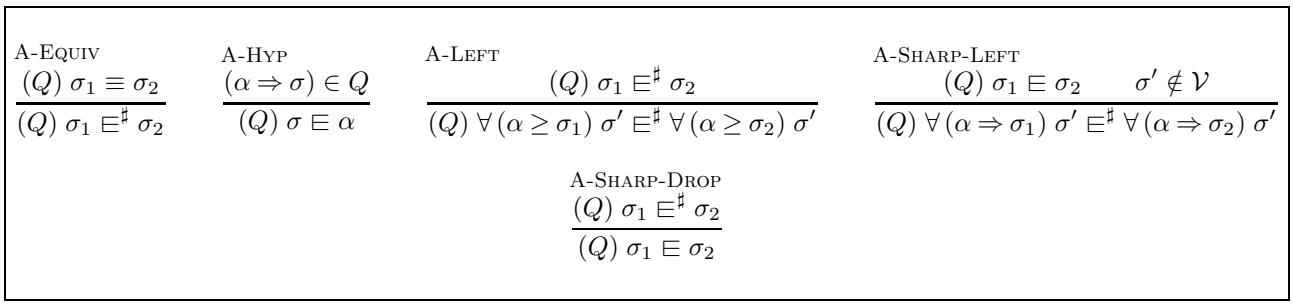
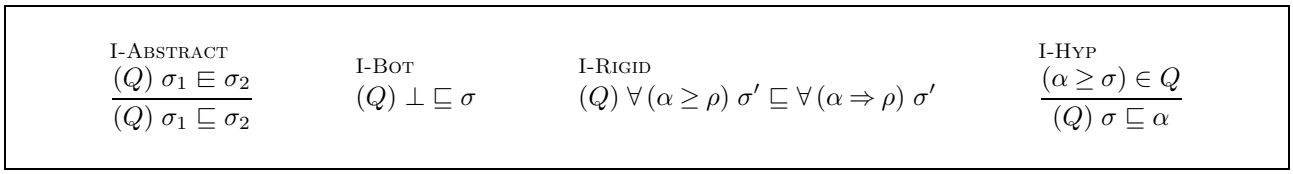


Figure 11: Instance in XML^F.



Rigid bindings are used to abstract explicit type schemes as implicit ones, by storing and sharing their definition via the prefix. The *abstraction relation* $\sqsubseteq$ describes whenever a type scheme is more abstract than another one. The relation is essentially structural, except for the key axiom that retrieves an assumption from the prefix (Rule A-Hyp of Figure 10). However, a peculiarity of the relation is that it is congruent in all contexts ending with a true rigid binding, that is, contexts of the form $\forall\,(\alpha \Rightarrow [\,])\ \sigma'$ where σ' is not equivalent to α. This condition is ensured by the stronger requirement $\sigma' \notin \mathcal{V}$ of Rule A-Sharp-Left. Omitting the condition would allow pathological contexts such as $\forall\,(\beta \geq \forall\,(\alpha \Rightarrow [\,])\ \alpha)\ \tau$, which are equivalent to $\forall\,(\beta \geq [\,])\ \tau$. Abstraction in this context would not be reversible, hence it would be unsound. Intuitively, a rigid binding behaves as a protection that prevents the underlying type from ever being instantiated and, as a consequence, allows the underlying type to be abstracted. Technically, we need to keep track of protected abstractions, as only those can be used in unprotected flexible contexts. For that purpose, we use an auxiliary relation $\sqsubseteq^\sharp$, called *protected abstraction*, that is recursively defined with the (unprotected) abstraction relation $\sqsubseteq$.

Definition 4.1.4 (Abstraction) The abstraction relation $\sqsubseteq$ and the protected abstraction relation $\sqsubseteq^\sharp$ are the smallest transitive relations satisfying the rules of Figure 10 as well as Rule Xmlf-All-Right. □

Rules may be read by first ignoring the difference between $\sqsubseteq$ and $\sqsubseteq^\sharp$, then realizing that the distinction only prevents uses of A-Hyp in pathological contexts for the reason explained above.

The instance relation differs from the one of IML^F in only two minor ways. Firstly, it extends not only the equivalence but also the abstraction relation. Secondly, Rule FI-Subst, which would no longer be well-formed, has been replaced by I-Rigid, introducing a rigid binding instead of performing the substitution in the conclusion. Other rules in Figure 11 are directly taken from those of IML^F (Figure 6).

Definition 4.1.5 (Instance) The instance relation, written $\sqsubseteq$, is the smallest transitive and $\geq$-congruent relation satisfying the rules of Figure 11. □

Notice the inclusions $(\equiv) \subset (\sqsubseteq^\sharp) \subset (\sqsubseteq) \subset (\sqsubseteq)$.

Soundness of instance and abstraction relations

The type soundness of XML^F is shown below by a translation of well-typed XML^F programs into well-typed IML^F programs, which in turn requires a translation of types and relations on types.

Trivial bindings such as $(\beta \geq \alpha)$ often lead to pathological cases, as they are just redirections. As a consequence, we often need to consider types schemes of $\mathcal{V}$ especially. While we write $\sigma \in \mathcal{V}$ (or $\sigma \equiv \beta$ when the identifier β is meaningful) for conciseness and clarity, this can always be understood—and computed—as $\mathsf{head}(\sigma) \in \vartheta$ (or $\mathsf{head}(\sigma) = \beta$).

The projection function, defined below, translates XML^F types into IML^F types.

Definition 4.1.6 (Type projection) The projection of an XML^F type into a IML^F type is defined as follows:

$$\llbracket\tau\rrbracket \triangleq \tau \qquad \llbracket\bot\rrbracket \triangleq \bot$$

$$\llbracket\forall\,(\alpha \Rightarrow \sigma_1)\;\sigma_2\rrbracket \triangleq \llbracket\sigma_2\rrbracket[\llbracket\sigma_1\rrbracket/\alpha]$$

$$\llbracket\forall\,(\alpha \geq \sigma_1)\;\sigma_2\rrbracket \triangleq \begin{cases} \llbracket\sigma_2\rrbracket[\beta/\alpha] & \text{if } \sigma_1 \equiv \beta \\ \forall\,(\alpha \geq \llbracket\sigma_1\rrbracket)\;\llbracket\sigma_2\rrbracket & \text{if } \sigma_1 \notin \mathcal{V} \end{cases}$$

□

The projection of a monotype τ is τ itself and the projection of $\bot$ is $\bot$. A binding $(\alpha \Rightarrow \sigma_1)$ is translated to a substitution $[\llbracket\sigma_1\rrbracket/\alpha]$, which is well-formed as $\llbracket\sigma_1\rrbracket$ is an F-like type. A binding $(\alpha \geq \sigma_1)$ is mapped as such, unless it is a trivial one, in which case the corresponding substitution is performed.

An important property of the translation is that it does not contain exposed type variables (Definition 3.3.6), unless the type scheme being translated is itself a type variable.

Lemma 4.1.7 *The set of exposed type variables of $\llbracket\sigma\rrbracket$ is included in the singleton $\{\mathsf{head}(\sigma)\}$.*

Whereas the projection of an XML^F type is an IML^F type, the projection of an XML^F prefix Q is a pair composed of an IML^F prefix that corresponds to flexible bindings of Q and a substitution that captures the rigid bindings of Q. Special care is again needed for trivial bindings.

Definition 4.1.8 (Prefix projection) The projection of a prefix Q is a pair (Q, θ), defined inductively as follows:

$$\llbracket\emptyset\rrbracket \triangleq (\emptyset, id) \qquad \frac{\llbracket Q\rrbracket = (Q', \theta)}{\llbracket(Q, \alpha \Rightarrow \sigma)\rrbracket \triangleq (Q', \theta \circ [\llbracket\sigma\rrbracket/\alpha])} \qquad \frac{\llbracket Q\rrbracket = (Q', \theta) \qquad \llbracket\sigma\rrbracket \notin \mathcal{V}}{\llbracket Q, (\alpha \geq \sigma)\rrbracket \triangleq ((Q', \alpha \geq \theta\llbracket\sigma\rrbracket), \theta)}$$

$$\frac{\llbracket Q\rrbracket = (Q', \theta) \qquad \llbracket\sigma\rrbracket \equiv \beta}{\llbracket Q, (\alpha \geq \sigma)\rrbracket \triangleq (Q', \theta \circ [\beta/\alpha])}$$

□

The following lemma states that type equivalence, type abstraction, and type instance relation are preserved by projections into IML^F.

Lemma 4.1.9 *Let (Q', θ) be $\llbracket Q\rrbracket$. We have the following implications*

i) *If $(Q)\;\sigma_1 \equiv \sigma_2$, then $(Q')\;\theta\llbracket\sigma_1\rrbracket \boxminus \theta\llbracket\sigma_2\rrbracket$.*

ii) *If $(Q)\;\sigma_1 \sqsubseteq \sigma_2$, then $(Q')\;\theta\llbracket\sigma_1\rrbracket \boxminus \theta\llbracket\sigma_2\rrbracket$.*

iii) *If $(Q)\;\sigma_1 \sqsubseteq \sigma_2$, then $(Q')\;\theta\llbracket\sigma_1\rrbracket \leq \theta\llbracket\sigma_2\rrbracket$.*

(Proof p. 47)

Completeness of instance and abstraction relations

We may conversely show that type instance and type abstraction in XML^F capture no more than type instance and type equivalence in IML^F.

Let us first introduce a translation from IML^F types to XML^F types, written $\lceil\sigma\rceil$ and defined inductively as follows:

$$\lceil\alpha\rceil \triangleq \alpha \qquad \lceil\bot\rceil \triangleq \bot \qquad \lceil\tau_1 \to \tau_2\rceil \triangleq \forall\,(\alpha_1 \Rightarrow \lceil\tau_1\rceil, \alpha_2 \Rightarrow \lceil\tau_2\rceil)\;\alpha_1 \to \alpha_2 \qquad \lceil\forall\,(\alpha \geq \sigma)\;\sigma'\rceil \triangleq \forall\,(\alpha \geq \lceil\sigma\rceil)\;\lceil\sigma'\rceil$$

The translation of variables, $\bot$, and flexible bindings are by a direct mapping. The translation of an arrow type $\tau_1 \to \tau_2$ uses auxiliary bindings $(\alpha_1 \Rightarrow \lceil\tau_1\rceil, \alpha_2 \Rightarrow \lceil\tau_2\rceil)$ as $\lceil\tau_1\rceil$ and $\lceil\tau_2\rceil$ are not guaranteed to be monotypes. They are guaranteed to be ρ-types, though. In case there are monotypes, the extra indirection is not problematic as it can always be eliminated by type-equivalence in XML^F.

The translation of an IML^F binding $(\alpha \geq \sigma)$ is a sequence of bindings $Q'(\alpha \geq \sigma')$, such that $\forall\,(Q')\;\sigma' = \lceil\sigma\rceil$ and Q' is rigid and as large as possible. As a consequence, the translation of a monotype binding $(\alpha \geq \tau_1 \to \tau_2)$ is the sequence $(\alpha_1 \Rightarrow \lceil\tau_1\rceil, \alpha_2 \Rightarrow \lceil\tau_2\rceil, \alpha \geq \alpha_1 \to \alpha_2)$. Notice that, by α-conversion of $\forall\,(Q')\;\sigma'$, the choice of the domain of Q' is free.

The translation of a prefix $Q = q_1 \ldots q_n$ is the concatenation of its pointwise translation: $\lceil Q\rceil = \lceil q_1\rceil \ldots \lceil q_n\rceil$, avoiding any capture by use of α-conversion.

As a preliminary result, we show that monomorphic substitution in IML^F is captured by the symmetric closure of abstraction in XML^F:

Lemma 4.1.10 *Let σ be an IML^F type scheme and Q be an IML^F prefix such that $(\alpha \geq \tau) \in Q$ with $\tau \in \mathcal{M}$. Then, we have $(\lceil Q\rceil)\;\lceil\sigma\rceil\;(\sqsubseteq^\sharp \cup \exists^\sharp)^*\;\lceil\sigma[\tau/\alpha]\rceil$ in XML^F.*

Figure 12: XML^F typing rules

$$\textsc{Var} \qquad \textsc{Fun} \qquad \textsc{App} \qquad \textsc{Gen} \qquad \textsc{Let}$$

$$\textsc{Inst}\ \frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \sqsubseteq \sigma'}{(Q)\ \Gamma \vdash a : \sigma'} \qquad \textsc{Annot}\ \frac{(Q)\ \Gamma \vdash a : \sigma' \qquad \bar\alpha \subseteq \mathsf{dom}(Q) \qquad (Q)\ \sigma \sqsubseteq \sigma'}{(Q)\ \Gamma \vdash (a : \exists\,(\bar\alpha)\ \sigma) : \sigma}$$

(Proof p. 47)

Lemma 4.1.11 *Let τ be an IML^F type, σ be an IML^F type scheme, and Q an XML^F prefix. Then, we have $(Q)\ \forall\,(\alpha \Rightarrow [\![\tau]\!])\ [\![\sigma]\!]\ (\sqsubseteq^\sharp \cup \exists^\sharp)^*\ [\![\sigma[\tau/\alpha]]\!]$ in XML^F.*

We may now show that type instance and type equivalence in IML^F map to type instance and the symmetric closure of type abstraction in XML^F, respectively.

Lemma 4.1.12 *Both properties hold :*

i) *If $(Q)\ \sigma_1 \boxminus \sigma_2$ holds, then $([\![Q]\!])\ [\![\sigma_1]\!]\ (\sqsubseteq^\sharp \cup \exists^\sharp)^*\ [\![\sigma_2]\!]$ holds.*

ii) *If $(Q)\ \sigma_1 \leq \sigma_2$ holds, then $([\![Q]\!])\ [\![\sigma_1]\!]\ (\sqsubseteq \cup \exists)^*\ [\![\sigma_2]\!]$ holds.*

(Proof p. 48)

Properties 4.1.9.ii and property 4.1.12.i show that the relations $\boxminus$ and $(\sqsubseteq \cup \exists)^*$ are in correspondence. There were also used indirectly to show Property 4.1.9.iii. Only Property 4.1.9.iii is further used, namely to show a close correspondence between type systems IML^F and XML^F.

4.2 Typing rules and type soundness

Terms of XML^F are those of IML^F extended with a new primitive construction for *type annotations* $(a : \exists\,(\bar\alpha)\ \sigma)$. The typing rules of XML^F, given in Figure 12, include all rules from the generic system $\mathsf{G}(\mathcal{T}_\mathsf{X}, \mathcal{S}_\mathsf{X}, \mathcal{Q}_\mathsf{X}, \sqsubseteq)$ and a new rule for type annotations. Notice, that the generic rule Inst is specialized, accordingly, using the relation $\sqsubseteq$ for type instance. Rule Annot is thus the only interesting rule in XML^F. The existentially quantified variables $\bar\alpha$ in annotations is to allow annotations to *partially* specify the type of the expression they annotate: their bounds are left unspecified (equivalently $\bar\alpha$ could be given a flexible bottom bound in the annotation) and thus must be inferred. Free type variables of σ must all be listed in $\bar\alpha$, so that the annotation $\exists\,(\bar\alpha)\ \sigma$ is itself closed. Variables $\bar\alpha$ are required to appear in the prefix Q, as specified by the premise $\bar\alpha \subseteq \mathsf{dom}(Q)$. The judgment $(Q)\ \sigma \sqsubseteq \sigma'$ allows to reveal σ, but no more. In particular, the bounds assigned to $\bar\alpha$ are shared between σ' and σ, which prevents to reveal more than explicitly specified in σ through implicit instantiation of its free type variables $\bar\alpha$. As a particular case, annotating an expression with $\exists\,(\alpha)\ \alpha$ is useless. Conversely, all inner bound variables of σ must be matched exactly—up to abstraction.

Syntactic sugar When σ is closed, we may simply write σ. We then recover the simplified rule given in the introduction (§ 2.3, page 16). In fact, by abuse of notation, we could also write $(a : \sigma)$ when σ is not closed to mean $(a : \exists\,(\mathsf{ftv}(\sigma))\ \sigma)$, but we prefer to remain more explicit about bound variables.

We also see abstractions $\lambda(x : \sigma)\ a$ as syntactic sugar for $\lambda(x)\ \mathsf{let}\ x = (x : \sigma)\ \mathsf{in}\ a$. Let-rebinding x to the annotated expression thus to avoid repeating the annotation on all occurrences of x in a. The effect is that $\lambda(x : \sigma)\ a$ is typed as if it were $\lambda(x)\ a[(x : \sigma)/x]$, but our syntactic sugar is more local. The annotated abstraction may also be typed directly, with the following derivable typing rule:

$$\textsc{Fun'}\ \frac{(Q)\ \Gamma, x : \rho \vdash a : \tau \qquad \bar\alpha \subseteq \mathsf{dom}(Q)}{(Q)\ \Gamma \vdash \lambda(x : \exists\,(\bar\alpha)\ \rho)\ a : \forall\,(\beta \Rightarrow \rho)\ \beta \to \tau}$$

In practice, most uses of annotations are actually in abstractions. The reason not to make annotated abstraction the primitive form and the other one the derived form is that $(a : \sigma)$ are much simpler to deal with technically.

Furthermore, for F-like type annotations, $(a : \rho)$ can just be seen as the application of a retyping primitive function (ρ) to the expression a. In Full ML^F, all annotations can be treated as such. We could restrict XML^F to F-like annotations. However, because types are stratified in Plain ML^F, we would then not reach all type annotations and XML^F would not be in close correspondence with IML^F any longer.

Example We first show that the (unannotated) identity function $\lambda(x)\ x$ is typable with type $\forall\,(\alpha \Rightarrow \rho)\ \alpha \to \alpha$ for any F-like type scheme ρ (which type corresponds to $\rho \to \rho$ in IML^F). Notice that ρ may be polymorphic. In the following, we write σ_{id} for $\forall\,(\alpha)\ \alpha \to \alpha$.

$$\textsc{Inst}\ \dfrac{\textsc{Gen}\ \dfrac{\textsc{Fun}\ \dfrac{\textsc{Var}\ \dfrac{}{(\alpha \geq \bot)\ x : \alpha \vdash x : \alpha}}{(\alpha \geq \bot)\ \emptyset \vdash \lambda(x)\ x : \alpha \to \alpha}}{(\emptyset)\ \emptyset \vdash \lambda(x)\ x : \sigma_{\mathsf{id}}} \qquad (\emptyset)\ \sigma_{\mathsf{id}} \sqsubseteq \forall\,(\alpha \Rightarrow \rho)\ \alpha \to \alpha}{(\emptyset)\ \emptyset \vdash \lambda(x)\ x : \forall\,(\alpha \Rightarrow \rho)\ \alpha \to \alpha}$$

Notice that a more direct derivation is possible:

$$\textsc{Gen}\ \dfrac{\textsc{Fun}\ \dfrac{\textsc{Var}\ \dfrac{}{(\alpha \Rightarrow \rho)\ x : \alpha \vdash x : \alpha}}{(\alpha \Rightarrow \rho)\ \emptyset \vdash \lambda(x)\ x : \alpha \to \alpha}}{(\emptyset)\ \emptyset \vdash \lambda(x)\ x : \forall\,(\alpha \Rightarrow \rho)\ \alpha \to \alpha}$$

For comparison, here is a derivation when the argument is annotated.

$$\textsc{Fun'}\ \dfrac{\textsc{Inst}\ \dfrac{\textsc{Var}\ \dfrac{}{(\alpha \Rightarrow \rho)\ x : \rho \vdash x : \rho} \qquad \dfrac{(\alpha \Rightarrow \rho)\ \rho \sqsubseteq \alpha}{(\alpha \Rightarrow \rho)\ \rho \sqsubseteq \alpha}}{(\alpha \Rightarrow \rho)\ x : \rho \vdash x : \alpha}}{(\alpha \Rightarrow \rho)\ \emptyset \vdash \lambda(x : \rho)\ x : \forall\,(\beta \Rightarrow \rho)\ \beta \to \alpha}$$

$$\textsc{Gen}\ \dfrac{(\alpha \Rightarrow \rho)\ \emptyset \vdash \lambda(x : \rho)\ x : \forall\,(\beta \Rightarrow \rho)\ \beta \to \alpha}{(\emptyset)\ \emptyset \vdash \lambda(x : \rho)\ x : \forall\,(\alpha \Rightarrow \rho)\ \forall\,(\beta \Rightarrow \rho)\ \beta \to \alpha}$$

The variable x has a polymorphic F-like type ρ, which is available with Rule Var. In the previous derivation, x had only a type α, which was bound to ρ in the prefix. This is a crucial difference between the two derivations. Indeed, in the latter derivation, the polymorphism can be instantiated, so that for example $\lambda(x : \sigma_{\mathsf{id}})\ x\ x$ is typable. On the contrary, we will show below (§4.5) that $\lambda(x)\ x\ x$ is not typable when the type annotation is missing. Another important remark is the use of abstraction (and Rule Inst) to hide the polymorphic type ρ of x as the abstract type α (defined to be ρ in the prefix). This is to prepare for rule Fun', which requires the codomain of the type of $\lambda(x : \rho)\ x$ to be a monotype and not a polytype ρ.

To see the role of existential quantification in type annotations, compare the two expressions $\lambda(x : \exists\,(\beta)\ \rho)\ x$ and $\lambda(x : \forall\,(\beta)\ \rho)\ x$ where ρ is $\forall\,(\alpha)\ \alpha \to \beta \to \alpha$—with a free single type variable β. Their respective types are $\forall\,(\beta)\ \forall\,(\gamma \Rightarrow \rho)\ \forall\,(\gamma' \Rightarrow \rho)\ \gamma \to \gamma'$ and $\forall\,(\gamma \Rightarrow \forall\,(\beta)\ \rho)\ \forall\,(\gamma' \Rightarrow \forall\,(\beta)\ \rho)\ \gamma \to \gamma'$. In the later case, the annotation requires the argument to be polymorphic in β—hence the result is also polymorphic in β. Conversely, in the former case, β is shared between the argument type and the result type and cannot be polymorphic within the expression, but only generalized afterward. Less polymorphism is required on the argument and so less polymorphism is asserted on the result—namely just as much as was promised to be received.

Derivable rules Rules App* and UnGen* (defined page 22) remains admissible in XML^F—of course, when read with types and type schemes taken in XML^F.

Expressiveness

We show that XML^F and IML^F are in close correspondence, and thus exactly as expressive. Dropping type annotations maps XML^F programs to IML^F programs directly.

Theorem 3 *Assume XML^F :: $(Q)\ \Gamma \vdash a : \sigma$ holds. Let a' be the erasure of a, let (Q', θ) be $\llbracket Q \rrbracket$. Then, IML^F :: $(Q')\ \theta(\Gamma) \vdash a' : \theta(\llbracket \sigma \rrbracket)$.*

(Proof p. 48)

Conversely, all IML^F programs can always be mapped to XML^F programs by inserting explicit type annotations.

Theorem 4 *If IML^F :: $(Q)\ \Gamma \vdash a : \sigma$ holds, then there exists a term a', such that a is a type-erasure of a' and XML^F :: $(\lceil Q \rceil)\ \lceil \Gamma \rceil \vdash a' : \lceil \sigma \rceil$ holds.*

(Proof p. 48)

Noticeably, the translation of an IML^F program is based on its typing derivation. It introduces a type annotation on every λ-abstraction, and possibly several ones on type instances and generalizations.

Type soundness

Type soundness is a corollary of Theorem 3, which ensures that XML^F is as safe as IML^F, and Theorem 2, which states type soundness of IML^F.

4.3 Translating System F into XML^F

The composition of theorems 2 and 4 states that there is a mapping of System-F terms to XML^F terms that proceeds only by insertion of well-chosen type annotations. Those theorems do not tell us where and what annotations to insert. However, their proof are constructive—given a type derivation of the input term, or equivalently, an explicitly typed input term. That is, we could have used the typed derivation given as input to explicitly produce a type derivation in XML^F as output.

However, the resulting term would contain many duplicated or scattered type annotations, unless we change the proof and show stronger (and difficult) lemmas that typed derivations could be rearranged in certain ways, so that for instance, type annotations can always be moved to function parameters.

