Information and Computation 207 (2009) 726–785

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/ic

Recasting ML^F

Didier Le Botlan[a,b], Didier Rémy[c,*]

[a] *CNRS, LAAS, 7 Avenue du colonel Roche, F-31077 Toulouse, France*
[b] *Université de Toulouse, UPS, INSA, INP, ISAE, LAAS, F-31077, France*
[c] *INRIA Paris – Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France*

ARTICLE INFO

Article history:
Received 11 July 2007
Accepted 28 May 2008
Available online 14 January 2009

Keywords:
ML
System F
Type inference
Type checking
Polymorphism
First-class polymorphism

ABSTRACT

The language ML^F is a proposal for a new type system that supersedes both ML and System F, allows for efficient, predictable, and complete type inference for partially annotated terms. In this work, we revisit the definition of ML^F, following a more progressive approach and focusing on the design-space and expressiveness. We introduce a Curry-style version $i\mathsf{ML}^\mathsf{F}$ of ML^F and provide an interpretation of $i\mathsf{ML}^\mathsf{F}$ types as instantiation-closed sets of System-F types, from which we derive the definition of type-instance in $i\mathsf{ML}^\mathsf{F}$. We give equivalent syntactic definition of the type-instance, presented as a set of inference rules. We also show an encoding of $i\mathsf{ML}^\mathsf{F}$ into the closure of Curry-style System F by let-expansion. We derive the Church-style version $e\mathsf{ML}^\mathsf{F}$ by refining types of $i\mathsf{ML}^\mathsf{F}$ so as to distinguish between given and inferred polymorphism. We show an embedding of ML in $e\mathsf{ML}^\mathsf{F}$ and a straightforward encoding of System F into $e\mathsf{ML}^\mathsf{F}$.

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 increase readability, to ease code reuse and to be robust to small code changes, and *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 functions have a unique ground type. However, higher-order functions, which receive among their arguments other functions 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 [25]: quantifiers

* Corresponding author.
E-mail addresses: didier.le-botlan@insa-toulouse.fr (D. Le Botlan), Didier.Remy@inria.fr (D. Rémy), *URLs*: http://wwwdgeinew.insa-toulouse.fr/∼lebotlan/ (D. Le Botlan), http://gallium.inria.fr/∼remy/ (D. Rémy).

0890-5401/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2008.12.006

are limited to the outermost position of types and the use of polymorphism is limited to definitions, as opposed to parameters of functions. For example, 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 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., a type whose instances are exactly all types of the program. 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 C# and the language Java, version 5.

However, polymorphic higher-order functions sooner or later need themselves to be passed as arguments to other functions.

Because of its limited form of polymorphism, ML is well suited for programming abstractions over known data-structures, such as the fold iterator over lists, but it does not allow to abstract iterators over arbitrary collections because it is not possible to write a function that visits a structure without knowing its constructors exactly. Another related limitation is that the Church encoding of data-structures does not usually work in ML. A crux of Haskell is the use of monads, but the lack of first-class polymorphism prevents the abstraction over monads. This has motivated the introduction of arbitrary-rank polymorphism in Haskell [13], but its limitation to predicative polymorphism is too restrictive and makes it inconvenient to use. While row variable polymorphism allows for a rather expressive object-oriented programming style in OCaml, the lack of polymorphic methods that sooner or later arise has motivated the introduction of a limited form of first-class polymorphism, based on Poly-ML [6]. Unfortunately, it remains contrived and *ad hoc*.

Many situations where first-class polymorphism is useful have been described in the literature. For instance, a series of examples in the context of Haskell are presented in [30]. Many of these particular cases are in fact similar and often involve in a way or another taking as argument a function that is applied to heterogeneous data, hence that has to be polymorphic. One may still wonder how often such situations arise. We believe that they are in fact unavoidable, for reasons explained below.

Perhaps, the most convincing and agreed limitations of ML is its inability to represent heterogeneous collections—except via explicit tagging which then prevents their treatment in a uniform way. Indeed, the other way to represent an heterogeneous collection is to give its elements an existential type of the form $\exists\alpha.\tau$, where in fact each element taken separately has a type of the form $\tau[\tau'/\alpha]$ (the type τ in which α has been replaced by τ') for some type τ'. Existential types are also needed to represent closures, i.e., delayed computations that, for instance, are to be run on some remote computation server.

It is well-known that an expression a of existential type $\exists\alpha.\tau$ may be encoded as the function $\lambda(f)\ f\ a$ taking as argument an arbitrary polymorphic function f of type $\forall(\beta)\ (\forall(\alpha)\ \tau \to \beta) \to \beta$, so that it can be applied to a but ignoring the part of a that is to be hidden. We write $\forall(\alpha)\ \tau$ for $\forall$-quantification of variable α in type τ. The scope of α extends to the right as far as possible. The arrow $\to$ is right associative and of lower priority than other type constructors such as the product $\times$. That is, $\forall(\alpha)\ \tau \times \tau' \to \tau \to \tau'$ means $\forall(\alpha)\ ((\tau \times \tau') \to (\tau \to \tau'))$.

Quite interestingly, providing existential types as primitive does not alleviate but, on the contrary, reinforces the need for passing polymorphic functions as arguments. Indeed, an expression a_1 of existential type must be opened before it can be used inside an expression a_2, which is usually written $\mathrm{open}\ a_1\ \mathrm{as}\ x, \beta\ \mathrm{in}\ a_2$. The type variable β stands for the witness of the existential type of a_1 and its scope is the expression a_2. When the body of a_2 is parametrized over some function f received as argument, the program becomes of the form $\lambda(f)\ \mathrm{open}\ a_1\ \mathrm{as}\ x, \beta\ \mathrm{in}\ a_2$ where f is used in a_2, typically with a type that involves β. In order for β not to escape from its scope, the function f must be polymorphic in some other variable that will be instantiated to some type containing β.

Such examples require quantifiers to appear within types—here on the left-hand side of arrow types, but more generally at any position within types. That is, polymorphic types should be treated as any other types, which is called *first-class polymorphism*. System F, the second-order polymorphic λ-calculus, is the reference for first-class polymorphism [8,45].

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 [50]. Moreover, System F does not have principal types.[1] That is, the types of an expression in a given typing context cannot be captured as the set of all instances of a particular (principal) type. Of course, the notion of principal types depends on a suitable definition of type-instance.

Therefore, in practice, one rather uses Church's view, where source programs contain explicit type information, so as to make type checking straightforward and decidable. 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 a list obtained by applying the first coordinate of each pair to the second coordinate.

We write $\forall(\alpha\alpha')\ \sigma$ for $\forall(\alpha)\ \forall(\alpha')\ \sigma$. We write $\lambda(x)\ a$ for the function that maps x to the expression a. We write $\Lambda\,\alpha.\ a$ and $a[\tau]$ for type abstraction and type application, respectively.

[1] Principal types should not be confused with principal typings [49].

Assume we are given a function listmap of type $\forall(\alpha\beta)\ (\alpha \to \beta) \to \mathit{list}(\alpha) \to \mathit{list}(\beta)$, as well as the projections fst and snd from pairs of respective types $\forall(\alpha\beta)\ \alpha \times \beta \to \alpha$ and $\forall(\alpha\beta)\ \alpha \times \beta \to \beta$.

```
Λα . Λβ . 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. That is, 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 $(\forall(\alpha)\ \alpha \times \alpha \to \mathit{list}(\alpha)) \times (\forall(\beta)\ \tau \times \tau)$, 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.

1.1. 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 [31]. 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 [36,28], 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 [9]. 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* [41,27] 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.

1.2. Our main contribution

We propose a new type system, called ML^F, that supersedes both ML and System F, allows for simple, efficient, predictable, and complete type inference for partially annotated terms. The language ML^F has been introduced in previous works [17,16], which we shall below refer to as Full ML^F to avoid ambiguity. In this work, we focus on a simplified version, here called ML^F for conciseness—or Shallow ML^F when there is ambiguity. While Shallow ML^F is less expressive than Full ML^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-style version of ML^F, called[2] $i\mathsf{ML}^\mathsf{F}$. 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 $i\mathsf{ML}^\mathsf{F}$ 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^F is theoretically small, although practically important as it increases modularity

[2] The *i* stands for *implicit*.

in an essential way. For the sake of comparison, one may also consider a weaker version, called[3] Simple ML^{F}, that has exactly the same expressiveness as System F. Although it retains most theoretical properties of ML^{F}, it is no longer an extension of ML and, as a result, has little practical interest.

Unsurprisingly, full type inference for *i*ML^{F} is undecidable. We thus also devise a Church-style version of ML^{F}, called[4] *e*ML^{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 type schemes. More importantly, it keeps track of sharing by distinguishing between $\forall(\alpha \Rightarrow \sigma, \alpha' \Rightarrow \sigma)\ \sigma'$ and $\forall(\alpha \Rightarrow \sigma)\ \sigma'[\alpha/\alpha']$, which is essential for type inference, as explained next.

Indeed, the ML^{F} type system maintains a clear separation between polymorphism that can be inferred and polymorphism that cannot (more precisely, that we choose not to infer).

Usual ML-style polymorphism belongs to the former category. It is represented by type schemes: the typing of the body of $\mathsf{let}\ x = a\ \mathsf{in}\ a'$ is of the form $x : \sigma \vdash a' : \sigma'$ for some type σ' to be inferred. That is, x is known to have a polymorphic type σ while typing a'. For instance, if a is $\lambda(y)\ y$, which we write id, then x is known to have type $\forall(\alpha)\ \alpha \to \alpha$, which we write σ_{id}.

On the contrary, the type of x in $\lambda(x)\ a$ belongs to the latter, and is represented by an abstract type while typing a. The typing of the body is of the form $(\alpha \Rightarrow \sigma)\ x : \alpha \vdash a : \sigma'$ (**1**). That is, x is only known to have an abstract type α, which is bound to σ in the prefix $(\alpha \Rightarrow \sigma)$, but which cannot be instantiated to σ or to anything else.[5] The binding $(\alpha \Rightarrow \sigma)$ can be discharged on the right of the typing judgement only when α does not appear in the typing environment, as prescribed by the usual generalisation rule. As an example, if a is x, then (1) becomes $(\alpha \Rightarrow \sigma)\ x : \alpha \vdash x : \alpha$. We may derive $(\alpha \Rightarrow \sigma)\ \vdash \lambda(x)\ x : \alpha \to \alpha$ and the binding $(\alpha \Rightarrow \sigma)$ may now be discharged into $\vdash \lambda(x)\ x : \forall(\alpha \Rightarrow \sigma)\ \alpha \to \alpha$. As a consequence $(\lambda(x)\ x)\ \omega^{\dagger}$ is typable without any type annotation on x, where the identifier $\omega^{\dagger}$ stands for the function $\lambda(y : \sigma_{\mathsf{id}})\ y\ y$. However, $\lambda(x)\ x$ is not typable with type $\forall(\alpha \Rightarrow \sigma, \alpha' \Rightarrow \sigma)\ \alpha \to \alpha'$, as this would have required typing $(\alpha \Rightarrow \sigma, \alpha' \Rightarrow \sigma)\ x : \alpha \vdash x : \alpha'$ (**2**), which does not hold.

Conversely, the type of an annotated argument belongs to the former category. The typing of a in $\lambda(x : \sigma)\ a$ involves a judgment of the form $x : \sigma \vdash a : \sigma'$. The annotation is *intuitively* discharged as $\vdash \lambda(x : \sigma)\ a : \sigma \to \sigma'$. However, this is not quite correct because type schemes may not appear under arrow types: instead, we write $\vdash \lambda(x : \sigma)\ a : \forall(\alpha \Rightarrow \sigma, \alpha' \Rightarrow \sigma')\ \alpha \to \alpha'$. For example, we have $x : \sigma \vdash x : \sigma$, and therefore, $\vdash \lambda(x : \sigma)\ x : \forall(\alpha \Rightarrow \sigma)\ \forall(\alpha' \Rightarrow \sigma)\ \alpha \to \alpha'$. Here, the type α' of the result, and the type α of the argument are decoupled, which is possible since the known type of x is σ and not a type variable abstracting σ.

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

The richer types and the let-polymorphism of *e*ML^{F} make it significantly superior to Church-style 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 *e*ML^{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 to be annotated. In particular, ML programs need no type annotations.

Although type inference is not addressed in this paper, it can be reduced to a first-order unification algorithm for (a form of) second-order types, combined with let-polymorphism *a la* ML, as shown in previous works by Le Boltan and Rémy [17,16]. While worst-case complexity is at least as hard as in ML, i.e., exponential-time complete, it seems to be quite reasonable in practice. In fact, type inference for ML^{F} has recently been shown to be as efficient as for ML [43,23].

Notice that both *i*ML^{F} and *e*ML^{F} are typed languages, of equivalent expressiveness, and only differ by whether (some) types are explicitly written in terms. We use the name *i*ML^{F} and *e*ML^{F} when this point of view matters. For instance, only *e*ML^{F} makes sense as a programming language. Otherwise, ML^{F} refers to either view, indifferently.

1.3. The full picture

Of course, something must have been lost while going from Full ML^{F} to Shallow ML^{F}. That is, it may be the case in Shallow ML^{F} that a local binding $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ is typable while its operationally equivalent form $(\lambda(x)\ a_2)\ a_1$ is not. Technically, Shallow ML^{F} polymorphism is second-order, yet not first-class.

To see the practical consequences of this limitation, we shall proceed by comparison with ML. We use unannotated terms by default, so as to ease the comparison between the different languages. Consider the following expression in ML:

```
let f = λ(x) x in ( f 42, f "foo")
```

[3] Simple ML^{F} was called Restricted ML^{F} in [17,16].

[4] The *e* stands for *explicit*.

[5] In the case where σ is a monotype τ, as in ML, α can be equivalently replaced by τ in $x : \alpha$ (indeed, monotypes can be inferred).

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

```
(λ (f) ( f 42, f "foo")) ( λ(x) 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 expression is typable in *i*ML$^{\mathsf{F}}$, which features second-order polymorphism. (In *e*ML$^{\mathsf{F}}$, one needs only add an explicit type-annotation, e.g., σ_{id}, on the parameter f.) However, this is only a simplistic example. For instance, consider a small variant of the previous expression:

```
let f = choose id in (f succ, choose f ω)
```

The identifier succ stands for the successor function of type $int \to int$. The identifier choose stands for a function that takes two arguments and returns either one, which could be defined as λ (x) λ (y) if true then x else y, of (principal) polymorphic type $\forall(\alpha)\ \alpha \to \alpha \to \alpha$. The identifier ω corresponds to the unannotated version of $\omega^{\dagger}$, that is, $\lambda(x)\ x\ x$. The expression above is not typable in System F. A formal proof of this fact is beyond the scope of this paper—see [33, Chapter 23] for hints on how to conduct such proofs. Interestingly, this expression is typable in *e*ML$^{\mathsf{F}}$, taking the annotated version $\omega^{\dagger}$ instead of ω. Indeed, 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 $(int \to int) \to (int \to int)$ and $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$, which are incompatible in System F.

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

```
(λ (f)(f succ, choose f ω)) (choose id)
```

which is not typable in *i*ML$^{\mathsf{F}}$. The problem is that the required flexible type $\forall(\alpha \geq \sigma_{\mathsf{id}})\ \alpha \to \alpha$ that could previously be assigned to the let-bound variable f, cannot be assigned to a λ-bound variable, as only System-F types are allowed for λ-bound variables in Shallow ML$^{\mathsf{F}}$.

This restriction is relaxed in Full ML$^{\mathsf{F}}$, which allows flexible quantification at arbitrary positions in types. Hence, the previous example is typable in Full ML$^{\mathsf{F}}$. (In Full *e*ML$^{\mathsf{F}}$, it suffices to add the type annotation $\forall(\alpha \geq \sigma_{\mathsf{id}})\ \alpha \to \alpha$ to the parameter f.)

The main outcomes of staying within Shallow ML$^{\mathsf{F}}$ 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 convey strong intuitions but also to recover the syntactic instance relation as set containment on semantic types.

The meta-theoretical study of Shallow ML$^{\mathsf{F}}$ presented hereafter builds on this semantic support and is thus significantly simpler than—and mostly independent from—the previous study of the full version [17,16].

The remaining gap between Shallow ML$^{\mathsf{F}}$ and Full ML$^{\mathsf{F}}$ is a small step for the user but another big step for a 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 shallow version. Hopefully, the intuitions built for the shallow 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 ML$^{\mathsf{F}}$ are summarized and put in close correspondence with existing languages in the Fig. 1 and Fig. 2.

All languages are presented in order of increasing expressiveness in the Fig. 1. The first group, on the top of the Fig. 1 and also appearing on the left-hand side of the Fig. 2, is composed of three well-known languages where polymorphism is based on the standard ∀-quantification. The second group, on bottom of the Fig. 1 and also appearing on the right-hand side on the Fig. 2, is composed of three variants of ML$^{\mathsf{F}}$ where polymorphism is based on the peculiar ≥-bounded quantification.

The correspondence between these two groups is explicitly represented on the Fig. 2 by the horizontal dotted arrows and on the Fig. 1 by the equality of the two submatrices left in white background on the diagonal (and the two constant submatrices in gray background). In this correspondence, the standard ∀-quantification in the first-group becomes the ≥-bounded quantification in the second group. In each group, expressiveness is successively gained by added quantification to the type of let-bound parameters and, then, to that of λ-bound parameters.

As mentioned earlier, Simple ML$^{\mathsf{F}}$ is exactly as expressive as System F. This is represented by the identity of the two lines at the junction of the two groups in the Fig. 1 and the double-line arrow relating them in the Fig. 2. The reason for this coincidence is that Simple ML$^{\mathsf{F}}$ is obtained by adding flexible quantification to System F, yet without any construction to exploit it. In this respect, Simple ML$^{\mathsf{F}}$ is to Shallow ML$^{\mathsf{F}}$ what simply-typed λ-calculus is to ML: simply-typed λ-calculus has universal type variables but no type-instantiation mechanism! Restated in the other direction, Shallow ML$^{\mathsf{F}}$ is to Simple ML$^{\mathsf{F}}$ 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$^{\mathsf{F}}$ is to Shallow ML$^{\mathsf{F}}$ what System F is to ML—it enables flexible polymorphism on function parameters and, more generally, to appear at arbitrary position in types: quantification is first-class—but of different power—in both Full ML$^{\mathsf{F}}$ and System F. By contrast, it can only be used at local bindings in both Shallow ML$^{\mathsf{F}}$ and ML.

The Systems $\mathsf{F}_{\wedge}$ (right-hand side of Fig. 2) is an extension of System F with intersection types [35], while $\mathsf{F}^{\mathsf{let}}$ below is the restriction of $\mathsf{F}_{\wedge}$ to rank-1 intersection types (see Section 3.6.1). 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 Shallow ML$^{\mathsf{F}}$ into $\mathsf{F}^{\mathsf{let}}$, which is proved below (Section 3.6), implies the correctness of Shallow ML$^{\mathsf{F}}$. The inclusion between Full ML$^{\mathsf{F}}$ and $\mathsf{F}_{\wedge}$ also holds but is not shown in this paper.