Instead, we propose a direct translation of explicitly typed System-F programs to XML^F that keep (actually translate) the type annotations on λ-abstractions, and throws away all type abstractions and type applications. Therefore, it returns an XML^F program that contains at most as much, and in general fewer, type information than the original System-F term.

The first step of the translation is the translation of types. Let us introduce an important design choice of this translation, informally. Rigid bindings are interpreted as substitutions (Definition 4.1.6). For example, $\forall\,(\alpha \Rightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha$ (**1**) is interpreted as the F-type $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$. However, $\forall\,(\alpha_1 \Rightarrow \sigma_{\mathsf{id}}, \alpha_2 \Rightarrow \sigma_{\mathsf{id}})\ \alpha_1 \to \alpha_2$ (**2**) is also interpreted likewise. Therefore, there are two candidates for the converse translation of $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$ into XML^F, namely (1) and (2). Observe that (2) is more general than (1) in XML^F (the latter is an instance of the former). Taking (2) is the approach chosen in Section 4.1 to translate IML^F types. Maybe surprisingly, we choose (1) to be the translation of $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$. That is, we always share similar bindings as much as possible, as formalized in Definition 4.3.3 below. The opposite choice, which would associate (2) to the translation of $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$, is also possible. Although this alternative is perhaps more elegant, its correctness proof is longer and much more involved [LB04]. We present the first approach here for sake of simplicity. Despite this choice, this section remains the most technical part of the paper[8]. It happens that proving the soundness of the translation from System F to XML^F is subtle and needs meticulous instrumentation. Let us explain why.

A single System F type, such as $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$, corresponds to possibly many types in XML^F, as a result of inlining of rigid bindings in System F. Consequently, XML^F types are more discriminatory, *i.e.* contain more information, than System-F types, which is crucial for permitting type inference. The downside is that, given a typing derivation in System F, we have to reconstruct the missing information and show that it is consistent with ML^F (typing rules). The purpose of the instrumentation is exactly to reconstruct and trace this information in a safe way.

Auxiliary definitions

We first define a few operators that are used to translate F-types into XML^F-types. The translation introduces rigid bindings and unconstrained flexible bindings, but never uses constrained flexible bindings.

As a first step, we translate an F-type into a pair of a prefix and a variable used as an entry point into that prefix. Following our design choice (exposed above), prefixes are maintained in shared form.

Definition 4.3.1 A prefix Q is *shared* when for all σ, $(\alpha_1 \Rightarrow \sigma) \in Q$ and $(\alpha_2 \Rightarrow \sigma) \in Q$ imply $\alpha_1 = \alpha_2$. □

Sharing is purely syntactic, based on type equality. While it would have been more natural to define sharing up to equivalence, this would require more technical machinery—and at least to present an algorithm for testing equivalence (which can be found [LB04]). The syntactic definition suffices and is simpler.

As a result of sharing, the insertion of a new binding into a prefix depends on bindings that are already present. Insertion, which is defined next, maintains another invariant: prefixes are *ordered* in the sense that rigid bindings are inserted as far to the left as possible. For example, the prefix $(\alpha, \beta \Rightarrow \gamma \to \gamma)$ is not ordered because the rigid bound of β does not depend on α. The ordered, equivalent prefix is $(\beta \Rightarrow \gamma \to \gamma, \alpha)$. Intuitively, ordering may move rigid bindings, but not flexible ones. This is a form of extrusion, which we enforce to ensure maximal sharing of rigid bounds. Indeed, this maximal sharing requires rigid bindings to have the wider possible

[8] We have marked as *auxiliary* those results that are not used in further sections of the paper and therefore can easily be skipped.

scope. Formally, the *bounds of a prefix* Q, written $\mathsf{bnds}(Q)$, is the set of all σ such that there exists a binding $(\alpha \Rightarrow \sigma)$ in Q.

Definition 4.3.2 The *insertion* of a type scheme σ into a prefix Q at variable α, written $Q \oplus_\alpha \sigma$, is a prefix defined in the two following cases: If $(\alpha \Rightarrow \sigma) \in Q$, then $Q \oplus_\alpha \sigma$ is Q. If $\alpha \notin \mathsf{dom}(Q)$ and $\sigma \notin \mathsf{bnds}(Q)$ then $Q \oplus_\alpha \sigma$ is defined recursively as follows:

- $\emptyset \oplus_\alpha \sigma$ is $(\alpha \Rightarrow \sigma)$,
- $(Q''Q') \oplus_\alpha \sigma$ is $(Q'' \oplus_\alpha \sigma)Q'$ if $\mathsf{ftv}(\sigma) \# \mathsf{dom}(Q')$,
- $(Q', \beta' \diamond \sigma') \oplus_\alpha \sigma$ is $(Q', \beta' \diamond \sigma', \alpha \Rightarrow \sigma)$ if $\beta' \in \mathsf{ftv}(\sigma)$.

We write $Q \otimes_\alpha \sigma$ for the pair $(Q \oplus_\alpha \sigma), \alpha$. □

Notice that insertion is partial. For example, $(\beta \Rightarrow \sigma) \oplus_\alpha \sigma$ is undefined. When defined, it returns the original prefix either exactly or with the binding $\alpha = \sigma$ inserted "at the right place" while preserving the ordering of other bindings.

We define an algorithm that computes the translation of an F-type t in two steps. We first define a relation that takes a prefix Q and the type t as input an and returns a pair of a new prefix Q' that extends Q and a type variable α as output. This is written $(Q)\ \langle\langle t \rangle\rangle : (Q', \alpha)$ (read "under prefix Q, a translation of t is the pair (Q', α)"). We then define the translation of t, written $\langle\langle t \rangle\rangle$ as a set of XML$^{\mathsf{F}}$ types.

The use of an auxiliary relation is to capture non-determinism that results from the choice of fresh variables during the translation. Lemma 4.3.6 below shows that a given input yields outputs that are similar, up to renaming.

Below, we use the notation $(Q)\ \langle\langle t \rangle\rangle : Q' \otimes \sigma$ (without any variable on $\otimes$) to mean that there exists α such that $Q' \otimes_\alpha \sigma$ is defined and $(Q)\ \langle\langle t \rangle\rangle : Q' \otimes_\alpha \sigma$ holds. This exposes the witness α in the relation $(Q)\ \langle\langle t \rangle\rangle : (Q', \alpha)$ and so introduces a source of non-determinism in the definition—the only one.

Definition 4.3.3 The translation relation is the smallest relation on quadruples of the form $(Q)\ \langle\langle t \rangle\rangle : (Q', \alpha)$ where Q and Q' are well-formed shared prefixes and t a type such that $\mathsf{ftv}(t) \cap \mathsf{dom}(Q') \subseteq \mathsf{dom}(Q)$ satisfying the following rules:

$$(Q)\ \langle\langle \alpha \rangle\rangle : (Q, \alpha) \qquad \frac{(Q)\ \langle\langle t_1 \rangle\rangle : (Q_1, \alpha_1) \qquad (Q_1)\ \langle\langle t_2 \rangle\rangle : (Q_2, \alpha_2)}{(Q)\ \langle\langle t_1 \to t_2 \rangle\rangle : Q_2 \otimes \alpha_1 \to \alpha_2} \qquad \frac{(Q\alpha)\ \langle\langle t \rangle\rangle : (Q_1 \alpha Q_2, \beta)}{(Q)\ \langle\langle \forall\,(\alpha)\ t \rangle\rangle : Q_1 \otimes \forall\,(\alpha Q_2)\ \beta}$$

□

The restriction on the free type variables of t and the domains of the prefixes can always be satisfied by an appropriate choice of fresh variables. The only possible translation of a type variable α under prefix Q is the pair (Q, α), whether α belongs to $\mathsf{dom}(Q)$ or not. A translation of an arrow type $t_1 \to t_2$ under Q is built using the translation (Q_1, α_1) of t_1 under Q and (Q_2, α_2) of t_2 under Q_1. It is defined as the insertion of $\alpha_1 \to \alpha_2$ into Q_2. Finally, the translation of a quantified type $\forall\,(\alpha)\ t$ is built using the translation of t under $Q\alpha$. Then, the resulting prefix is split into Q_1 and Q_2 and the result is the insertion of $\forall\,(\alpha Q_2)\ \beta$ into Q_1. Since $Q_1 \alpha Q_2$ is ordered, all the bindings of Q_2 actually depends on α, possibly indirectly. This means that the prefix Q_2 to appear in the quantification (αQ_2) is as small as possible.

The inclusion of prefixes, written $Q \subseteq Q'$, means that Q' is obtained from Q by none or several insertions. When $(Q)\ \langle\langle t \rangle\rangle : (Q', \alpha)$ holds, we have $Q \subseteq Q'$ and $\mathsf{ftv}(Q') \subseteq \mathsf{ftv}(Q) \cup \mathsf{ftv}(t)$. (This easily follows from the observation that $Q\alpha \subseteq Q_1 \alpha Q_2$ implies $Q \subseteq Q_1$.) Another invariant of the definition is that all bindings that are in Q' but not in Q are rigid.

Definition 4.3.4 (Translation of types and prefixes) The translation of an F-type t, written $\langle\langle t \rangle\rangle$ is the set of all XML$^{\mathsf{F}}$ type schemes $\forall\,(Q)\ \alpha$ such that there exists a rigid, shared prefix Q' with domain disjoint from $\mathsf{ftv}(t)$ verifying $(Q')\ \langle\langle t \rangle\rangle : (Q, \alpha)$. The translation of an F-typing environment A, written $\langle\langle A \rangle\rangle$, is the set of typing environments Γ that maps each x in $\mathsf{dom}(A)$ to some type scheme in $\langle\langle A(x) \rangle\rangle$. □

The prefix Q must actually be rigid and shared. This follows from definition of the translation relation and that fact that Q' is itself rigid.

Note that σ_{id} is both a type of System F and of XML$^{\mathsf{F}}$. We have $(\emptyset)\ \langle\langle \sigma_{\mathsf{id}} \rangle\rangle : (\beta \Rightarrow \sigma_{\mathsf{id}}), \beta$. We also have $(Q)\ \langle\langle \sigma_{\mathsf{id}} \rangle\rangle : (Q)\ (\beta \Rightarrow \sigma_{\mathsf{id}}), \beta$, for any rigid, shared prefix Q that does already has a binding for σ_{id}. We then have $(\beta \Rightarrow \sigma_{\mathsf{id}})\ \langle\langle \sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}} \rangle\rangle : (\beta \Rightarrow \sigma_{\mathsf{id}})\ (\gamma \Rightarrow \beta \to \beta), \gamma$.

We shall see below that all type schemes in the translation of an F-type are in fact equivalent (Corollary 4.3.10), and similarly for the translation of typing environments.

Auxiliary results We now establish several properties about the translation algorithm that will be used to prove the main result of this section, Lemma 5, from which theorems 6 and 7—two variants of the same result—immediately follows. All of these properties address the following informal question: *How can the output vary for some small changes to the input?* For instance, the following lemmas answer this question in particular cases:

- Lemma 4.3.5 states that inputs and outputs can be consistently renamed.
- Lemma 4.3.6 characterizes non determinism: the outputs can be renamed while leaving the input unchanged provided some capture-avoiding side conditions.
- Lemma 4.3.7 states a form of idempotence: i) the input may be be replaced by the output (leaving the output unchanged); ii) once the input and output are equal, they may be extended simultaneously.
- Lemma 4.3.9 states that the relation is closed by equivalence.
- Lemma 4.3.12 characterizes the effect of applying a substitution to the input type: the output prefix must be substituted and shared again—along a sharing relation defined below.
- Lemma 4.3.13 states that when removing an unconstrained binding from the input, the bindings of the output may have to be reordered.

As mentioned above, given a prefix Q and a type t, the algorithm may return different results due to different choices of fresh variables. This is captured by saying that renamings preserve the translation.

Lemma 4.3.5 (Stability by renaming) *If* $(Q)\ \langle\!\langle t \rangle\!\rangle : (Q', \alpha)$ *holds, then* $(\phi(Q))\ \langle\!\langle \phi(t) \rangle\!\rangle : (\phi(Q'), \phi(\alpha))$ *holds for any renaming* ϕ*.*

Conversely, the choice of fresh variables is the only source of non-determinism, so that outputs for a single input are always equal up to renaming. Additionally, the renaming can be chosen to be invariant on any "fresh" set of variables I.

Lemma 4.3.6 (Determinism up to α-conversion) *If* $(Q)\ \langle\!\langle t \rangle\!\rangle : (Q_1, \alpha_1)$ *and* $(Q)\ \langle\!\langle t \rangle\!\rangle : (Q_1', \alpha_1')$*, then for all finite set* I *disjoint from* $\mathsf{dom}(Q_1) \cup \mathsf{dom}(Q_1')$*, there exists a renaming* ϕ *such that*

$$\mathsf{dom}(\phi) \# \mathsf{dom}(Q) \cup \mathsf{ftv}(t) \cup I \qquad \phi(Q_1) = Q_1' \qquad \phi(\alpha_1) = \alpha_1'$$

As a consequence, $\forall\,(Q_1)\ \alpha_1$ is equal to $\forall\,(Q_1')\ \alpha_1'$ by α-conversion. The translation is also stable by iteration: translating a type under a prefix already containing the bindings of the translation returns the same prefix.

Lemma 4.3.7 (Idempotence) *If* $(\emptyset)\ \langle\!\langle t \rangle\!\rangle : (Q, \alpha)$*, then* $(QQ')\ \langle\!\langle t \rangle\!\rangle : (QQ', \alpha)$ *for any* Q' *such that* QQ' *is well-formed.*

(Proof p. 49)

The insertion of a type σ in a prefix Q (that is, $Q \otimes \sigma$) is defined so as to maximize sharing. The result depends on the initial prefix Q. Therefore, the translation of a type t under Q (that is, $(Q)\ \langle\!\langle t \rangle\!\rangle$) also depends on Q. We wish to show that the translation of a single type under different initial prefixes yields comparable prefixes, up to some equivalence relation that we define now.

The equivalence relation on shared prefixes $\equiv^I$ is the smallest equivalence (reflexive, symmetric, and transitive) relation also satisfying the two following rules:

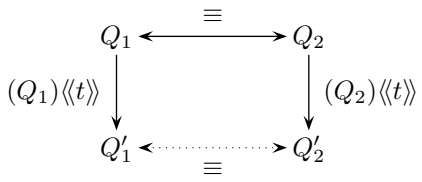
$$\textsc{Free}\ \frac{\alpha \notin I \cup \mathsf{dom}(Q) \cup \mathsf{ftv}(Q) \qquad \sigma \notin \mathsf{bnds}(Q)}{Q \equiv^I (Q, \alpha \Rightarrow \sigma)} \qquad \textsc{Comm}\ \frac{\alpha_1 \notin \mathsf{ftv}(\sigma_2) \qquad \alpha_2 \notin \mathsf{ftv}(\sigma_1)}{(Q, \alpha_1 \Rightarrow \sigma_1, \alpha_2 \Rightarrow \sigma_2, Q') \equiv^I (Q, \alpha_2 \Rightarrow \sigma_2, \alpha_1 \Rightarrow \sigma_1, Q')}$$

The superscript I is a finite set of type variables called the *interface*. Bindings of variables in I are "exposed" to equivalence. Rule Free allows the insertion (or removal) of bindings not in the interface I. Side conditions ensure that the prefix is kept well-formed and shared. Rule Comm allows the commutation of independent binders.

Unsurprisingly, two types built with equivalent prefixes are equivalent.

Lemma 4.3.8 *If* $Q_1 \equiv^I Q_2$ *and* $\mathsf{ftv}(\sigma) \subseteq I$*, then* $(Q)\ \forall\,(Q_1)\ \sigma \equiv \forall\,(Q_2)\ \sigma$ *holds under any suitable* Q*.*

Equivalent prefixes also yield equivalent translations. Informally, this may be illustrated by the commutative diagram below. The translation of t under two equivalent prefixes Q_1 and Q_2, yield equivalent prefixes Q'_1 and Q'_2.



Notations To lighten the notation below, we let Q mean $\mathsf{dom}(Q)$ and α mean $\{\alpha\}$ when a set of variables is non-ambiguously expected from context. For example, given a set of variables J, a variable α, and a prefix Q, we may write $J \cup \alpha$ for $J \cup \{\alpha\}$, $J \setminus \alpha$ for $J \setminus \{\alpha\}$, $J \cup Q$ for $J \cup \mathsf{dom}(Q)$, $J \cup (Q_1 \cap Q_2)$ for $J \cup (\mathsf{dom}(Q_1) \cap \mathsf{dom}(Q_2))$, and $\alpha \cap Q$ for $\{\alpha\} \cap \mathsf{dom}(Q)$.

Lemma 4.3.9 (Equiv) *Let I be $\mathsf{ftv}(t)$. We assume Q_1 rigid and $Q_1 \equiv^I Q_2$ holds. If $(Q_1)\ \langle\langle t\rangle\rangle : (Q'_1, \alpha_1)$ and $(Q_2)\ \langle\langle t\rangle\rangle : (Q'_2, \alpha_2)$ hold, then there exist a set J and a renaming ϕ such that:*

$$\mathsf{dom}(\phi) \# I \qquad I \cup \{\alpha_1\} \subseteq J \qquad Q'_1 \equiv^J \phi(Q'_2) \qquad \phi(\alpha_2) = \alpha_1$$

(Proof p. 49)

Corollary 4.3.10 *If $\sigma_1, \sigma_2 \in \langle\langle t\rangle\rangle$, then $(Q)\ \sigma_1 \equiv \sigma_2$ under any suitable Q.*

(Proof p. 52)

We now consider the effect of type instance on the translation. More precisely, substituting α by t' in a type t has an effect on sharing, in the sense that the translation of $t[t'/\alpha]$ is not only a substitution of the translation of t. For example, consider the type t equal to $(\sigma_{\mathsf{id}} \to \alpha) \to (\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}})$. A valid translation of t under an empty prefix is (after harmless simplification) $\forall\,(\alpha_1 \Rightarrow \sigma_{\mathsf{id}}, \alpha_2 \Rightarrow \alpha_1 \to \alpha, \alpha_3 \Rightarrow \alpha_1 \to \alpha_1)\ \alpha_2 \to \alpha_3$ (**1**). Substituting α by σ_{id} in t, we get $t[\sigma_{\mathsf{id}}/\alpha]$, that is, $(\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}) \to (\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}})$. A valid translation of the latter is $\forall\,(\alpha_1 \Rightarrow \sigma_{\mathsf{id}}, \alpha_2 \Rightarrow \alpha_1 \to \alpha_1)\ \alpha_2 \to \alpha_2$ (**2**). We see that α_2 and α_3 have been merged. In order to transform (1) into (2), the substitution $[\alpha_1/\alpha]$ must be applied first, then similar bindings must be shared again (namely, α_2 and α_3).

To this end, we define an algorithm $\Rrightarrow_\phi$ that shares prefixes as much as possible. The subscript ϕ is a substitution that keeps track of sharing that has already been performed. In the example above, ϕ would be $[\alpha_2/\alpha_3]$. The algorithm $\Rrightarrow$ is recursively defined by as a set of (deterministic) inference rules. It is written $(Q)\ Q_1 \Rrightarrow_\phi Q_2$, where Q and Q_1 are inputs and ϕ and Q_2 are outputs. For types, the algorithm is written $\Rrightarrow$, without subscript (Rule Sh-Types). As usual, the prefix Q may be omitted in $(Q)\ Q_1 \Rrightarrow_\phi Q_2$ or $(Q)\ \sigma_1 \Rrightarrow \sigma_2$ when it is empty.

Sh-Empty
$$(Q)\ \emptyset \Rrightarrow_{\mathsf{id}} \emptyset$$

Sh-Flex
$$\frac{(Q\alpha)\ Q_1 \Rrightarrow_\phi Q_2}{(Q)\ \alpha Q_1 \Rrightarrow_\phi \alpha Q_2}$$

Sh-Subst
$$\frac{(Q)\ \sigma \Rrightarrow \sigma' \qquad (\alpha' \Rightarrow \sigma') \in Q \qquad (Q)\ Q_1[\alpha'/\alpha] \Rrightarrow_\phi Q_2}{(Q)\ (\alpha \Rightarrow \sigma, Q_1) \Rrightarrow_{\phi \circ [\alpha'/\alpha]} Q_2}$$

Sh-Context
$$\frac{(Q)\ \sigma \Rrightarrow \sigma' \qquad \sigma' \notin \mathsf{bnds}(Q) \qquad (Q, \alpha \Rightarrow \sigma')\ Q_1 \Rrightarrow_\phi Q_2}{(Q)\ (\alpha \Rightarrow \sigma, Q_1) \Rrightarrow_\phi (\alpha \Rightarrow \sigma', Q_2)}$$

Sh-Types
$$\frac{(Q)\ Q_1 \Rrightarrow_\phi Q_2}{(Q)\ \forall\,(Q_1)\ \tau \Rrightarrow \forall\,(Q_2)\ \phi(\tau)}$$

Rules Sh-Empty, Sh-Flex, and Sh-Context are context rules that do not perform any sharing. On the contrary, Rule Sh-Subst detects and shares two similar bindings.

The next lemma describes properties of this algorithm.

Lemma 4.3.11

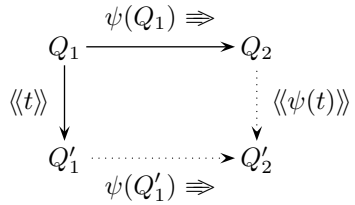
i) *If $(Q)\ Q_1 \Rrightarrow_\phi Q_2$, then $\mathsf{dom}(\phi) = \mathsf{dom}_=(Q_1) - \mathsf{dom}(Q_2)$ and $\mathsf{dom}(Q_1) = \mathsf{dom}(Q_2) \cup \mathsf{dom}(\phi)$.*

ii) *If $(Q)\ Q_1 \Rrightarrow_\phi Q_2$ holds, then for any type σ closed under QQ_1, we have $(Q)\ \forall\,(Q_1)\ \sigma \sqsubseteq \forall\,(Q_2)\ \phi(\sigma)$.*

Item i) is a technical invariant: the domain of Q_1 is the disjoint sum of the domain of Q_2 and the domain of ϕ. More precisely, the domain of ϕ is included in the rigid domain of Q_1 (which means that only rigid bindings are shared). Item ii) asserts that the sharing performed by the algorithm corresponds to abstraction at the type level.

(Proof p. 52)

The following lemma, composed of three properties, specifies the effect of a substitution ψ of the form $[\alpha/\beta]$ over a translation $\langle\!\langle t \rangle\!\rangle$. Property P-I is used in the proof of the following commutative diagram (Property P-II):



Moreover, the effect of ψ on the translation is equivalent to the substitution $[t'/\beta]$, provided (Q', α) is a translation of t' (Property P-III).

Lemma 4.3.12 *Let ψ be the substitution $[\alpha/\beta]$. We assume that the binding of α in Q. We say that Q is α-*rigid *if Q is rigid or of the form $Q', \alpha \diamond \sigma, Q''$ with Q' rigid. The following implications hold:*

P-I
$$\frac{\psi(Q) \Rrightarrow_\phi Q' \qquad \beta \neq \alpha' \qquad Q \text{ is } \alpha\text{-rigid} \qquad \beta \notin \mathsf{dom}(Q) \qquad Q \oplus_{\alpha'} \sigma \text{ is defined} \qquad \phi \circ \psi(\sigma) \Rrightarrow \sigma'}{\exists \phi' \text{ such that } \psi(Q \otimes_{\alpha'} \sigma) \Rrightarrow_{\phi' \circ \phi} Q' \otimes_{\phi' \circ \phi(\alpha')} \sigma' \qquad \mathsf{dom}(\phi') \subseteq \{\alpha'\}}$$

P-II
$$\frac{\begin{array}{c} Q_1 \text{ shared} \\ Q_1 \text{ is } \alpha_1\text{-rigid} \qquad \beta \notin \mathsf{dom}(Q'_1) \qquad \mathsf{ftv}(t) \# \mathsf{dom}_=(Q'_1) \qquad (Q_1)\ \langle\!\langle t \rangle\!\rangle : (Q'_1, \alpha_1) \qquad \psi(Q_1) \Rrightarrow_\phi Q_2 \end{array}}{\begin{array}{c} \exists Q'_2, \alpha_2, \phi' \text{ s.t. } (Q_2)\ \langle\!\langle \psi(t) \rangle\!\rangle : (Q'_2, \alpha_2) \qquad \psi(Q'_1) \Rrightarrow_{\phi' \circ \phi} Q'_2 \\ \phi' \circ \phi \circ \psi(\alpha_1) = \alpha_2 \end{array}}$$

P-III
$$\frac{(\emptyset)\ \langle\!\langle t' \rangle\!\rangle : (Q', \alpha) \qquad (Q')\ \langle\!\langle \psi(t) \rangle\!\rangle : (Q'_2, \alpha_2)}{(Q')\ \langle\!\langle t[t'/\beta] \rangle\!\rangle : (Q'_2, \alpha_2)}$$

(Proof p. 53)

In the following lemma, we show that an unconstrained binding (here β) may be removed from the input prefix of the translation only requires some reordering of binders in the output. To this end, we define the relation $\approx$ as the smallest equivalence relation on prefixes satisfying Comm. Noticeably, $Q \approx Q'$ implies $Q \equiv^I Q'$ for any I.

Lemma 4.3.13 *Let Q_1 and Q_2 be rigid prefixes. Let P and P' be two prefixes, each of which is either empty or starts with an unconstrained binding. Then, the following implication holds:*

$$\frac{(Q_1 \beta Q_2 P)\ \langle\!\langle t \rangle\!\rangle : (Q'_1 \beta Q'_2 P', \alpha) \qquad Q_1 Q_2 \approx Q_3}{\exists Q'_3 \qquad (Q_3 P)\ \langle\!\langle t \rangle\!\rangle : (Q'_3 P', \alpha) \qquad \wedge \qquad Q'_1 Q'_2 \approx Q'_3}$$

(Proof p. 54)

We are finally equipped to show the correctness of the translation from System F into ML^F. We shall use the two following derivable rules as short-cuts in the proof.

Lemma 4.3.14 *The following rules are derivable:*

$$\frac{Q' \text{ rigid}}{(QQ')\ \forall\, (Q')\ \sigma \sqsubseteq \sigma}\ \textsc{Shift}^\star \qquad\qquad \frac{}{(QQ')\ \forall\, (Q')\ \tau \sqsubseteq \tau}\ \textsc{Share}^\star$$

Theorem 5 *If $\mathsf{F} :: \Gamma' \vdash a : t$ holds, then there exists an expression a' whose type erasure is a and such that $\mathsf{XML}^\mathsf{F} :: (Q)\ \Gamma \vdash a' : \sigma$ holds for any $\Gamma \in \langle\!\langle \Gamma' \rangle\!\rangle$, $\sigma \in \langle\!\langle t \rangle\!\rangle$ and suitable prefix Q.*

(Proof p. 54)

The main result follows as a corollary.

Theorem 6 *Any term typable in implicit System F is typable in ML^F by adding some type annotations on function arguments.*

Noticing that type annotations on function arguments depend only on the type of the argument, and not on the rest of the typing derivation, a more precise statement is the following:

Theorem 7 *Any term typable in explicit System F is typable in ML^F by dropping type abstractions and type applications and by translating type annotations.*

Remarkably, all the System F terms that differ only in their type abstractions and type applications are translated towards the same XML^F program. Since every XML^F program admits a principal type (this result is not shown in this paper, but shown for Full ML^F as well as a small variant of XML^F in [LB04]), this type captures all the possible type abstractions and type applications of the term.

4.4 Embedding ML into XML^F

We show that ML is a conservative extension to ML. That is, we consider ML raw terms. *i.e.* XML^F terms that do not use type annotations, and show that well-typedness in ML implies well-typedness in XML^F. Conversely, closed ML raw terms that are well-typed in XML^F are also well-typed in ML.

ML types are a subset of F-types where quantification is allowed only at outermost level.

The instance relation for $\leq_{\mathsf{ML}}$ has been defined in the introduction (§2.1, Page 10). We recall that it is composed of exactly all pairs of the form $\forall\,(\bar\alpha)\ \sigma \leq_{\mathsf{ML}} \forall\,(\bar\beta)\ \sigma[\bar\tau/\bar\alpha])$ with $\bar\beta \# \mathsf{ftv}(\forall\,(\bar\alpha)\ \sigma)$. Notice that variables $\bar\alpha$ may only be substituted by monotypes $\bar\tau$. The following chain of relations in XML^F shows that $\leq_{\mathsf{ML}}$ is a subrelation of $\sqsubseteq$.

$$
\begin{array}{rcll}
\forall\,(\bar\alpha)\ \sigma & = & \forall\,(\alpha_1,..,\alpha_n)\ \sigma & \text{by notation}\\
& = & \forall\,(\alpha_1 \geq \bot,..,\alpha_n \geq \bot)\ \sigma & \text{by notation}\\
& \equiv & \forall\,(\bar\beta)\ \forall\,(\alpha_1 \geq \bot,..,\alpha_n \geq \bot)\ \sigma & \text{by Eq-Free}\\
& \sqsubseteq & \forall\,(\bar\beta)\ \forall\,(\alpha_1 \geq \tau_1,..,\alpha_n \geq \tau_n)\ \sigma & \text{by I-Nil and context rule}\\
& \sqsubseteq & \forall\,(\bar\beta)\ \forall\,(\alpha_1 \Rightarrow \tau_1,..,\alpha_n \Rightarrow \tau_n)\ \sigma & \text{by I-Rigid}\\
& \equiv & \forall\,(\bar\beta)\ \sigma[\tau_1/\alpha_1]..[\tau_n/\alpha_n] & \text{by Eq-Mono}\\
& = & \forall\,(\bar\beta)\ \sigma[\bar\tau/\bar\alpha] & \text{by notation}
\end{array}
$$

ML terms are in XML^F The typing rules of ML are exactly those of XML^F, namely Var, Fun, App, Inst, Gen, Let without Annot, and of course, modulo the restriction to ML types and prefixes and the use of $\leq_{\mathsf{ML}}$ instead of $\sqsubseteq$ in rule Inst. Consequently, any typing derivation in ML is also a typing derivation in XML^F (which is not, however, a most principal derivation in general).

Theorem 8 *Any term typable in ML is typable in XML^F as such.*

Conversely, terms that are typable in XML^F are not necessarily typable in ML. Indeed, XML^F contains the full power of System F, but ML does not. However, given an unannotated term of XML^F, it does also typecheck in ML.

Unannotated XML^F terms are in ML We proof this by translating XML^F typing derivations of unannotated terms into ML typing derivations in two steps. First, rigid bindings are removed from the initial derivation by "flexifying" the derivation (Definition 4.4.1). The result which contains only flexible types, is still a valid derivation (Lemma 4.4.3). Last, all quantifiers are extruded to the outermost level. The final derivation is still correct and it is a derivation in ML (Lemma 4.4.7).

Definition 4.4.1 XML^F types that do not contain rigid bindings are said to be *flexible*. We say that a derivation is *flexible* if it does not contain any rigid bindings in types nor in any prefix appearing in the derivation. A judgment is flexible if it has a flexible derivation. □

Let flex be the function defined on XML^F types and prefixes that transforms every rigid binding into a flexible binding. For instance, $\mathsf{flex}\,(\forall\,(\alpha \Rightarrow \sigma_1, \beta \geq \sigma_2)\ \sigma)$ is $\forall\,(\alpha \geq \mathsf{flex}\,(\sigma_1), \beta \geq \mathsf{flex}\,(\sigma_2))\ \mathsf{flex}\,(\sigma)$. The following lemma shows that flexifying an instance relation is indeed correct.

Lemma 4.4.2

i) *If* $(Q)\ \sigma_1 \equiv \sigma_2$, *then* $(\mathsf{flex}\,(Q))\ \mathsf{flex}\,(\sigma_1) \equiv \mathsf{flex}\,(\sigma_2)$ *is flexible.*

ii) *If* $(Q)\ \sigma_1 \sqsupseteq \sigma_2$, *then* $(\mathsf{flex}\,(Q))\ \mathsf{flex}\,(\sigma_1) \sqsubseteq \mathsf{flex}\,(\sigma_2)$ *is flexible.*

iii) *If* $(Q)\ \sigma_1 \sqsubseteq \sigma_2$, *then* $(\mathsf{flex}\,(Q))\ \mathsf{flex}\,(\sigma_1) \sqsubseteq \mathsf{flex}\,(\sigma_2)$ *is flexible.*

(Proof p. 55)

We lift the function flex to typing environments and to typing judgments in the natural way. This operation preserves typing judgments.

Lemma 4.4.3 *If* $(Q)\ \Gamma \vdash a : \sigma$ *holds in* XML^F*, then so does* $\mathsf{flex}\ ((Q)\ \Gamma \vdash a : \sigma)$.

(Proof p. 55)

We recall a standard result of ML.

Lemma 4.4.4 *If we have* $\forall(\bar\alpha)\ \tau_1 \leq_{\mathsf{ML}} \forall(\bar\beta)\ \tau_2$*, then for any* σ *such that* $\mathsf{ftv}(\sigma) \# \bar\alpha \cup \bar\beta$*, we have* $\forall(\bar\alpha)\ \sigma[\tau_1/\gamma] \leq_{\mathsf{ML}} \forall(\bar\beta)\ \sigma[\tau_2/\gamma]$

We now show how a flexible XML^F type is transformed into an ML type by extrusion of quantifiers. We first transform prefixes. A flexible prefix is transformed into a pair whose first element is a set of quantifiers and the second element is a monotype substitution.

Definition 4.4.5 The ML approximation of a flexible prefix Q, written $\langle\!\langle Q \rangle\!\rangle$, and the ML approximation of a flexible type σ, written $\langle\!\langle \sigma \rangle\!\rangle$, are defined recursively as follows (overloading the notation used for System F should not raise any ambiguity):

$$\langle\!\langle \emptyset \rangle\!\rangle = (\emptyset, \mathsf{id}) \qquad \frac{\langle\!\langle Q \rangle\!\rangle = (\bar\alpha, \theta) \qquad \langle\!\langle \sigma \rangle\!\rangle = \forall(\bar\beta)\ \tau \qquad \bar\alpha \# \bar\beta}{\langle\!\langle Q, \alpha \geq \sigma \rangle\!\rangle = (\bar\alpha\bar\beta, \theta \circ [\tau/\alpha])} \qquad \langle\!\langle \bot \rangle\!\rangle = \forall(\alpha)\ \alpha \qquad \frac{\langle\!\langle Q \rangle\!\rangle = (\bar\alpha, \theta)}{\langle\!\langle \forall(Q)\ \tau \rangle\!\rangle = \forall(\bar\alpha)\ \theta(\tau)}$$

□

Notice that the ML approximation of a prefix Q is a pair $(\bar\alpha, \theta)$ that may be renamed. For example, the approximation of $(\alpha \geq \sigma_{\mathsf{id}})$ is $(\beta, [\beta \to \beta/\alpha])$ which is considered equivalent to the pair $(\gamma, [\gamma \to \gamma/\alpha])$. As a consequence, we may always assume freshness conditions on the new variables introduced by the approximation. We omit the details.

Lemma 4.4.6

i) *For any* σ *and any monotype substitution* θ*, we have* $\langle\!\langle \theta(\sigma) \rangle\!\rangle = \theta(\langle\!\langle \sigma \rangle\!\rangle)$.

ii) *If* $(Q)\ \sigma_1 \sqsubseteq \sigma_2$ *is flexible, and* $\langle\!\langle Q \rangle\!\rangle = (\bar\alpha, \theta)$*, then* $\theta(\langle\!\langle \sigma_1 \rangle\!\rangle) \leq_{\mathsf{ML}} \theta(\langle\!\langle \sigma_2 \rangle\!\rangle)$ *holds.*

(Proof p. 56)

We lift $\langle\!\langle \cdot \rangle\!\rangle$ to typing environments in the obvious way.

Lemma 4.4.7 *If there exists a flexible derivation of* $(Q)\ \Gamma \vdash a : \sigma$ *in* XML^F *then there exists a derivation of* $(\bar\alpha)\ \theta(\langle\!\langle \Gamma \rangle\!\rangle) \vdash a : \theta(\langle\!\langle \sigma \rangle\!\rangle)$ *in* ML *where* $(\bar\alpha, \theta)$ *is* $\langle\!\langle Q \rangle\!\rangle$.

(Proof p. 56)

Theorem 9 *Any unannotated term typable in* XML^F *under a flexible typing environment* Γ *(including an empty one) is typable in* ML *under* $\langle\!\langle \Gamma \rangle\!\rangle$.

(Proof p. 57)

4.5 Programs that we intendedly reject

We have shown that XML^F is a type system that is as powerful as System F (§4.3). While the encoding introduced an annotation on every λ-abstraction, these annotations may be omitted when the argument is not used polymorphically, as shown by the embedding of ML into XML^F.

Conversely, terms that are insufficiently annotated are rejected. Thus, although the full power of System F is available in XML^F, it must be gently introduced by means of explicit type annotations. Few type annotations are needed (the encoding of System F is already concise and still sometimes redundant), but some are mandatory: *annotations on function arguments that are used polymorphically.* This provides a clear intuition to the programmer with respect to *when* to put type annotations

Rather than a weakness, it is the strength of XML^F to enforce such annotations. Since such a clear difference can be made between implicit and explicit polymorphism and programs rejected accordingly, type inference never has to guess polymorphism and, as a result, is decidable (and tractable).

As an example of a function that requires an annotation, $\lambda(x)\ x\ x$ is not typable in XML^F—otherwise it would also be typable in ML. However, $\lambda(x : \sigma_{\mathsf{id}})\ x\ x$ is typable in XML^F, as discussed on page 31.

5 Related works

Our work continues a long line of research efforts concerned with type inference with first-class polymorphism. Unsurprisingly, this problem has been tackled from two opposite directions, either performing (partial) type inference for (variants of) System F and attempting to reach most of ML programs (§5.1), or encapsulating first-class polymorphic values within first-order ML types (§5.2) in more and more transparent ways.

5.1 Type inference for System F

Several interesting works on type inference for System-F-like type systems had already been carried out before it was proved to be undecidable for System F [Wel99] and for some of its variants.

Type containment. In the late 80's, Mitchell noticed that System F might not be the "right" system for studying type inference [Mit88]. He introduced the closure of System F by η-conversion, known as F$^{\eta}$, and showed that well-typedness in F$^{\eta}$ could be obtained by replacing the instance relation $\leq_{\mathsf{F}}$ by a larger relation $\leq_{\eta}$, called *type containment* (see §2.1). He also showed that uses of type-containment were equivalent to the applications of retyping functions—functions whose type-erasure η-reduces to the identity—in System F. Type inference for System F modulo η-expansion is now known to be also undecidable [Wel96].

Our treatment of type annotations as type-revealing primitives resembles the use of retyping functions. Moreover, $\leq_{\eta}$ and our type instance relation $\leq$ have a few interesting cases in common. However, they also differ significantly. Type containment is implicit, automatically driven by the type structure, and propagated according to polarities of occurrences (*e.g.* contravariantly on the left-hand side of arrows and covariantly anywhere else). By contrast, our type instantiation is always explicitly specified via flexibly bound variables, and may be used at occurrences of arbitrary polarities and in particular, it can be applied simultaneously at occurrences of opposite polarities, so that the weaker the argument, the weaker the result. Of course, typing rules will only allow type instantiation at some occurrences and will enforce non variance via rigid bindings anywhere else. As a result, the two relations are incomparable. The resemblance between type containment and ML$^{\mathsf{F}}$ is only superficial.

Polymorphic Subtyping. System F$_{<:}$ is another extension of System F with a richer instance relation $<:$ (see its definition in §2.1). In F$_{<:}$ as in ML$^{\mathsf{F}}$, each type variable is also given a bound. However, it is an upper bound in F$_{<:}$ while it is a lower bound in ML$^{\mathsf{F}}$. As for type-containment, the subtyping relation $<:$ is structural, which makes a huge difference with our instance relation. Type inference for F$_{<:}$ is undecidable. Even, type checking is undecidable for some variants of the subtyping relation $<:$ [Pie94].

Type inference based on second-order unification. Second-order unification, although known to be undecidable, has been used to explore the practical effectiveness of type inference for System F by Pfenning [Pfe88]. Despite our opposite choice, that is *not* to support second-order unification, there are at least two comparisons to be made. Firstly, Pfenning's work does not cover the language ML *per se*, but only the λ-calculus, since let-bindings are expanded prior to type inference. Indeed, ML is not the simply-typed λ-calculus and type inference in ML cannot, *in practice*, be reduced to type inference in the simply-typed λ-calculus after expansion of let-bindings. Secondly, one proposal seems to require annotations exactly where the other can skip them: Pfenning's system requires place holders (without type information) for type abstractions and type applications but never need type information on arguments of functions. Conversely, ML$^{\mathsf{F}}$ requires type information on *some* arguments of functions, but no information for type abstractions or applications.

While Pfenning's system relies on second-order unification to really infer polymorphic types, we strictly keep a first-order unification mechanism and never infer polymorphic types—we just propagate them. It might be interesting to see whether our form of unification could be understood as a particular case of second-order unification. For instance, using a constraint-based presentation of second-order unification [DHKP96], could flexible bounds help capture certain multi-sets of unification constraints in a more principal manner, and so reduce the amount of backtracking?

Another restriction of second-order unification is unification under a mixed prefix [Mil92]. However, our notion of prefix and its role in abstracting polytypes is quite different. In particular, mixed prefixes mention universal and existential quantification, whereas ML$^{\mathsf{F}}$ prefixes are universally quantified. Besides, ML$^{\mathsf{F}}$ prefixes associate a bound to each variable, whereas mixed prefixes are always unconstrained.

Partial type inference in System F. Several people have considered partial type inference for System F [Boe85, JWOG89, Pfe93, Sch98] and stated undecidability results for some particular variants. For instance, Boehm [Boe85] and Pfenning [Pfe93] considered programs of System F where λ-abstractions can be unannotated, and only the locations of type applications were given, not the actual type argument. They both showed that type reconstruction then becomes undecidable as it can encode second-order unification problems. The encoding introduces an unannotated λ-abstraction whose argument is used polymorphically. This is precisely what we avoid in ML$^{\mathsf{F}}$: all polymorphic λ-abstraction must be annotated, whereas type abstractions and type applications are inferred.

As another example, Schubert [Sch98] considers *sequent decision problems* for System F in both Curry's style and Church's style. They, in fact, correspond to type inference problems in System F, as already studied by Wells [Wel99], and are known to be undecidable. An inverse typing problem in Church's style System F consists in finding the typing environment Γ that makes a fully annotated program M typable. Schubert proves that this problem is undecidable in general by encoding a restricted form of second-order unification, which is then proved equivalent to the problem of termination for two-counters automatons. We see, that although the program is fully annotated, the knowledge of the typing environment is necessary to typecheck it in a decidable way. It is then unsurprising that systems with intersection types, and more generally systems aiming at principal typings, which have to infer both the type and the typing environment, are undecidable.

On the contrary, the typing context is always known in the approach followed in ML^F—-as in ML.

Decidable fragments of System F. Several approaches have considered fragments of System F, for which complete type reconstruction may be performed: Rank-2 polymorphic types [KW94], called Λ_2, and rank-2 intersection types [Jim95], called I_2 actually type the same programs. They have been generalized to even-rank polymorphic types and odd-rank intersection types [Jim00]. However, none of these system is compositional, because of the rank limitation: one may not abstract over arbitrary values of the language. Since first-class polymorphism is precisely needed to introduce a higher level of abstraction, we think this is a fundamental limitation that is not acceptable in practice. Besides, their type inference algorithm in Λ_2 requires rewriting programs according to some non-intuitive set of reduction rules. Hence, no simple intuitive specification of well-typedness is provided to the user. Worse, type inference can only be performed on full programs: it is thus not possible to split a program into several modules and typecheck them independently. Noticeably, I_2 has better properties than Λ_2, such as principal typings. However, the equivalence between I_2 and Λ_2 is shown by means of rewriting techniques; thus, although a typing in I_2 can be inferred in a modular way, it does not give a modular typing in Λ_2.

Intersection types and System E. Wells and Carlier have proposed a type system, called System E, that generalizes intersection types with expansion variables [CPWK04]. Although their work is quite different in nature, as type inference is undecidable and only a semi-algorithm is given, there are interesting connections to be made. In particular, both works attempt to share several derivations of a same term, using implicit sharing via expansion variables in the case of System E or more explicit sharing via auxiliary quantifiers in the case of ML^F.

Local type inference. Local type inference [Car93, PT98] uses typing constraints between adjacent nodes to propagate type information locally, as opposed to the global propagation that is performed by unification as used in ML^F (or ML). This technique is quite successful at leaving implicit many (but not all) eliminations of both subtyping and universal polymorphism. However, it usually performs poorly for their introduction forms, which remain mandatory in most cases. The technique has been tested in practice and ambiguous results were reported: While many *dummy* type annotations can be removed, a few of them remained necessary and sometimes in rather unpredictable ways. One difficulty arises from anonymous functions as well as so-called *hard-to-synthesize* arguments [HP99b].

The technique is actually fragile and does not resist to simple program transformations. As an example, the application $\mathsf{app}\ f\ x$ may be untypable with local type inference when f is polymorphic[9]. Principal types are ensured by finding a "best argument" each time a polymorphic type is instantiated. If no best argument can be found, the typechecker signals an error. Such errors do not exist in ML nor ML^F, where every typable expression has a principal type.

It should be noticed that finding a "best argument", and thus inferring principal types in local type systems is sometimes made more difficult because of the presence of subtyping, which ML^F does not consider. In particular, local type inference and its refinement described in the next paragraph are the only partial type inference techniques that deal with both second-order polymorphism and subtyping. The extension of ML^F with subtyping has not been explored at all.

Colored local type inference [OZZ01] is considered as an improvement over local type inference, although some terms typable in the latter are not typable in the former. It enriches local type inference by allowing only partial type information to be propagated.

[9]The problem disappears in the uncurrified form, but uncurrifying is not always possible, or it may amount to introducing anonymous functions with an explicit type annotation.

Stratified type inference. Beyond its treatment of subtyping, local type inference also brings the idea that explicit type annotations can be propagated up and down the source tree according to fixed well-defined rules. This can sometimes be viewed as a preprocessing pass on the source term, which we then called *stratified type inference*. When the preprocessing step is simple and intuitive it need not be defined through logical typing rules, but may instead be defined algorithmically. Such a mechanism was already used in the first prototype of ML^F to move annotations of toplevel definitions to annotations of their respective parameters. Here, stratified type inference is used as a secondary tool that helps writing annotations at different places rather than removing them. The use of stratified type inference as the main tool for performing type inference for System F has not be shown satisfactory, even for its rather limited predicative fragment [Rém05].

5.2 Embedding first-class polymorphism in ML

ML programmers did not wait for solutions to the problem of type inference with first-class polymorphic types to introduce them in existing languages. Boxing polymorphism is a backup solution that consists in embedding polymorphic values into first-class ML values. Initially introduced for existential types it was quickly applied to universal types and later turned into more and more sophisticated proposals, some of which are now in use in OCaml or Haskell.