Variant	let-∀	λ-∀	let-≥	λ-≥
Simple Types	–	–	–	–
ML	√	–	–	–
System F	√	√	–	–
Simple ML$^{\mathsf{F}}$	√	√	–	–
Shallow ML$^{\mathsf{F}}$	√	√	√	–
Full ML$^{\mathsf{F}}$	√	√	√	√

Fig. 1. Expressiveness of variants of ML$^{\mathsf{F}}$.

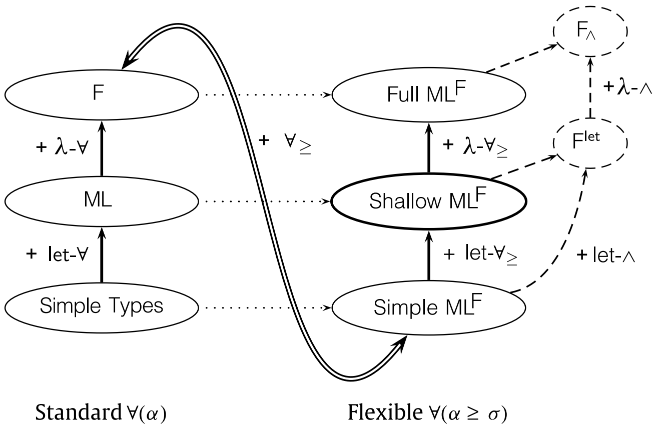


Fig. 2. The small hierarchy of ML$^{\mathsf{F}}$ variants.

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

We believe that a programming language should be based on the full rather than the shallow version of ML$^{\mathsf{F}}$. However, other extensions such as higher-order types may be easier to explore in the simple version. Hence, the shallow 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.

1.4. Type inference

We do not address type inference here, as it is a technically orthogonal issue and is not significantly easier for Shallow ML$^{\mathsf{F}}$ than for Full ML$^{\mathsf{F}}$. The reader is referred to [17,16] or independent study in subsequent work [42]. As ML$^{\mathsf{F}}$ was designed with first-order type inference and let-polymorphism in mind, polymorphism never needs to be guessed: it is only picked at local bindings or user-provided type annotations, and is propagated to use sites 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-style version *e*ML$^{\mathsf{F}}$.

1.5. Outline of the paper

While previous studies focused on Full *e*ML$^{\mathsf{F}}$, this work is limited to the study of Shallow 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 Section 2. The Curry's view is explored in details, including discussions of type soundness and of expressiveness in Section 3. The Church's view is studied formally in Section 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.

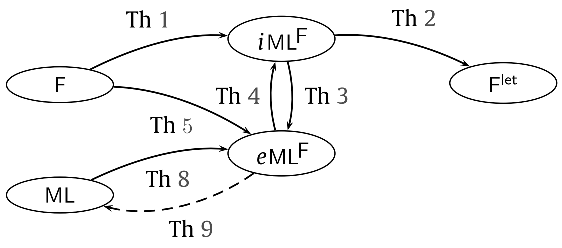


Fig. 3. Encodings.

We also address expressiveness of the Church's view by showing that it subsumes both ML and System F. Related works are discussed in Section 5 before concluding remarks. For clarity of exposition, all proofs have been moved to appendices.

Fig. 3 summarizes the seven encodings presented in Sections 3 and 4. Encoding are represented by edges and labeled with the corresponding theorem. (The interest of the direct encoding of System F to eML$^{\mathsf{F}}$ is to introduce fewer annotations than the composition of encodings via iML$^{\mathsf{F}}$, and in particular, fewer type annotations than present in the original term.) The encoding from eML$^{\mathsf{F}}$ to ML is partial, defined on the subset of eML$^{\mathsf{F}}$ programs that do not contain any annotations.

1.6. 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 $e_1, \ldots, e_n$. 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 (**3**) as a binding annotation in text or formulas and normal font as (3) 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$^{\mathsf{F}}$

This section is primarily an informal introduction to ML$^{\mathsf{F}}$. The only prerequisite is a good knowledge of ML and some knowledge of System F. We first remind Curry-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-style and Church-style versions of ML$^{\mathsf{F}}$ as instances of G. We provide intuitions on the flexible and rigid quantification that are at the heart of ML$^{\mathsf{F}}$, by means of examples. We also discuss—still informally—some of the advantages of ML$^{\mathsf{F}}$ compared to System F, besides type inference. This section may also be read back after some technical knowledge of ML$^{\mathsf{F}}$ has been acquired to deepen one's understanding.

2.1. A generic Curry-style second-order type system

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

$$\begin{array}{ll} a ::= x \mid \lambda(x)\ a \mid a_1\ a_2 & \text{Terms} \\ \quad \mid \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 & \text{Local bindings (optional)} \end{array}$$

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.

2.1.1. 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

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

Types should at least contain type variables α and arrow types $\tau \to \tau$, but may also contain other forms represented by the ellipsis. Of course, types could also contain other type constructors such as products, sums, etc., without any technical complication. For conciseness, we do not include these in the formal presentation.

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 the 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:

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

Indeed, this definition must be completed to cover the possible extra forms in the definition of $\mathcal{T}$. The extra cases are usually obvious. 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:

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

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:

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

Quantifiers may appear in types directly and thus at arbitrary positions in F-types—as expected. In this case, types are closed by outermost quantification and coincide with type schemes. We use letter t instead of letter τ to range over F-types. This will be useful below in $i\mathsf{ML}^{\mathsf{F}}$ to distinguish F-types from other forms of τ-types.

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

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

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

2.1.2. Prefixes

As type variables come with bounds, most operations on types will be defined under a type environment, called a prefix, that assigns bounds to their free type variables. A *prefix* Q is a sequence of bindings $q_1, \ldots, q_n$. 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). Besides, bounds may themselves have free type variables written $\mathsf{ftv}(Q)$ that are 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 $q_1, \ldots, q_n$, $\mathsf{dom}(q_i)$ may only intersect $\mathsf{ftv}(q_j)$ for $j > i$. A prefix $q_1, \ldots, q_n$ is often used to build types of the form $\forall(q_1), \ldots, \forall(q_n)\ \tau$, which we simply write $\forall(q_1, \ldots, q_n)\ \tau$. Prefixes may also be used as type environments in judgments under prefix, e.g., as type-instance defined next.

We assume that the following well-formedness condition holds: for each prefix Q of the form $q_i^{i \in 1,\ldots,n}$ and for each i and j in $1, \ldots, n$, (1) $i \neq j$ implies $\mathsf{dom}(q_i) \neq \mathsf{dom}(q_j)$, and (2) $i \geq j$ implies $\mathsf{dom}(q_i) \notin \mathsf{ftv}(q_j)$.

2.1.3. 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-style type systems.

The instance relation for Curry-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 $\bar\tau$.

$$\text{VAR}\quad \frac{x : \sigma \in \Gamma}{(Q)\ \Gamma \vdash x : \sigma} \qquad \text{FUN}\quad \frac{(Q)\ \Gamma, x : \tau \vdash a : \tau'}{(Q)\ \Gamma \vdash \lambda(x)\ a : \tau \to \tau'}$$

$$\text{APP}\quad \frac{(Q)\ \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad (Q)\ \Gamma \vdash a_2 : \tau_2}{(Q)\Gamma \vdash a_1\ a_2 : \tau_1} \qquad \text{INST}\quad \frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \leqslant \sigma'}{(Q)\Gamma \vdash a : \sigma'}$$

$$\text{GEN}\quad \frac{(Q, q)\ \Gamma \vdash a : \sigma \qquad \mathsf{dom}(q) \notin \mathsf{ftv}\,(\Gamma)}{(Q)\ \Gamma \vdash a : \forall(q)\ \sigma} \qquad \text{LET}\quad \frac{(Q)\Gamma \vdash a : \sigma \qquad (Q)\,\Gamma, x : \sigma \vdash a' : \sigma'}{(Q)\ \Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \sigma'}$$

Fig. 4. Typing rules for $\mathsf{G}(\mathcal{T}, \mathcal{S}, \mathcal{Q}, \leqslant)$.

The definition of the type-instance relation $\leq_{\mathsf{ML}}$ for ML is exactly the same except that it applies to weaker sets of types and type schemes. Alternatively, the relation $\leq_{\mathsf{ML}}$ is also 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 prefixes Q ranging in the set of all prefixes. 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 inverse of a prefixed relation $\mathcal{R}$ is the relation $\mathcal{R}^{-1}$ defined by taking the inverse of $(\mathcal{R}_Q)$ 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 80s [26]. 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 contravariantly on the left-hand side of arrow types (and covariantly everywhere else) and distributes $\forall$-quantifiers over arrows (see [26]). 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.

Both $\leq_\eta$ and $<:$ treat the arrow type asymmetrically, covariantly on the right-hand side and contra-variantly on the left-hand side. When generalizing the language of types to arbitrary types constructors, they would require type constructors to be declared with a signature defining their variance in each of their argument. By contrast, the type-instance relations of ML, System F, or ML^{F} are symmetric and need not know about variance of type-constructors.

2.1.4. Type equivalence

We assume that any suitable instance relation $\leqslant$ is a preorder. Then, an instance relation induces an equivalence under prefix, defined as the kernel of $\leqslant$, i.e., $\leqslant \cap \geqslant$.

In the only case of F-types where the equivalence does not depend on prefixes, we actually consider 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.

2.1.5. The generic Curry-style second-order type system

The generic Curry-style second-order *type system*, written $\mathsf{G}(\mathcal{T}, \mathcal{S}, \mathcal{Q}, \leqslant)$, is parametrized 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 with finite domains from program variables to type schemes. The free type-variables of a typing context are the union of the free type variables of its codomain. We write $\emptyset$ the empty mapping 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 Fig. 4. 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 forall-elimination step. This approach is preferable in a Curry-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. In particular, the bound expression is assigned a type scheme rather than a type. This improves over the default rule that would consist in typechecking let $x = a_1$ 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 the 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.

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.

Hypotheses. In order for the type system to have interesting 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:

Renaming
$$\frac{(Q)\ \sigma_1 \leqslant \sigma_2 \qquad \phi \text{ renaming}}{(\phi(Q))\ \phi(\sigma_1) \leqslant \phi(\sigma_2)}$$

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 rely on both axioms.

2.1.6. Particular instances of System G.

Curry-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 [26]. 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, σ_id, which is an abbreviation for $\forall(\alpha)\ \alpha \to \alpha$, 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 $\mathsf{F}_{<:}$ proposed by Cardelli [3] 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-style presentation while $\mathsf{F}_{<:}$ is usually presented in Church style.

2.2. *i*ML$^\mathsf{F}$: Curry-style ML$^\mathsf{F}$

Returning to our goal, we seek for a language *e*ML$^\mathsf{F}$ with partial type annotations that is at least as expressive as System F (each term of Curry-style System F is the type erasure of some term of *e*ML$^\mathsf{F}$), supersedes ML (all terms of ML are in *e*ML$^\mathsf{F}$, without any annotation), and for which we can perform type inference using a form of first-order unification. Based and improving on previous experiences, we wish to manipulate second-order types transparently, so as to bypass 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 just their juxtaposition, so that the best of each approach also strengthens the other one.

2.2.1. 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 considered 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 $\forall(\alpha)\ \alpha \to \alpha \to \alpha$ (choose 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 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 choose v as long as τ' is an instance of τ. In other words, the correct types for choose v form a set $\{\tau' \to \tau' \mid \tau \leq \tau'\}$.[6] 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 the intuitive and formal perspectives.

One may be tempted to use infinite intersection types $\bigwedge\{\tau' \to \tau' \mid \tau \leq \tau'\}$, as suggested by Leivant in another context [22]. 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.

2.2.2. 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 (Section 3.1). 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 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 $i\mathsf{ML}^\mathsf{F}$.

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 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 instantiate types on the right-hand side of arrow types (or more generally on positive occurrences in types), we do not do so. We may always use a type scheme $\forall(\alpha \geq \tau)\ \tau' \to \alpha$ instead 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* $\perp$ (read *bottom*) to mean *any* type. We may then see $\forall(\alpha)\ \alpha \to \alpha$ as syntactic sugar for $\forall(\alpha \geq \perp)\ \alpha \to \alpha$. Intuitively, $\perp$ could itself be seen as representing the set of all types and, indeed, $\perp$ is equivalent to $\forall(\alpha \geq \perp)\ \alpha$ 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 $\perp$ 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. 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 $i\mathsf{ML}^\mathsf{F}$ (Curry-style ML^F) are defined as follows:

$$
\begin{array}{llr}
\tau \in \mathcal{T}_i ::= \alpha \mid \tau \to \tau \mid \forall(\alpha \geq \perp)\ \tau & & i\mathsf{ML}^\mathsf{F}\text{ types} \\
\sigma \in \mathcal{S}_i ::= \tau \mid \forall(q)\ \sigma \mid \perp & & i\mathsf{ML}^\mathsf{F}\text{ types schemes} \\
q \in \mathcal{Q}_i ::= \alpha \geq \sigma & & i\mathsf{ML}^\mathsf{F}\text{ bindings}
\end{array}
$$

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 let $x = a_1$ 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).

[6] 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.

We thus consider the instance $\mathsf{G}(\mathcal{T}_i, \mathcal{S}_i, \mathcal{Q}_i, \leq)$ of Generic Curry-style System F with flexible quantification and local definitions. This intermediate language is in Curry style and all type information is still left implicit. For this reason, we also call it *i*ML$^\mathsf{F}$ (read *implicit* ML$^\mathsf{F}$). Remarkably, the power of *i*ML$^\mathsf{F}$ 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. *e*ML$^\mathsf{F}$: Church-style ML$^\mathsf{F}$

The language *i*ML$^\mathsf{F}$ is our stepping stone to *e*ML$^\mathsf{F}$. By comparison with System F, more expressions have principal types and many expressions have more general types. However, *i*ML$^\mathsf{F}$ 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 *e*ML$^\mathsf{F}$—a Church-style version of *i*ML$^\mathsf{F}$ that enables type inference[7].

2.3.1. First-order type inference with second-order types

Our goal is to perform type inference based on first-order unification but in a language with second-order types. In addition, we wish to reach all ML programs without type annotations and all 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, 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 self-application ω, defined as $\lambda(x)\ x\ x$ (**1**), unless we explicitly annotate the parameter as in the expression $\lambda(x : \sigma_{\mathsf{id}})\ x\ x$ (which we write $\omega^\dagger$).

Our goal is to design the system so that the second-order polymorphism is never guessed. 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 $(\lambda(z)\ z)\ \omega^\dagger$ and a_2 defined as $(\lambda(x)\ x\ x)\ \mathsf{id}$. 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 $(\lambda(z : \sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}})\ z)\ \omega^\dagger$. 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 $(\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}) \to (\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}})$, which may be more concisely summarized by its fully annotated form $(\Lambda\, \alpha.\ \lambda(z : \alpha)\ z)\ [\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}]\ \omega^\dagger$ in Church-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 obtain the polymorphic type $(\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}) \to (\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}})$ of $\lambda(z)\ z$. Fortunately, we may—and must, as argued in Section 2.2—devise ML$^\mathsf{F}$ 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 ω (1). Indeed, any closed subterm of a well-typed term must also be well-typed. Thus a_2, which contains the ill-typed closed subterm ω 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 $\lambda(z)\ (z\ \omega^\dagger)$, 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 $(\lambda(y)\ \lambda(z)\ z\ y)\ \omega^\dagger$, which we refer to as a_4. Arguing as for a_1, it is clear that 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 when the redexes are let-bindings or applications of unannotated λ-abstractions. As a cross-checking final example, should the expression a_5 defined as $\lambda(z)\ \omega^\dagger\ \ z$ 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 $(\lambda(x)\ \lambda(y)\ x\ y)\ \omega^\dagger$ is typable.

It is actually a remarkable and *essential* property of ML$^\mathsf{F}$ 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, such properties ensure that well-typed programs are stable under some minor but useful program transformations. More type-preserving program transformations are given in Section 4.6.

2.3.2. 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 [6]. 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 allows the polymorphic type $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$ to be represented by the monotype $\alpha \to \alpha$.

[7] Arguably, *e*ML$^\mathsf{F}$ could be considered half way between Curry style and Church style since some type reconstruction based on first-order unification is still needed for parameters of functions that are left unannotated.

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 $(\alpha \Rrightarrow \sigma)$, which should be read "$\alpha$ abstracts σ." Resuming with the typechecking of a_5, we may write $(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ z : \alpha \vdash \omega^{\dagger}\ \ z : \alpha$ (**3**). The hypothesis that α abstracts σ_{id} allows to abstract the type $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$ of $\omega^{\dagger}$ as $\alpha \to \alpha$. We may then assign the type $\alpha \to \alpha$ to $\omega^{\dagger}$ and type α to the application of $\omega^{\dagger}\ \ z$ when z is assumed to be of type α. Note that abstraction is an asymmetric relation and it is not the case that σ_{id} abstracts α. In particular, $(\alpha \Rrightarrow \sigma)\ z : \alpha \vdash z : \sigma_{\mathsf{id}}$ 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 $(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ \vdash a_5 : \alpha \to \alpha$. Finally discharging the prefix (like in Rule Gen), we may conclude $\vdash a_5 : \forall(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha$. This can be read as "$a_5$ 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 $(\alpha \Rrightarrow \sigma_{\mathsf{id}})$ a *rigid binding* and the position of σ_{id} in this type scheme 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 $\forall(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha$ is not equivalent to $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$, nor to $\forall(\alpha \Rrightarrow \sigma_{\mathsf{id}}, \alpha' \Rrightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha'$, 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 $\forall(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ (\forall(\alpha' \Rrightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha')$ and abstract σ_{id} as α in the binding $(\alpha' \Rrightarrow \sigma_{\mathsf{id}})$, leading to $\forall(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ (\forall(\alpha' \Rrightarrow \alpha)\ \alpha \to \alpha')$, which is equivalent to $\forall(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha$. The later step holds because monotype bounds, which have no other instances but themselves, are "transparent" and can always be inlined, i.e., substituted for the variable that binds them.