Boxed polymorphism refers to the encapsulation of first-class polymorphic values into monomorphic ones via injection and projection functions. In their most basic version, injections and projections are explicit, even though, in practice, they can be attached to datatype constructors [LO94, Rém94]. Typically, preliminary type definitions are made for all polymorphic types that appear in the program. For instance, the following program defines two flavors of auto and applies them to id:

$$
\begin{array}{ll}
\mathsf{type\ sid} = \mathsf{Sid\ of} \ \forall\,(\alpha)\ \alpha \to \alpha & \\
\mathsf{let\ id} = \mathsf{Sid}\ (\lambda(x)\ x) & : \mathsf{sid} \\
\mathsf{let\ auto}_1 = \lambda(x)\ \mathsf{let\ Sid}\ z = x\ \mathsf{in}\ z\ z & : \forall\,(\beta)\ \mathsf{sid} \to (\beta \to \beta) \\
\mathsf{let\ auto}_2 = \lambda(x)\ \mathsf{let\ Sid}\ z = x\ \mathsf{in}\ z\ x & : \mathsf{sid} \to \mathsf{sid} \\
(\mathsf{auto}_1\ \ \mathsf{id},\ \mathsf{auto}_2\ \ \mathsf{id}) & : \forall\,(\beta)\ (\beta \to \beta) \times \mathsf{sid}
\end{array}
$$

The symbol Sid is both used as a constructor in the creation of the polymorphic value id (second line) and as a destructor when it appears on the left-hand side of let-bindings (third and fourth lines)—or in place of parameters as in $\lambda(\mathsf{Sid}\ x)\ x\ x$, which could be an alternative definition of auto_1. Notice the difference between auto_1 and auto_2: the former returns the unboxed identity while the latter returns the boxed identity. Here, the coercion between the two forms must be explicit. This may be quite annoying in practice, as already suggested by the involved encoding of System F into boxed polymorphism [OL96].

Poly-ML [GR99] is an improvement over this mechanism that replaces the projection from monotypes to polytypes by a simple place holder $\langle\cdot\rangle$, indicating the need for a projection but eluding the projection itself. It also introduces a notation $[\cdot : \sigma]$ for embedding polymorphic values into monomorphic ones, which alleviates the need for prior type definitions. The previous example may be rewritten as follows (where σ_{id} is a meta-level abbreviation for $\forall\,(\alpha)\ \alpha \to \alpha$):

$$
\begin{array}{lcl}
\mathsf{let\ id} = [\lambda(x)\ x : \sigma_{\mathsf{id}}] & : & [\sigma_{\mathsf{id}}] \\
\mathsf{let\ auto}_1 = \lambda(x : [\sigma_{\mathsf{id}}])\ \langle x\rangle\ \langle x\rangle & : & \forall\,(\beta)\ [\sigma_{\mathsf{id}}] \to (\beta \to \beta) \\
\mathsf{let\ auto}_2 = \lambda(x : [\sigma_{\mathsf{id}}])\ \langle x\rangle\ x & : & [\sigma_{\mathsf{id}}] \to [\sigma_{\mathsf{id}}] \\
(\mathsf{auto}_1\ \ \mathsf{id},\ \mathsf{auto}_2\ \ \mathsf{id}) & : & \forall\,(\beta)\ (\beta \to \beta) \times [\sigma_{\mathsf{id}}] \\
\langle \mathsf{id}\rangle & : & \forall\,(\beta)\ (\beta \to \beta)
\end{array}
$$

Explicit type information is still required when creating a polymorphic value (first line). Abstracting over an (unknown) polymorphic value also requires explicit type information (second and third lines). However, type information may be omitted when using a *known* polymorphic value (last line). In fact, polymorphism must always be *known* in order to be used. For instance, $\lambda(x)\ \langle x\rangle$ would be rejected. Notice that, we could also have written auto_1 as $\lambda(x : [\sigma_{\mathsf{id}}])\ \mathsf{let}\ z = \langle x\rangle\ \mathsf{in}\ z\ z$, so as to avoid repeating the projection, much as for the treatment of type annotations in XML^F. (But this is unsurprising, since XML^F was much inspired by Poly-ML.)

The progress made between boxed polymorphism and Poly-ML is significant, which can already be seen on the encoding of System F into Poly-ML— much simpler than the encoding into boxed polymorphism. Yet, Poly-ML is not quite satisfactory. In particular, each polymorphic value must still be embedded and so requires an explicit type annotation at its creation (first line). The explicit type information necessary to build a polymorphic value is utterly redundant: can a programmer accept to write down a type that is already inferred by the typechecker? Moreover, this information may be much larger than what one would need to write in System F. For example, $\Lambda\alpha.\ \lambda(x : \alpha \to \alpha)\ x$ must be encoded as $[\lambda(x)\ x : \forall\,(\alpha)\ (\alpha \to \alpha) \to (\alpha \to \alpha)]$.

Boxy types [VWJ06] goes one step-further than Poly-ML by removing the "coercion box" of Poly-ML from the level of expressions—retaining it only at the level of types. In Poly-ML one could define the ordinary polymorphic identity function $\lambda(x)\ x$ with (toplevel) type scheme $\forall\,(\alpha)\ \alpha \to \alpha$ or the boxed first-class polymorphic value $[\lambda(x)\ x : \forall\,(\alpha)\ \alpha \to \alpha]$ with polytype $[\forall\,(\alpha)\ \alpha \to \alpha]$. With boxy types, there is no syntax to express this difference and instead, it is left to the typechecker to infer which form is meant. While there is an obvious competition between these two forms, the typing rules are presented in an algorithmic fashion, *i.e.* as an algorithm, that (silently) resolves competing cases in favor of one or the other view. The type system has principal types, but with respect to its algorithmic specification. Unfortunately, it is unknown whether there is a logic specification of the type system equivalent to the algorithmic presentation. Actually, this is unlikely, as the logic rules would somehow have to encode the left-to-right evaluation order followed by the algorithmic rules.

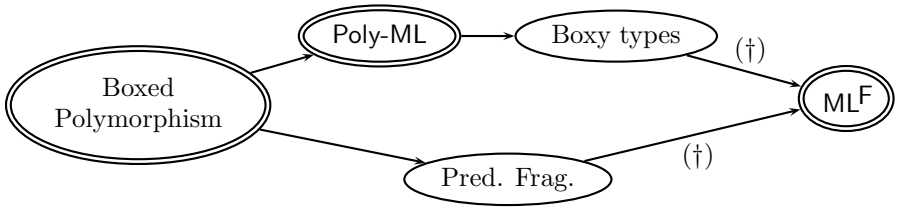
Because boxy types have an algorithmic specification it is difficult to compare them with ML^F, precisely. As they (arbitrarily) privilege propagation of type information from the function type to the argument type, they can type examples where ML^F would require an annotation and thus fail. Conversely, there are many examples that ML^F can type and that boxy types cannot—for some much deeper reason. In particular, if $a_1\ a_2$ is typable, then $\mathsf{app}\ a_1\ a_2$ is not necessarily typable with boxy types—a severe problem.

We believe that besides a few biases of the algorithmic propagation, boxy-types are significantly inferior to ML^F types. The authors of boxy types claim that one advantage is that expressions remains a subset of System F and that they can be elaborated as terms of explicit System F. In our opinion, this is rather a weakness than a strength as typing derivations in System F are not always as modular as desired, as argued above (§3.7). Moreover, $\mathsf{F}^{\mathsf{let}}$ or $\mathsf{F}_\wedge$ should be as valuable as System F when used as intermediate languages.

As boxy types, ML^F also removes the "coercion boxes" of Poly-ML, but do so in a more symmetric way, without arbitrary choices, by enriching the types of System F just as little as needed to represent all possible choices in derivations within (unique, principal) types.

(Partial) Type inference for the predicative fragment Odersky and Läufer have also extended boxed polymorphism to implicit predicative instantiation of rank-2 polymorphism [OL96], which was later improved to arbitrary-rank types by Peyton Jones and Shields [JS04, JVWS07]. Technically, this approach mixes local type inference with ML-style, unification-based type inference. However, this approach has two serious problems. Inherited from local type inference, and as boxy types it makes algorithmically specified, arbitrary choices. Moreover, the restriction to predicative polymorphism is far too drastic (see [Rém05] for detailed arguments). As a result, this approach is not sufficiently powerful and can only be used in combination with the more basic form of boxed polymorphism to recover the full power of System F. This results in a rather complicated language with several not well-integrated idioms to accomplish similar goals. In fact, this proposal seems to have been dropped in favor of boxy types by the common authors of both works.

Summary. The different proposals for embedding first-class polymorphism within first-class values can be schematically summarized below: arrows mean *is weaker than*; double lines are used for type systems enjoying principal types.



5.3 Comparison with (other presentations of) ML$^{\mathsf{F}}$

Comparison with the Original We recall that we restricted our study to Plain ML$^{\mathsf{F}}$ which version was named Shallow ML$^{\mathsf{F}}$ in earlier works. However, this correspondence is not exact. Our syntactic definition of XML$^{\mathsf{F}}$ differs from the original version. This is not just a matter of presentation—in fact, the original definition is simpler. The truth is that we enlarged the abstraction $\sqsubseteq$ (which forced us changing the presentation of type abstraction using the auxiliary relation $\sqsubseteq^\sharp$ to prevent pathological uses of unprotected abstraction that would break soundness, as explained above, in §4.1). In practice, the difference is unimportant. However, the new definition is really the most appropriate as its makes XML$^{\mathsf{F}}$ coincide exactly with IML$^{\mathsf{F}}$, whose type-instance relation, defined as sets of System-F types, is itself somehow canonical.

For instance, consider the types σ_1 and σ_2, respectively defined as $\forall\,(\alpha \geq \forall\,(\beta \Rightarrow \sigma_{\mathsf{id}})\ \beta \to \beta)\ \alpha \to \alpha$ and $\forall\,(\beta \Rightarrow \sigma_{\mathsf{id}})\ (\beta \to \beta) \to \beta \to \beta$. We have $\sigma_1 \sqsubseteq \sigma_2$, hence $\sigma_2 \sqsupseteq \sigma_1$ and $\sigma_1 \boxminus \sigma_2$. That is, types σ_1 and σ_2 can be (explicitly) converted to one another. However, in the original version we do not have $\sigma_1 \sqsubseteq_0 \sigma_2$ but only $\sigma_1 \sqsubseteq_0 \sigma_2$ (subscript 0 stands for the original version). There, only σ_1 is (implicitly) convertible to σ_2 by type instance but σ_2 is not convertible to σ_1. More precisely, the program $\lambda(x : \sigma_2)\ (x : \sigma_1)$ is in our version of ML$^{\mathsf{F}}$ but not in the original one.

This improvement was first suggested by François Pottier. However, its naive formalization in the original version was not correct, because of pathological contexts (see page 27). Quite interestingly, the improvement simplifies the presentation when taking a graphic presentation of type instance [RY07], by contrast with the syntactic presentation we have used.

There are other minor differences with the original presentation. For instance, the encoding of System F into XML$^{\mathsf{F}}$ we have presented is different from the original one, as explained page 32. More precisely, the abstraction relation defines a lattice over types, as shown in [LB04]. A given type of System F corresponds to several types in the lattice, any of which could be chosen as its default encoding. In [LB04], we chose the meet of all candidates, which is easier to build. In this paper, we chose their join, favoring the shortness of the proof.

Comparison with the graphical presentation A graphic presentation of ML$^{\mathsf{F}}$ types and type-instance has recently been proposed [RY07] and their application to graphical typing constraints is on going work. These works are complementary, as both bring different enlightenment on ML$^{\mathsf{F}}$ and its instance relation. In this work the instance relation is derived from a more canonical definition as sets of System F types. In the graphic presentation, it is derived from type instance on first-order types and natural simple operations on the binding tree. While, the former does not easily generalizes to Full ML$^{\mathsf{F}}$, the graphic presentation in fact directly defined in Full ML$^{\mathsf{F}}$. The graphic presentation is also targeted at performing efficient type inference, which we did not address in this work.

6 Conclusion and future work

In our quest for better integration of first-class polymorphism within ML, we have come up with ML$^{\mathsf{F}}$—a new type system for second-order polymorphism that is actually two-fold.

The Curry's style version IML$^{\mathsf{F}}$ just extends second-order types with flexible bindings so as to capture instances of a given type as a single type scheme. This is written $\forall\,(\alpha \geq \sigma)\ \sigma'$, meaning that α may be replaced by any instance of σ in σ'. Types schemes are interpreted by sets of System-F types. The instance relation on types is defined as set inclusion of their semantics. The language IML$^{\mathsf{F}}$ is a subset of F$^{\mathsf{let}}$, an extension of System F with a very restricted form of intersection types that contains exactly all let-expansions of System-F expressions.

We have also proposed a Church's style version XML$^{\mathsf{F}}$ that permits type inference. Expressions of XML$^{\mathsf{F}}$, given with some explicit type annotations, always have principal types[10]. Technically, XML$^{\mathsf{F}}$ introduces rigid bindings written $\forall\,(\alpha \Rightarrow \sigma)\ \sigma'$ to mediate between explicit type information σ and and its implicit view α within σ'. Interestingly, XML$^{\mathsf{F}}$ is a conservative extension of ML as fully unannotated programs are typable in XML$^{\mathsf{F}}$ if and only if they are typable in ML. Moreover, all System F programs can be turned into XML$^{\mathsf{F}}$ programs by simply dropping type abstractions and type applications and by a simple translation of type annotations. Interestingly, only function arguments that are used polymorphically need a type annotation in XML$^{\mathsf{F}}$. This provides a clear specification of *when* and *how* to annotated type parameters.

[10] This is shown in other works, not this article.

We believe that ML^F is a user-friendly extension of ML with first-class polymorphism. Additionally, without significantly departing from System F, programs in IML^F have "more principal" types than in System F and therefore are more modular.

Ongoing works

In this work, we have also explored the design space and related several variants of ML^F to a hierarchy of known existing languages. We have focused on $\mathsf{Plain}\ \mathsf{ML}^\mathsf{F}$ as it is a good compromise between its expressiveness and its simple and intuitive semantics: while $\mathsf{Simple}\ \mathsf{ML}^\mathsf{F}$ would loose useful modular properties, $\mathsf{Full}\ \mathsf{ML}^\mathsf{F}$ would loose a simple semantics in terms of System-F types.

However, from a syntactic point of view $\mathsf{Full}\ \mathsf{ML}^\mathsf{F}$ is not any harder than ML^F: unification and type inference algorithms remain the same in both cases but are only called on a subset of possible inputs in the case of ML^F. Therefore, the study of type inference has been left out of this paper, referring to its original presentation in the context of $\mathsf{Full}\ \mathsf{ML}^\mathsf{F}$ [LBR03, LB04]. More recently, a graph-based representation of types has been introduced and studied in some other parallel work [RY07]. Originally targeted at finding efficient unification and type inference algorithms, graphs also bring syntactic and semantic types closer. Interestingly, the graph representation applies indifferently to the plain or full versions, and might be a means to also give a semantics to $\mathsf{Full}\ \mathsf{ML}^\mathsf{F}$. Besides, the graph representation has enabled a new, efficient unification algorithm for ML^F types. In ongoing work, graphs are also used to revisit, improve and modularize type inference for ML^F.

Future works

The recent study of the interaction between ML^F and qualified types (upon which Haskell type classes are based) [LL05] opens the road to the integration of ML^F and $\mathsf{Haskell}$. However, ML^F-$\mathsf{Haskell}$'s current compilation schema into System-F terms is unsatisfactory as it passes useless coercion functions at runtime. Perhaps, F^let could be advantageously used instead of System F as the targeted language.

ML^F types are strictly—but only slightly—more expressive than System-F types. One may wonder whether they could be subsumed by higher-order types. We think that the two mechanisms are complementary and equally desired. We already see two solutions to higher-order polymorphism.

A limited form of higher-order polymorphism can be obtained with the use of higher-order kinds, which treats type operators as first-order type variables. This allows abstraction and instantiation of type operators, but not any reduction at the level of types. Technically it treats type application $F(\tau)$ where F is a type operator as an application $@(F, \tau)$ where $@$ is an application operator and F treated as a normal type, but of a higher-order kind. Then, there is not significant differences but keeping track of kinds. This should already work in ML^F without any problem. Interestingly, instantiation of type operators will be inferred as all other type-instantiations in ML^F.

Another solution to really integrate higher-order types is to reintroduce a new F-like universal quantifier in types with fully explicit type abstractions and type applications constructs in the language. As long as there is no implicit conversion between the two form of quantification, this should not raise any problem, even if explicit quantification is allowed at higher-order types. This would provide the full power of F^ω, but instantiation of type operators will always remain explicit.

Although, inferring instantiation of type operators seems possible in many interesting cases, it would not work in general. In fact, type-level computation resulting from instantiation of higher-order types seems to be conflicting with maintaining the precise sharing between polymorphic types at the core of ML^F. Studying this interaction remains an interesting research direction.

Extending ML^F with recursive types is another track. In the presence of type inference, one would expect implicit equi-recursive types to be used, as in ML. While we expect no difficulty with "monomorphic" recursion, the problem seems much harder when recursion crosses polymorphic boundaries. A solution might have to combine implicit monomorphic *equi* and explicit polymorphic *iso* recursive types, altogether.

Extensions of ML^F-types with subtyping, type constraints, assertions, *etc.* are of course also worth exploring.

A Proofs

Proof of Lemma 3.3.8

By structural induction on σ.

∘ Case α: Obviously, α is exposed in σ, therefore t must be monomorphic. To conclude, observe that both $\{\!\{\lfloor\theta\rfloor(\sigma)\}\!\}$ and $\theta(\{\!\{\sigma\}\!\})$ are equal to $\{t\}$.
∘ Case β with $\beta \neq \alpha$: Then, both $\{\!\{\lfloor\theta\rfloor(\sigma)\}\!\}$ and $\theta(\{\!\{\sigma\}\!\})$ are equal to $\{\beta\}$.
∘ Case $\tau_1 \to \tau_2$: Then, both $\{\!\{\lfloor\theta\rfloor(\sigma)\}\!\}$ and $\theta(\{\!\{\sigma\}\!\})$ are equal to $\{\theta(\tau_1) \to \theta(\tau_2)\}$.
∘ Case $\bot$: Let t' be in $\{\!\{\theta(\bot)\}\!\}$, that is $\{\!\{\bot\}\!\}$, with $\alpha \notin \mathsf{ftv}(t')$. Then $\theta(t') = t'$, which we may also write $t' \in \theta(\{t'\})$. Thus, $t' \in \theta(\{\!\{\bot\}\!\})$ holds.
∘ Case $\forall\,(\beta \geq \sigma_1)\ \sigma_2$: Let t' be in $\{\!\{\lfloor\theta\rfloor(\sigma)\}\!\}$ with $\alpha \notin \mathsf{ftv}(t')$ (**1**). We may assume $\beta \# \mathsf{dom}(\theta) \cup \mathsf{codom}(\theta)$ *w.l.o.g.* Then, $\lfloor\theta\rfloor(\sigma)$ is equal to $\forall\,(\beta \geq \lfloor\theta\rfloor(\sigma_1))\ \lfloor\theta\rfloor(\sigma_2)$. By Definition 3.2.1, t' is of the form $\forall\,(\bar\gamma)\ t_2[t_1/\beta]$ with $\bar\gamma \# \mathsf{ftv}(\lfloor\theta\rfloor(\sigma))$, $t_1 \in \{\!\{\lfloor\theta\rfloor(\sigma_1)\}\!\}$ and $t_2 \in \{\!\{\lfloor\theta\rfloor(\sigma_2)\}\!\}$ (**2**). We may assume $\alpha \notin \bar\gamma$, *w.l.o.g.* If α is not exposed in σ, it must also be not exposed in σ_1 and in σ_2 by Definition 3.3.6. Moreover, (1) implies both $\alpha \notin \mathsf{ftv}(t_2)$ and $\alpha \notin \mathsf{ftv}(t_1)$. By induction hypothesis applied to (2), we get $t_2 \in \theta(\{\!\{\sigma_2\}\!\})$ and $t_1 \in \theta(\{\!\{\sigma_1\}\!\})$. That is, t_1 and t_2 are of the form $\theta(t'_1)$ and $\theta(t'_2)$ with $t'_1 \in \{\!\{\sigma_1\}\!\}$ and $t'_2 \in \{\!\{\sigma_2\}\!\}$. Therefore, t' is equal to $\forall\,(\bar\gamma)\ \theta(t'_2)[\theta(t'_1)/\beta]$, which implies that t' is also equal to $\theta(\forall\,(\bar\gamma)\ t'_2[t'_1/\beta])$. By definition 3.2.1, we have $t' \in \theta(\{\!\{\sigma\}\!\})$, as expected. ■

Proof of Lemma 3.3.9

Each rule is considered separately.
∘ Case FE-Comm: Let σ_a be $\forall\,(\alpha_1 \geq \sigma_1)\ \forall\,(\alpha_2 \geq \sigma_2)\ \sigma$ and σ_b be $\forall\,(\alpha_2 \geq \sigma_2)\ \forall\,(\alpha_1 \geq \sigma_1)\ \sigma$, with $\alpha_1 \notin \mathsf{ftv}(\sigma_2)$ (**1**) and $\alpha_2 \notin \mathsf{ftv}(\sigma_1)$ (**2**). Our goal is to show $\{\!\{\sigma_a\}\!\} = \{\!\{\sigma_b\}\!\}$, which implies the expected result $\theta(\{\!\{\sigma_a\}\!\}) = \theta(\{\!\{\sigma_b\}\!\})$ for all $\theta \in \{\!\{Q\}\!\}$. By symmetry, it suffices to show $\{\!\{\sigma_a\}\!\} \subseteq \{\!\{\sigma_b\}\!\}$. Let t_a be an F-type in $\{\!\{\sigma_a\}\!\}$. We show that t_a is also in $\{\!\{\sigma_b\}\!\}$ (**3**). By Definition 3.2.1, t_a is of the form $\forall\,(\bar\beta)\ t'[t_1/\alpha_1]$ (**4**) with $\bar\beta \# \mathsf{ftv}(\sigma_a)$, $t' \in \{\!\{\forall\,(\alpha_2 \geq \sigma_2)\ \sigma\}\!\}$, and $t_1 \in \{\!\{\sigma_1\}\!\}$. By Definition 3.2.1, t' is in turn of the form $\forall\,(\bar\beta')\ t[t_2/\alpha_2]$ (**5**) with $\bar\beta' \# \mathsf{ftv}(\forall\,(\alpha_2 \geq \sigma_2)\ \sigma)$, $t \in \{\!\{\sigma\}\!\}$, and $t_2 \in \{\!\{\sigma_2\}\!\}$ (**6**). By α-conversion, we may assume, *w.l.o.g.*, $\alpha_2 \notin \mathsf{ftv}(t_1) \cup \{\alpha_1\}$ (**7**) and $\bar\beta' \# \mathsf{ftv}(t_1) \cup \{\alpha_1\} \cup \mathsf{ftv}(\sigma_1)$. By inlining (5) in (4), it appears that t_a is equal to $\forall\,(\bar\beta\bar\beta')\ t[t_2/\alpha_2][t_1/\alpha_1]$. By (7), we may commute the two substitutions in t_a and obtain $\forall\,(\bar\beta\bar\beta')\ t[t_1/\alpha_1][t_2[t_1/\alpha_1]/\alpha_2]$. Let t'_2 be $t_2[t_1/\alpha_1]$. It follows from (6) that t'_2 belongs to $\{\!\{\sigma_2\}\!\}[t_1/\alpha_1]$, which is included in $\{\!\{\sigma_2[t_1/\alpha_1]\}\!\}$ by Lemma 3.3.5. The latter equals $\{\!\{\sigma_2\}\!\}$, given (1). In summary, t_a is equal to $\forall\,(\bar\beta\bar\beta')\ t[t_1/\alpha_1][t'_2/\alpha_2]$ with $t'_2 \in \{\!\{\sigma_2\}\!\}$ and $\bar\beta\bar\beta' \# \mathsf{ftv}(\sigma_b)$, which implies (3) by Definition 3.2.1.
∘ Case FE-Free: Let σ with $\alpha \notin \mathsf{ftv}(\sigma)$. We show $\{\!\{\sigma\}\!\} = \{\!\{\forall\,(\alpha \geq \sigma')\ \sigma\}\!\}$ by considering both inclusions separately. Assume $t \in \{\!\{\sigma\}\!\}$. We may as well assume $\alpha \notin \mathsf{ftv}(t)$, *w.l.o.g.* Then, t is trivially of the form $t[t'/\alpha]$, by choosing some arbitrary t' in $\{\!\{\sigma'\}\!\}$, which implies $t \in \{\!\{\forall\,(\alpha \geq \sigma')\ \sigma\}\!\}$. Conversely, assume $t \in \{\!\{\forall\,(\alpha \geq \sigma')\ \sigma\}\!\}$. By Definition 3.2.1, it is of the form $\forall\,(\bar\beta)\ t''[t'/\alpha]$ with $t'' \in \{\!\{\sigma\}\!\}$ and $t' \in \{\!\{\sigma'\}\!\}$. Thus, $t \in \forall\,(\bar\beta)\ \{\!\{\sigma\}\!\}[t'/\alpha]$. By Lemma 3.3.5, this implies $t \in \forall\,(\bar\beta)\ \{\!\{\sigma[\lfloor t'\rfloor/\alpha]\}\!\}$, that is, $t \in \{\!\{\sigma\}\!\}$ by Lemma 3.3.4.
∘ Case FE-Var is by definition and Lemma 3.3.4.
∘ Case FE-Mono: by hypothesis, $(\alpha \geq \tau) \in Q$. Thus, Q is of the form $(Q_1, \alpha \geq \tau, Q_2)$ for some Q_1 and Q_2. Let θ be in $\{\!\{Q\}\!\}$. By definition, a canonical decomposition of θ is of the form $\theta_1 \circ [t_a/\alpha] \circ \theta_2$ (**8**) for some $\theta_1 \in \{\!\{Q_1\}\!\}$, $\theta_2 \in \{\!\{Q_2\}\!\}$, and $t_a \in \{\!\{\tau\}\!\}$. Since by hypothesis τ is a monotype, this implies $t_a = \lceil\tau\rceil$ (**9**). By Lemma 3.2.6, θ is equal to $\theta \circ [t_a/\alpha]$. Therefore, $\theta(\{\!\{\sigma\}\!\}) = \theta(\{\!\{\sigma\}\!\}[\lceil\tau\rceil/\alpha])$, which implies $\theta(\{\!\{\sigma\}\!\}) \subseteq \theta(\{\!\{\sigma[\tau/\alpha]\}\!\})$ by Lemma 3.3.5. As for the converse inclusion, let t be in $\theta(\{\!\{\sigma[\tau/\alpha]\}\!\})$. There exists t' such that $t = \theta(t')$ and $t' \in \{\!\{\sigma[\tau/\alpha]\}\!\}$. Let t'' be $t'[\lceil\tau\rceil/\alpha]$. We notice that $\theta(t'') = \theta \circ [\lceil\tau\rceil/\alpha](t') = \theta(t')$ (**10**) $= t$, where (10) is obtained by Lemma 3.2.6, (8), and (9). Besides, by Lemma 3.3.5, t'' belongs to $\{\!\{\sigma[\tau/\alpha]\}\!\}$. Since, by construction, $\alpha \notin \mathsf{ftv}(t'')$, we get $t'' \in \{\!\{\sigma\}\!\}[\lceil\tau\rceil/\alpha]$ as a consequence of Lemma 3.3.8, Therefore, t belongs to $\theta(\{\!\{\sigma\}\!\}[\lceil\tau\rceil/\alpha])$, that is, $\theta(\{\!\{\sigma\}\!\})$. ■