Intuitively, polytype bounds may also be expanded. For instance, $\forall(\alpha \Rrightarrow \sigma_{\mathsf{id}}, \alpha' \Rrightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha'$ intuitively stands for $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$. However, this is not technically 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 $\forall(\alpha \Rrightarrow \sigma_{\mathsf{id}}, \alpha' \Rrightarrow \sigma_{\mathsf{id}}, \alpha'' \Rrightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha' \to \alpha''$ or $\forall(\alpha \Rrightarrow \sigma_{\mathsf{id}}, \alpha' \Rrightarrow \sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}})\ \alpha \to \alpha'$ where $\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$ needs in turn to be expanded. However, both are not considered equivalent in *i*ML$^{\mathsf{F}}$. 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 *i*ML$^{\mathsf{F}}$, 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}; the annotation then reveals σ_{id} as the type of (the annotated) x instead of the (weaker) abstract type α. We thus have $(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ x : \alpha \vdash (x : \sigma_{\mathsf{id}}) : \sigma_{\mathsf{id}}$, using a derived rule of the form

$$\frac{(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ x : \alpha \vdash x : \alpha \qquad (\alpha \Rrightarrow \sigma_{\mathsf{id}})\ \alpha \sqsupseteq \sigma_{\mathsf{id}}}{(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ x : \alpha \vdash (x : \sigma_{\mathsf{id}}) : \sigma_{\mathsf{id}}}$$

Once the type σ_{id} has been revealed, it may be instantiated, e.g., into $\alpha \to \alpha$. Therefore, we have $(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ x : \alpha \vdash (x : \sigma_{\mathsf{id}})\ x : \alpha$ and, finally, $\vdash \lambda(x)\ (x : \sigma_{\mathsf{id}})\ x : \forall(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha$. There is still one subtlety when typechecking the simpler program $\lambda(x)\ (x : \sigma_{\mathsf{id}})$. From the intermediate step $(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ x : \alpha \vdash (x : \sigma_{\mathsf{id}}) : \sigma_{\mathsf{id}}$ we may not conclude $(\alpha \Rrightarrow \sigma_{\mathsf{id}})\ \vdash (x : \sigma_{\mathsf{id}}) : \alpha \to \sigma_{\mathsf{id}}$ 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 $(\alpha \Rrightarrow \sigma_{\mathsf{id}}, \alpha' \geq \sigma_{\mathsf{id}})\ \vdash (x : \sigma_{\mathsf{id}}) : \alpha'$. We may then discharge both the context and the prefix and conclude that $\lambda(x : \sigma_{\mathsf{id}})\ x$ has type $\forall(\alpha \Rrightarrow \sigma_{\mathsf{id}}, \alpha' \geq \sigma_{\mathsf{id}})\ \alpha \to \alpha'$. 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(s) $\forall(\alpha \Rrightarrow \sigma, \alpha' \geq \sigma)\ \alpha \to \alpha'$.

2.3.3. Fitting it together into *e*ML$^{\mathsf{F}}$

The type system *e*ML$^{\mathsf{F}}$ we have devised so far does not fit directly into the Curry-style System G that does not permit *any* annotations on source terms. As type abstractions and type instantiation remain implicit in *e*ML$^{\mathsf{F}}$, 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.

$$\begin{array}{llr}
\tau \in \mathcal{T}_e ::= \alpha \mid \tau \to \tau & & e\mathsf{ML}^{\mathsf{F}}\text{ types} \\
\sigma \in \mathcal{S}_e ::= \tau \mid \forall(q)\ \sigma \mid \bot & & e\mathsf{ML}^{\mathsf{F}}\text{ type schemes} \\
q \in \mathcal{Q}_e ::= \alpha \geq \sigma \mid \alpha \Rrightarrow \rho & & e\mathsf{ML}^{\mathsf{F}}\text{ bindings} \\
\rho \in \mathcal{R}_e ::= \tau \mid \forall(\alpha \geq \bot)\ \rho \mid \forall(\alpha \Rrightarrow \rho)\ \rho & & \text{F-like type schemes}
\end{array}$$

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

Of course, we must adapt the instance relation $\leq$ of *i*ML$^{\mathsf{F}}$ to an instance relation $\sqsubseteq$ on *e*ML$^{\mathsf{F}}$ 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 *i*ML$^{\mathsf{F}}$ and *e*ML$^{\mathsf{F}}$. 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 Section 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 ρ as opposed to the abstract type α—we may say that it reveals the concrete type ρ of α. Hence, the let-bound variable x has type ρ and may be used within a with different instances. Of course, we may derive the following typing rule for annotated abstractions:

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

Following this approach, *e*ML$^\mathsf{F}$ 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 prevents ρ to be an arbitrary type scheme σ. While this restriction is not a problem in practice, it is more restrictive than necessary and does not allow to directly map derivations of *i*ML$^\mathsf{F}$ programs to *e*ML$^\mathsf{F}$ programs.

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

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

This allows all type schemes to be *explicitly* coerced along the inverse of type abstraction, called *revelation*. There is still no surprise in the typing rules of *e*ML$^\mathsf{F}$: the power of *e*ML$^\mathsf{F}$ lies in its types, the enforcement of a clear separation between types schemes and types, and the decomposition of the instance relation of *i*ML$^\mathsf{F}$ into a smaller implicit instance relation and explicit revelation, which we explain below in more details.

2.3.4. Design space

The semantics of *e*ML$^\mathsf{F}$ is given by translation into *i*ML$^\mathsf{F}$ by both dropping type annotations and inlining rigid bindings. For the sake of comparison between the instance relation $\sqsubseteq$ of *e*ML$^\mathsf{F}$ and the instance relation $\leq$ of *i*ML$^\mathsf{F}$, we may always see types of *e*ML$^\mathsf{F}$ as types of *i*ML$^\mathsf{F}$ by inlining all rigid bindings. While $\leq$ is uniquely determined by the encoding of its types into sets of System-F types (given in Section 3.2), the relation $\sqsubseteq$ is only a subrelation of $\leq$, so as to allow type inference in *e*ML$^\mathsf{F}$. Although type inference is out of the scope of this paper and described elsewhere [17,16,43], we may explain the choice of $\sqsubseteq$ as follows.

A design choice for *e*ML$^\mathsf{F}$ is to treat polymorphism as first-class, as in *i*ML$^\mathsf{F}$, but never have to guess it. Furthermore, typable expressions must have principal types, so that type inference is both tractable and practical. Concretely, types of *e*ML$^\mathsf{F}$ are enriched so as to distinguish between types $\forall(\alpha \Rightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha$ and $\forall(\alpha \Rightarrow \sigma_{\mathsf{id}}, \alpha' \Rightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha'$, say σ_1 and σ_2, that are confounded in *i*ML$^\mathsf{F}$. In other words, the equivalence relation $\boxminus$, the kernel of $\leq$, which is too coarse, is replaced by the asymmetric, more discriminative abstraction relation $\sqsubseteq$, whose kernel $\equiv$, is smaller than $\boxminus$ and such that $(\sqsubseteq \cup \sqsupseteq)^*$ is $\boxminus$ (**1**), where $\sqsupseteq$, called revelation, is the inverse of $\sqsubseteq$. This ensures that nothing is really lost: type 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 use revelation in *e*ML$^\mathsf{F}$ via explicit type annotations.

Therefore, the design space mainly lies in the choice of the preorder $\sqsubseteq$, which is already constrained by (1), so that *e*ML$^\mathsf{F}$ and *i*ML$^\mathsf{F}$ coincide up to type annotations. The abstraction must not be too coarse, as otherwise *e*ML$^\mathsf{F}$ would coincide with *i*ML$^\mathsf{F}$ exactly, require type annotation and have undecidable type inference. The abstraction must not be too discriminative either, as in order for *e*ML$^\mathsf{F}$ to be a conservative extension of ML, $\equiv$ should at least contain the restriction of $\boxminus$ to ML type schemes. For instance, if $\equiv$ were the identity relation, then *e*ML$^\mathsf{F}$ would distinguish types such as two representations $\forall(\alpha \Rightarrow \tau)\ \alpha \to \alpha$ and $\forall(\alpha \Rightarrow \tau, \alpha' \Rightarrow \tau)\ \alpha \to \alpha'$ of the same type $\tau \to \tau$.

We may remove some degrees of freedom in the definition of $\sqsubseteq$, by requiring revelation to be *confluent*, i.e., $\sqsubseteq ; \sqsupseteq$ (one step of $\sqsubseteq$ followed by one step of $\sqsupseteq$) to be a subrelation of $\sqsupseteq ; \sqsubseteq$. This also implies that $\boxminus$ is in fact equal to $\sqsupseteq ; \sqsubseteq$ (**2**). When splitting $\boxminus$ into $\sqsupseteq ; \sqsubseteq$, there is in fact a natural orientation of the rules that is induced by $\leq$. For example, consider the type schemes σ'_1, defined as $\forall(\alpha \geq \sigma_{\mathsf{id}})\ \alpha \to \alpha$, and σ'_2, defined as $\forall(\alpha \geq \sigma_{\mathsf{id}}, \alpha' \geq \sigma_{\mathsf{id}})\ \alpha \to \alpha'$, obtained from the types σ_1 and σ_2 (defined two paragraphs above) by replacing rigid bounds with flexible ones. Type σ'_1 is an instance of σ'_2 in *i*ML$^\mathsf{F}$. That is, flexible bindings can be shared along the instance relation in *i*ML$^\mathsf{F}$. Therefore, we choose the same direction for rigid bindings and let $\sigma_2 \sqsubseteq \sigma_1$ hold in *e*ML$^\mathsf{F}$.

We thus seek for a confluent pre-order $\sqsupseteq$, such that $\sqsupseteq ; \sqsubseteq$ is $\boxminus$ and $\sqsubseteq \cap \sqsupseteq$, i.e., $\equiv$, contains at least the equivalence of ML type schemes. In addition, we should minimize $\sqsupseteq \setminus \sqsubseteq$ which determines when explicit type annotations are required. That is, maximize $\equiv$ within $\boxminus$ (so as preserve type soundness) while retaining both the *decidability of type inference* and the *existence of principal types* (**3**).

Finally, once $\sqsubseteq$ is determined, the relation $\sqsubseteq$ must be such that $\sqsubseteq \cap \sqsupseteq$ is $\equiv$ and $(\sqsubseteq \cup \sqsupseteq)^*$ is $\leq$. In addition, $\sqsubseteq$ should be chosen as large as possible, so as to minimize the number of type annotations.

Beyond this point, we may only provide hints, as a detailed discussion of type inference is out of the scope of this paper. The preservation of principal types requires the commutation of $\equiv$ with type inference rules [43] and imposes further constraints on $\equiv$. While the original choice for $\equiv$ [17] does verify the criteria (3) above, it is not the largest such relation. The relation proposed in this work is more general. It also verifies the criteria (3), as shown in recent works based on graphs by Rémy and Yakobowski [43,42], although with a different presentation. Whether it is optimal—if we exclude considering special cases—is a difficult question, although the graph presentation, which exposes the commutation properties between type inference and type-instance more clearly, might help find an answer.

In summary, there is actually little choice for the definition of $\sqsubseteq$, except changing the relation $\leq$ itself—or considering arbitrary special cases. The design choice made in this paper is ultimately justified by type inference for Full $e\mathsf{ML}^\mathsf{F}$, shown in two other works with minor differences—a slightly weaker relation in the original work or a slight difference in the definition of the relation in the more recent works based on graphs. Interestingly, while Full ML^F uses a richer set of types (where rigid bounds can be arbitrary σ-types rather and not only ρ-types), the restriction of its type-instance relation to ML^F types is exactly the type-instance relation of ML^F. In summary, the semantics of types of $i\mathsf{ML}^\mathsf{F}$ determines type-instance $\leq$ in $i\mathsf{ML}^\mathsf{F}$ which in turn leaves little freedom for $\sqsubseteq$ and $\vDash$ in $e\mathsf{ML}^\mathsf{F}$ and also strongly constraints their generalization to Full $e\mathsf{ML}^\mathsf{F}$.

3. *i*ML$^\mathsf{F}$, Curry-style ML$^\mathsf{F}$

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

3.1. Types and prefixes

Flexible bindings are the main novelty of $i\mathsf{ML}^\mathsf{F}$. So as to be self-contained, we remind their definition here:

$$\begin{array}{lr} \tau \in \mathcal{T}_i ::= \alpha \mid \tau \to \tau \mid \forall(\alpha \geq \bot)\ \tau & \text{Types} \\ \sigma \in \mathcal{S}_i ::= \tau \mid \forall(q)\ \sigma \mid \bot & \text{Type Schemes} \\ q \in \mathcal{Q}_i ::= \alpha \geq \sigma & \text{Bindings} \end{array}$$

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 2.1.1.

Inert types. In $i\mathsf{ML}^\mathsf{F}$, a type is *inert* 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 inert types is written $\mathcal{I}_i$. Intuitively, inert types have no other instance but themselves (up to equivalence), i.e., if τ is in $\mathcal{I}_i$ and τ is an instance of τ' then τ' is an instance of τ—and so equivalent to τ.

Polytypes. The set $\mathcal{T}_i$ of $i\mathsf{ML}^\mathsf{F}$ 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 $\mathcal{T}_i$ 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 $\mathcal{T}_i$, 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.

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 $\{\forall(\alpha)\ t \mid t \in S\}$ and $\theta(S)$ for $\{\theta(t) \mid t \in S\}$.

Definition 3.2.1 (*Semantics of types*). The semantics of a type τ, written $\{\!\{\tau\}\!\}$ is the instance closure of its translation to 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.

The instance relation of System F induces an instance relation in $i\mathsf{ML}^{\mathsf{F}}$. 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 $\circ$. That is, $f \circ g$ is the function $x \mapsto f(g(x))$. Given a family of functions $(f_i)^{i\in 1..n}$, we write $\circ^{i\in 1,\ldots,n} f_i$ for $f_1 \circ \cdots \circ f_n$.

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 $\circ^{i\in 1,\ldots,n}(\alpha_i \mapsto t_i)$ where $t_i \in \{\!\{\sigma_i\}\!\}$ for i in $1,\ldots,n$. 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 *canonical decomposition* of an F-substitution is a decomposition of the form $\circ^{i\in 1..n}\theta_i$ where all θ_i's are F-substitutions (hence 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. *Given a prefix of the form $Q_i^{i\in 1..n}$, any member of $\{\!\{Q_i^{i\in 1..n}\}\!\}$ 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 F-substitution, then $\theta_2 \circ (\theta_1 \circ \theta_2) = (\theta_1 \circ \theta_2) \circ \theta_1 = (\theta_1 \circ \theta_2)$.*

Definition 3.2.7 (*Type-instance*). The instance relation $\leq$ in $i\mathsf{ML}^{\mathsf{F}}$ 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 $\leq$.

Remarkably, type-instance treats both sides of the arrow symmetrically. Thus, the generalization of types to allow arbitrary type constructors would not need a notion of variance for type constructors. Technically, type-instance does not directly operate under arrow types, similarly to $\leq_{\mathsf{F}}$ and by contrast with both $<:$ and $\leq_\eta$. However, it does operate inside bindings, which may indirectly affect arrow types. For instance, $\forall(\alpha \geq \sigma_{\mathsf{id}})\ \tau \to \alpha$ can be instantiated into $\forall(\alpha \geq \sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}})\ \tau \to \alpha$ which is equivalent to $\tau \to (\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}})$. However, $\tau \to (\sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}})$ is not an instance of $\tau \to \sigma_{\mathsf{id}}$. Thus, the replacement of $\tau \to \sigma_{\mathsf{id}}$ by $\forall(\alpha \geq \sigma_{\mathsf{id}})\ \tau \to \alpha$ is a way to tell explicitly within types that the right-hand side of the arrow may be instantiated. We could also write $\forall(\alpha \geq \sigma_{\mathsf{id}})\ \alpha \to \tau$ to allow instantiation on the left-hand side of the arrow, although such a type may not be more useful than $\forall(\alpha)\ \alpha \to \tau$. Much more interesting, we may allow instantiation on several occurrences simultaneously, as in the type $\forall(\alpha \geq \sigma_{\mathsf{id}})\ \alpha \to \alpha$ (of choose id). It is actually essential that type instantiation be allowed under arrow types and simultaneously in several occurrences, so that programs such as choose id have principal types. It is also important that this mechanism does not depend on variances of type constructors, which would prevent the use of first-order unification for type inference.

Expanding the definition of equivalence, 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—see in Section 2.1.4. We may thus invert the definition $\lceil\cdot\rceil$ into a function from $\mathcal{T}_{\mathsf{F}}$ to $\mathcal{T}_i$. For any type t in $\mathcal{T}_{\mathsf{F}}$, we write $\lfloor t\rfloor$ the type τ of $\mathcal{T}_i$ defined up to equivalence such that t is $\lceil\tau\rceil$. We extend the $\lfloor\cdot\rfloor$-mapping pointwise 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 $\{\!\{Q_1\}\!\} \supseteq \{\!\{Q_2\}\!\}$.

*i*Con-AllLeft
$$\frac{(Q)\ \sigma_1\ \mathcal{R}\,\sigma_2}{(Q)\ \forall(\alpha \geq \sigma_1)\ \sigma\ \mathcal{R}\,\forall(\alpha \geq \sigma_2)\ \sigma}$$

*i*Con-AllRight
$$\frac{(Q, \alpha \geq \sigma)\ \sigma_1\ \mathcal{R}\,\sigma_2}{(Q)\ \forall(\alpha \geq \sigma)\ \sigma_1\ \mathcal{R}\,\forall(\alpha \geq \sigma)\ \sigma_2}$$

*i*Con-Arrow
$$\frac{(Q)\ \tau_1\ \mathcal{R}\,\tau_2 \qquad (Q)\ \tau_1'\ \mathcal{R}\,\tau_2'}{(Q)\ \tau_1 \to \tau_1'\ \mathcal{R}\,\tau_2 \to \tau_2'}$$

Fig. 5. Type congruence in *i*ML$^{\mathsf{F}}$.

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

*i*Equ-Free
$$\frac{\alpha \notin \mathsf{ftv}(\sigma)}{(Q)\ \forall(\alpha \geq \sigma')\ \sigma \boxminus \sigma}$$

*i*Equ-Inert
$$\frac{(\alpha \geq \sigma') \in Q \qquad (\emptyset)\ \sigma' \boxminus \tau \qquad \tau \in \mathcal{I}_i}{(Q)\ \sigma \boxminus \sigma[\tau/\alpha]}$$

*i*Equ-Var
$$(Q)\ \forall(\alpha \geq \sigma)\ \alpha \boxminus \sigma$$

Fig. 6 Syntactic type equivalence $\boxminus$ in *i*ML$^{\mathsf{F}}$.

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 simpler (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.

Definition 3.3.1 (*Congruence*). A relation $\mathcal{R}$ is *$\geq$-congruent* when it satisfies both *i*Con-AllLeft and *i*Con-AllRight (Fig. 5). It is *congruent* if it is $\geq$-congruent and moreover satisfies *i*Con-Arrow (Fig. 5).

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 Fig. 6.

We intendedly use the same symbol $\boxminus$ for syntactic and semantic equivalence, because the two definitions coincide, as explained below.

Rule *i*Equ-Comm allows for commutation of bindings with disjoint domains; Rule *i*Equ-Free allows for removal of useless bindings; Rule *i*Equ-Inert allows for inlining of inert bindings (inert types are defined in Section 3.1.). Rule *i*Equ-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 *i*Equ-Inert 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 [17] or [16, Chapter 3] for more details.

Rules *i*Equ-Inert, *i*Equ-Free, and *i*Equ-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 [17] or [16, 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 Section 3.2.

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

Fig. 7. Syntactic type-instance $\leq$ in $i\mathsf{ML}^{\mathsf{F}}$.

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 any type constructor (i.e., the arrow in our setting), recursively defined as follows:

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

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 inert or α is not exposed in σ. If $t' \in \{\!\{\lfloor\theta\rfloor(\sigma)\}\!\}$ and $\alpha \notin \mathsf{ftv}(t')$, then $t' \in \theta(\{\!\{\sigma\}\!\})$.*

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 in Appendix)

Lemma 3.3.9 (Soundness of syntactic equivalence). *The semantic type equivalence relation satisfies all rules of Fig. 6.*

(Proof in Appendix)

Definition 3.3.10 (*Syntactic type-instance*). The syntactic instance relation is the smallest transitive and $\geq$-congruent relation on closed judgments satisfying rules of Fig. 7.

We intendedly use the same symbol for syntactic and semantic type-instance because the two definitions coincide, as explained below.

Rule *i*INS-EQUIV ensures that type-instance contains type equivalence. Rule *i*INS-BOT states that $\bot$ is the most general type. Rule *i*INS-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). *i*INS-SUBST inlines a flexible binding $(\alpha \geq \tau)$ with a type bound τ, 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 is 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. Note that this condition also prevents the case where σ is itself α, as in the type-instance $(Q)\ \forall(\alpha \geq \tau)\ \alpha \leq \tau$. However, this judgment is a particular case of equivalence, and thus still provable using rule *i*INS-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 Section 2.1.5.*
(ii) *If $(Q)\ \tau_1 \boxminus \tau_2$ holds, then $(Q)\ \sigma[\tau_1/\alpha] \boxminus \sigma[\tau_2/\alpha]$ holds.*

(Proof in Appendix)

Lemma 3.3.12 (Soundness of syntactic instance). *The type-instance relation satisfies all rules of Fig. 7.*

(Proof in Appendix)

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 Fig. 6. Any type-instance relation can be derived with rules of Fig. 7.*

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 Section 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 [42]) 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 *i*ML$^{\mathsf{F}}$ is an instance of System G, its typing rules are as described in Fig. 4. Some examples of typings have been introduced informally in Section 1. Other examples will be presented with more details in Section 3.7.

There is an interesting admissible rule in *i*ML$^{\mathsf{F}}$ that helps typing abstractions:

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

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 *i*Ins-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.

3.4.1. 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

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

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:

$$\begin{array}{llll} (Q, \alpha \geq \sigma) & \forall(\alpha \geq \sigma)\ \sigma' = & \forall(\beta \geq \sigma)\ \sigma'[\beta/\alpha] & \text{by } \alpha\text{-conversion} \\ & \leq & \forall(\beta \geq \alpha)\ \sigma'[\beta/\alpha] & \text{by } i\text{Ins-Hyp} \\ & & & \text{and } i\text{Con-AllLeft} \\ & \boxminus & \sigma' & \text{by } i\text{Equ-Inert} \end{array}$$

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:

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

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 Section 3.7.

3.4.2. Recursion

There is no explicit construction for recursion in *i*ML$^{\mathsf{F}}$. From a typing point of view, recursion can always be treated as the application of a fixed point combinator, given as a constant with its type scheme in the initial typing environment. For instance, $\mathsf{let\ rec}\ f = \lambda(x)\ a_1\ \mathsf{in}\ a_2$ in ML may be seen as $\mathsf{let}\ f = \mathrm{fix}\ (\lambda(f)\ \lambda(x)\ a_1)\ \mathsf{in}\ a_2$ where fix behaves, for instance, as

$$\lambda(f)\ (\lambda(x) f\ (\lambda(y)\ x\ x\ y))\ (\lambda(x) f\ (\lambda(y)\ x\ x\ y))$$

and is assigned the type scheme $\forall(\gamma \geq \forall(\alpha\beta)\ \alpha \to \beta)\ (\gamma \to \gamma) \to \gamma$. The derived typing rule is:

$$\textsc{Rec}^{\star}\quad \frac{(Q, Q')\ \Gamma, f : \tau \vdash \lambda(x)\ a_1 : \tau \qquad (Q)\ \Gamma, f : \forall(Q')\ \tau \vdash a_2 : \tau_2}{(Q)\ \Gamma \vdash \mathsf{let\ rec}\ f = \lambda(x)\ a_1\ \mathsf{in}\ a_2 : \tau_2}$$

This view implies that f must be assigned a type τ and not a type scheme while type checking a_1. Thus, f may be used polymorphically during the recursion, but only with System-F polymorphism.

Using a primitive construct for recursion would allow f to be assigned a type scheme, which would be slightly more general. However, this extension would break the encoding of ML$^{\mathsf{F}}$ into System F, described below in Section 3.6. The key is that where f is assigned a type, the recursion itself can be seen as monomorphic, as every occurrence of f in a_1 sees f with the *same* polymorphic type, which may then be instantiated appropriately. By contrast, when f is a type scheme σ, several occurrences of f in a_1 may view f with different types (that are all instances of σ, of course).

3.5. System F as a subset of (Simple) *i*ML$^{\mathsf{F}}$

We recall that Simple ML$^{\mathsf{F}}$ is *i*ML$^{\mathsf{F}}$ without terms with local bindings. Before showing the inclusion of System F in Simple ML$^{\mathsf{F}}$, we show the inclusion of their instance relations.

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

(Proof in Appendix)

The next lemma states the inclusion of System F into Simple ML$^{\mathsf{F}}$.

The translation $\lfloor \Gamma \rfloor$ of a typing context Γ into one of Simple ML$^{\mathsf{F}}$, 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 iML$^{\mathsf{F}}$ $:: (Q)\ \lfloor \Gamma \rfloor \vdash a : \lfloor t \rfloor$ holds under any prefix Q binding the free variables of Γ and t.*

(Proof in Appendix)

3.6. Type soundness, by viewing *i*ML$^{\mathsf{F}}$ as a subset of F$^{\mathsf{let}}$

In this section, we introduce a new type system, called F$^{\mathsf{let}}$ that extends System F with let-bindings *a la ML*, which is already large enough to contain *i*ML$^{\mathsf{F}}$. The language F$^{\mathsf{let}}$ is actually a subset of the more notorious extension of System F with intersection types $\mathsf{F}_{\wedge}$. The type soundness of $\mathsf{F}_{\wedge}$, which is folklore knowledge, ensures the type soundness of *i*ML$^{\mathsf{F}}$.

3.6.1. From ML and System F to F$^{\mathsf{let}}$

The language 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 variable 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 resulting expression may be exponentially larger than the original one. 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[8].

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 $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$

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

as 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.

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

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

Since let-reduction may duplicate let-redexes but not create new ones, it must terminate (by Levy's finite development theorem for the λ-calculus [1]). 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:

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

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:

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

This system is a particular case of the extension of System F with intersection types studied by [35],[9] 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.

3.6.2. Type soundness of F^{let}

Type soundness relates the static semantics of programs, i.e., well-typedness, to their dynamic semantics, i.e., evaluation. In pure λ-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 $\mathsf{F}^{\mathsf{let}}$.

The 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 out of the scope of this paper.

3.6.3. Encoding iML^F into F^{let}

We show that $i\mathsf{ML}^{\mathsf{F}}$ is a subset of $\mathsf{F}^{\mathsf{let}}$ by translating typing derivations of $i\mathsf{ML}^{\mathsf{F}}$ into typing derivations of $\mathsf{F}^{\mathsf{let}}$. For that purpose, we instrument typing judgments of $i\mathsf{ML}^{\mathsf{F}}$ $(Q)\ \Gamma \vdash a : \sigma$ 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 $i\mathsf{ML}^{\mathsf{F}} :: (Q)\ a \vdash \sigma$ and returns a context Δ. These judgments are defined by typing rules of Fig. 8.

In the translation, we distinguish let-bound variables, written x, from λ-bound variables, written y. Hence, the two rules *i*F-Var-Let and *i*F-Var-Fun corresponding to the usual unique rule for variables. Notice that only the *i*F-Var-Let inserts

[9] The presentation of [35] is in Church style, but this is irrelevant here.

*i*F-Var-Let

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

*i*F-Var-Fun

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

*i*F-Fun

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

*i*F-App

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

*i*F-Inst

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

*i*F-Gen

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

*i*F-Let-o

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

*i*F-Let

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

Fig. 8. Translating *i*ML$^{\mathsf{F}}$ to F$^{\mathsf{let}}$.

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 *i*F-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 *i*F-Var-Let. Rule *i*F-Let-o is for the degenerate case where x does not appear free in a'. We must still typecheck the premise once to ensure that a is well-typed—since in a call-by-value strategy 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 in Appendix)

The next two lemmas show the soundness of *i*ML$^{\mathsf{F}}$ by translation into F$^{\mathsf{let}}$, 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 in Appendix)

Theorem 2. *iML^{F} is a subset of F^{let}. More precisely, if the judgment iML^{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 in Appendix)

Type soundness. Type soundness is a corollary of Theorem 2, as it ensures that $i\mathsf{ML}^\mathsf{F}$ is as safe as $\mathsf{F}^{\mathsf{let}}$, 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 $\mathsf{F}^{\mathsf{let}}$ 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

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

We already know that at least one of the two last inclusions is strict, as the term a^ω_{IK} defined as $\mathsf{let}\ y = \omega\ \mathsf{in}\ K\ (y\ I)\ (y\ K)$ is in $\mathsf{F}^{\mathsf{let}}$ (as it let-reduces to a term in F) but it is not in F [7]. (Notice that a^ω_{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 show an example that is typable in ML^F but not in System F in the following subsection. We now argue (informally) that a^ω_{IK} is not in ML^F.

Intuitively, Rule Inter is much more powerful than rule *i*F-Let since a conjunctive type may be an arbitrary (finite) set of types in Inter, whereas *i*F-Let only allows to form conjunctions between types that belong to a common type scheme. To see that a^ω_{IK} is not in ML^F, we may reproduce the argument used for F by [7] by analyzing all possible derivations of a^ω_{IK} in ML^F. In fact, the parameter y will be assigned a type σ that must be a type for ω. In turn, as the parameter x of ω 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 reasoning of [7] (see also [33, Section 23] and [50]), t must be of the 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^ω_{IK} is not typable in Full ML^F either.

Subject reduction. The subject reduction property holds in Simple $i\mathsf{ML}^\mathsf{F}$ as a consequence of the two-directional encoding between System F and Simple $i\mathsf{ML}^\mathsf{F}$.

We expect subject reduction to hold in $i\mathsf{ML}^\mathsf{F}$, since it holds in Full ML^F [16]. However, we did not check this result, as type soundness has already been established by encoding $i\mathsf{ML}^\mathsf{F}$ into $\mathsf{F}^{\mathsf{let}}$ in Section 3.6.

3.7. Expressiveness and modularity

As typing derivations of System F can be mapped directly to typing derivations of $i\mathsf{ML}^\mathsf{F}$, the language $i\mathsf{ML}^\mathsf{F}$ performs at least as well as System F with regard to typechecking. We claim that $i\mathsf{ML}^\mathsf{F}$ is strictly more expressive than System F in a rather unusual but practically meaningful sense, as it is more modular than System F. Indeed, we show that constructing data-structures containing polymorphic values is not modular in System F, while it is in $i\mathsf{ML}^\mathsf{F}$.

For that purpose, we exhibit an unannotated expression b, which, as far as typing is concerned, can be compared to a data-structure, say a list, containing a polymorphic value, say id. This expression is typable in System F, hence also in $i\mathsf{ML}^\mathsf{F}$. Then, b is inserted[10] both into a monomorphic data-structure (for the sake of conciseness, we use constants in expressions) and into a (truly) polymorphic data-structure. In System F, though, two typing derivations are necessary for b, depending on the type of the data-structure it will be inserted into. Worse, the number of differences between both typing derivations is proportional to the size of b. On the contrary, in $i\mathsf{ML}^\mathsf{F}$, we show that the very same derivation can be used for b in both cases. (This is of course simply a consequence of principal types.)

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

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 [16] where it is shown how a single type of $e\mathsf{ML}^\mathsf{F}$ captures all type abstractions and type applications of a given expression in System F. However, this formal result uses the principal type property of $e\mathsf{ML}^\mathsf{F}$, which we do not show here, as we do not address type inference—hence, our informal, but simpler explanation.

Although the following example is presented in $i\mathsf{ML}^\mathsf{F}$ and based on the modularity of typing derivations, it can be reproduced in $e\mathsf{ML}^\mathsf{F}$ since only type annotations on function parameters are needed in $e\mathsf{ML}^\mathsf{F}$ 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 choose, id, comp, and ω that satisfy the following signature in $i\mathsf{ML}^\mathsf{F}$ (their signature in System F follows by translation).

[10] Insertion is modeled by mere application.

$$\begin{array}{rcll} \text{choose} & : & \forall(\alpha)\ \alpha \to \alpha \to \alpha & \\ \text{id} & : & \forall(\alpha)\ \alpha \to \alpha & \stackrel{\triangle}{=} \sigma_{\text{id}} \\ \text{comp} & : & (\text{i} \to \text{i}) \to (\text{i} \to \text{i}) & \stackrel{\triangle}{=} \sigma_{\text{comp}} \\ \omega & : & \sigma_{\text{id}} \to \sigma_{\text{id}} & \stackrel{\triangle}{=} \sigma_{\omega} \end{array}$$

For instance, comp could be the expression $\lambda(g)\ (\lambda(x)\ g\ (\text{succ}\ x))$ where succ is the successor function for integers. From a *typing* point of view, we claim that choose is similar to a datatype constructor. Indeed, choose v w can be compared to the two-element list $[v; w]$, since v and w are required to have the same type. Of course, choose v w does not *behave* as a list, since it forgets its elements after application (at least one of them). Similarly, choose v can be compared to the singleton list $[v]$. In particular, the typing of choose id raises the same difficulties than the typing of [id] as far as the position of quantifiers is concerned: namely, is its type $list(\sigma_{\text{id}})$, or $\forall(\alpha)\ list(\alpha \to \alpha)$, or even $\forall(\alpha)\ list(\forall(\beta)\ (\alpha \to \beta) \to (\alpha \to \beta)$?

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

$$a^0 \stackrel{\triangle}{=} a \qquad a^{n+1} \stackrel{\triangle}{=} \text{choose}\ \ a^n \qquad \sigma^0 \stackrel{\triangle}{=} \sigma \qquad \sigma^{n+1} \stackrel{\triangle}{=} \forall(\alpha \geq \sigma^n)\ \alpha \to \alpha$$

(From a typing point of view, a^n is like $[..[a]..]$, the nested singleton list ending at depth n with a as a leaf.) Assume $\Gamma_0 \vdash a^n : \sigma^n$ (**1**). By Lemma 2.1.1.i, we have $(\alpha \geq \sigma^n)\ \Gamma_0 \vdash a^n : \sigma^n$ (**2**). Then, we have the following derivation of $\Gamma_0 \vdash a^{n+1} : \sigma^{n+1}$ (**3**) in *i*ML$^{\mathsf{F}}$:

$$\dfrac{\dfrac{\dfrac{(\alpha \geq \sigma^n)\ \Gamma_0 \vdash \text{choose} : \forall(\alpha)\ \alpha \to \alpha \to \alpha}{(\alpha \geq \sigma^n)\ \Gamma_0 \vdash \text{choose} : \alpha \to \alpha \to \alpha}\ \textsc{Inst} \qquad \dfrac{(2) \qquad (\alpha \geq \sigma^n)\ \sigma^n \leq \alpha}{(\alpha \geq \sigma^n)\ \Gamma_0 \vdash a^n : \alpha}\ \textsc{Inst}}{(\alpha \geq \sigma^n)\ \Gamma_0 \vdash \text{choose}\ a^n : \alpha \to \alpha}\ \textsc{App}}{(3)}\ \textsc{Gen}$$

Hence, by induction judgement (1) holds for every integer n provided $\Gamma_0 \vdash a : \sigma$ holds. Observe that, by construction, we have $\Gamma_0 \vdash \text{id} : \sigma_{\text{id}}$ (**4**), $\Gamma_0 \vdash \text{comp} : \sigma_{\text{comp}}$ (**5**), and $\Gamma_0 \vdash \omega: \sigma_\omega$ (**6**).

Assume moreover that $\sigma_{\text{id}} \leq \sigma$. Using Rule *i*Con-AllLeft repeatedly, we obtain $\sigma_{\text{id}}^n \leq \sigma^n$. In particular, $\sigma_{\text{id}}^{n+1} \leq \forall(\alpha \geq \sigma^n)\ \alpha \to \alpha$ (7). Let b be id^{n+1}. We have $\Gamma \vdash b : \sigma_{\text{id}}^{n+1}$, which gives $\Gamma \vdash b : \forall(\alpha \geq \sigma^n)\ \alpha \to \alpha$ by Rule Inst and (7). Using generalized Rule App* we have $\Gamma_0 \vdash b\ a^n : \sigma^n$ (**8**).As both σ_ω and σ_{comp} are instances of σ_{id}, we may thus conclude that both applications $b\ \omega^n$ and b comp^n are typable in *i*ML$^{\mathsf{F}}$. 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 *i*ML$^{\mathsf{F}}$.

In System F, both applications are typable as well, but unlike in *i*ML$^{\mathsf{F}}$, 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 id^{n+1} ω^n (compared to id^{n+1} comp^n) induces changes in the typing of the body of the function a_{n+1} up to depth n.

As explained above, this example can easily be read back with lists. More generally, it shows that when incrementally building a collection of objects in ML$^{\mathsf{F}}$, each intermediate collection may be kept as polymorphic as possible, since it can be seen as a collection of a less polymorphic values at any time and at no cost. In particular, a new element that is less polymorphic than the previous elements of the collection may later be added to the collection. By contrast, in System F, one must know the best common type of all the elements before collecting the very first one.

To wrap up the example, it follows that the expressions a_{let}^n defined as let $x =$ id^{n+1} in let $z = x\ \omega^n$ in x comp^n for any n, and, as a particular case, the expression let $x =$ choose id in let $z = x\ \omega$ in x comp are not typable in System F. They are all typable in *i*ML$^{\mathsf{F}}$, indeed.

Notice that although F$^{\text{let}}$ is larger than *i*ML$^{\mathsf{F}}$, it is not necessarily better: while a_{let}^n is also typable in F$^{\text{let}}$ its typing derivation in F$^{\text{let}}$ is still problematic as the typing derivation for id contains two similar sub-derivations specialized for ω and comp, respectively, and joined with $\wedge$ rather than a principal typing derivation independent of further applications, as could be done in *i*ML$^{\mathsf{F}}$. 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_{let}^n.

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

In summary, the main benefit of *i*ML$^{\mathsf{F}}$ over System F is that its types are more principal, so that typing derivations of *i*ML$^{\mathsf{F}}$ are more modular than typing derivations in System F. This is a key for the design of *e*ML$^{\mathsf{F}}$ that permits simple type inference. In fact, many modularity properties of *i*ML$^{\mathsf{F}}$ are kept in *e*ML$^{\mathsf{F}}$ (see type preserving transformations in Section 4.6).