Proof of Property 3.3.11

The proof of (i) is by structural induction on the derivation. The proof of (ii) is by structural induction on σ. ■

Proof of Lemma 3.5.1

Necessarily, t is $\forall\,(\bar\alpha)\ \tau'$ and t' is $\forall\,(\bar\beta)\ \tau'[\bar\tau/\bar\alpha]$ with $\bar\beta \# \mathsf{ftv}(t)$. If τ' is some variable α with $\alpha \in \bar\alpha$, then $\lfloor t \rfloor$ is equivalent to $\bot$ by FE-Var, and we conclude directly by FI-Bot. From now on, we assume t is not a variable α in $\bar\alpha$. As a consequence, all α's in $\bar\alpha$ are not exposed in $\lfloor t \rfloor$ (**1**). We have

$$
\begin{array}{lcll}
\lfloor t \rfloor & = & \forall\,(\bar\alpha)\ \lfloor \tau' \rfloor & \\
& = & \forall\,(\bar\alpha \geq \bot)\ \lfloor \tau' \rfloor & \text{by notation} \\
& \boxminus & \forall\,(\bar\beta)\ \forall\,(\bar\alpha \geq \bot)\ \lfloor \tau' \rfloor & \text{by FE-Free} \\
& \leq & \forall\,(\bar\beta)\ \forall\,(\bar\alpha \geq \bar\tau)\ \lfloor \tau' \rfloor & \text{by FI-Bot and congruence} \\
& \leq & \forall\,(\bar\beta)\ \lfloor \tau' \rfloor[\lfloor \bar\tau \rfloor/\bar\alpha] & \text{by FI-Subst and (1)} \\
& = & \lfloor \forall\,(\bar\beta)\ \tau'[\bar\tau/\bar\alpha] \rfloor & \text{by definition} \\
& = & \lfloor t' \rfloor &
\end{array}
$$

We conclude by transitivity of $\leq$. ■

Proof of Theorem 1

By induction on the derivation of $\mathsf{F} :: \Gamma \vdash a : t$. Let Q be a prefix binding the free variables of Γ and t.

∘ Case Var: By hypothesis, we have $x : t \in \Gamma$. By definition of $\lfloor \Gamma \rfloor$, we have $x : \lfloor t \rfloor \in \lfloor \Gamma \rfloor$, and $\lfloor \Gamma \rfloor$ is closed under Q. Hence, $(Q)\ \lfloor \Gamma \rfloor \vdash x : \lfloor t \rfloor$ holds by Var.

∘ Case App: Then, a is of the form $a_1\ a_2$ and the premises are $\mathsf{F} :: \Gamma \vdash a_1 : t_2 \to t_1$ and $\mathsf{F} :: \Gamma \vdash a_2 : t_2$. Let Q' be an unconstrained prefix binding $\mathsf{ftv}(t_2) \setminus \mathsf{dom}(Q)$. This implies $\mathsf{dom}(Q') \# \mathsf{ftv}(\lfloor A \rfloor)$ (**1**) and $\mathsf{dom}(Q') \# \mathsf{ftv}(t_1)$ (**2**). By induction hypothesis, we have both $(QQ')\ \lfloor \Gamma \rfloor \vdash a_1 : \lfloor t_2 \to t_1 \rfloor$ (**3**) and $(QQ')\ \lfloor \Gamma \rfloor \vdash a_2 : \lfloor t_2 \rfloor$ (**4**). By definition, $\lfloor t_2 \to t_1 \rfloor$ is $\lfloor t_2 \rfloor \to \lfloor t_1 \rfloor$. Hence, (3) becomes $(QQ')\ \lfloor \Gamma \rfloor \vdash a_1 : \lfloor t_2 \rfloor \to \lfloor t_1 \rfloor$. By App, we get $(QQ')\ \lfloor \Gamma \rfloor \vdash a_1\ a_2 : \lfloor t_1 \rfloor$. By Gen and (1), we get $(Q)\ \lfloor \Gamma \rfloor \vdash a_1\ a_2 : \forall\,(Q')\ \lfloor t_1 \rfloor$. By equivalence (Imlf-Inst and FE-Free with (2)), we obtain the expect result $(Q)\ \lfloor \Gamma \rfloor \vdash a_1\ a_2 : \lfloor t_1 \rfloor$.

∘ Case Inst: The premises are $\mathsf{F} :: \Gamma \vdash a : t'$ and $t' \leq_{\mathsf{F}} t$ (**5**). By induction hypothesis, we have $(Q)\ \lfloor \Gamma \rfloor \vdash a : \lfloor t' \rfloor$ (**6**). By Lemma 3.5.1 and (5), we have $(Q)\ \lfloor t' \rfloor \leq \lfloor t \rfloor$. We conclude by Inst.

∘ Case Fun: The premise is $\mathsf{F} :: \Gamma, x : t_1 \vdash a : t_2$. By induction, we have $(Q)\ \lfloor \Gamma \rfloor, x : \lfloor t_1 \rfloor \vdash a : \lfloor t_2 \rfloor$. By Rule Fun, we get the expected result $(Q)\ \lfloor \Gamma \rfloor \vdash \lambda(x)\ a : \lfloor t_1 \rfloor \to \lfloor t_2 \rfloor$.

∘ Case Gen: The premise is $\mathsf{F} :: \Gamma \vdash a : t'$ and $\alpha \notin \mathsf{ftv}(\Gamma)$ (**7**). By induction, we have $(Q, \alpha)\ \lfloor \Gamma \rfloor \vdash a : \lfloor t' \rfloor$. Besides, (7) implies $\alpha \notin \mathsf{ftv}(\lfloor \Gamma \rfloor)$. By Rule Gen, we get $(Q)\ \lfloor \Gamma \rfloor \vdash a : \forall\,(\alpha)\ \lfloor t' \rfloor$, that is, $(Q)\ \lfloor \Gamma \rfloor \vdash a : \lfloor \forall\,(\alpha)\ t' \rfloor$. ■

Proof of Lemma 3.6.1

By induction on the derivation of $(Q)\ \Gamma \vdash a : \sigma$. We simultaneously show that for any $x : \wedge(t_i)^{i \in I}$ in Δ, we have $x : \sigma$ in Γ such that $(\theta(t_i) \in \theta(\{\!\{\sigma\}\!\}))^{i \in I}$. All cases are straightforward, except Imlf-Inst, which relies on Lemma 3.3.12. ■

Proof of Lemma 3.6.2

By induction on the derivation of $(Q \ni \theta)\ \Gamma \vdash a : \sigma \ni t \Rightarrow \Delta$. We show a stronger result, that is, $\mathsf{F}^{\mathsf{let}} :: \theta(\{\!\{\Gamma\}\!\}), \Delta \wedge \Delta' \vdash a : \theta(t)$ holds for any context Δ' such that $\mathsf{dom}(\Delta') \# \mathsf{dom}(\{\!\{\Gamma\}\!\})$ (**1**). All cases are straightforward, except Imlf-Gen, which can be shown as follows. We reuse the notations of rule Imlf-Gen. By definition, we must have $\theta(\forall\,(\bar\beta)\ t'[t/\alpha])$ in $\theta(\{\!\{\forall\,(\alpha \geq \sigma)\ \sigma'\}\!\})$. As the last rule of the derivation is Imlf-Gen, we must have $(Q, \alpha \geq \sigma \ni \theta \circ [t/\alpha])\ \Gamma \vdash a : \sigma' \ni t' \Rightarrow \Delta$ (**2**), $\alpha \notin \mathsf{ftv}(\Gamma)$ (**3**), and $\bar\beta \# \mathsf{dom}(Q)$ (**4**) hold. By induction hypothesis applied to (2), we get $\mathsf{F}^{\mathsf{let}} :: \theta \circ [t/\alpha](\{\!\{\Gamma\}\!\}), \Delta \wedge \Delta' \vdash a : \theta(t'[t/\alpha])$ (**5**) for any Δ' satisfying the hypothesis (1). From (3), we have $\theta \circ [t/\alpha](\{\!\{\Gamma\}\!\}) = \theta(\{\!\{\Gamma\}\!\})$. Thus, from (5), we have $\mathsf{F}^{\mathsf{let}} :: \theta(\{\!\{\Gamma\}\!\}), \Delta \wedge \Delta' \vdash a : \theta(t'[t/\alpha])$ (**6**). From (4), we have $\bar\beta \# \mathsf{ftv}(\{\!\{\Gamma\}\!\}) \cup \mathsf{ftv}(\Delta)$. By α-conversion, we may also assume $\bar\beta \# \mathsf{ftv}(\Delta')$, *w.l.o.g.* We may thus conclude with rule Gen applied to (6). ■

Proof of Theorem 2

This is an immediate consequence of Lemmas 3.6.1 and 3.6.2. ■

Proof of Lemma 4.1.9

Each statement is shown separately, by induction on the given derivation.

Let us consider the equivalence relation first (statement i). Reflexivity is immediate. Transitivity and symmetry are by induction hypothesis. As for congruence (rules XMLF-ALL-LEFT and XMLF-ALL-RIGHT), we consider two cases:

◦ CASE $\Rightarrow$-congruence: By hypothesis, σ_1 is $\forall\,(\alpha{\Rightarrow}\sigma_a)\ \sigma'_a$ and σ_2 is $\forall\,(\alpha{\Rightarrow}\sigma_b)\ \sigma'_b$. The premises are $(Q)\ \sigma_a \equiv \sigma_b$ and $(Q, \alpha \Rightarrow \sigma_b)\ \sigma'_a \equiv \sigma'_b$. We have to show $(Q')\ \theta(\llbracket\sigma'_a\rrbracket[\llbracket\sigma_a\rrbracket/\alpha]) \boxminus \theta(\llbracket\sigma'_b\rrbracket[\llbracket\sigma_b\rrbracket/\alpha])$. By induction hypothesis, we have $(Q')\ \theta\llbracket\sigma_a\rrbracket \boxminus \theta(\llbracket\sigma_b\rrbracket)$ (**1**) and $(Q')\ \theta(\llbracket\sigma'_a\rrbracket[\llbracket\sigma_b\rrbracket/\alpha]) \boxminus \theta(\llbracket\sigma'_b\rrbracket[\llbracket\sigma_b\rrbracket/\alpha])$ (**2**). By Lemma 3.3.11.ii (page 21) and (1), we have $(Q')\ \theta(\llbracket\sigma'_a\rrbracket)[\theta(\llbracket\sigma_a\rrbracket)/\alpha] \boxminus \theta(\llbracket\sigma'_a\rrbracket)[\theta(\llbracket\sigma_b\rrbracket)/\alpha]$, that is, $(Q')\ \theta(\llbracket\sigma'_a\rrbracket[\llbracket\sigma_a\rrbracket/\alpha]) \boxminus \theta(\llbracket\sigma'_a\rrbracket[\llbracket\sigma_b\rrbracket/\alpha])$. We conclude by transitivity and (2).

◦ CASE $\geq$-congruence: By hypothesis, σ_1 is $\forall\,(\alpha{\geq}\sigma_a)\ \sigma'_a$ and σ_2 is $\forall\,(\alpha{\geq}\sigma_b)\ \sigma'_b$. The premises are $(Q)\ \sigma_a \equiv \sigma_b$ and $(Q, \alpha \geq \sigma_b)\ \sigma'_a \equiv \sigma'_b$. By induction hypothesis, the former gives $(Q')\ \theta\llbracket\sigma_a\rrbracket \boxminus \theta\llbracket\sigma_b\rrbracket$ (**3**). We consider two subcases:

SUBCASE $\llbracket\sigma_b\rrbracket \notin \mathcal{V}$: Then, we get by induction hypothesis $(Q', \alpha \geq \theta(\llbracket\sigma_b\rrbracket))\ \theta\llbracket\sigma'_a\rrbracket \boxminus \theta\llbracket\sigma'_b\rrbracket$. By IMLF-ALL-LEFT, IMLF-ALL-RIGHT and (3), we get $(Q')\ \forall\,(\alpha \geq \theta(\llbracket\sigma_a\rrbracket))\ \theta\llbracket\sigma'_a\rrbracket \boxminus \forall\,(\alpha \geq \theta(\llbracket\sigma_b\rrbracket))\ \theta\llbracket\sigma'_b\rrbracket$ (**4**). If $\llbracket\sigma_a\rrbracket \notin \mathcal{V}$, this is the expected result. Otherwise, $\llbracket\sigma_a\rrbracket \equiv \beta$, which implies that $(Q')\ \theta(\llbracket\sigma_a\rrbracket) \boxminus \theta(\beta)$ holds by FE-FREE. Consequently, $(Q')\ \forall\,(\alpha \geq \theta(\llbracket\sigma_a\rrbracket))\ \theta\llbracket\sigma'_a\rrbracket \boxminus \forall\,(\alpha \geq \theta(\beta))\ \theta\llbracket\sigma'_a\rrbracket$ (**5**) holds by IMLF-ALL-LEFT. Besides, we have $(Q')\ \forall\,(\alpha \geq \theta(\beta))\ \theta\llbracket\sigma'_a\rrbracket \boxminus \theta(\llbracket\sigma'_a\rrbracket[\beta/\alpha])$ (**6**) by FE-MONO. We conclude by (6), (5), (4) and transitivity.

SUBCASE $\llbracket\sigma_b\rrbracket \in \mathcal{V}$ and $\llbracket\sigma_b\rrbracket \equiv \beta$: By induction hypothesis, we have $(Q')\ \theta(\llbracket\sigma'_a\rrbracket)[\theta(\beta)/\alpha] \boxminus \theta(\llbracket\sigma'_b\rrbracket)[\theta(\beta)/\alpha]$. By FE-FREE and (3), we have $(Q')\ \theta\llbracket\sigma_a\rrbracket \boxminus \theta(\beta)$ (**7**). If $\llbracket\sigma'_a\rrbracket \equiv \gamma$, then $(Q')\ \theta\llbracket\sigma_a\rrbracket \boxminus \theta(\gamma)$ by FE-FREE, and so we get $(Q')\ \theta(\gamma) \equiv \theta(\beta)$ from (7). We conclude by Lemma 3.3.11.ii (page 21) then. Otherwise, $\llbracket\sigma'_a\rrbracket \notin \mathcal{V}$. We conclude by (7), IMLF-ALL-LEFT, IMLF-ALL-RIGHT, and FE-MONO.

◦ CASE EQ-COMM: This rule commutes two binders. Each one is either flexible or rigid. Because of symmetry, we only have to consider three subcases: either both bindings are rigid, or both are flexible, or one is flexible and one is rigid. The first subcase is shown by commutation of the two substitutions. The second subcase is shown by Rule FE-COMM. The last subcase is by reflexivity.

◦ CASE EQ-VAR: We distinguish two subcases, depending on the binding being flexible or rigid. If it is flexible, we use FE-VAR. Otherwise, we use reflexivity.

◦ CASE EQ-FREE: Similarly, we use FE-FREE if the binding is flexible. Otherwise, we use reflexivity.

◦ CASE EQ-MONO: We use FE-MONO if the binding is flexible. Otherwise, we use reflexivity.

As for the abstraction relation (statement ii), transitivity is by induction hypothesis. Rigid-congruence (Rules A-LEFT, A-SHARP-LEFT) is shown as above, like for equivalence. Rule A-EQUIV is a consequence of i. Finally, Rule A-HYP is by reflexivity.

The instance relation (statement iii) is shown similarly. Transitivity is by induction hypothesis. Flexible-congruence is shown as above (see the equivalence case). I-ABSTRACT is a consequence of ii and FI-EQUIV. I-BOT is shown with FI-BOT. I-HYP is shown with FI-HYP. Finally, I-RIGID is shown with FI-SUBST, using Lemma 4.1.7 (page 29). ■

Proof of Lemma 4.1.10

By structural induction on σ. Both cases $\bot$ and β (with $\beta \neq \alpha$) are immediate. We remind that $\equiv$ is a subrelation of $\sqsubseteq^\sharp$.

◦ CASE $\sigma = \alpha$. Assume $(\alpha \geq \tau) \in Q$. We show $(\lceil Q\rceil)\ \alpha\ (\sqsubseteq^\sharp \cup \ni^\sharp)^*\ \lceil\tau\rceil$ (**1**) by cases on τ. If τ is a type variable γ, then (1) holds by EQ-MONO; otherwise, τ is an arrow type $\tau_1 \to \tau_2$ and the translation of $(\alpha \geq \tau)$, which is $(\alpha_1 \Rightarrow \lceil\tau_1\rceil, \alpha_2 \Rightarrow \lceil\tau_2\rceil, \alpha \geq \alpha_1 \to \alpha_2)$, appears in $\lceil Q\rceil$. Hence, we may derive

$$
\begin{array}{rcll}
(\lceil Q\rceil)\ \lceil\tau\rceil = & \forall\,(\alpha'_1 \Rightarrow \lceil\tau_1\rceil, \alpha'_2 \Rightarrow \lceil\tau_2\rceil)\ \alpha'_1 \to \alpha'_2 & \text{by Definition} \\
\sqsubseteq^\sharp & \forall\,(\alpha'_1 \Rightarrow \alpha_1, \alpha'_2 \Rightarrow \alpha_2)\ \alpha'_1 \to \alpha'_2 & \text{by A-HYP, A-SHARP-LEFT} \\
\equiv & \alpha_1 \to \alpha_2 & \text{by EQ-MONO} \\
\equiv & \alpha & \text{by EQ-MONO, Congruence}
\end{array}
$$

This is the expected result (1).

◦ CASE $\sigma = \tau_1 \to \tau_2$ is by induction hypothesis and A-SHARP-LEFT.

◦ CASE $\sigma = \forall\,(\alpha \geq \sigma_1)\ \sigma_2$ is by induction hypothesis and A-LEFT. ■

Proof of Lemma 4.1.12

i) is shown by induction on the derivation of $(Q)\ \sigma_1 \boxminus \sigma_2$. Reflexivity, transitivity, and symmetry are immediate by definition of $(\Subset \cup \exists)^*$.

- Case FE-Comm: by Rule Eq-Comm.
- Case FE-Free: by Rule Eq-Free.
- Case FE-Mono: by Lemma 4.1.10.
- Case FE-Var: by Rule Eq-Var.

Congruence of $\boxminus$ is defined by rules Imlf-All-Left, Imlf-All-Right, and Imlf-Arrow:

- Case Imlf-All-Left: by induction hypothesis and Rule A-Left.
- Case Imlf-All-Right: by induction hypothesis and Rule Xmlf-All-Right.
- Case Imlf-All-Arrow: σ_1 is $\tau_1 \to \tau_1'$ and σ_2 is $\tau_2 \to \tau_2'$. By hypothesis, both $(Q)\ \tau_1 \boxminus \tau_2$ and $(Q)\ \tau_1' \boxminus \tau_2'$ hold, and so by induction hypothesis, we get both $(\llbracket Q \rrbracket)\ \llbracket \tau_1 \rrbracket\ (\Subset^\sharp \cup \exists^\sharp)^*\ \llbracket \tau_2 \rrbracket$ (**1**) and $(\llbracket Q \rrbracket)\ \llbracket \tau_1' \rrbracket\ (\Subset^\sharp \cup \exists^\sharp)^*\ \llbracket \tau_2' \rrbracket$ (**2**). by definition, $\llbracket \tau_1 \to \tau_1' \rrbracket$ is $\forall\,(\alpha_1 \Rightarrow \llbracket \tau_1 \rrbracket, \alpha_1' \Rightarrow \llbracket \tau_1' \rrbracket)\ \alpha_1 \to \alpha_1'$. by Rule A-Left, (1) and (2), we get $(\llbracket Q \rrbracket)\ \llbracket \tau_1 \to \tau_1' \rrbracket\ (\Subset^\sharp \cup \exists^\sharp)^*\ \forall\,(\alpha_1 \Rightarrow \llbracket \tau_2 \rrbracket, \alpha_1' \Rightarrow \llbracket \tau_2' \rrbracket)\ \alpha_1 \to \alpha_1'$, that is, $(\llbracket Q \rrbracket)\ \llbracket \tau_1 \to \tau_1' \rrbracket\ (\Subset^\sharp \cup \exists^\sharp)^*\ \llbracket \tau_2 \to \tau_2' \rrbracket$, which is the expected result.

ii) is shown by induction on the derivation of $(Q)\ \sigma_1 \leq \sigma_2$. Transitivity is by definition of $(\sqsubseteq \cup \exists)^*$.

- Case FI-Equiv: by property i), and rules A-Sharp-Drop and I-Abstract.
- Case FI-Bot: by I-Bot
- Case FI-Hyp: by I-Hyp, A-Hyp and A-Sharp-Left.
- Case FI-Subst: by I-Rigid and Lemma 4.1.11.

The $\geq$-congruence of $\leq$ is defined by the rules Imlf-All-Left and Imlf-All-Right.

- Case Imlf-All-Left: by induction hypothesis and Rule Xmlf-Flex-Left.
- Case Imlf-All-Right: by induction hypothesis and Rule Xmlf-All-Right.

■

Proof of Theorem 3

By a simple induction on the typing derivation of $(Q)\ \Gamma \vdash a : \sigma$. The interesting cases are Annot and Inst, which immediately follow from properties 4.1.9.ii (page 29) and 4.1.9.iii (page 29). ■

Proof of Theorem 4

The proof is constructive as it explicitly builds the translated term a'. Thus, an algorithm that returns a' given a derivation of a could extracted from the proof. By induction on the derivation of $\mathsf{IML}^\mathsf{F} :: (Q)\ \Gamma \vdash a : \sigma$.