4. *e*ML$^{\mathsf{F}}$, Church-style ML$^{\mathsf{F}}$

In this section, we step on modular typechecking properties of *i*ML$^{\mathsf{F}}$ to design a version with optional type annotations, called *e*ML$^{\mathsf{F}}$, that has a clear and intuitive specification of *where* and *when* to put type annotations. After a presentation of *e*ML$^{\mathsf{F}}$ types (Section 4.2) and relations between them, we introduce terms, typing rules and show type safety (Section 4.3) by translation of well-typed programs into *i*ML$^{\mathsf{F}}$. We also exhibit a translation of System F into *e*ML$^{\mathsf{F}}$ that shows the intrinsic expressiveness of *e*ML$^{\mathsf{F}}$ and its very low demand on the amount of explicit type information (Section 4.4). We show that *e*ML$^{\mathsf{F}}$ is a conservative extension to ML (Section 4.5). Finally, we present a few useful program transformations that are type preserving in *e*ML$^{\mathsf{F}}$ (Section 4.6).

4.1. Specifying where and when to put type annotations

In *i*ML$^{\mathsf{F}}$, 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 are 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 permitted, as it would allow—and hence force—type inference to guess (impredicative) 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.

4.2. Types, prefixes, and type relations

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

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

As in ML, monotypes are simple types, for the purpose of type inference. This contrasts with System F or *i*ML$^{\mathsf{F}}$, for which type inference is not considered.

We use the symbol $\diamond$ as a meta-variable that denotes either $\geq$ or $\Rightarrow$. For example, $(\alpha \diamond \sigma)$ stands for either $(\alpha \Rightarrow \sigma)$ or $(\alpha \geq \sigma)$, which are called *rigid* (respectively, flexible) *bindings*. We also say that α is *rigidly* (respectively, *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 *e*ML$^{\mathsf{F}}$ types do not form a superset of *i*ML$^{\mathsf{F}}$ types, since for type inference purposes, they cannot have quantifiers. This may be surprising. However, all of *i*ML$^{\mathsf{F}}$ types have a counterpart in *e*ML$^{\mathsf{F}}$ (as implied by Theorem 4). For instance, the *i*ML$^{\mathsf{F}}$ type $(\forall(\alpha_1)\ \tau_1) \to (\forall(\alpha_2)\ \tau_2)$ is not an *e*ML$^{\mathsf{F}}$ type. However, it can be represented as the *e*ML$^{\mathsf{F}}$ type $\forall(\beta_1 \Rightarrow \sigma_1, \beta_2 \Rightarrow \sigma_2)\ \beta_1 \to \beta_2$ via extra rigid bindings. In fact, *e*ML$^{\mathsf{F}}$ types are more precise as there may be several types of *e*ML$^{\mathsf{F}}$ mapped to the same *i*ML$^{\mathsf{F}}$ type. This refinement of types is used in *e*ML$^{\mathsf{F}}$ to distinguish polymorphism that is given from polymorphism that is assumed.

Inert types $\mathcal{I}_e$. A type is inert if it is of the form $\forall(Q)\ \tau$, where $\tau \notin \vartheta$ and Q is rigid. The set of inert types is written $\mathcal{I}_e$. This definition is indeed the counterpart of inert types $\mathcal{I}_i$ in *i*ML$^{\mathsf{F}}$.

Relations between types. The equivalence and instance relations in *i*ML$^{\mathsf{F}}$ are adapted to *e*ML$^{\mathsf{F}}$ to deal with rigid bindings. As explained earlier 2.3, type equivalence $\boxminus$ in *i*ML$^{\mathsf{F}}$ is too large to permit type inference and has been split into two inverse relations: abstraction $\sqsubseteq$ and revelation $\sqsupseteq$. More precisely, type equivalence $\boxminus$ in *i*ML$^{\mathsf{F}}$ corresponds to the transitive closure of $\sqsubseteq \cup \sqsupseteq$, while type equivalence $\equiv$ in *e*ML$^{\mathsf{F}}$ 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.

$$\text{eCon-FlexLeft}\quad \frac{(Q)\ \sigma_1\ \mathcal{R}\ \sigma_2}{(Q)\ \forall(\alpha \geq \sigma_1)\ \sigma \mathcal{R}\ \forall\ (\alpha \geq \sigma_2)\ \sigma} \qquad \text{eCon-AllLeft}\quad \frac{(Q)\ \sigma_1\ \mathcal{R}\ \sigma_2}{(Q)\ \forall(\alpha \diamond \sigma_1)\ \sigma \mathcal{R} \forall\ (\alpha \diamond \sigma_2)\ \sigma}$$

$$\text{eCon-AllRight}\quad \frac{(Q, \alpha\ \diamond \sigma)\ \sigma_1\ \mathcal{R}\ \sigma_2}{(Q)\ \forall(\alpha \diamond \sigma)\ \sigma_1\ \mathcal{R}\ \forall(\alpha \diamond \sigma)\ \sigma_2}$$

Fig. 9. Type congruence in *e*ML$^{\mathsf{F}}$.

$$\text{eEqu-Comm}\quad \frac{\alpha_1 \notin \mathsf{ftv}\,(\sigma_2) \qquad \alpha_2 \notin\ \mathsf{ftv}\,(\sigma_1)}{(Q)\ \forall(\alpha_1\ \diamond_1\ \sigma_1)\ \forall(\alpha_2\ \diamond_2\ \sigma_2)\ \sigma \equiv \forall(\alpha_2\ \diamond_2\ \sigma_2)\ \forall(\alpha_1\ \diamond_1\ \sigma_1)\ \sigma} \qquad \text{eEqu-Free}\quad \frac{\alpha \notin \mathsf{ftv}(\sigma)}{(Q)\ \forall(\alpha \diamond \sigma')\ \sigma \equiv \sigma}$$

$$\text{eEqu-Mono}\quad \frac{(\alpha \diamond \tau) \in Q}{(Q)\ \sigma \equiv \sigma[\tau/\alpha\ ]} \qquad \text{eEqu-Var}\quad (Q)\ \forall(\alpha \diamond \sigma)\ \alpha \equiv \sigma \qquad \text{eEqu-Inert}\quad \frac{\sigma \in \mathcal{I}_e}{(Q)\ \forall(\alpha \geq \sigma)\ \sigma' \equiv \forall(\alpha \Rightarrow \sigma)\ \sigma'}$$

Fig. 10. Type equivalence in *e*ML$^{\mathsf{F}}$.

Definition 4.2.1 (*Congruence*). A relation $\mathcal{R}$ is $\geq$-*congruent* if it satisfies both *e*CON-FLEXLEFT and *e*CON-ALLRIGHT (Fig. 9). It is *congruent* if it satisfies both *e*CON-ALLLEFT and *e*CON-ALLRIGHT.

Definition 4.2.2 (*Equivalence*). The equivalence relation, written $\equiv$, is the smallest congruent equivalence relation satisfying the rules of Fig. 10.

Rules of Fig. 10 are straightforward adaptations of those on Fig. 6. The only difference is the replacement of single $\geq$ bound in *i*ML$^{\mathsf{F}}$ by the two possible bounds $\geq$ or $\Rightarrow$ in *e*ML$^{\mathsf{F}}$. Rule *i*EQU-INERT is split into two rules: *e*EQU-MONO inlines directly monotypes, whereas *e*EQU-INERT uses a rigid binding as an alias for an inert type σ (since direct substitution would not be possible in general).

We remind that ϑ is the set of type variables. We write $\mathcal{V}$ for the set of type schemes that are $\equiv$-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 4.2.3 (*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 heads of all elements of $\mathcal{V}$ are type variables. We can show the converse—a type scheme σ, the head of which is a type variable, is in $\mathcal{V}$—by an easy structural induction on σ. Hence, $\mathcal{V}$ is also the set $\{\sigma \in \mathcal{S} \mid \mathsf{head}(\sigma) \in \vartheta\}$ of type schemes the head of which are type variables.

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.

To explain abstraction, we introduce type scheme contexts, written S, as a type scheme with a single hole "$\cdot$", which may be defined by the following BNF grammar:

$$S ::= \ \cdot\ \mid \forall(\alpha \diamond S)\ \sigma \mid \forall(\alpha \diamond \sigma)\ S$$

Abstraction is essentially structural, except for the key axiom that retrieves an assumption from the prefix (Rule *e*ABS-HYP of Fig. 11). However, a peculiarity of abstraction is that it is congruent *only* in all contexts ending with a true rigid binding, that is, type scheme contexts of the form $\forall(\alpha \Rightarrow\ \cdot\ )\ \sigma$ where σ is not equivalent to α. This condition is ensured by the stronger requirement $\sigma \notin \mathcal{V}$ of Rule *e*ABS-SHARPLEFT.

Omitting the condition would allow pathological contexts such as $\forall(\beta \geq \forall(\alpha \Rightarrow\ \cdot\ )\ \alpha)\ \tau$, which are equivalent to $\forall(\beta \geq\ \cdot\ )\ \tau$. In particular, this would allow to conclude $(Q)\ \forall(\alpha \geq \sigma_1)\ \tau \sqsubseteq \forall(\alpha \geq \sigma_2)\ \tau$ from $(Q)\ \sigma_1 \sqsubseteq \sigma_2$. For example, we would have $(\alpha \Rightarrow \sigma)\ \forall(\beta \geq \sigma)\ \tau \sqsubseteq \forall(\beta \geq \alpha)\ \tau$, which implies $\forall(\alpha \Rightarrow \sigma)\ \forall(\beta \geq \sigma)\ \tau \sqsubseteq \forall(\alpha \Rightarrow \sigma)\ \forall(\beta \geq \alpha)\ \tau$, i.e., $\forall(\alpha \Rightarrow \sigma)\ \forall(\beta \geq \sigma)\ \tau \sqsubseteq \forall(\alpha \Rightarrow \sigma)\ \tau[\alpha/\beta]$, by Rule *e*CON-ALLRIGHT. However, this is unsound with respect to its counter-

$$\textit{e}\textsc{Abs-Equiv}\ \frac{(Q)\ \sigma_1 \equiv \sigma_2}{(Q)\ \sigma_1 \mathrel{\sqsubset\!\!\!\!-}^{\sharp} \sigma_2} \qquad \textit{e}\textsc{Abs-Hyp}\ \frac{(\alpha \Rightarrow \sigma) \in Q}{(Q)\ \sigma \mathrel{\sqsubset\!\!\!\!-} \alpha} \qquad \textit{e}\textsc{Abs-Left}\ \frac{(Q)\ \sigma_1 \mathrel{\sqsubset\!\!\!\!-}^{\sharp} \sigma_2}{(Q)\ \forall(\alpha \geq \sigma_1)\ \sigma' \mathrel{\sqsubset\!\!\!\!-}^{\sharp} \forall(\alpha \geq \sigma_2)\ \sigma'}$$

$$\textit{e}\textsc{Abs-SharpLeft}\ \frac{(Q)\ \sigma_1 \mathrel{\sqsubset\!\!\!\!-} \sigma_2 \qquad \sigma' \notin \mathscr{V}}{(Q)\ \forall(\alpha \Rightarrow \sigma_1)\ \sigma' \mathrel{\sqsubset\!\!\!\!-}^{\sharp} \forall(\alpha \Rightarrow \sigma_2)\ \sigma'} \qquad \textit{e}\textsc{Abs-SharpDrop}\ \frac{(Q)\ \sigma_1 \mathrel{\sqsubset\!\!\!\!-}^{\sharp} \sigma_2}{(Q)\ \sigma_1 \mathrel{\sqsubset\!\!\!\!-} \sigma_2}$$

Fig. 11. Type abstraction in *e*ML$^{\mathsf{F}}$.

$$\textit{e}\textsc{Ins-Abstract}\ \frac{(Q)\ \sigma_1 \mathrel{\sqsubset\!\!\!\!-} \sigma_2}{(Q)\ \sigma_1 \sqsubseteq \sigma_2} \qquad \textit{e}\textsc{Ins-Hyp}\ \frac{(\alpha \geq \sigma) \in Q}{(Q)\ \sigma \sqsubseteq \alpha} \qquad \textit{e}\textsc{Ins-Bot}\ (Q)\ \bot \sqsubseteq \sigma$$

$$\textit{e}\textsc{Ins-Rigid}\ (Q)\ \forall(\alpha \geq \rho)\ \sigma' \sqsubseteq \forall(\alpha \Rightarrow \rho)\ \sigma'$$

Fig. 12. Type-instance in *e*ML$^{\mathsf{F}}$.

part in $\leq$. To see this, take σ_{id} for σ and $\beta \to \alpha$ for τ. Writing σ' for $\forall(\alpha \Rightarrow \sigma_{\mathsf{id}})\ \forall(\beta \geq \sigma_{\mathsf{id}})\ \beta \to \alpha$, we would have $\sigma' \mathrel{\sqsubset\!\!\!\!-} \forall(\alpha \Rightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha$ in *e*ML$^{\mathsf{F}}$ while $\forall(\beta \geq \sigma_{\mathsf{id}})\ \beta \to \sigma_{\mathsf{id}} \equiv \sigma_{\mathsf{id}} \to \sigma_{\mathsf{id}}$ does not hold in *i*ML$^{\mathsf{F}}$. This is indeed incorrect, as it would allow the other direction $\forall(\alpha \Rightarrow \sigma_{\mathsf{id}})\ \alpha \to \alpha \mathrel{-\!\!\!\!\sqsupset} \sigma'$ to be used explicitly via an annotation, and so make $(\omega^{\dagger}\colon \sigma')\ \omega^{\dagger}$ well-typed, which is of course incorrect as its evaluation loops. (Perhaps, another more intuitive example using primitive integers is $(\omega^{\dagger}\colon \sigma')$ succ.)

Intuitively, a rigid binding behaves as a protection that prevents the underlying type from ever being instantiated and, as a consequence, allows it 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 $\mathrel{\sqsubset\!\!\!\!-}^{\sharp}$, called *protected abstraction*, that is recursively defined with the (unprotected) abstraction relation $\mathrel{\sqsubset\!\!\!\!-}$.

Definition 4.2.4 (*Abstraction*). The abstraction relation $\mathrel{\sqsubset\!\!\!\!-}$ and the protected abstraction relation $\mathrel{\sqsubset\!\!\!\!-}^{\sharp}$ are the smallest transitive relations satisfying the rules of Fig. 11 and Rule *e*Con-AllRight.

Rules may be read by first ignoring the difference between $\mathrel{\sqsubset\!\!\!\!-}$ and $\mathrel{\sqsubset\!\!\!\!-}^{\sharp}$, then realizing that the distinction only prevents uses of *e*Abs-Hyp in pathological contexts for the reason explained above.

In fact, an alternative presentation is to remove Rule *e*Equ-Inert from the equivalence, and instead add a rule *e*Abs-Inert

$$\textit{e}\textsc{Abs-Inert}\ \frac{\sigma \in \mathcal{I}_e}{(Q)\ \forall(\alpha \geq \sigma)\ \sigma' \mathrel{\sqsubset\!\!\!\!-}^{\sharp} \forall(\alpha \Rightarrow \sigma)\ \sigma'}$$

This only slightly weakens the equivalence relation (inert types can then only be changed from flexible to rigid bound and not conversely). It changes the set of types a program has, but does not change the set of typable programs. The advantage of this alternate presentation is a simplification of equivalence which then only deals with commutation and removal of useless binders and inlining of monomorphic nodes. It also seems more natural in the graphical presentation of ML$^{\mathsf{F}}$ [52]—as all other operations on inert types must remain part of abstraction. However, we prefer here to stay with the (slightly) larger equivalence relation.

Whereas the abstraction relation $\mathrel{\sqsubset\!\!\!\!-}$ can abstract a type scheme into a variable (see Rule *e*Abs-Hyp), the protected abstraction $\mathrel{\sqsubset\!\!\!\!-}^{\sharp}$ cannot do the same, except in degenerate cases.

Lemma 4.2.5. *If* $(Q)\ \sigma_1 \mathrel{\sqsubset\!\!\!\!-}^{\sharp} \sigma_2$ *and either* $\sigma_1 \in \mathcal{V}$ *or* $\sigma_2 \in \mathcal{V}$*, then* $(Q)\ \sigma_1 \equiv \sigma_2$.

The instance relation differs from the one of *i*ML$^{\mathsf{F}}$ in only two minor ways. First, it extends not only the equivalence but also the abstraction relation. Second, Rule *i*Ins-Subst, which would no longer be well-formed, has been replaced by *e*Ins-Rigid, introducing a rigid binding instead of performing the substitution in the conclusion. Other rules in Fig. 12 are directly taken from those of *i*ML$^{\mathsf{F}}$ (Fig. 7).

Definition 4.2.6 (*Instance*). The instance relation, written $\sqsubseteq$, is the smallest transitive and $\geq$-congruent relation satisfying the rules of Fig. 12.

Notice the inclusions $(\equiv) \subseteq (\sqsubseteq^{\sharp}) \subseteq (\sqsubseteq) \subseteq (\sqsubseteq)$.

The next lemma shows that instantiating (and a fortiori abstracting) a type τ has no effect up to equivalence, in contrast with abstraction of type schemes.

Lemma 4.2.7. *For any type τ and type scheme σ, if $(Q)\ \tau \sqsubseteq \sigma$ holds, then $(Q)\ \tau \equiv \sigma$. Besides, there exists a type τ' such that $(\emptyset)\ \sigma \equiv \tau'$.*

4.2.1. Soundness of instance and abstraction relations

The type soundness of *e*ML$^{\mathsf{F}}$ is shown below by a translation of well-typed *e*ML$^{\mathsf{F}}$ programs into well-typed *i*ML$^{\mathsf{F}}$ ones, 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 type schemes of $\mathcal{V}$ in a special way. 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 $\mathsf{head}(\sigma) = \beta$).

Definition 4.2.8 (*Type projection*). The projection of an *e*ML$^{\mathsf{F}}$ type into an *i*ML$^{\mathsf{F}}$ type is defined as follows:

$$\llbracket \tau \rrbracket \triangleq \tau \qquad \llbracket \forall(\alpha \Rightarrow \sigma_1)\ \sigma_2 \rrbracket \triangleq \llbracket \sigma_2 \rrbracket [\llbracket \sigma_1 \rrbracket/\alpha]$$

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

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 translated to $(\alpha \geq \llbracket \sigma_1 \rrbracket)$, unless it is a trivial one, in which case the corresponding substitution is performed.

An important property of the projection is, intuitively, that the projection of a type that is not itself a type variable never contains exposed type variables (defined in Section 3.3.6).

Lemma 4.2.9. *The set of exposed type variables of $\llbracket \sigma \rrbracket$ is included in the singleton $\{\mathsf{head}(\sigma)\}$.*

Whereas the projection of an *e*ML$^{\mathsf{F}}$ type is an *i*ML$^{\mathsf{F}}$ type, the projection of an *e*ML$^{\mathsf{F}}$ prefix Q is a pair composed of an *i*ML$^{\mathsf{F}}$ 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 4.2.10 (*Prefix projection*). The projection of a prefix Q is a pair (Q, θ), defined inductively as follows:

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

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

The following lemma states that type equivalence, type abstraction, and type instance relation are all preserved by projections into *i*ML$^{\mathsf{F}}$.

Lemma 4.2.11. *Let (Q', θ) be $\llbracket Q \rrbracket$. We have the following implications:*
(i) *If $(Q)\ \sigma_1 \equiv \sigma_2$, then $(Q')\ \theta \llbracket \sigma_1 \rrbracket \sqsubseteq \theta \llbracket \sigma_2 \rrbracket$.*
(ii) *If $(Q)\ \sigma_1 \sqsubseteq \sigma_2$, then $(Q')\ \theta \llbracket \sigma_1 \rrbracket \sqsubseteq \theta \llbracket \sigma_2 \rrbracket$.*
(iii) *If $(Q)\ \sigma_1 \sqsubseteq \sigma_2$, then $(Q')\ \theta \llbracket \sigma_1 \rrbracket \leq \theta \llbracket \sigma_2 \rrbracket$.*

(Proof in Appendix)

Properties 4.2.11.ii and 4.2.14.i show that the relations $\boxminus$ and $(\sqsubseteq \cup \sqsupseteq)^*$ are in correspondence. They were also used indirectly to show Property 4.2.11.iii. Only Property 4.2.11.iii is further used, namely to show the close correspondence between *i*ML$^{\mathsf{F}}$ and *e*ML$^{\mathsf{F}}$.

4.2.2. Completeness of instance and abstraction relations

We may conversely show that type-instance and type abstraction in *e*ML$^{\mathsf{F}}$ capture no more than type-instance and type equivalence in *i*ML$^{\mathsf{F}}$.

Var Fun App Gen Let

Inst
$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \sqsubseteq \sigma'}{(Q)\ \Gamma \vdash a : \sigma'}$$

Annot
$$\frac{\bar\alpha \subseteq \operatorname{dom}(Q) \qquad (Q)\ \Gamma \vdash a : \sigma' \qquad (Q)\ \sigma \sqsupseteq\!\!\!\!\sqsubset \sigma'}{(Q)\ \Gamma \vdash (a : \exists(\bar\alpha)\ \sigma) : \sigma}$$

Fig. 13. *e*ML$^\mathsf{F}$ typing rules.

Let us first introduce a translation from *i*ML$^\mathsf{F}$ types to *e*ML$^\mathsf{F}$ ones, written $\lceil\sigma\rceil$, defined inductively as follows:

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

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 $(\alpha_1 \Rightarrow \lceil\tau_1\rceil, \alpha_2 \Rightarrow \lceil\tau_2\rceil)$, since $\lceil\tau_1\rceil$ and $\lceil\tau_2\rceil$ are guaranteed to be ρ-types, but not necessarily monotypes. In case they *are* monotypes, the extra indirection is not problematic, as it can always be eliminated by type-equivalence in *e*ML$^\mathsf{F}$.

The translation of an *i*ML$^\mathsf{F}$ binding $(\alpha \geq \sigma)$ is a binding $(\alpha \diamond \lceil\sigma\rceil)$ that is rigid if and only if σ is equivalent to an inert type. Formally,

$$\lceil(\alpha \geq \sigma)\rceil = \begin{cases} (\alpha \Rightarrow \lceil\sigma\rceil) & \text{if there exists } \tau \in \mathcal{I}_i \text{ such that } (\emptyset)\ \sigma \boxminus \tau \\ (\alpha \geq \lceil\sigma\rceil) & \text{otherwise} \end{cases}$$

Checking if a type scheme σ is equivalent to an inert type can be performed quite easily, for example by taking normal forms—see [16] for details. The translation of a prefix Q is the pointwise translation of its bindings.

As a preliminary result, we show that inert substitution in *i*ML$^\mathsf{F}$ is captured by the symmetric closure of abstraction in *e*ML$^\mathsf{F}$:

Lemma 4.2.12. *Let σ be an iML$^\mathsf{F}$ type scheme and Q be an iML$^\mathsf{F}$ prefix such that $(\alpha \geq \sigma') \in Q$ with $(\emptyset)\ \lceil\sigma'\rceil\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \lceil\tau\rceil$ and $\tau \in \mathcal{I}_i$. Then, we have $(\lceil Q\rceil)\ \lceil\sigma\rceil\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \lceil\sigma[\tau/\alpha]\rceil$ in eML$^\mathsf{F}$.*

(Proof in Appendix)

Lemma 4.2.13. *Let τ be an iML$^\mathsf{F}$ type, σ be an iML$^\mathsf{F}$ type scheme, and Q an eML$^\mathsf{F}$ prefix. Then, we have $(Q)\ \forall(\alpha \Rightarrow \lceil\tau\rceil)\ \lceil\sigma\rceil\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \lceil\sigma[\tau/\alpha]\rceil$ in eML$^\mathsf{F}$.*

We may now show that type-instance and type equivalence in *i*ML$^\mathsf{F}$ map to type-instance and the symmetric closure of type abstraction in *e*ML$^\mathsf{F}$.

Lemma 4.2.14.
(i) *If $(Q)\ \sigma_1 \boxminus \sigma_2$, then $(\lceil Q\rceil)\ \lceil\sigma_1\rceil\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \lceil\sigma_2\rceil$.*
(ii) *If $(Q)\ \sigma_1 \leq \sigma_2$, then $(\lceil Q\rceil)\ \lceil\sigma_1\rceil\ (\sqsubseteq \cup \sqsupseteq)^*\ \lceil\sigma_2\rceil$.*

(Proof in Appendix)

4.3. Terms, typing rules, and type soundness

Terms of *e*ML$^\mathsf{F}$ are those of *i*ML$^\mathsf{F}$ extended with a new primitive construction for *type annotations* $(a : \exists(\bar\alpha)\ \sigma)$. Notice that the existential $\exists(\bar\alpha)\ \sigma$ is not an existential type, but a syntactic notation for introducing meta-variables $\bar\alpha$ standing for some types $\bar\sigma'$ appearing in σ to be inferred, as can be done in the language ML [39, p. 102; 12]. That is, the BNF grammar of expressions of *e*ML$^\mathsf{F}$ is:

$a ::= x \mid \lambda(x)\ a \mid a_1\ a_2 \mid \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$	Expressions of *i*ML$^\mathsf{F}$
$\quad (a : \exists(\bar\alpha)\ \sigma)$	Type annotations

The typing rules of *e*ML$^\mathsf{F}$, given in Fig. 13, include all rules from the generic system $\mathsf{G}(\mathcal{T}_e, \mathcal{S}_e, \mathcal{Q}_e, \sqsubseteq)$ 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 *e*ML$^\mathsf{F}$. The existentially quantified variables $\bar\alpha$ in annotations are used 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 $(Q)\ \sigma \mathrel{\sqsubseteq\!\!\!\!\!-}\ \sigma'$ allows to reveal σ, but no more. In particular, the bounds assigned to $\bar{\alpha}$ are shared between σ' and σ, which prevents from revealing 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 the annotation σ instead of $\exists\,(\emptyset)\ \sigma$. We so recover the simplified rule given in the introduction (Section 2.3.3). 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 $\lambda(x)\ \mathsf{let}\ x = (x : \sigma)\ \mathsf{in}\ a$. Rebinding x to its annotated version $(x : \sigma)$ avoids repeating the annotation on each occurrence 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:

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

In practice, most uses of annotations are actually in abstractions. The reason not to make annotated abstraction the primitive form and the annotations the derived form is that annotations 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$^{\mathsf{F}}$, all annotations can be treated as such. We could restrict *e*ML$^{\mathsf{F}}$ to F-like annotations. However, because types are stratified in Shallow ML$^{\mathsf{F}}$, we would then not reach all type annotations and *e*ML$^{\mathsf{F}}$ would loose its close correspondence with *i*ML$^{\mathsf{F}}$.

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 would be written $\rho \to \rho$ in *i*ML$^{\mathsf{F}}$. 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 also 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.

$$\textsc{Gen}\ \dfrac{\textsc{Fun}^{\star}\ \dfrac{\textsc{Inst}\ \dfrac{\textsc{Var}\ \dfrac{}{(\alpha \Rightarrow \rho)\ x : \rho \vdash x : \rho} \qquad \dfrac{(\alpha \Rightarrow \rho)\ \rho \mathrel{\sqsubseteq\!\!\!\!\!-}\ \alpha}{(\alpha \Rightarrow \rho)\ \rho \sqsubseteq \alpha}}{(\alpha \Rightarrow \rho)\ x : \rho \vdash x : \alpha}}{(\alpha \Rightarrow \rho)\ \emptyset \vdash \lambda(x : \rho)\ x : \forall(\beta \Rightarrow \rho)\ \beta \to \alpha}}{(\emptyset)\ \emptyset \vdash \lambda(x : \rho)\ x : \forall(\alpha \Rightarrow \rho)\ \forall(\beta \Rightarrow \rho)\ \beta \to \alpha}$$

Here, the variable x has a polymorphic F-like type ρ, which is directly accessible with Rule Var. In the previous derivation, x had only a type α, which was bound to ρ in the prefix, hence not directly available. 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 (Section 4.4) 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 (rigidly bound to ρ in the prefix). This is to prepare for rule Fun$^{\bigstar}$ that 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 claimed on the result—namely just as much as promised on the argument.

Derivable rules. Rules App* and UnGen* (defined in Section 3.4) remains admissible in *e*ML$^{\mathsf{F}}$—with types and type schemes now taken in *e*ML$^{\mathsf{F}}$, of course.

Recursion. Viewing recursion as a fixed point combinator means that the identifier f of $\mathsf{let\ rec}\ f = \lambda(x)\ a_1$ in a_2 is treated as the argument of a function and must have a ρ type. Then, f needs an annotation if it is used polymorphically in the body of a_1. This treatment of recursion allows for polymorphic recursion but with explicit type annotations.

Using a primitive construct for recursion would allow f to be assigned a type scheme instead of a type, via an explicit annotation. Omitting the annotation would amount to inferring polymorphic recursion, which would be undecidable in ML$^{\mathsf{F}}$, as it is already undecidable in ML.

Expressiveness. We show that *e*ML$^{\mathsf{F}}$ and *i*ML$^{\mathsf{F}}$ are in close correspondence, and thus exactly as expressive. Dropping type annotations directly maps *e*ML$^{\mathsf{F}}$ programs to *i*ML$^{\mathsf{F}}$ ones.

Theorem 3. *Assume eML$^{\mathsf{F}}$ :: $(Q)\ \Gamma \vdash a : \sigma$ holds. Let a' be the erasure of a and (Q', θ) be $\llbracket Q \rrbracket$. Then, iML$^{\mathsf{F}}$ :: $(Q')\ \theta(\Gamma) \vdash a' : \theta(\llbracket \sigma \rrbracket)$ holds.*

(Proof in Appendix)

Conversely, all *i*ML$^{\mathsf{F}}$ programs can always be mapped to *e*ML$^{\mathsf{F}}$ ones by inserting explicit type annotations.

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

(Proof in Appendix)

Noticeably, the translation of an *i*ML$^{\mathsf{F}}$ 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 *e*ML$^{\mathsf{F}}$ is as safe as *i*ML$^{\mathsf{F}}$, and Theorem 2, which states type soundness of *i*ML$^{\mathsf{F}}$.

4.4. Translating System F into eML$^{\mathsf{F}}$

The composition of Theorems 2 and 4 states that there is a mapping of System-F terms to *e*ML$^{\mathsf{F}}$ 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 proofs 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 *e*ML$^{\mathsf{F}}$ 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 *e*ML$^{\mathsf{F}}$ that keep (actually translate) the type annotations on λ-abstractions, and throws away all type abstractions and type applications. Therefore, it returns an *e*ML$^{\mathsf{F}}$ 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 explain an important design choice of this translation, informally. Rigid bindings are interpreted as substitutions (Definition 4.2.8). 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 *e*ML$^{\mathsf{F}}$, namely (1) and (2). Observe that (2) is more general than (1) in *e*ML$^{\mathsf{F}}$ (the latter is an instance of the former). Taking (2) is the approach chosen in Section 4.2.2 to translate *i*ML$^{\mathsf{F}}$ 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 4.4.3 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 [16]. We present the first approach here for the sake of simplicity. Despite this choice, this section remains the most technical part of the paper. It happens that proving the soundness of the translation from System F to *e*ML$^{\mathsf{F}}$ 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 *e*ML$^{\mathsf{F}}$, as a result of the inlining of rigid bindings in System F. Consequently, *e*ML$^{\mathsf{F}}$ 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 *e*ML$^{\mathsf{F}}$ typing rules. The purpose of the instrumentation is exactly to reconstruct and trace this information in a safe way.

As the rest of the paper does not depend on the technical details of this section, the user may skip them and directly jump to Theorems 6 and 7 at the end of this section.

4.4.1. Auxiliary definitions.

We first define a few operators that are used to translate F-types into *e*ML$^{\mathsf{F}}$-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 4.4.1. 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 (as, for instance, the one of [16]). 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 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 4.4.2. 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 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 *e*ML$^{\mathsf{F}}$ types.

The use of an auxiliary relation instead of simply a function is to capture non-determinism that results from the choice of fresh variables during the translation. Lemma 4.4.6 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 4.4.3. 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:

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

$$\frac{(Q\alpha)\ \langle\!\langle t \rangle\!\rangle : (Q_1 \alpha Q_2, \beta)}{(Q)\ \langle\!\langle \forall(\alpha)\ t \rangle\!\rangle : Q_1 \otimes \forall(\alpha Q_2)\ \beta}$$

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 4.4.4 (*Translation of types and prefixes*). The translation of an F-type t, written $\langle\langle t\rangle\rangle$ is the set of all $e\mathsf{ML}^\mathsf{F}$ 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 the fact that Q' is itself rigid.

Note that σ_{id} is both a type of System F and of $e\mathsf{ML}^\mathsf{F}$. We have $(\emptyset)\ \langle\langle\sigma_{\mathsf{id}}\rangle\rangle : (\beta \Rightarrow \sigma_{\mathsf{id}}), \beta$. We then have $(\beta \Rightarrow \sigma_{\mathsf{id}})\ \langle\langle\sigma_{\mathsf{id}} \rightarrow \sigma_{\mathsf{id}}\rangle\rangle : (\beta \Rightarrow \sigma_{\mathsf{id}})\ (\gamma \Rightarrow \beta \rightarrow \beta), \gamma$.

We shall see below that all type schemes in the translation of an F-type are in fact equivalent (Corollary 4.4.10), and similarly for the translation of typing environments. We call *rigid* an $e\mathsf{ML}^\mathsf{F}$ type scheme that is in the translation of an F-type.

Auxiliary results. We now establish several properties about the translation algorithm that will be used to prove the main result of this section, Theorem 5, from which Theorems 6 and 7—two variants of the same result—immediately follow. 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 4.4.5 states that inputs and outputs can be consistently renamed.
- Lemma 4.4.6 characterizes non-determinism: the outputs can be renamed while leaving the input unchanged provided that some capture-avoiding side conditions hold.
- Lemma 4.4.7 states a form of idempotence: (i) the input may be replaced by the output (leaving the output unchanged); (ii) once the input and output are equal, they may be extended simultaneously.
- Lemma 4.4.9 states that the relation is closed by equivalence.
- Lemma 4.4.12 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 4.4.13 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 4.4.5 (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 4.4.6 (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 4.4.7 (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 in Appendix)

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:

*e*SHA-FREE
$$\frac{\alpha \notin I \cup \mathsf{dom}(Q) \cup \mathsf{ftv}(Q) \qquad \sigma \notin \mathsf{bnds}(Q)}{Q \equiv^I (Q, \alpha \Rightarrow \sigma)}$$

*e*SHA-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 *e*SHA-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 *e*SHA-COMM allows the commutation of independent binders.

Unsurprisingly, two types built with equivalent prefixes are equivalent.

$$\textit{e}\text{TSH-TYPES}\quad \frac{(Q)\ Q_1 \Rrightarrow_{\phi} Q_2}{(Q)\ \forall(Q_1)\ \tau \Rrightarrow \forall(Q_2)\ \phi(\tau)} \qquad \textit{e}\text{TSH-EMPTY}\quad (Q)\ \emptyset \Rrightarrow_{\mathsf{id}} \emptyset \qquad \textit{e}\text{TSH-FLEX}\quad \frac{(Q\alpha)\ Q_1 \Rrightarrow_{\phi} Q_2}{(Q)\ \alpha\, Q_1 \Rrightarrow_{\phi} \alpha Q_2}$$

$$\textit{e}\text{TSH-SUBST}\quad \frac{(Q)\ \sigma \Rrightarrow \sigma' \qquad (\alpha' \Rightarrow \sigma') \in Q \qquad (Q)\ Q_1[\alpha'/\alpha] \Rrightarrow_{\phi} Q_2}{(Q)\ (\alpha \Rightarrow \sigma, Q_1) \Rrightarrow_{\phi\circ[\alpha'/\alpha]} Q_2}$$

$$\textit{e}\text{TSH-CONTEXT}\quad \frac{(Q)\ \sigma \Rrightarrow \sigma' \qquad \sigma' \notin \mathsf{bnds}(Q) \qquad (Q, \alpha \Rightarrow \sigma')\ Q_1 \Rrightarrow_{\phi} Q_2}{(Q)\ (\alpha \Rightarrow \sigma, Q_1) \Rrightarrow_{\phi} (\alpha \Rightarrow \sigma', Q_2)}$$

Fig. 14. Translation of types and prefixes.

Lemma 4.4.8. *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, yields equivalent prefixes Q_1' and Q_2'.

$$\begin{tikzcd} Q_1 \arrow[r, leftrightarrow, "\equiv"] \arrow[d, "(Q_1)\langle\langle t\rangle\rangle"'] & Q_2 \arrow[d, "(Q_2)\langle\langle t\rangle\rangle"] \\ Q_1' \arrow[r, leftrightarrow, dotted, "\equiv"'] & Q_2' \end{tikzcd}$$

Lemma 4.4.9 (Equiv). *Let* I *be* $\mathsf{ftv}(t)$. *We assume that* Q_1 *is 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 in Appendix)

Corollary 4.4.10. *If* $\sigma_1, \sigma_2 \in \langle\langle t\rangle\rangle$, *then* $(Q)\ \sigma_1 \equiv \sigma_2$ *under any suitable* Q.

(Proof in Appendix)

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 as a set of (deterministic) inference rules given in Fig. 14. The algorithm is written $(Q)\ \sigma_1 \Rrightarrow \sigma_2$ for types, where Q and σ_1 are inputs and σ_2 in an output (Rule *e*TSH-TYPES). It is written $(Q)\ Q_1 \Rrightarrow_{\phi} Q_2$ for prefixes, where Q and Q_1 are inputs and ϕ and Q_2 are outputs. 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. Rules *e*TSH-EMPTY, *e*TSH-FLEX, and *e*TSH-CONTEXT are context rules that do not perform any sharing. On the contrary, Rule *e*TSH-SUBST detects and shares two similar bindings.