- Case Var: necessarily, a is a variable x such that $x : \sigma$ belongs to Γ, and so $x : \llbracket \sigma \rrbracket$ belongs to $\llbracket \Gamma \rrbracket$. We conclude by Rule Var, taking $a' = x$.
- Case Fun: a is of the form $\lambda(x)\ b$, σ is $\tau \to \tau'$, and the premise is $\mathsf{IML}^\mathsf{F} :: (Q)\ \Gamma, x : \tau \vdash b : \tau'$. By induction hypothesis, there exists b' (whose type erasure is b) such that $\mathsf{XML}^\mathsf{F} :: (\llbracket Q \rrbracket)\ \llbracket \Gamma \rrbracket, x : \llbracket \tau \rrbracket \vdash b' : \llbracket \tau' \rrbracket$ (**3**) holds. Let $\bar\alpha$ be $\mathsf{ftv}(\llbracket \tau \rrbracket)$ and a' be $\lambda(x : \exists\,(\bar\alpha)\ \llbracket \tau \rrbracket)\ b'$. We have

(**4**)	$(\llbracket Q \rrbracket)$	$\llbracket \Gamma \rrbracket, x : \llbracket \tau \rrbracket \vdash b' : \forall\,(\alpha_2 \Rightarrow \llbracket \tau' \rrbracket)\ \alpha_2$
(**5**)	$(\llbracket Q \rrbracket, \alpha_2 \Rightarrow \llbracket \tau' \rrbracket)$	$\llbracket \Gamma \rrbracket, x : \llbracket \tau \rrbracket \vdash b' : \alpha_2$
(**6**)	$(\llbracket Q \rrbracket, \alpha_2 \Rightarrow \llbracket \tau' \rrbracket)$	$\llbracket \Gamma \rrbracket \vdash a' : \forall\,(\alpha_1 \Rightarrow \llbracket \tau \rrbracket)\ \alpha_1 \to \alpha_2$
(**7**)	$(\llbracket Q \rrbracket)$	$\llbracket \Gamma \rrbracket \vdash a' : \forall\,(\alpha_1 \Rightarrow \llbracket \tau \rrbracket, \alpha_2 \Rightarrow \llbracket \tau' \rrbracket)\ \alpha_1 \to \alpha_2$
(**8**)	$(\llbracket Q \rrbracket)$	$\llbracket \Gamma \rrbracket \vdash a' : \llbracket \tau \to \tau' \rrbracket$

We get (4) by equivalence from (3) (Rule Inst and Rule Eq-Var). We obtain (5) by Rule UnGen*, and (6) by Rule Fun'. By Rule Gen, we get (7), which is equal to the expected result (8).

- Case App: a is $a_1\ a_2$, and the premises are $\mathsf{IML}^\mathsf{F} :: (Q)\ \Gamma \vdash a_1 : \tau_2 \to \tau_1$ (**9**) and $\mathsf{IML}^\mathsf{F} :: (Q)\ \Gamma \vdash a_2 : \tau_2$ (**10**), where τ_1 is σ. By induction hypothesis, there exist both a_1' and a_2' (whose type erasures are a_1 and a_2, respectively) such that $\mathsf{XML}^\mathsf{F} :: (\llbracket Q \rrbracket)\ \llbracket \Gamma \rrbracket \vdash a_1' : \forall\,(\alpha_2 \Rightarrow \llbracket \tau_2 \rrbracket)\ \forall\,(\alpha_1 \Rightarrow \llbracket \tau_1 \rrbracket)\ \alpha_2 \to \alpha_1$ (**11**) and $\mathsf{XML}^\mathsf{F} :: (\llbracket Q \rrbracket)\ \llbracket \Gamma \rrbracket \vdash a_2' : \llbracket \tau_2 \rrbracket$. The latter can as well be written $(\llbracket Q \rrbracket)\ \llbracket \Gamma \rrbracket \vdash a_2' : \forall\,(\alpha_2 \Rightarrow \llbracket \tau_2 \rrbracket)\ \alpha_2$ by equivalence (Rule Inst and Rule Eq-Var). We conclude by Rule App* take $a_1'\ a_2'$ for a'.
- Case Inst: The premises are $\mathsf{IML}^\mathsf{F} :: (Q)\ \Gamma \vdash a : \sigma'$ (**12**) and $(Q)\ \sigma' \leq \sigma$ (**13**). By induction hypothesis, there exists a_0 such that $\mathsf{XML}^\mathsf{F} :: (\llbracket Q \rrbracket)\ \llbracket \Gamma \rrbracket \vdash a_0 : \llbracket \sigma' \rrbracket$. By Lemma 4.1.12.ii and (13), $(\llbracket Q \rrbracket)\ \llbracket \sigma' \rrbracket\ (\sqsubseteq \cup \exists)^*\ \llbracket \sigma \rrbracket$

holds, that is, there exist a sequence $\sigma_0, \sigma_1, \ldots, \sigma_n$, with $\sigma_0 = \llbracket\sigma'\rrbracket$ and $\sigma_n = \llbracket\sigma\rrbracket$, such that $(\llbracket Q\rrbracket)\ \sigma_i \sqsubseteq \sigma_{i+1}$ if i is even, and $(\llbracket Q\rrbracket)\ \sigma_i \ni \sigma_{i+1}$ if i is odd. We show by induction on i, for $i \leqslant n$, that there exists a_i, whose type erasure is a, such that $(\llbracket Q\rrbracket)\ \llbracket\Gamma\rrbracket \vdash a_i : \sigma_i$ (**14**). Then, the expected result $(\llbracket Q\rrbracket)\ \llbracket\Gamma\rrbracket \vdash a' : \llbracket\sigma\rrbracket$ is obtained with $i = n-1$ and $a' = a_{n-1}$. The judgement (14) holds for $i = 0$, as shown above. By induction hypothesis, we assume it holds for some $i < n$. If i is even, we get judgement (14) for $i+1$ using Rule Inst. If i is odd, we get judgement (14) for $i+1$ using Rule Annot, taking $a_{i+1} = (a_i : \sigma_{i+1})$. *In fact, the sequence of instance and annot could be rearranged into a single instance followed by a single annotation, as* $(\leqslant \cup \ni)^* \subset (\leqslant \circ \ni)$. *However, a proof of this commutation lemma, which has been done in the graphical presentation, would be very tedious, syntactically.*

◦ Case Gen: The premise is $\mathsf{IML}^F :: (Q, \alpha \geq \sigma_1)\ \Gamma \vdash a : \sigma_2$, with $\alpha \notin \mathsf{ftv}(\Gamma)$, and σ is $\forall\,(\alpha \geq \sigma_1)\ \sigma_2$. By induction hypothesis, there exists a'' such that $\mathsf{XML}^F :: (\llbracket Q, \alpha \geq \sigma_1\rrbracket)\ \llbracket\Gamma\rrbracket \vdash a'' : \llbracket\sigma_2\rrbracket$ (**15**). Let $\forall\,(Q_1)\ \sigma_1'$ be $\llbracket\sigma_1\rrbracket$, with Q_1 rigid and as large as possible. Then (15) is $(\llbracket Q\rrbracket, Q_1, \alpha \geq \sigma_1')\ \llbracket\Gamma\rrbracket \vdash a'' : \llbracket\sigma_2\rrbracket$. We notice that the domain of Q_1 can be chosen disjoint from $\mathsf{ftv}(\Gamma)$. By repeated uses of Rule Gen, we get $(\llbracket Q\rrbracket)\ \llbracket\Gamma\rrbracket \vdash a'' : \forall\,(Q_1, \alpha \geq \sigma_1')\ \llbracket\sigma_2\rrbracket$. Since $(\llbracket Q\rrbracket)\ \forall\,(Q_1, \alpha \geq \sigma_1')\ \llbracket\sigma_2\rrbracket \ni \forall\,(\alpha \geq \forall\,(Q_1)\ \sigma_1')\ \llbracket\sigma_2\rrbracket$ holds by Xmlf-All-Right, A-Left, and repeated uses of A-Hyp, we conclude by Rule Annot, taking $a' = (a'' : \llbracket\sigma\rrbracket)$.

◦ Case Let: by induction hypothesis and Rule Let. ■

Proof of Lemma 4.3.7

This is a particular case of the following result, taking P, P', and P'' empty. In the following, P' may be an empty prefix or a prefix starting with an unconstrained binding. Similarly for P''. Also, P and Q must be rigid (Q is indeed rigid in the Lemma, since it is returned under an empty input prefix).

$$\frac{(PP')\ \langle\langle t\rangle\rangle : (PQP'', \alpha) \qquad PQQ'P'' \text{ well-formed}}{(PQQ'P')\ \langle\langle t\rangle\rangle : (PQQ'P'', \alpha)}$$

This is shown by structural induction on t. ■

Proof of Lemma 4.3.9

This lemma is a simplification of the invariant stated further, making also use of the following result, which is shown by structural induction on t (we omit its proof)

Assume $(Q)\ \langle\langle t\rangle\rangle : (Q', \alpha)$ *holds. Let* γ *be outside* $\mathsf{ftv}(t) \cup \mathsf{dom}(Q')$. *Then,* $(Q\gamma)\ \langle\langle t\rangle\rangle : (Q'\gamma, \alpha)$ *holds.*

Lemma 4.3.9 is not strong enough to be proved directly. Instead, we consider the following huge much stronger invariant, (we recover the Lemma by taking P, P_1', P_2' empty, $I = \mathsf{ftv}(t)$ and using the previous short result to introduce the unconstrained binding (α) in prefixes).

Assume Q_1 *rigid,* $Q_1 \equiv^I Q_2$, $\mathsf{ftv}(P) \subseteq I \cup \alpha$, *and* $\mathsf{dom}(Q_i'\alpha P_i') \cap I = \mathsf{dom}(Q_i\alpha P) \cap I$ *for* $i = 1, 2$. *Assume, moreover, that one of the following set of condition holds:*

$$\mathsf{ftv}(\sigma) \subseteq I \cup \alpha P \qquad Q_1\alpha P \otimes_{\alpha_1} \sigma = (Q_1'\alpha P_1', \alpha_1) \qquad Q_2\alpha P \otimes_{\alpha_2} \sigma = (Q_2'\alpha P_2', \alpha_2)$$

or

$$\mathsf{ftv}(t) \subseteq I \cup \alpha P \qquad (Q_1\alpha P)\ \langle\langle t\rangle\rangle : (Q_1'\alpha P_1', \alpha_1) \qquad (Q_2\alpha P)\ \langle\langle t\rangle\rangle : (Q_2'\alpha P_2', \alpha_2)$$

Then, there exists a set J *and a renaming* ϕ *such that*

$$\mathsf{dom}(\phi) \# I \cup \alpha P \qquad I \cup (\alpha_1 \cap Q_1') \subseteq J \qquad Q_1' \equiv^J \phi(Q_2') \qquad \phi(\alpha_2) = \alpha_1 \qquad \phi(P_2') = P_1'$$

$$\mathsf{ftv}(P_1') \subseteq J \cup \alpha$$

We show the result for each of the two sets of conditions separately. The first result is shown by induction on $Q_1 \equiv^I Q_2$. The second result is shown by structural induction on t.
First result: Instead of showing $\mathsf{ftv}(P_1') \subseteq J \cup \alpha$ (last predicate), we show $\mathsf{ftv}(P_1') \subseteq I \cup \alpha$, which is stronger. Also, the hypothesis $\mathsf{dom}(Q_i' \alpha P_i') \cap I = \mathsf{dom}(Q_i \alpha P_i) \cap I$ is equivalent to $\alpha_i \notin \mathsf{dom}(Q_i P_i) \Longrightarrow \alpha_i \notin I$.
∘ Case Reflexivity: We have $Q_1 = Q_2$. Let J be $I \cup (\alpha_1 \cap Q_1')$. If $\alpha_1 = \alpha_2$, we take $\phi = \mathsf{id}$. Otherwise, let ϕ be the renaming of domain $\{\alpha_1, \alpha_2\}$ that swaps α_1 and α_2. The binding $(\alpha_1 \Rightarrow \sigma)$ is inserted in Q_1 or in P. In both cases, $\phi(Q_2') = Q_1'$ and $\phi(P_2') = P_1'$ hold. The former implies $Q_1' \equiv^J \phi(Q_2')$ by reflexivity. Also, $\mathsf{ftv}(P_1') \subseteq \mathsf{ftv}(P) \cup \mathsf{ftv}(\sigma) - \mathsf{dom}(P)$ which implies $\mathsf{ftv}(P_1') \subseteq I \cup \alpha$.
∘ Case Transitivity: To ease readability (with respect to indices), we assume $Q_1 \equiv^I Q_2$ (**1**) and $Q_2 \equiv^I Q_3$ (**2**) hold and we show the conclusion where the index 2 is replaced by 3. We take α_2 such that $Q_2 \alpha P \otimes_{\alpha_2} \sigma$ is defined (it always exists). By renaming, we may also freely assume $\alpha_2 \notin \mathsf{dom}(Q_2 P_2) \Longrightarrow \alpha_2 \notin I$. The hypotheses are

$$\mathsf{ftv}(P) \subseteq I \cup \alpha \ (\mathbf{3}) \qquad \mathsf{ftv}(\sigma) \subseteq I \cup \alpha P \ (\mathbf{4}) \qquad Q_1 \alpha P \otimes_{\alpha_1} \sigma = (Q_1' \alpha P_1', \alpha_1) \qquad Q_2 \alpha P \otimes_{\alpha_2} \sigma = (Q_2' \alpha P_2', \alpha_2) \ (\mathbf{5})$$

$$Q_3 \alpha P \otimes_{\alpha_3} \sigma = (Q_3' \alpha P_3', \alpha_3) \ (\mathbf{6})$$

By induction hypothesis and (1), there exist J_1 and ϕ_1 such that

$$\mathsf{dom}(\phi_1) \# I \cup \alpha P \ (\mathbf{7}) \qquad I \cup (\alpha_1 \cap Q_1') \subseteq J_1 \ (\mathbf{8}) \qquad Q_1' \equiv^{J_1} \phi_1(Q_2') \ (\mathbf{9}) \qquad \phi_1(\alpha_2) = \alpha_1 \ (\mathbf{10})$$

$$\phi_1(P_2') = P_1' \ (\mathbf{11}) \qquad \mathsf{ftv}(P_1') \subseteq I \cup \alpha \ (\mathbf{12})$$

Note that (7), (3), and (4) imply $\phi_1(P) = P$ and $\phi_1(\sigma) = \sigma$. From (2), we get $\phi_1(Q_2) \equiv^I \phi_1(Q_3)$. From (5) and (6), we get the following:

$$\phi_1(Q_2) \alpha P \otimes_{\phi_1(\alpha_2)} \sigma = (\phi_1(Q_2') \alpha \phi_1(P_2'), \phi_1(\alpha_2)) \qquad \phi_1(Q_3) \alpha P \otimes_{\phi_1(\alpha_3)} \sigma = (\phi_1(Q_3') \alpha \phi_1(P_3'), \phi_1(\alpha_3))$$

By (10) and (11), the former gives

$$\phi_1(Q_2) \alpha P \otimes_{\alpha_1} \sigma = (\phi_1(Q_2') \alpha P_1', \alpha_1)$$

By induction hypothesis, there exist J_2 and ϕ_2 such that

$$\mathsf{dom}(\phi_2) \# I \cup \alpha P \ (\mathbf{13}) \qquad I \cup (\alpha_1 \cap \phi_1(Q_2')) \subseteq J_2 \ (\mathbf{14}) \qquad \phi_1(Q_2') \equiv^{J_2} \phi_2 \circ \phi_1(Q_3') \ (\mathbf{15})$$

$$\phi_2 \circ \phi_1(\alpha_3) = \alpha_1 \ (\mathbf{16}) \qquad \phi_2 \circ \phi_1(P_3') = P_1' \ (\mathbf{17})$$

Let ϕ be $\phi_2 \circ \phi_1$ and J be $J_1 \cap J_2$. We have to show the following:

$$\mathsf{dom}(\phi) \# I \cup \alpha P \cup \mathsf{ftv}(Q_3') \ (\mathbf{18}) \qquad I \cup (\alpha_1 \cap Q_1') \subseteq J \ (\mathbf{19}) \qquad Q_1' \equiv^J \phi(Q_3') \ (\mathbf{20}) \qquad \phi(\alpha_3) = \alpha_1 \ (\mathbf{21})$$

$$\phi(P_3') = P_1' \ (\mathbf{22}) \qquad \mathsf{ftv}(P_1') \subseteq I \cup \alpha \ (\mathbf{23})$$

We have (18) as a consequence of of (7) and (13). From (8) and (14), we have $I \subseteq J_1 \cap J_2$. Also, if $\alpha_1 \in \mathsf{dom}(Q_1')$, then $\alpha_1 \in J_1$ from (8) and $\alpha_1 \in \mathsf{dom}(\phi_1(Q_2'))$ from (9). The latter implies $\alpha_1 \in J_2$ from (14). Consequently, if $\alpha_1 \in Q_1'$, then $\alpha_1 \in J_1 \cap J_2$. This implies (19).
We get (20) from (9) and (15). Also, (21) is (16), (22) is (17), and (23) is (12). This concludes the case.
∘ Case Symmetry: by induction hypothesis and by taking $\phi^{-1}(J)$ for J and ϕ^{-1} for ϕ.
∘ Case Comm: Similar to reflexivity.
∘ Case Free: By hypothesis, Q_2 is $(Q_1, \beta \Rightarrow \sigma')$ with $\beta \notin I \cup \mathsf{dom}(Q_1) \cup \mathsf{ftv}(Q_1)$. Also, $\sigma' \notin \mathsf{bnds}(Q_1)$. We consider two subcases:
Subcase $\sigma \neq \sigma'$: This case is similar to reflexivity. Noticeably, if $\alpha_1 \in \mathsf{dom}(Q_1')$, we get $\phi(Q_2') \equiv Q_1'$ by Comm (commuting the bindings of α_1 and β), instead of $\phi(Q_2') = Q_1'$.

SUBCASE $\sigma = \sigma'$: Then, Q'_1 is $(Q_1, \alpha_1 \Rightarrow \sigma)$, $Q'_2 = Q_2$ and $\beta = \alpha_2$. If $\alpha_1 = \beta$, then we take $\phi = \mathsf{id}$ and $J = I \cup \alpha_1$. Otherwise, ϕ is the renaming of domain $\{\alpha_1, \beta\}$ swapping α_1 and β. In both cases, $Q'_1 \equiv^J \phi(Q'_2)$ is derivable by reflexivity.

Second result (by structural induction on t):
◦ CASE β: Then, $Q'_1 = Q_1$, $Q'_2 = Q_2$, $P'_1 = P'_2 = P$, and $\alpha_1 = \alpha_2 = \beta$. We get the expected result by taking $J = I$ and $\phi = \mathsf{id}$.
◦ CASE $t_1 \to t_2$: By hypothesis, we have $\mathsf{ftv}(t) \subseteq I \cup \alpha P$ (**24**) as well as

$$(Q_1 \alpha P) \langle\!\langle t_1 \rangle\!\rangle : (Q_1^a \alpha P_1^a, \alpha_1^a) \text{ (**25**)} \qquad (Q_2 \alpha P) \langle\!\langle t_1 \rangle\!\rangle : (Q_2^a \alpha P_2^a, \alpha_2^a) \text{ (**26**)} \qquad (Q_1^a \alpha P_1^a) \langle\!\langle t_2 \rangle\!\rangle : (Q_1^b \alpha P_1^b, \alpha_1^b) \text{ (**27**)}$$

$$(Q_2^a \alpha P_2^a) \langle\!\langle t_2 \rangle\!\rangle : (Q_2^b \alpha P_2^b, \alpha_2^b) \text{ (**28**)} \qquad (Q'_1 \alpha P'_1, \alpha_1) = Q_1^b \alpha P_1^b \otimes_{\alpha_1} \alpha_1^a \to \alpha_1^b \text{ (**29**)}$$

$$(Q'_2 \alpha P'_2, \alpha_2) = Q_2^b \alpha P_2^b \otimes_{\alpha_2} \alpha_2^a \to \alpha_2^b \text{ (**30**)}$$

By induction hypothesis, (25) and (26), there exist a set J_1 and a renaming ϕ_1 such that

$$\mathsf{dom}(\phi_1) \# I \cup \alpha P \text{ (**31**)} \qquad I \cup (\alpha_1^a \cap Q_1^a) \subseteq J_1 \qquad Q_1^a \equiv^{J_1} \phi_1(Q_2^a) \text{ (**32**)} \qquad \phi_1(\alpha_2^a) = \alpha_1^a \qquad \phi_1(P_2^a) = P_1^a$$

$$\mathsf{ftv}(P_1^a) \subseteq J_1 \cup \alpha$$

Note that (31) and (24) imply $\phi_1(t) = t$. By Lemma 4.3.5 (page 34) and (28), we get

$$(\phi_1(Q_2^a) \alpha P_1^a) \langle\!\langle t_2 \rangle\!\rangle : (\phi_1(Q_2^b) \alpha \phi_1(P_2^b), \phi_1(\alpha_2^b))$$

By induction hypothesis (taking J_1 for I), (32) and (27), there exist a set J_2 and a renaming ϕ_2 such that

$$\mathsf{dom}(\phi_2) \# J_1 \cup \alpha P_1^a \qquad J_1 \cup (\alpha_1^b \cap Q_1^b) \subseteq J_2 \qquad Q_1^b \equiv^{J_2} \phi_2 \circ \phi_1(Q_2^b) \qquad \phi_2 \circ \phi_1(\alpha_2^b) = \alpha_1^b \qquad \phi_2 \circ \phi_1(P_2^b) = P_1^b$$

$$\mathsf{ftv}(P_1^b) \subseteq J_2 \cup \alpha$$

Let ϕ' be $\phi_2 \circ \phi_1$. The results above can be rewritten like this:

$$\mathsf{dom}(\phi') \# I \cup \alpha P \qquad I \cup (\alpha_1^b \cap Q_1^b) \subseteq J_2 \qquad Q_1^b \equiv^{J_2} \phi'(Q_2^b) \qquad \phi'(\alpha_2^b) = \alpha_1^b \qquad \phi'(\alpha_2^a) = \alpha_1^a \qquad \phi'(P_2^b) = P_1^b$$

$$\mathsf{ftv}(P_1^b) \subseteq J_2 \cup \alpha$$

By applying ϕ' to (30), we get

$$(\phi'(Q'_2) \alpha \phi'(P'_2), \phi'(\alpha_2)) = \phi'(Q_2^b) \alpha P_1^b \otimes_{\phi'(\alpha_2)} \alpha_1^a \to \alpha_1^b$$

Then, by using the first result of the Lemma, there exists a set J and a renaming ψ such that

$$\mathsf{dom}(\psi) \# J_2 \cup \alpha P_1^b \qquad J_2 \cup (\alpha_1 \cap Q'_1) \subseteq J \qquad Q'_1 \equiv^J \psi \circ \phi'(Q'_2) \qquad \psi \circ \phi'(\alpha_2) = \alpha_1 \qquad \psi \circ \phi'(P'_2) = P'_1$$

$$\mathsf{ftv}(P'_1) \subseteq J \cup \alpha$$

Let ϕ be $\psi \circ \phi'$. We get

$$\mathsf{dom}(\phi) \# I \cup \alpha P \qquad I \cup (\alpha_1 \cap Q'_1) \subseteq J \qquad Q'_1 \equiv^J \phi(Q'_2) \qquad \phi(\alpha_2) = \alpha_1 \qquad \phi(P'_2) = P'_1 \qquad \mathsf{ftv}(P'_1) \subseteq J \cup \alpha$$