The next lemma describes properties of this algorithm.

Lemma 4.4.11.
(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 in Appendix)

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).

$$\begin{tikzcd}
Q_1 \arrow[r, "\psi(Q_1)\Rrightarrow"] \arrow[d, "\langle\!\langle t \rangle\!\rangle"'] & Q_2 \arrow[d, dotted, "\langle\!\langle \psi(t) \rangle\!\rangle"] \\
Q_1' \arrow[r, dotted, "\psi(Q_1')\Rrightarrow"'] & Q_2'
\end{tikzcd}$$

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 4.4.12. *Let ψ be the substitution $[\alpha/\beta]$. We assume that the α is bound 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:*

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

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

$$\text{P-III}\quad \frac{(\emptyset)\ \langle\!\langle t' \rangle\!\rangle : (Q', \alpha) \qquad (Q')\ \langle\!\langle \psi(t) \rangle\!\rangle : (Q_2', \alpha_2)}{(Q')\ \langle\!\langle t[t'/\beta] \rangle\!\rangle : (Q_2', \alpha_2)}$$

(Proof in Appendix)

In the following lemma, we show that an unconstrained binding (here β) may be removed from the input prefix of the translation and 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 *e*Sha-Comm. Noticeably, $Q \approx Q'$ implies $Q \equiv^{I} Q'$ for any I.

Lemma 4.4.13. *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 in Appendix)

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 4.4.14. *The following rules are derivable:*

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

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

(Proof in Appendix)

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 $e\mathsf{ML}^\mathsf{F}$ program. Since every $e\mathsf{ML}^\mathsf{F}$ program admits a principal type (this result is not shown in this paper, but has been shown for Full ML^F as well as a small variant of $e\mathsf{ML}^\mathsf{F}$ by [16]), this type captures all the possible type abstractions and type applications of the term.

While the encoding of System F terms into $e\mathsf{ML}^\mathsf{F}$ uses an annotation on every λ-abstraction, these annotations could in fact be omitted when the argument is not used polymorphically, as shown below by the embedding of ML into $e\mathsf{ML}^\mathsf{F}$.

4.5. Embedding ML into eML^F

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

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

The instance relation for $\leq_{\mathsf{ML}}$ has been defined in the introduction (Section 2.1.3). 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 $e\mathsf{ML}^\mathsf{F}$ shows that $\leq_{\mathsf{ML}}$ is a subrelation of $\sqsubseteq$.

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

ML terms are in eML^F. The typing rules of ML are exactly those of $e\mathsf{ML}^\mathsf{F}$, 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 $e\mathsf{ML}^\mathsf{F}$ (which is not a derivation of the most principal type in $e\mathsf{ML}^\mathsf{F}$, in general).

Theorem 8. *Any term typable in ML is also typable in eML^F.*

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

Unannotated eML^F terms are in ML. We prove this by translating $e\mathsf{ML}^\mathsf{F}$ 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 4.5.1). The result which contains only flexible types, is still a valid derivation (Lemma 4.5.3). Last, all quantifiers are extruded to the outermost level. The final derivation is still correct and it is a derivation in ML (Lemma 4.5.7).

Definition 4.5.1. $e\mathsf{ML}^\mathsf{F}$ 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 binding in any type 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 $e\mathsf{ML}^\mathsf{F}$ 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 4.5.2.
(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* $(Q)\ \sigma_1 \sqsubseteq\!\!\!\!\!-\ \sigma_2$, *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 in Appendix)

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

Lemma 4.5.3. *For any unannotated term a, if* $(Q)\ \Gamma \vdash a : \sigma$ *holds in eML^F, then so does* $\mathsf{flex}\ ((Q)\ \Gamma \vdash a : \sigma)$.

(Proof in Appendix)

We recall a standard result of ML.

Lemma 4.5.4. *If we have* $\forall(\bar\alpha)\ \tau_1 \leq_{ML} \forall(\bar\beta)\ \tau_2$, *then for any* σ *such that* $\mathsf{ftv}(\sigma) \# \bar\alpha \cup \bar\beta$, *we have* $\forall(\bar\alpha)\ \sigma[\tau_1/\gamma] \leq_{ML} \forall(\bar\beta)\ \sigma[\tau_2/\gamma]$

We now show how a flexible *e*ML$^{\mathsf{F}}$ type is transformed into an ML type by extrusion of quantifiers. We first transform prefixes. A flexible prefix is transformed into the pair of a set of quantifiers and a monotype substitution.

Definition 4.5.5. 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 (we overload the notation used for System F, which should not raise any ambiguity):

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

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 4.5.6.
(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* $\theta(\langle\!\langle \sigma_1\rangle\!\rangle) \leq_{ML} \theta(\langle\!\langle \sigma_2\rangle\!\rangle)$ *holds*.

(Proof in Appendix)

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

Lemma 4.5.7. *If there exists a flexible derivation of* $(Q)\ \Gamma \vdash a : \sigma$ *in eML*$^{\mathsf{F}}$ *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 in Appendix)

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

(Proof in Appendix)

This shows why terms that are insufficiently annotated are rejected. For example, ω is not typable in *e*ML$^{\mathsf{F}}$, as we claimed earlier. By contrast, the annotated version $\omega^\dagger$ is, as explained in Section 4.3.

Thus, although the full power of System F is available in *e*ML$^{\mathsf{F}}$, 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 *e*ML$^{\mathsf{F}}$ 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).

Although our guideline to put *annotations on function arguments that are used polymorphically* is intuitive and accurate, based on our experience, it lacks a formal definition. Unfortunately, it is difficult to find an exact characterization that does not paraphrase the typing rules of *e*ML$^{\mathsf{F}}$, since typechecking an expression $\lambda(x)\ a$ may combine all the expressive features of ML$^{\mathsf{F}}$ while typechecking its body *a*.

Interestingly, there are approximations of this criteria based on the translation of System F into *i*ML$^{\mathsf{F}}$ and thus restricted to programs that are typable in System F. Of course, such criteria are only meaningful if we restrict to type annotations on parameters of abstractions, that is, if we replace rules Fun and Annot by Fun*.

Unsurprisingly, if $\lambda(x : t)\ a$ is typable in System F and *t* is a monotype, then $\lambda(x)\ a$ is typable in ML$^{\mathsf{F}}$ without an annotation on x. This implies, for example, that $(\lambda(x)\ x)\ \omega^\dagger$ needs no annotation on x.

This simple criterion is weak, but can be combined with type preserving transformations in ML$^{\mathsf{F}}$ to show that more involved programs, such as $\lambda(x)\ \mathsf{choose}\ \omega^\dagger\ x$, need no annotation either—without explicitly referring to the typing rules of ML$^{\mathsf{F}}$, but to those of System-F.

4.6. Type-preserving program transformations

In a type-inference system, it is important that type inference does not interfere with the programming style. Therefore, programs must be stable under some small local transformations, such as the permutation of the order of arguments, the introduction of auxiliary bindings, etc. Such common transformations should therefore be type preserving. We list a few useful type-preserving transformations below and use them to better explain how functions can require some of their parameters to be polymorphic but still use them parametrically, in which case an annotation is not required in *e*ML$^{\mathsf{F}}$.

We write $a \subseteq a'$ to mean that all typings of *a* are typings of a', that is, for all prefix Q, for all typing context Γ and all type τ, the judgment $(Q)\ \Gamma \vdash a : \tau$ implies $(Q)\ \Gamma \vdash a' : \tau$.

Invariance of typings by let-conversion. The invariance of typings by let-conversion is a particular case of subject reduction—for a liberal reduction strategy that allows reduction of non-evaluated expressions. We have $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 \subseteq a_2[a_1/x]$. The proof of this property in $\mathsf{ML^F}$ is the same as in ML.

The converse property, $a_2[a_1/x] \subseteq \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ also holds in $\mathsf{ML^F}$, whenever x appears free in a_2. In $e\mathsf{ML^F}$, this result easily follows from the existence of principal types, which is however not shown in this paper. Principal types allow to factor out the types of the different occurrences of a_1 in $a_2[a_1/x]$ as several instantiations of the principal type of a_1. This property also holds in ML and in Church-style System F, both of which have principal types—by design in Church-style System F since typing derivations are unique. However, this property does not hold in all systems with principal types [29,19,48].

Preservation of typings by β-reduction. Although, we do not prove subject reduction for $e\mathsf{ML^F}$ in this paper, it has been proved in the case of Full $e\mathsf{ML^F}$ in [17,16], provided new type annotations may be introduced during reduction—but only to replace existing type annotations. For instance, no propagation of annotations is necessary for applications of unannotated abstractions, and we have $(\lambda(x)\ a)\ a' \subseteq a[a'/x]$[11]. In particular, if $(\lambda(x')\ \lambda(x)\ a)\ a'$ needs no annotations on x and x', then $\lambda(x)\ (a[a'/x'])$ needs no annotation on x.

Invariance of typings by abstraction of applications. Typing of applications is "first class" in $e\mathsf{ML^F}$, in the sense that the primitive application of $a_1\ a_2$ can be redefined as the application of apply to arguments a_1 and a_2, where apply is the abstraction of the application $\lambda(x)\ \lambda(y)\ x\ y$. Formally, we have $a_1\ a_2 \subseteq \mathsf{apply}\ \ a_1\ a_2$, and conversely, $\mathsf{apply}\ \ a_1\ a_2 \subseteq a_1\ a_2$.

<u>Proof</u>: The inverse inclusion follows from the preservation of typings by β-reduction. We thus only need to check the direct inclusion. Assume $(Q)\ \Gamma \vdash a_1\ a_2 : \tau$, a derivation of this judgment will end with a sequence (**1**) of rules Inst and Gen eventually preceded by a rule App of the form:

$$\textsc{App}\ \frac{(QQ')\ \Gamma \vdash a_1 : \tau_2 \to \tau_1\ (\mathbf{2}) \qquad (QQ')\ \Gamma \vdash a_2 : \tau_2\ (\mathbf{3})}{(QQ')\ \Gamma \vdash a_1\ a_2 : \tau_1}$$

We can build the following derivation (where anchors must be replaced by the premises introduced above):

$$\textsc{App}\ \frac{\textsc{App}\ \dfrac{(QQ')\ \Gamma \vdash \mathsf{apply} : (\tau_2 \to \tau_1) \to \tau_2 \to \tau_1 \qquad (2)}{(QQ')\ \Gamma \vdash \mathsf{apply}\ a_1 : \tau_2 \to \tau_1} \qquad (3)}{(QQ')\ \Gamma \vdash \mathsf{apply}\ a_1\ a_2 : \tau_1}$$

which, when followed by the sequence (1), ends with $(Q)\ \Gamma \vdash \mathsf{apply}\ a_1\ a_2 : \tau$.

As the order of arguments is insignificant in $e\mathsf{ML^F}$, we also have $a_1\ a_2 \subseteq \mathsf{revapply}\ \ a_1\ a_2$, and conversely, where revapply stands for the application receiving its argument in reverse order $\lambda(x)\ \lambda(y)\ y\ x$.

Preservation of typings by η-expansion. Of course, this may only hold for functional expressions. Formally, if $(Q)\ \Gamma \vdash a : \forall(Q')\ \tau \to \tau'$ and x is not free in a, then $(Q)\ \Gamma \vdash \lambda(x)\ a\ x : \forall(Q')\ \tau \to \tau'$.

<u>Proof</u>: We may first show the lemma for monotypes, i.e., when Q' is empty. Assume $(Q)\ \Gamma \vdash a : \tau \to \tau'$ and x is not free in a. By weakening (a standard property that could be shown by induction on the height of the derivation), we have $(Q)\ \Gamma, x : \tau \vdash a : \tau \to \tau'$ (**1**). We thus have the following typing derivation:

$$\textsc{App}\ \frac{\textsc{Fun}\ \dfrac{(1) \qquad (Q)\ \Gamma, x : \tau \vdash x : \tau}{(Q)\ \Gamma, x : \tau \vdash a\ x : \tau'}}{(Q)\ \Gamma \vdash \lambda(x)\ a\ x : \tau \to \tau'}$$

The general case reduces to the monotype case, by applying a sequence of rules UnGen* beforehand and a sequence of rules Gen afterward.

Parametric uses of polymorphism. Our guideline for $e\mathsf{ML^F}$ *annotate only parameters of functions that are used polymorphically* can also be explained by its corollary: parameters of functions that are *used* monomorphically need not be annotated. These include parameters of function that are monomorphic, as we have seen above with, for example, x in $(\lambda(x)\ x)\ \omega^{\dagger}$, but also parameters with polymorphic types that are used parametrically in the body of the function, such as x in expressions $\lambda(x)\ x\ \omega^{\dagger}$, $\lambda(x)\ \omega^{\dagger}\ x$, or $\lambda(x)\ \mathsf{choose}\ \omega^{\dagger}\ x$. Any typing in System F of these expressions will require a polymorphic type for x.

Still, we can show that x is used parametrically by exhibiting in each case a type preserving transformations in which x can be typed monomorphically.

[11] Formally, this result has only been shown in Full $e\mathsf{ML^F}$ for closed expressions.

For example, $\lambda(x)$ choose $\omega^\dagger$ x is the reduct of $(\lambda(y)\ \lambda(x)$ choose $y\ x)\ \omega^\dagger$ where both x and y are monomorphic—as can easily be seen on a typing derivation in System F—and thus needs no annotation. Thus, by the subject reduction property, x needs no annotation in the original term.

Similarly, $\lambda(x)\ x\ \omega^\dagger$ is the reduct of $(\lambda(y)\ \lambda(x)\ x\ y)\ \omega^\dagger$, where again x may be assigned a monomorphic type in System F. For $\lambda(x)\ \omega^\dagger\ x$, one needs not even refer to the typing in System F, as it is also the η-expansion of $\omega^\dagger$ and is thus typable in $e\mathsf{ML^F}$, since $\omega^\dagger$ is.

By contrast $\lambda(x)\ x\ x$ cannot be typed without an annotation on x. Indeed, all types of this expression in System F must assign x a type of the form $\forall(\alpha)\ t$ where α occurs on the leftmost part of t. In this example, x is explicitly used at two incompatible types. However, x may also be indirectly required to be polymorphic, as in $\lambda(x)$ let $y = x$ in $y\ y$. This programs becomes $\lambda(x)\ (\lambda(y)\ y\ y)\ x$ after let-expansion, which, as we already know, needs an annotation, e.g., σ_{id} on y in $\mathsf{ML^F}$. This annotation is transferred during β-reduction to the annotation σ_{id} on x in the resulting term. So it is unsurprising that the original program $\lambda(x)$ let $y = x$ in $y\ y$ also needs an annotation, e.g., σ_{id}, on its parameter x.

We have identified some typical parametric uses of polymorphic parameters for which no annotations are required and, conversely, some polymorphic uses of parameters that require type annotations. Interestingly, we could easily classify these examples without directly referring to the typing rules of $e\mathsf{ML^F}$.

There are also examples that combine several features of $\mathsf{ML^F}$ in sequence, for which typechecking is harder—but still possible—to predict. Consider the expression $\lambda(x)$ let $y = \omega^\dagger\ x$ in $y\ y$ (**1**). We may abstract $\omega^\dagger$ away in $(\lambda(z)\ \lambda(x)$ let $y = z\ x$ in $y\ y)\ \omega^\dagger$. The expression $(\lambda(z)\ \lambda(x)$ let $y = z\ x$ in $y\ y)$ is not typable in ML and thus z or x needs a type annotation, but we cannot tell whether both of them needs one. It happens that a single annotation $\exists(\alpha)\ \alpha \to \sigma_{\mathsf{id}}$ on z that specifies the codomain of z is sufficient. Then, x needs not an annotation. We can explain this on the original form as follows: in the type of $\omega^\dagger$ the domain and the codomain share no type variable, hence the type of $\omega^\dagger\ x$ does not depend on the type of x (as long as $\omega^\dagger\ x$ is typable), and can therefore be generalized in y and instantiated independently for the two occurrences of y in $y\ y$. Thus, the expression (1) is typable in $\mathsf{ML^F}$.

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 (Section 5.1), or encapsulating first-class polymorphic values within first-order ML types (Section 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 [50] and for some of its variants.

Type containment. In the late 80s, Mitchell noticed that System F might not be the "right" system for studying type inference [26]. He introduced the closure of System F by η-conversion, known as F^η, and showed that well-typedness in F^η could be obtained by replacing the instance relation $\le_{\mathsf{F}}$ by a larger relation $\le_\eta$, called *type containment* (see Section 2.1.3). 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 [51].

Our treatment of type annotations as type-revealing primitives resembles the use of retyping functions. Moreover, $\le_\eta$ and our type-instance relation $\le$ 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 (eg., contravariantly on the left-hand side of arrows and covariantly anywhere else). By contrast, our form of 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 only allow type instantiation at some occurrences and prevent it via rigid bindings anywhere else. As a result, the two relations are incomparable. The resemblance between type containment and $\mathsf{ML^F}$ is only superficial.

Polymorphic subtyping System $\mathsf{F}_{<:}$ is another extension of System F with a richer instance relation $<:$ (see its definition in Section 2.1.1). In $\mathsf{F}_{<:}$ as in $\mathsf{ML^F}$, each type variable is also given a bound. However, it is an upper bound in $\mathsf{F}_{<:}$ while it is a lower bound in $\mathsf{ML^F}$. As for type-containment, the subtyping relation $<:$ is structural, which makes a huge difference with our instance relation. Type inference for $\mathsf{F}_{<:}$ is undecidable. Even type checking is undecidable for some variants of the subtyping relation $<:$ [34].

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 [31]. Despite our opposite choice, that is *not* to support second-order unification, there are at least two comparisons to be made. First, 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. Second, 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^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 [5], 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 [24]. 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^F prefixes are universally quantified. Besides, ML^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 [2,3,46] and stated undecidability results for some particular variants. For instance, Boehm [2] and Pfenning [32] 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 the parameter of which is used polymorphically. This is precisely what we avoid in ML^F: all polymorphic λ-abstraction must be annotated, whereas type abstractions and type applications are inferred.

As another example, Schubert [46] considers *sequent decision problems* for System F in both Curry style and Church style. They, in fact, correspond to type inference problems in System F, as already studied by Wells [50], and are known to be undecidable. An inverse typing problem in Church-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-counter 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 [14], called Λ_2, and rank-2 intersection types [11], called I_2 actually type the same programs. They have been generalized to even-rank polymorphic types and odd-rank intersection types [10]. 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 [4]. 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 [3,36] 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 [9].

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[12]. Principal types are ensured by finding a "best

[12] 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.

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, to our knowledge, local type inference and its refinement described in the next paragraph are still 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 [28] 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.