◦ Case $\forall\,(\beta)\ t_0$: By hypothesis, we have

$$(Q_1\alpha P\beta)\ \langle\!\langle t_0 \rangle\!\rangle : (Q_1^a\alpha P_1^a\beta P_1^b, \beta_1)\ (\mathbf{33}) \qquad (Q_2\alpha P\beta)\ \langle\!\langle t_0 \rangle\!\rangle : (Q_2^a\alpha P_2^a\beta P_2^b, \beta_2)\ (\mathbf{34})$$

$$(Q_1'\alpha P_1', \alpha_1) = Q_1^a\alpha P_1^a \otimes_{\alpha_1} \forall\,(\beta P_1^b)\ \beta_1\ (\mathbf{35}) \qquad (Q_2'\alpha P_2', \alpha_2) = Q_2^a\alpha P_2^a \otimes_{\alpha_2} \forall\,(\beta P_2^b)\ \beta_2\ (\mathbf{36})$$

By induction hypothesis, there exist a set J' and a renaming ϕ' such that

$$\mathsf{dom}(\phi') \# I \cup \alpha P\beta \qquad I \cup (\beta_1 \cap Q_1^a) \subseteq J' \qquad Q_1^a \equiv^{J'} \phi'(Q_2^a) \qquad \phi'(\beta_2) = \beta_1 \qquad \phi'(P_2^a\beta P_2^b) = P_1^a\beta P_1^b\ (\mathbf{37})$$

$$\mathsf{ftv}(P_1^a\beta P_1^b) \subseteq J' \cup \alpha\ (\mathbf{38})$$

Note that (37) implies both $\phi'(P_2^a) = P_1^a$ and $\phi'(P_2^b) = P_1^b$. Additionally, (38) implies both $\mathsf{ftv}(P_1^a) \subseteq J' \cup \alpha$ and $\mathsf{ftv}(P_1^b) \subseteq J' \cup \alpha\beta P_1^a$. From (36), we have

$$(\phi'(Q_2')\alpha\phi'(P_2'), \phi'(\alpha_2)) = \phi'(Q_2^a)\alpha P_1^a \otimes_{\phi'(\alpha_2)} \phi'(\forall\,(\beta P_2^b)\ \beta_2)$$

We note that $\phi'(\forall\,(\beta P_2^b)\ \beta_2)$ is an alpha-conversion of $\forall\,(\beta\phi'(P_2^b))\ \phi'(\beta_2)$, that is $\forall\,(\beta P_1^b)\ \beta_1$. By using the first result of the lemma, there exists a set J and a renaming ψ such that

$$\mathsf{dom}(\psi) \# J' \cup \alpha P_1^a \qquad J' \cup (\alpha_1 \cap Q_1') \subseteq J \qquad Q_1' \equiv^J \psi \circ \phi'(Q_2') \qquad \psi \circ \phi'(\alpha_2) = \alpha_1 \qquad \psi \circ \phi'(P_2') = P_1'$$

$$\mathsf{ftv}(P_1') \subseteq J \cup \alpha$$

Let ϕ be $\psi \circ \phi'$. We have

$$\mathsf{dom}(\phi) \# I \cup \alpha P \qquad I \cup (\alpha_1 \cap Q_1') \subseteq J \qquad Q_1' \equiv^J \phi(Q_2') \qquad \phi(\alpha_2) = \alpha_1 \qquad \phi(P_2') = P_1' \qquad \mathsf{ftv}(P_1') \subseteq J \cup \alpha$$

This is the expected result. ■

Proof of Corollary 4.3.10

Let (Q', α') be the translation of t under an empty prefix (formally, $(\emptyset)\ \langle\!\langle t \rangle\!\rangle : (Q', \alpha')$ (**1**) holds). We show below that, *for any σ in $\langle\!\langle t \rangle\!\rangle$, we have* $(Q)\ \sigma \equiv \forall\,(Q')\ \alpha'$ *(**2**) under any suitable Q.* Indeed, we then have $\forall\,(Q)\ \sigma_i \equiv \forall\,(Q')\ \alpha'$ for i in $\{1, 2\}$ and the result follows by symmetry and transitivity of $\equiv$.

Let us show (2). Let I be $\mathsf{ftv}(t)$ and σ in $\langle\!\langle t \rangle\!\rangle$. By hypothesis, there exist shared rigid prefixes Q_1 and Q_1' such that:

$$\sigma = \forall\,(Q_1')\ \alpha_1\ (\mathbf{3}) \qquad (Q_1)\ \langle\!\langle t \rangle\!\rangle : (Q_1', \alpha_1)\ (\mathbf{4}) \qquad I \# \mathsf{dom}(Q_1)\ (\mathbf{5})$$

Using (5) and Free repeatedly, one may derive $\emptyset \equiv^I Q_1$. By Lemma 4.3.9 (page 35), there exist a set J and a renaming ϕ such that

$$\mathsf{dom}(\phi) \# I\ (\mathbf{6}) \qquad I \cup \alpha_1 \subseteq J\ (\mathbf{7}) \qquad Q_1' \equiv^J \phi(Q')\ (\mathbf{8}) \qquad \phi(\alpha') = \alpha_1$$

From (8), (7), and Lemma 4.3.8 (page 34), we get $(Q)\ \forall\,(Q_1')\ \alpha_1 \equiv \forall\,(\phi(Q'))\ \alpha_1$ (**9**) under any suitable Q. The left-hand type is σ. The right-hand term is $\forall\,(\phi(Q'))\ \phi(\alpha')$, which is alpha-convertible to $\phi(\forall\,(Q')\ \alpha')$. Noting that $\mathsf{ftv}(Q') \subseteq I$ holds from (1), we get $\phi(\forall\,(Q')\ \alpha') = \forall\,(Q')\ \alpha'$ from (6). Therefore, (9) can be written $(Q)\ \sigma \equiv \forall\,(Q')\ \alpha'$, which is the expected result (2). ■

Proof of Property 4.3.11

Both properties are shown by induction on $Q_1 \Rrightarrow_\phi Q_2$. ■

Proof of Lemma 4.3.12

Each property is shown separately.

P-I: By hypothesis, $Q \otimes_{\alpha'} \sigma$ is defined and $\psi(Q) \Rrightarrow_{\phi} Q'$ (**1**) holds. Also, $\phi \circ \psi(\sigma) \Rrightarrow \sigma'$ (**2**) holds. There are two subcases:

SUBCASE $\alpha' \in \mathsf{dom}(Q)$: Then, $(\alpha' \Rightarrow \sigma) \in Q$ and $Q \otimes_{\alpha'} \sigma$ is Q. Therefore, $(\alpha' \Rightarrow \psi(\sigma)) \in \psi(Q)$. From (1) and (2), we get $(\phi(\alpha') \Rightarrow \sigma') \in Q'$. Consequently, $Q' \otimes_{\phi(\alpha')} \sigma'$ is defined and equals Q'. This is the expected result, taking $\phi' = \mathsf{id}$.

SUBCASE $\alpha' \notin \mathsf{dom}(Q)$: Let Q be $Q_0\alpha_1 Q_1..\alpha_n Q_n$ with $Q_1, .., Q_n$ rigid. From (1), we know that Q' equals $Q'_0\alpha_1 Q'_1..\alpha_n Q'_n$ with $Q'_1, .., Q'_n$ rigid and $\phi_{i-1} \circ .. \circ \phi_0 \circ \psi(Q_i) \Rrightarrow_{\phi_i} Q'_i$ holds for all $0 \leqslant i \leqslant n$. Besides, $\phi = \phi_n \circ .. \circ \phi_0$. Also, $\phi(\alpha') = \alpha'$ since $\mathsf{dom}(\phi) \subseteq \mathsf{dom}(Q)$.

Let i be the index corresponding to the insertion of $(\alpha' \Rightarrow \sigma)$. Then, $Q \otimes_{\alpha'} \sigma$ is $Q_0\alpha_1 Q_1..\alpha_i Q_i, (\alpha' \Rightarrow \sigma), ..\alpha_n Q_n$.

We distinguish two other subcases: either $(\gamma \Rightarrow \sigma') \in Q'$ for some γ or not. In the latter case, $Q' \otimes_{\alpha'} \sigma'$ is defined and equals $Q'_0\alpha_1 Q'_1..\alpha_i Q'_i, (\alpha' \Rightarrow \sigma'), ..\alpha_n Q'_n$. We note that $\phi_{i-1} \circ .. \circ \phi_0 \circ \psi(Q_i, \alpha' \Rightarrow \sigma) \Rrightarrow (Q'_i, \alpha' \Rightarrow \sigma')$. Therefore, $\psi(Q \otimes_{\alpha'} \sigma) \Rrightarrow_{\phi} Q' \otimes_{\alpha'} \sigma'$. This is the expected result taking $\phi' = \mathsf{id}$.

We now consider the last remaining case, when $(\gamma \Rightarrow \sigma') \in Q'$ for some γ. Let ϕ' be $[\gamma/\alpha']$. Then, $\psi(Q \otimes_{\alpha'} \sigma) \Rrightarrow_{\phi' \circ \phi} Q'$. This is the expected result since $Q' \otimes_{\gamma} \sigma'$ is Q' and γ equals $\phi' \circ \phi(\alpha')$.

P-II: by structural induction on t.

SUBCASE $t = \gamma$ (with $\gamma \neq \beta$ or $\gamma = \beta$): Then $Q'_1 = Q_1$ and $\alpha_1 = \gamma$. We conclude by taking $Q'_2 = Q_2$, $\alpha_2 = \psi(\gamma)$ and $\phi' = \mathsf{id}$.

SUBCASE $t_1 \to t_2$: By hypothesis, $(Q_1)\ \langle\!\langle t \rangle\!\rangle : (Q'_1, \alpha_1)$ holds. The premises are $(Q_1)\ \langle\!\langle t_1 \rangle\!\rangle : (Q^a_1, \alpha^a_1)$ (**3**) and $(Q^a_1)\ \langle\!\langle t_2 \rangle\!\rangle : (Q^b_1, \alpha^b_1)$ (**4**), and Q'_1 is $Q^b_1 \otimes_{\alpha_1} \alpha^a_1 \to \alpha^b_1$. By induction hypothesis and (3), there exist Q^a_2, α^a_2 and ϕ^a such that

$$(Q_2)\ \langle\!\langle \psi(t_1) \rangle\!\rangle : (Q^a_2, \alpha^a_2)\ (\mathbf{5}) \qquad \psi(Q^a_1) \Rrightarrow_{\phi^a \circ \phi} Q^a_2 \qquad \phi^a \circ \phi \circ \psi(\alpha^a_1) = \alpha^a_2$$

By induction hypothesis and (4), there exist Q^b_2, α^b_2 and ϕ^b such that

$$(Q^a_2)\ \langle\!\langle \psi(t_2) \rangle\!\rangle : (Q^b_2, \alpha^b_2)\ (\mathbf{6}) \qquad \psi(Q^b_1) \Rrightarrow_{\phi^b \circ \phi^a \circ \phi} Q^b_2\ (\mathbf{7}) \qquad \phi^b \circ \phi^a \circ \phi \circ \psi(\alpha^b_1) = \alpha^b_2$$

We note that $\phi^b \circ \phi^a \circ \phi \circ \psi(\alpha^a_1 \to \alpha^b_1)$ equals $\alpha^a_2 \to \alpha^b_2$ and so $\phi^b \circ \phi^a \circ \phi \circ \psi(\alpha^a_1 \to \alpha^b_1) \Rrightarrow \alpha^a_2 \to \alpha^b_2$ holds. Thus, Property P-I and (7), imply that there exists ϕ^c such that $\mathsf{dom}(\phi^c) \subseteq \{\alpha_1\}$ (**8**) and $\psi(Q'_1) \Rrightarrow_{\phi^c \circ \phi^b \circ \phi^a \circ \phi} Q^b_2 \otimes_{\phi^c \circ \phi^b \circ \phi^a \circ \phi(\alpha_1)} \alpha^a_2 \to \alpha^b_2$ (**9**). Let ϕ' be $\phi^c \circ \phi^b \circ \phi^a$. Let Q'_2 be $Q^b_2 \otimes_{\phi' \circ \phi(\alpha_1)} \alpha^a_2 \to \alpha^b_2$. From (5) and (6), we have $(Q_2)\ \langle\!\langle \psi(t) \rangle\!\rangle : (Q'_2, \alpha_2)$ by taking $\alpha_2 = \phi' \circ \phi(\alpha_1)$ (**10**). From (9), we have $\psi(Q'_1) \Rrightarrow_{\phi' \circ \phi} Q'_2$. Since α_1 is not β, we have $\psi(\alpha_1) = \alpha_1$ which gives $\alpha_2 = \phi' \circ \phi \circ \psi(\alpha_1)$ from (10). This is the expected result.

SUBCASE $\forall\,(\gamma)\ t'$: By hypothesis, $(Q_1)\ \langle\!\langle t \rangle\!\rangle : (Q'_1, \alpha_1)$ holds. The premise is $(Q_1\gamma)\ \langle\!\langle t' \rangle\!\rangle : (Q^a_1 \gamma Q^b_1, \delta)$ and Q'_1 is $Q^a_1 \otimes_{\alpha_1} \forall\,(\gamma Q^b_1)\ \delta$. We observe that $\psi(Q_1\gamma) \Rrightarrow_{\phi} Q_2\gamma$ holds. Thus, by induction hypothesis, there exist Q^a_2, Q^b_2, δ' and ϕ^a such that

$$(Q_2\gamma)\ \langle\!\langle \psi(t') \rangle\!\rangle : (Q^a_2 \gamma Q^b_2, \delta')\ (\mathbf{11}) \qquad \psi(Q^a_1 \gamma Q^b_1) \Rrightarrow_{\phi^a \circ \phi} Q^a_2 \gamma Q^b_2\ (\mathbf{12}) \qquad \phi^a \circ \phi \circ \psi(\delta) = \delta'\ (\mathbf{13})$$

We note that (12) implies that there exist ϕ^b and ϕ^c such that $\psi(Q^a_1) \Rrightarrow_{\phi^b \circ \phi} Q^a_2$ (**14**) and $\phi^b \circ \phi \circ \psi(Q^b_1) \Rrightarrow_{\phi^c} Q^b_2$ with $\phi^a = \phi^c \circ \phi^b$ (**15**). As a consequence, $\phi^b \circ \phi \circ \psi(\forall\,(\gamma Q^b_1)\ \delta) \Rrightarrow \forall\,(\gamma Q^b_2)\ \phi^c \circ \phi^b \circ \phi \circ \psi(\delta)$ holds. Remarking that $\phi^c \circ \phi^b \circ \phi \circ \psi(\delta)$ equals δ' from (13) and (15), we may use Property P-I with (14) which provides ϕ^d such that $\psi(Q'_1) \Rrightarrow_{\phi^d \circ \phi^b \circ \phi} Q^a_2 \otimes_{\phi^d \circ \phi^b \circ \phi(\alpha_1)} \forall\,(\gamma Q^b_2)\ \delta'$. Let ϕ' be $\phi^d \circ \phi^b$, α_2 be $\phi' \circ \phi(\alpha_1)$ (**16**) and Q'_2 be $Q^a_2 \otimes_{\alpha_2} \forall\,(\gamma Q^b_2)\ \delta'$ (**17**). We have $\psi(Q'_1) \Rrightarrow_{\phi' \circ \phi} Q'_2$, $(Q_2)\ \langle\!\langle \psi(t) \rangle\!\rangle : (Q'_2, \alpha_2)$ (from (11) and (17)), and $\phi' \circ \phi \circ \psi(\alpha_1) = \alpha_2$ (from (16) and noting that $\psi(\alpha_1) = \alpha_1$).

P-III: Let θ be the substitution $[t'/\beta]$. Property P-III is a particular case of the following rule, taking $P = \emptyset$:

$$\frac{(\emptyset)\ \langle\!\langle t' \rangle\!\rangle : (Q', \alpha) \qquad (Q'P)\ \langle\!\langle \psi(t) \rangle\!\rangle : (Q'P', \alpha')}{(Q'P)\ \langle\!\langle \theta(t) \rangle\!\rangle : (Q'P', \alpha')}$$

The proof is by structural induction on t.

- CASE $t = \gamma$ with $\gamma \neq \beta$: Then, $P' = P$ and $\alpha' = \gamma$. The result is immediate.

◦ Case $t = \beta$: Then, $\psi(t)$ is α and so P' is P and α' is α. Using Lemma 4.3.7 (page 34), we have $(Q'P)\ \langle\!\langle t' \rangle\!\rangle : (Q'P, \alpha)$, which is the expected result.
◦ Case $t_1 \to t_2$: by induction hypothesis.
◦ Case $\forall\,(\gamma)\ t_1$: by induction hypothesis. ■

Proof of Lemma 4.3.13

By structural induction on t. We assume $(Q_1\beta Q_2 P)\ \langle\!\langle t \rangle\!\rangle : (Q'_1\beta Q'_2 P', \alpha)$ holds (**1**).
◦ Case t is γ: In order for (1) to hold, α must be γ and $Q_1\beta Q_2 P$ must be $Q'_1\beta Q'_2 P'$ (exactly in the same order). Hence Q'_1 is Q_1 and Q'_2 is Q_2. We take Q'_3 for Q_3.
◦ Case t is $t_a \to t_b$: The premises of (1) are

$$(Q_1\beta Q_2 P)\ \langle\!\langle t_a \rangle\!\rangle : (Q^a_1\beta Q^a_2 P^a, \alpha_a)\ (\mathbf{2}) \qquad (Q^a_1\beta Q^a_2 P^a)\ \langle\!\langle t_b \rangle\!\rangle : (Q^b_1\beta Q^b_2 P^b, \alpha_b)\ (\mathbf{3})$$

$$(Q'_1\beta Q'_2 P', \alpha) = Q^b_1\beta Q^b_2 P^b \otimes_\alpha \alpha_a \to \alpha_b\ (\mathbf{4})$$

By induction hypothesis applied to (2) and (3), we get $(Q_3 P)\ \langle\!\langle t_a \rangle\!\rangle : (Q^a_3 P^a, \alpha_a)$ with $Q^a_1 Q^a_2 \approx Q^a_3$ and $(Q^a_3 P^a)\ \langle\!\langle t_b \rangle\!\rangle : (Q^b_3 P^b, \alpha_b)$ with $Q^b_1 Q^b_2 \approx Q^b_3$. Let Q''_3 be $Q^b_3 P^b \oplus_\alpha \alpha_a \to \alpha_b$ (**5**). We have $(Q_3 P)\ \langle\!\langle t \rangle\!\rangle : (Q''_3, \alpha)$.

It remains to show that Q''_3 is of the form $Q'_3 P'$ with $Q'_3 \approx Q_3$. If $\alpha \in \mathsf{dom}(P')$, then Q'_1 is Q^b_1 and Q'_2 is Q^b_2 and $\alpha \notin \mathsf{dom}(Q'_1 Q'_2)$. It follows from (4) that α was inserted in at most one of Q^b_1, Q^b_2, or P^b, and as left as possible. If α was inserted in Q^b_1 or Q^b_2, leaving P' equal to P_b then (5) would also insert α in Q^b_3, leaving P_b unchanged, hence Q''_3 is of the form $Q'_3 P'$ and $Q'_3 \approx Q'_1 Q'_2$. Otherwise, α could not be inserted in $Q^b_1 Q^b_2$ and was inserted in P^b leading to P'. Thus (5) could not either insert α in Q^b_3 but in P'. Hence again, Q''_3 is of the form $Q'_3 P'$ and $Q'_3 \approx Q'_1 Q'_2$.
◦ Case t is $\forall\,(\gamma)\ t_a$: For (1) to hold, we must have $(Q'_1\beta Q'_2 P', \alpha)$ equal to $Q^a_1\beta Q^a_2 P^a \otimes_\alpha \forall\,(\gamma P^b)\ \alpha'$ and $(Q_1\beta Q_2 P\gamma)\ \langle\!\langle t_a \rangle\!\rangle : (Q^a_1\beta Q^a_2 P^a \gamma P^b, \alpha')$ (**6**) By induction hypothesis applied to (6), we get $(Q_3 P\gamma)\ \langle\!\langle t_a \rangle\!\rangle : (Q^a_3 P^a \gamma P^b, \alpha')$ with $Q^a_1 Q^a_2 \approx Q^a_3$. Let $Q'_3 P'$ be $Q^a_3 P^a \otimes_\alpha \forall\,(\gamma P^b)\ \alpha'$. It remains only to show that $Q'_3 \approx Q'_1 Q'_2$. This follows by a case analysis as in the previous case. ■

Proof of Theorem 5

By induction on $\mathsf{F} :: \Gamma' \vdash a : t$ (**1**). Let Γ be in $\langle\!\langle \Gamma' \rangle\!\rangle$. Thanks to Corollary 4.3.10 (page 35) and Inst, it suffices to show $\mathsf{XML}^{\mathsf{F}} :: (Q)\ \Gamma \vdash a' : \sigma$ for *one* σ in $\langle\!\langle t \rangle\!\rangle$ instead of *all* of them. By default, we let typing judgment be in $\mathsf{XML}^{\mathsf{F}}$.
◦ Case F-Var: a is x and (1) implies $x : t \in \Gamma'$. Hence, $x : \sigma \in \Gamma$ with $\sigma \in \langle\!\langle t \rangle\!\rangle$. By Var, we have $(Q)\ \Gamma \vdash x : \sigma$, which is the expected result.
◦ Case F-App: a is $a_1\ a_2$ and (1) implies $\mathsf{F} :: \Gamma' \vdash a_1 : t_2 \to t$ (**2**) and $\mathsf{F} :: \Gamma' \vdash a_2 : t_2$ (**3**). Let $\forall\,(Q')\ \alpha$ be in $\langle\!\langle t_2 \to t \rangle\!\rangle$. By definition, there exists a prefix Q_0 such that $(Q_0)\ \langle\!\langle t_2 \to t \rangle\!\rangle : (Q', \alpha)$ holds. The premises of this judgment are $(Q_0)\ \langle\!\langle t_2 \rangle\!\rangle : (Q_1, \alpha_1)$ and $(Q_1)\ \langle\!\langle t \rangle\!\rangle : (Q_2, \alpha_2)$ (**4**) with Q' being $Q_2 \otimes_\alpha \alpha_1 \to \alpha_2$ (**5**)

By induction hypothesis and (2), there exists a'_1 such that $(Q)\ \Gamma \vdash a'_1 : \forall\,(Q')\ \alpha$. We note from (5) that $(Q)\ \forall\,(Q')\ \alpha \equiv \forall\,(Q_2)\ \alpha_1 \to \alpha_2$ holds by Eq-Var. Thus, by Inst and Eq-Var, we have $(Q)\ \Gamma \vdash a'_1 : \forall\,(Q_2)\ \alpha_1 \to \alpha_2$ (**6**).

By induction hypothesis and (3), there exists a'_2 such that $(Q)\ \Gamma \vdash a'_2 : \forall\,(Q_1)\ \alpha_1$. Since $Q_1 \subseteq Q_2$ holds, we have $(Q)\ \forall\,(Q_1)\ \alpha_1 \equiv \forall\,(Q_2)\ \alpha_1$ by Eq-Free. Consequently, $(Q)\ \Gamma \vdash a'_2 : \forall\,(Q_2)\ \alpha_1$ (**7**) holds by Inst.

From App* (page 22), (6) and (7), we get $(Q)\ \Gamma \vdash a'_1\ a'_2 : \forall\,(Q_2)\ \alpha_2$. We conclude by taking $a' = a'_1\ a'_2$ and noting that $\forall\,(Q_2)\ \alpha_2 \in \langle\!\langle t \rangle\!\rangle$ holds from (4).
◦ Case F-Fun: a is $\lambda(x)\ a_1$, and (1) implies that t is $t_1 \to t_2$ and $\mathsf{F} :: \Gamma', x : t_1 \vdash a_1 : t_2$ (**8**) holds. Let $\forall\,(Q')\ \alpha$ be in $\langle\!\langle t_1 \to t_2 \rangle\!\rangle$. By definition, there exists Q_0 such that $(Q_0)\ \langle\!\langle t_1 \to t_2 \rangle\!\rangle : (Q', \alpha)$ holds. The premises are $(Q_0)\ \langle\!\langle t_1 \rangle\!\rangle : (Q_1, \alpha_1)$ and $(Q_1)\ \langle\!\langle t_2 \rangle\!\rangle : (Q_2, \alpha_2)$ with Q' being $Q_2 \otimes \alpha_1 \to \alpha_2$ (**9**).

By induction hypothesis and (8), there exists a'_1 such that $(QQ_2)\ \Gamma, x : \forall\,(Q_1)\ \alpha_1 \vdash a'_1 : \forall\,(Q_2)\ \alpha_2$ holds. We may freely assume that $\mathsf{dom}(Q_2) \# \mathsf{ftv}(\Gamma)$. Noting that $(QQ_2)\ \forall\,(Q_2)\ \alpha_2 \sqsubseteq \alpha_2$ holds by Share*, we derive $(QQ_2)\ \Gamma, x : \forall\,(Q_1)\ \alpha_1 \vdash a'_1 : \alpha_2$ by Inst. Let a' be $\lambda(x : \forall\,(Q_1)\ \alpha_1)\ a'_1$. By Fun', we get $(QQ_2)\ \Gamma \vdash a' : \forall\,(\beta_1 \Rightarrow \forall\,(Q_1)\ \alpha_1)\ \beta_1 \to \alpha_2$. Using Gen, we get $(Q)\ \Gamma \vdash a' : \forall\,(Q_2)\ \forall\,(\beta_1 \Rightarrow \forall\,(Q_1)\ \alpha_1)\ \beta_1 \to \alpha_2$. We note that $Q_1 \subseteq Q_2$, hence $(Q)\ \forall\,(Q_1)\ \alpha_1 \equiv \forall\,(Q_2)\ \alpha_1$ (**10**) holds by Eq-Free. Then, the following holds:

$$\begin{array}{lll} & \forall\,(Q_2)\,\forall\,(\beta_1 \Rightarrow \forall\,(Q_1)\,\alpha_1)\,\beta_1 \to \alpha_2 & \\ \equiv & \forall\,(Q_2)\,\forall\,(\beta_1 \Rightarrow \forall\,(Q_2)\,\alpha_1)\,\beta_1 \to \alpha_2 & \text{from (10)} \\ \sqsubseteq & \forall\,(Q_2)\,\forall\,(\beta_1 \Rightarrow \alpha_1)\,\beta_1 \to \alpha_2 & \text{by SHIFT}^\star \\ \equiv & \forall\,(Q_2)\,\alpha_1 \to \alpha_2 & \text{by EQ-MONO} \\ \equiv & \forall\,(Q')\,\alpha & \text{from (9)} \end{array}$$

Thus, by INST, we get $(Q)\ \Gamma \vdash a' : \forall\,(Q')\,\alpha$. This is the expected result.

∘ CASE F-GEN: (1) implies that τ is of the form $\forall\,(\alpha)\,t'$ with $\alpha \notin \mathsf{ftv}(\Gamma')$ and $\mathsf{F} :: \Gamma' \vdash a : t'$ (**11**). Let Q' and α' be such that $(\emptyset)\ \langle\!\langle \forall\,(\alpha)\,t' \rangle\!\rangle : (Q', \alpha')$. The premise is $(\alpha)\ \langle\!\langle t' \rangle\!\rangle : (Q_1 \alpha Q_2, \beta)$ (**12**) and Q' is $Q_1 \otimes_{\alpha'} \forall\,(\alpha Q_2)\,\beta$ (**13**). By Lemma 4.3.13 (page 36) and (12), there exists Q'_3 such that $(\emptyset)\ \langle\!\langle t' \rangle\!\rangle : (Q_3, \beta)$ holds with $Q_3 \approx Q_1 Q_2$ (**14**). Thus, by induction hypothesis and (11), there exists a' such that $(Q\alpha)\ \Gamma \vdash a' : \forall\,(Q_3)\,\beta$ holds. From (14), we get $(Q\alpha)\ \forall\,(Q_3)\,\beta \equiv \forall\,(Q_1 Q_2)\,\beta$. Thus, $(Q\alpha)\ \Gamma \vdash a' : \forall\,(Q_1 Q_2)\,\beta$ holds by INST. By GEN, we get $(Q)\ \Gamma \vdash a' : \forall\,(\alpha Q_1 Q_2)\,\beta$. By EQ-COMM and INST, we get $(Q)\ \Gamma \vdash a' : \forall\,(Q_1 \alpha Q_2)\,\beta$. We get the expected result by noting that $(Q)\ \forall\,(Q')\,\alpha' \equiv \forall\,(Q_1 \alpha Q_2)\,\beta$ holds from (13).

∘ CASE F-INST: (1) implies that t is of the form $t_0[t'/\beta]$ and $\mathsf{F} :: \Gamma' \vdash a : \forall\,(\beta)\,t_0$ (**15**). Let (Q', α) be such that $(\emptyset)\ \langle\!\langle t' \rangle\!\rangle : (Q', \alpha)$ (**16**) and $\mathsf{ftv}(t) \# \mathsf{dom}(Q')$ (**17**) hold. We may freely assume that $\beta \notin \mathsf{dom}(Q') \cup \mathsf{ftv}(Q')$.

Let (Q_0, α_0) be such that $(Q')\ \langle\!\langle \forall\,(\beta)\ t_0 \rangle\!\rangle : (Q_0, \alpha_0)$. The premise of this judgment is $(Q'\beta)\ \langle\!\langle t_0 \rangle\!\rangle : (Q_1 \beta Q_2, \gamma)$ (**18**) and Q_0 is $Q_1 \otimes_{\alpha_0} \forall\,(\beta Q_2)\,\gamma$. By EQ-VAR, this implies $(Q)\ \forall\,(Q_0)\,\alpha_0 \equiv \forall\,(Q_1 \beta Q_2)\,\gamma$ (**19**). By induction hypothesis and (15), there exists a' such that $(Q)\ \Gamma \vdash a' : \forall\,(Q_0)\,\alpha_0$ holds. From (19) and INST, we get $(Q)\ \Gamma \vdash a' : \forall\,(Q_1 \beta Q_2)\,\gamma$ (**20**).

Let ψ be $[\alpha/\beta]$. From (18), we have $\beta \notin \mathsf{dom}(Q_1) \cup \mathsf{ftv}(Q_1)$. Therefore, $\psi(Q_1) = Q_1$. Then, we derive the following:

$$\begin{array}{lll} (Q) & \forall\,(Q_1 \beta Q_2)\,\gamma & \\ = & \forall\,(Q_1)\,\forall\,(\beta \geq \bot)\,\forall\,(Q_2)\,\gamma & \text{by notation} \\ \sqsubseteq & \forall\,(Q_1)\,\forall\,(\beta \geq \alpha)\,\forall\,(Q_2)\,\gamma & \text{by I-NIL and flexible-congruence} \\ \equiv & \forall\,(Q_1)\,\psi(\forall\,(Q_2)\,\gamma) & \text{by EQ-MONO and EQ-FREE} \\ = & \forall\,(\psi(Q_1 Q_2))\,\psi(\gamma) & \end{array}$$

Therefore, $(Q)\ \Gamma \vdash a' : \forall\,(\psi(Q_1 Q_2))\,\psi(\gamma)$ (**21**) holds from (20) by INST.

We know from (16) that Q' is already shared, that is, $Q' \Rrightarrow_{\mathsf{id}} Q'$ holds. Additionally, $\psi(Q') = Q'$. Thus, $\psi(Q') \Rrightarrow_{\mathsf{id}} Q'$ holds.

From (18) and Lemma 4.3.13, there exists Q_3 such that $(Q')\ \langle\!\langle t_0 \rangle\!\rangle : (Q'_3, \gamma)$ with $Q'_3 \approx Q_1 Q_2$ (**22**). Then, by Property P-II and (17), there exist Q'_2, α_2 and ϕ' such that

$$(Q')\ \langle\!\langle \psi(t_0) \rangle\!\rangle : (Q'_2, \alpha_2)\ (\mathbf{23}) \qquad \psi(Q_3) \Rrightarrow_{\phi'} Q'_2\ (\mathbf{24}) \qquad \phi' \circ \psi(\gamma) = \alpha_2\ (\mathbf{25})$$

From (21), (22), and INST, we get $(Q)\ \Gamma \vdash a' : \forall\,(\psi(Q_3))\,\psi(\gamma)$ (**26**). From (24) and Lemma 4.3.11.ii (page 35), we have $(Q)\ \forall\,(\psi(Q_3))\,\psi(\gamma) \sqsubseteq \forall\,(Q'_2)\,\phi' \circ \psi(\gamma)$. By (25), that is $(Q)\ \forall\,(\psi(Q_3))\,\psi(\gamma) \sqsubseteq \forall\,(Q'_2)\,\alpha_2$. Hence, we get $(Q)\ \Gamma \vdash a' : \forall\,(Q'_2)\,\alpha_2$ (**27**) by INST and PI1.

From (23) and P-III, we get $(Q')\ \langle\!\langle t \rangle\!\rangle : (Q'_2, \alpha_2)$, which implies $\forall\,(Q'_2)\,\alpha_2 \in \langle\!\langle t \rangle\!\rangle$ and so (27) is the expected result. ■

Proof of Property 4.4.2

Each property is proved by induction on the derivation. Properties i and iii are easy. As for ii, the case A-HYP is replaced by I-HYP and congruence is replaced by flexible congruence. Finally, I-RIGID is replaced by reflexivity (that is, by the equivalence relation). ■

Proof of Lemma 4.4.3

By induction on the derivation of $(Q)\ \Gamma \vdash a : \sigma$. Case VAR is immediate. Cases APP, FUN and LET are by induction hypothesis. Case INST is by Property 4.4.2.iii (page 37). Case GEN: We have $(Q)\ \Gamma \vdash a : \forall\,(\alpha \diamond \sigma_1)\,\sigma_2$, and the premise is $(Q, \alpha \diamond \sigma_1)\ \Gamma \vdash a : \sigma_2$, with $\alpha \notin \mathsf{ftv}(\Gamma)$. Note that $\alpha \notin \mathsf{ftv}(\mathsf{flex}\ (\Gamma))$ either. By induction hypothesis, $(\mathsf{flex}\ (Q), \alpha \geq \mathsf{flex}\ (\sigma_1))\ \mathsf{flex}\ (\Gamma) \vdash a : \mathsf{flex}\ (\sigma_2)$ holds. Hence, $(\mathsf{flex}\ (Q))\ \mathsf{flex}\ (\Gamma) \vdash a : \forall\,(\alpha \geq \mathsf{flex}\ (\sigma_1))\ \mathsf{flex}\ (\sigma_2)$ holds by GEN. By definition, this means $(\mathsf{flex}\ (Q))\ \mathsf{flex}\ (\Gamma) \vdash a : \mathsf{flex}\ (\forall\,(\alpha \diamond \sigma_1)\ \sigma_2)$, which is the expected result. ■

Proof of Property 4.4.6

Property i): It is a consequence of the following: If $\langle\!\langle Q \rangle\!\rangle = (\bar{\alpha}, \theta')$ and $\mathsf{dom}(\theta) \# \bar{\alpha} \cup \mathsf{dom}(Q)$, then $\langle\!\langle \theta(Q) \rangle\!\rangle = (\bar{\alpha}, \theta \circ \theta')$.
Property ii): As a preliminary result, we show the same property for equivalence, that is, if $(Q)\ \sigma_1 \equiv \sigma_2$ holds, then $\theta(\langle\!\langle \sigma_1 \rangle\!\rangle)$ and $\theta(\langle\!\langle \sigma_2 \rangle\!\rangle)$ are equivalent in ML. The proof is by induction on the derivation of $(Q)\ \sigma_1 \equiv \sigma_2$. Transitivity is by induction hypothesis. Reflexivity and symmetry are immediate: the ML equivalence relation is reflexive and symmetric. For congruence, the proof is similar to the one for the instance relation (see below). It remains to consider the following cases:

- Case Eq-Free: We may as well add or remove useless binders in an ML type scheme.
- Case Eq-Comm: We may as well commute binders in an ML type scheme.
- Case Eq-Var: By definition, $\langle\!\langle \forall\,(\alpha \geq \sigma)\ \alpha \rangle\!\rangle$ and $\langle\!\langle \sigma \rangle\!\rangle$ are identical.
- Case Eq-Mono: Then σ_2 is of the form $\sigma_1[\tau/\alpha]$ and the premise is $(\alpha \geq \tau) \in Q$. Hence, Q is of the form $(Q_1, \alpha \geq \tau, Q_2)$. Therefore, θ equals $\theta_1 \circ [\tau/\alpha] \circ \theta_2$ with θ_1 and θ_2 being the substitutions associated with Q_1 and Q_2 respectively. By well-formedness, $\mathsf{ftv}(\tau) \# \mathsf{dom}(\theta_2)$ and $\alpha \notin \mathsf{dom}(\theta_2)$. As a consequence, we also have $\theta = \theta \circ [\tau/\alpha]$. We have $\langle\!\langle \sigma_2 \rangle\!\rangle = \langle\!\langle \sigma_1 \rangle\!\rangle[\tau/\alpha]$ from Property i). Then, $\theta(\langle\!\langle \sigma_1 \rangle\!\rangle) = \theta \circ [\tau/\alpha](\langle\!\langle \sigma_1 \rangle\!\rangle) = \theta(\langle\!\langle \sigma_2 \rangle\!\rangle)$. This implies the expected result by reflexivity of $\leq_{\mathsf{ML}}$. The ends the proof for the case of equivalence.

The proof for the general case is by induction on the derivation of $(Q)\ \sigma_1 \sqsubseteq \sigma_2$. Transitivity is by induction hypothesis and transitivity of the $\leq_{\mathsf{ML}}$. Rule I-Rigid cannot occur since it mentions a rigid binding, whereas the derivation is assumed to be flexible.

- Case I-Bot: $\forall\,(\alpha)\ \alpha \leq_{\mathsf{ML}} \sigma$ holds for any ML type σ.
- Case I-Hyp: We have $(\alpha \geq \sigma_1) \in Q$ and σ_2 is α. Let $\forall\,(\bar{\beta})\ \tau_1$ be $\langle\!\langle \sigma_1 \rangle\!\rangle$ and $(\bar{\alpha}, \theta)$ be $\langle\!\langle Q \rangle\!\rangle$. By definition of $\langle\!\langle Q \rangle\!\rangle$, $\theta(\alpha) = \theta(\tau_1)$ (**1**). Besides, $\theta(\forall\,(\bar{\beta})\ \tau_1)$ is $\forall\,(\bar{\beta})\ \theta(\tau_1)$ (**2**), and $\forall\,(\bar{\beta})\ \theta(\tau_1) \leq_{\mathsf{ML}} \theta(\tau_1)$ (**3**) holds. By combining (2), (3), and (1), we have $\theta(\forall\,(\bar{\beta})\ \tau_1) \leq_{\mathsf{ML}} \theta(\alpha)$, as expected.
- Case I-Abstract: The premise is $(Q)\ \sigma_1 \sqsubseteq \sigma_2$. Necessarily, we have $(Q)\ \sigma_1 \equiv \sigma_2$ (Rule A-Equiv), because other possible rules for abstraction mention rigid bindings. Then, we conclude using the preliminary result.
- Case Flexible Congruence: We recall the congruence rule:

$$\frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2\ (\mathbf{5}) \qquad (Q, \alpha \geq \sigma_2)\ \sigma_1' \sqsubseteq \sigma_2'\ (\mathbf{4})}{(Q)\ \forall\,(\alpha \geq \sigma_1)\ \sigma_1' \sqsubseteq \forall\,(\alpha \geq \sigma_2)\ \sigma_2'}$$

Keeping the notations of this rule, we have to show $\theta(\langle\!\langle \forall\,(\alpha \geq \sigma_1)\ \sigma_1' \rangle\!\rangle) \leq_{\mathsf{ML}} \theta(\langle\!\langle \forall\,(\alpha \geq \sigma_2)\ \sigma_2' \rangle\!\rangle)$ (**6**). Let $\forall\,(\bar{\alpha_1})\ \tau_1$ be $\langle\!\langle \sigma_1 \rangle\!\rangle$ and $\forall\,(\bar{\alpha_2})\ \tau_2$ be $\langle\!\langle \sigma_2 \rangle\!\rangle$. By induction hypothesis and (5), we have $\theta(\forall\,(\bar{\alpha_1})\ \tau_1) \leq_{\mathsf{ML}} \theta(\forall\,(\bar{\alpha_2})\ \tau_2)$ (**7**) Using the definition of $\langle\!\langle \cdot \rangle\!\rangle$, we may rewrite (6) as follows:

$$\theta(\forall\,(\bar{\alpha_1})\ \langle\!\langle \sigma_1' \rangle\!\rangle[\tau_1/\alpha]) \leq_{\mathsf{ML}} \theta(\forall\,(\bar{\alpha_2})\ \langle\!\langle \sigma_2' \rangle\!\rangle[\tau_2/\alpha])$$

We show this result in two steps, as follows:

$$\theta(\forall\,(\bar{\alpha_1})\ \langle\!\langle \sigma_1' \rangle\!\rangle[\tau_1/\alpha]) \leq_{\mathsf{ML}} \theta(\forall\,(\bar{\alpha_2})\ \langle\!\langle \sigma_1' \rangle\!\rangle[\tau_2/\alpha])\ (\mathbf{8}) \qquad \theta(\forall\,(\bar{\alpha_2})\ \langle\!\langle \sigma_1' \rangle\!\rangle[\tau_2/\alpha]) \leq_{\mathsf{ML}} \theta(\forall\,(\bar{\alpha_2})\ \langle\!\langle \sigma_2' \rangle\!\rangle[\tau_2/\alpha])\ (\mathbf{9})$$

The first step (8) is a consequence of Lemma 4.4.4 (page 38) and (7). The second step (9) is by induction hypothesis applied to (4). ■

Proof of Lemma 4.4.7

By induction on the derivation. Case Var is immediate. Cases Fun, App, and Let are by induction hypothesis. Case Inst is a direct consequence of Property 4.4.6.ii (page 38). Case Gen: The premise is $(Q, \alpha \geq \sigma_1)\ \Gamma \vdash a : \sigma_2$. Let $\forall\,(\bar{\beta})\ \tau_1$ be $\langle\!\langle \sigma_1 \rangle\!\rangle$ (**1**), and θ_1 be $[\tau_1/\alpha]$ (**2**). We choose $\bar{\beta}$ such that $\bar{\beta} \# \mathsf{ftv}(\Gamma)$ (**3**). By definition, we have $\langle\!\langle (Q, \alpha \geq \sigma_1) \rangle\!\rangle = (\bar{\alpha}\bar{\beta}, \theta \circ \theta_1)$. By induction hypothesis, we have $\theta \circ \theta_1(\langle\!\langle \Gamma \rangle\!\rangle) \vdash a : \theta \circ \theta_1(\langle\!\langle \sigma_2 \rangle\!\rangle)$ in ML. Since $\alpha \notin \mathsf{ftv}(\Gamma)$, we have $\theta_1(\langle\!\langle \Gamma \rangle\!\rangle) = \langle\!\langle \Gamma \rangle\!\rangle$. Hence, $\theta(\langle\!\langle \Gamma \rangle\!\rangle) \vdash a : \theta \circ \theta_1(\langle\!\langle \sigma_2 \rangle\!\rangle)$ holds. From (3), we get by Rule Gen of ML, $\theta(\langle\!\langle \Gamma \rangle\!\rangle) \vdash a : \forall\,(\bar{\beta})\ \theta \circ \theta_1(\langle\!\langle \sigma_2 \rangle\!\rangle)$. Notice that $\forall\,(\bar{\beta})\ \theta \circ \theta_1(\langle\!\langle \sigma_2 \rangle\!\rangle)$ is equal to $\theta(\forall\,(\bar{\beta})\ \theta_1(\langle\!\langle \sigma_2 \rangle\!\rangle))$, which, by definition of $\langle\!\langle \cdot \rangle\!\rangle$, (1), and (2) is also $\theta(\langle\!\langle \forall\,(\alpha \geq \sigma_1)\ \sigma_2 \rangle\!\rangle)$. We thus have $\theta(\langle\!\langle \Gamma \rangle\!\rangle) \vdash a : \theta(\langle\!\langle \forall\,(\alpha \geq \sigma_1)\ \sigma_2 \rangle\!\rangle)$, as expected. ■

Proof of Theorem 9

Direct consequence of Lemmas 4.4.7 and 4.4.3. ■

References

[Bar84] Henk P. Barendregt. *The Lambda Calculus*. North Holland, revised edition, 1984.

[Boe85] H.-J. Boehm. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*, pages 339–345. IEEE Computer Society Press, October 1985.

[Car93] Luca Cardelli. An implementation of FSub. Research Report 97, Digital Equipment Corporation Systems Research Center, 1993.

[CPWK04] Sébastien Carlier, Jeff Polakow, J.B. Wells, and A.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.

[DHKP96] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Higher-order unification via explicit substitutions: the case of higher-order patterns. In M. Maher, editor, *Joint international conference and symposium on logic programming*, pages 259–273, 1996.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arith-mé-tique d'ordre supérieur*. Thèse d'état, University of Paris VII, 1972.

[GR88] P. Giannini and S. Ronchi Della Rocca. Characterization of typings in polymorphic type discipline. In *Third annual Symposium on Logic in Computer Science*, pages 61–70. IEEE, 1988.

[GR99] Jacques Garrigue and Didier Rémy. Extending ML with Semi-Explicit higher-order polymorphism. *Journal of Functional Programming, vol 155*, pages pages 134–169, 1999. A preliminary version appeared in TACS'97.

[HP99a] Haruo Hosoya and Benjamin C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.

[HP99b] Haruo Hosoya and Benjamin C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.

[Jim95] 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.

[Jim00] T. Jim. A polar type system. *ICALP Workshops*, 8:323–338, 2000.

[JS04] S. Peyton Jones and M. Shields. Practical type inference for arbitrary-rank types, 2004.

[JVWS07] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, 2007.

[JWOG89] Jr. James William O'Toole and David K. Gifford. Type reconstruction with first-class polymorphic values. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989. ACM. also in ACM SIGPLAN Notices 24(7), July 1989.

[KW94] 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.

[LB04] Didier Le Botlan. *MLF : Une extension de ML avec polymorphisme de second ordre et instanciation implicite.* PhD thesis, Ecole Polytechnique, May 2004. 326 pages.

[LBR03] Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of System-F. In *Proceedings of the International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 27–38. ACM Press, August 2003.

[Lei90] Daniel Leivant. Discrete polymorphism (summary). In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 288–297, 1990.

[LL05] Daan Leijen and Andres Löh. Qualified types for mlf. In *The International Conference on Functional Programming (ICFP'05)*. ACM Press, September 2005.

[LO94] 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.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.

[Mil92] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[Mit88] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 2/3(76):211–249, 1988.

[OL96] 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.

[OZZ01] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. *ACM SIG-PLAN Notices*, 36(3):41–53, March 2001.

[Pfe88] 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.

[Pfe93] Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1,2):185–199, 1993. Preliminary version available as Technical Report CMU-CS-92-105, School of Computer Science, Carnegie Mellon University, January 1992.

[Pie91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

[Pie94] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Massachusetts Institute of Technology Cambridge, Massachusetts 02142, 2002.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of the 25th ACM Conference on Principles of Programming Languages*, 1998. Full version in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1), January 2000, pp. 1–44.

[Rém94] 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.

[Rém05] Didier Rémy. Simple, partial type-inference for System F based on type-containment. In *Proceedings of the tenth International Conference on Functional Programming*, September 2005.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[RY07] Didier Rémy and Boris Yakobowski. A graphical presentation of MLF types with a linear-time unification algorithm. In *TLDI'07: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 27–38, Nice, France, January 2007. ACM Press.

[Sch98] Aleksy Schubert. Second-order unification and type inference for Church-style polymorphism. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 279–288, New York, NY, 1998.

[VWJ06] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: inference for higher-rank types and impredicativity. *SIGPLAN Not.*, 41(9):251–262, 2006. Proceedings of the 2006 International Conference on Functional Programming.

[Wel96] Joe B. Wells. *Type Inference for System F with and without the Eta Rule*. PhD thesis, Boston University, 1996.

[Wel99] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1–3):111–156, 1999.



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

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

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