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 [16]. 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 been shown unsatisfactory, even for its rather limited predicative fragment [40]. Stratified type inference has been applied more successfully for performing type inference in ML in the presence of Guarded Abstract Data Types (GADT) [38].

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 have now been in use in OCaml or Haskell for several years.

First-class polymorphism in FX. The language FX [29] seems to be a precursor of all extensions of ML with first-class polymorphism. Although its expressiveness is somewhat limited—it approximately fits between *boxed polymorphism* and Poly-ML described below—FX already contained several of the ideas reused in some later works. Roughly, FX allows ML implicit topmost quantifiers as in ML type schemes and explicit first-class quantifiers as in System F, plus constructions to coerce between the two. By definition, it supersedes both ML and System F, hence its expressiveness. However, expressiveness is achieved by relying more on explicit than on implicit polymorphism. In particular, an important restriction of FX, which makes its inference mechanism less expressive than the one of Poly-ML, is that implicit type instantiation remains predicative. That is, implicit quantifiers can only be instantiated by simple types—types that do not contain any quantifier. To recover impredicativity, implicit polymorphism may (and often need to) be converted to explicit polymorphism. Moreover, the type system specification requires typing derivations to be principal, disallowing any form of weakening, whenever first-class polymorphism is being used. Furthermore, implicit quantifiers do not commute and are instead inserted in some fixed left-to-right order so that they can later be unambiguously converted to explicit quantifiers the order of which matters. As a result, although the user need not always write quantifiers explicitly, he must still usually think in terms of explicit polymorphism.

Boxed polymorphism. This 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 [15; 41]. Typically, preliminary type definitions are made for all polymorphic types that appear in the program. For instance, the following program defines three versions of ω and applies them to id:

type sid = Sid of $\forall(\alpha)\ \alpha \to \alpha$	
let id = Sid $(\lambda(x)\ x)$	: sid
let ω_1= $\lambda(x)$ let Sid $z = x$ in $z\ z$	: $\forall(\beta)$ sid $\to (\beta \to \beta)$
let ω_2= $\lambda(x)$ let Sid $z = x$ in $z\ x$	: sid $\to$ sid
let ω_3= $\lambda(x)\ \omega_2\ \ x$	: sid $\to$ sid
$(\omega_1$ id, ω_2 id, ω_3 id)	: $\forall(\beta)\ (\beta \to \beta) \times$ sid $\times$ sid

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 ω_1. Notice the difference between ω_1 and ω_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 [27]. The fifth line shows that an η-expansion need no type annotation even when the argument is polymorphic.

Poly-ML [6]. This is an improvement over boxed polymorphism 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 $\forall(\alpha)\ \alpha \to \alpha$, and boxed polymorphism is represented inside square brackets):

let id $= [\lambda(x)\ x : \sigma_{\text{id}}]$	:	$[\sigma_{\text{id}}]$
let $\omega_1 = \lambda(x : [\sigma_{\text{id}}])\ \langle x\rangle\ \langle x\rangle$	:	$\forall(\beta)\ [\sigma_{\text{id}}] \to (\beta \to \beta)$
let $\omega_2 = \lambda(x : [\sigma_{\text{id}}])\ \langle x\rangle\ x$	:	$[\sigma_{\text{id}}] \to [\sigma_{\text{id}}]$
let $\omega_3 = \lambda(x)\ \omega_2\ \ x$	:	$[\sigma_{\text{id}}] \to [\sigma_{\text{id}}]$
$(\omega_1$ id, ω_2 id, ω_3 id$)$	:	$\forall(\beta)\ (\beta \to \beta) \times [\sigma_{\text{id}}] \times [\sigma_{\text{id}}]$
$\langle$id$\rangle$	:	$\forall(\beta)\ (\beta \to \beta)$

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 ω_1 as $\lambda(x : [\sigma_{\text{id}}])$ let $z = \langle x\rangle$ in $z\ z$, so as to avoid repeating the projection, much as for the treatment of type annotations in eML$^{\mathsf{F}}$. (But this is unsurprising, since eML$^{\mathsf{F}}$ was much inspired by Poly-ML.) The fourth line shows that η-expansion need no type annotation even when the argument is polymorphic.

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 $[\lambda(x)\ x : \forall(\alpha)\ (\alpha \to \alpha) \to (\alpha \to \alpha)]$.

Still, when using core Poly-ML to typecheck objects, every polymorphic-method invocation must be explicitly marked as an instantiation site. To avoid this burden, an extension that has been proposed (and now in use in OCaml) considers that every method invocation is implicitly a possible instantiation site. However, this extension requires typing derivations to be principal so as to be non ambiguous—and thus well-defined—in a pathological case. A similar restriction was already used in FX, but in a more crucial way for the main cases.

Boxy types [47]. Going one step-further than Poly-ML, they remove the “coercion box” at 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$^{\mathsf{F}}$, precisely. As they (arbitrarily) privilege propagation of type information from the function type to the argument type, they can type examples where ML$^{\mathsf{F}}$ would require an annotation and thus fail. Conversely, there are many examples that ML$^{\mathsf{F}}$ can type and that boxy types cannot—for some much deeper reason. In particular, if $a_1\ a_2$ is typable, then app $a_1\ a_2$ is not necessarily typable with boxy types—a severe problem. More generally, boxy types are not conservative with respect to small program transformations, as seen earlier (Section 4.6).

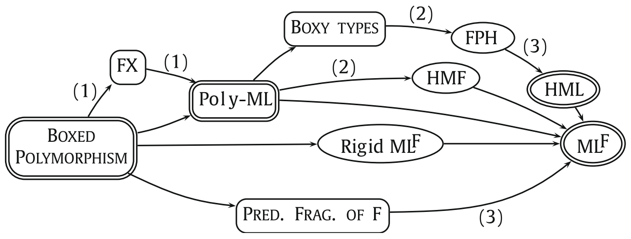
As boxy types, ML$^{\mathsf{F}}$ also removes the “coercion boxes” of Poly-ML, but does so in a more symmetric way, without arbitrary choices, by enriching the types of System F just as little as needed to represent all possible choices in derivations within (unique, principal) types.

Rigid ML$^{\mathsf{F}}$. This is a restriction of eML$^{\mathsf{F}}$ where the typechecking of let-bindings can only use rigid type schemes for bound expressions [18]. Interestingly, Rigid ML$^{\mathsf{F}}$ can only type programs of System F, as Simple ML$^{\mathsf{F}}$. In the absence of let-bindings, it does not perform better than ML$^{\mathsf{F}}$. However, it also allows implicit generalization at let-bindings, as ML$^{\mathsf{F}}$, but remaining within System F, as Simple ML$^{\mathsf{F}}$. Hence, Rigid ML$^{\mathsf{F}}$ can use System F as an intermediate explicitly typed language, which ML$^{\mathsf{F}}$ still lacks. Having an intermediate explicitly typed language is useful for compilers, such as the Haskell compiler, that simultaneously perform aggressive program transformations and maintain precise type information during compilation. As a corollary, Rigid ML$^{\mathsf{F}}$ can be used as a front end to Haskell.

Unfortunately, Rigid ML^F looses principal types in the usual sense—technically it recovers principal types by using an *ad hoc*, non logical side condition in typing rules to rule out some otherwise correct typing derivations—much as in FX. As a result, Rigid ML^F is fragile to some minor, but useful program transformations; It also requires more type annotations than ML^F—and the additional type annotations are often annoying.

(Partial) Type inference for the predicative fragment. Odersky and Läufer have also extended boxed polymorphism to implicit predicative instantiation of rank-2 polymorphism [27], which was later improved to arbitrary-rank types by Peyton Jones and Shields [13,30]. 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 [40] 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 are schematically summarized below. Unnanotated arrows mean *is weaker than*; annotated arrows are only approximations that applies to the main aspects of the systems but some particular programs may typecheck in the source and not in the target, with the following meaning for annotations: (1) up to small details; (2) the target is not stable by η-conversion while the source is; (3) except when relying on the left-to-right bias in the order of arguments in the source. Double lines are used for type systems enjoying principal types. Oval nodes are used for all restriction variants of ML^F.



The languages FPH, HMF, and HML that have been proposed more recently are discussed in Section 5.4.

5.3. Comparison with (other presentations of) ML^F

We have restricted our investigation to (Shallow) ML^F, because it is based on a semantics of types as sets of System-F types. The original presentation [17,16], instead focuses on the more expressive Full ML^F system and introduces Shallow ML^F as a restriction of Full ML^F. We first compare (Shallow) ML^F, with the original, Full and Shallow versions, successively. We then discuss the more recent graphical presentation of ML^F [42,43].

Comparison with Full ML^F. Although the original presentation was about Full eML^F, and the comparison should a priori be made with eML^F, there is little to say about type inference, but more about expressiveness. We thus consider the simpler, implicitly typed version Full iML^F and compare it with iML^F.

Types of iML^F are stratified: they distinguish between types, which are isomorphic to System-F types, and can be used to form arrow types, from type schemes, which can only be used as such or in the bound of variables to form other type schemes. By contrast, types are not stratified in Full iML^F and instead are identified with type schemes, as follows:

$$\tau, \sigma \in \mathcal{T}_{\mathsf{Full}} ::= \alpha \mid \tau \to \tau \mid \forall(\alpha \geq \tau)\ \tau \mid \bot$$

In particular, Full iML^F allows proper flexible types σ, e.g., $\forall(\alpha \geq \sigma')\ \tau$ to appear in positions where polymorphism is *required*, as in $\sigma \to \forall(\alpha)\ \alpha$, which is a type of Full iML^F but not of iML^F. For this reason, our semantics for iML^F does not easily extend to Full iML^F.

The problem may be illustrated by considering the semantics of the simpler type $\sigma \to \mathsf{int}$. In iML^F, σ is restricted to be a type τ of System F, i.e., whose semantics is a singleton $\{t\}$. Hence, the semantics of $\sigma \to \mathsf{int}$ could be simply defined as the singleton $\{t \to \mathsf{int}\}$. In Full iML^F, σ may be a proper flexible type, i.e., one whose semantics is not a singleton type. What should be the semantics of $\sigma \to \mathsf{int}$ in this case? Certainly not $\{t \to \mathsf{int} \mid t \in \{\!\{\sigma\}\!\}\}$, as this is the semantics of $\forall(\alpha \geq \sigma)\ \alpha \to \mathsf{int}$, which is quite different from $\sigma \to \mathsf{int}$. Intuitively, $\{\!\{\sigma \to \mathsf{int}\}\!\}$ should describe the type of functions that require their argument to have *all* types in $\{\!\{\sigma\}\!\}$. A temptation is to write "$\{\!\{\sigma\}\!\} \to \mathsf{int}$", but this is ill-formed since $\{\!\{\sigma\}\!\}$ is a set of types, not a type.

It is actually instructive to draw an analogy with intersection types (which can be used to model finite sets of types) where the intersection $\wedge$ distributes on the right-hand side of arrow types, but not on its left-hand side. $(\tau_1 \wedge \tau_2) \to \tau$ is not equivalent to $(\tau_1 \to \tau) \wedge (\tau_2 \to \tau)$. Hence, intersection types of the form $(\tau_1 \wedge \tau_2) \to \tau$, which are not intersection of System-F types, are essential. This suggests that extending the semantics of *i*ML$^{\mathsf{F}}$ to Full *i*ML$^{\mathsf{F}}$ would have to go beyond sets of System-F types, and perhaps, introduce a form of infinitary intersection types. However, these would have to be studied on their own first, which is likely to be more difficult than studying Full *i*ML$^{\mathsf{F}}$ types alone!

Comparison with the original. There remain a few differences between ML$^{\mathsf{F}}$ we presented here and the original Shallow ML$^{\mathsf{F}}$ [17] that are not just a matter of presentation. For the sake of comparison and by contrast with above, we here consider the explicitly typed versions.

The abstraction relation $\sqsubseteq$ in this version of *e*ML$^{\mathsf{F}}$ is larger than the original one so that *e*ML$^{\mathsf{F}}$ coincides exactly with *i*ML$^{\mathsf{F}}$. This is an improvement, as it follows from the semantics of *i*ML$^{\mathsf{F}}$ and its canonical definition of type-instance by interpretation of types as sets of System-F types.

This difference in the definition of abstraction is unimportant in practice, even though we may easily build an example in *e*ML$^{\mathsf{F}}$ that is not in the original version. For instance, consider the types σ_1 and σ_2, respectively, defined as $\forall(\alpha \geq \forall(\beta \Rrightarrow \sigma_{\mathsf{id}})\ \beta \to \beta)\ \alpha \to \alpha$ and $\forall(\beta \Rrightarrow \sigma_{\mathsf{id}})\ (\beta \to \beta) \to \beta \to \beta$. We have $\sigma_1 \sqsubseteq \sigma_2$ and $\sigma_2 \sqsupseteq \sigma_1$, hence $\sigma_1 \boxminus \sigma_2$. That is, types σ_1 and σ_2 can be (explicitly) converted to one another. However, in the original version we do not have $\sigma_1 \sqsubseteq{}_o\sigma_2$ but only $\sigma_1 \sqsubseteq_o \sigma_2$ (subscript *o* stands for the original version). Therefore, only σ_1 is (implicitly) convertible to σ_2 by type instance but σ_2 is not convertible to σ_1 in the original version. 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 in the definition of the abstraction relation was first suggested by François Pottier. However, its naive formalization in the original version would not be correct, because of pathological contexts in which uses of unprotected abstraction would break type soundness, as explained in Section 4.2. In order to prevents these misuses of abstraction, we introduced the auxiliary relation $\sqsubseteq^{\sharp}$ in the definition of $\sqsubseteq$. Quite interestingly, this change, which appears to be a small complication in the syntactic presentation of abstraction is actually a simplification when we view types as graphs [42], as described below.

There are other minor differences with the original presentation. For instance, the encoding of System F into *e*ML$^{\mathsf{F}}$ we have presented is different from the original one, as explained in Section 4.3. More precisely, the abstraction relation defines a lattice over types, as shown in [16]. A given type of System F corresponds to several types in the lattice, any of which could be chosen as its default encoding. In [16], 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 [42] and their application to graphical typing constraints is ongoing 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 semantics definition of type-instance does not easily generalize to Full ML$^{\mathsf{F}}$, the graphic presentation is 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.

5.4. Ongoing works around ML^{F}

Although *e*ML$^{\mathsf{F}}$ requires very few type annotations, and has a simple criterion for where and when to provide them, two criticisms have been made, namely the lack of an internal explicitly typed language and the introduction of "unfamiliar" types. These have motivated ongoing works on finding restrictions of ML$^{\mathsf{F}}$ that would circumvent these issues. Unsurprisingly, any limitation come at some price, increasing the number of required type annotations and losing robustness to program transformations and other interesting properties of ML$^{\mathsf{F}}$—such as preservation of typing by reduction.

An internal, explicitly typed language for ML^{F}. An argument against the use ML$^{\mathsf{F}}$ has been that programs cannot be elaborated into terms of System F. The system F$^{\mathsf{let}}$ is certainly not meant to be a replacement for System F as an internal language because the translation from ML$^{\mathsf{F}}$ to F$^{\mathsf{let}}$ does not preserve the modularity properties of ML$^{\mathsf{F}}$.

The problem with finding an internal language for *e*ML$^{\mathsf{F}}$ is that source terms contain type annotations that must be transformed during reduction, as shown in the original work [17].

Fortunately, an internal language, called *x*ML$^{\mathsf{F}}$, has just been proposed for ML$^{\mathsf{F}}$ [44]. This language generalizes type abstraction of System F by introducing flexible bounds on type variables and replaces type applications by instantiations—a small sub-language describing exactly how types are instantiated. Type annotations of ML$^{\mathsf{F}}$ are translated by type instantiations in *x*ML$^{\mathsf{F}}$, which can then be transformed during reduction so as to maintain well-typedness.

This improves over the only previously known alternative that is to insert coercion functions so as to remain within System F [21]. Indeed, those coercions are retained at runtime with an unnecessary runtime cost, in particular due to duplication of data-structures. Furthermore, it has never been proved that the insertion of these coercion functions actually preserves the semantics of programs.

Strange looking types. Types of *e*ML^F may look unfamiliar because they use two new forms of quantification: flexible and rigid bindings. It has been argued that this might hinder the adoption of ML^F. We agree that types of ML^F are sometimes harder to print and read because they are richer. However, the introduction of both flexible and rigid quantification has been driven by properties and program transformations that should hold in a reasonable type inference system.

Flexible types are used to factor into a *principal* typing derivation of ML^F, correct but incomparable derivations of System F. As a side effect, they also increase expressiveness, although this was not their first purpose. Flexible types now have some simple semantic explanation, which may help them look more familiar.

Rigid types do not increase expressiveness, but introduce extra information that partially remembers the way types have been derived, and in particular, distinguishes polymorphism that has been requested from polymorphism that has been provided. Even though rigid types may not be justified by semantic arguments, they mainly mimic flexible types in the way they keep track of sharing. Hopefully, they will look more familiar once flexible types will be well-understood and used.

Eliminating the need for rigid bindings. Leijen has identified a quite interesting restriction of ML^F, called HML, that does not use rigid bindings at all, but still permits type inference [20]. His proposal, inspired by *i*ML^F, could have been described as a stable subset of *e*ML^F, which we explain below.

Leijen's restriction proposal can be summarized as *disallowing the use of rigid bindings in the prefix of typing judgments*. As a result, all other rigid bindings, i.e., appearing in the typing environment or the result type become local and can be locally maximally shared. Below, we call *restricted* typing judgments of *e*ML^F that do not use rigid binding in prefixes and where all other types are maximally shared. When rigid bindings are maximally shared in types, they may be printed inlined and look as *i*ML^F types.

In order to allow all judgments to be restricted, we use derived rules App* and Fun** instead of App and Annot in derivations, which we call restricted derivations. The rule Fun** is the following variant of Fun* :

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

That avoids the need for putting rigid bindings in the prefix Q. We still allow rigid bindings to appear in derivations of instantiation and abstraction judgments (although we could also introduce derived rules instead).

Interestingly, under restricted prefixes, $\exists; \sqsubseteq$ steps may always be turned into $\sqsubseteq; \exists$ steps—a result that does not hold in general. This implies that under restricted prefixes, imposing maximal sharing in τ and Γ does not restrict typability. Formally, if Q is a restricted prefix and $(Q)\ \Gamma \vdash a : \sigma$ and $\Gamma \sqsubseteq \Gamma'$ (when $\sqsubseteq$ is extended to type environments pointwise), then there exists σ' such that $(Q)\ \Gamma' \vdash a : \sigma'$ and $\sigma \sqsubseteq \sigma'$. This enables to transform any restricted derivation of a judgment in restricted form into one where all intermediate judgments are also in restricted form. In such derivations, revelation needs never be used. By forcing maximal sharing in Γ, we may be forced to do more sharing in σ' as well; however, this is harmless, as this sharing could always—but never need to—be recovered a posteriori by a single final revelation.

The cost of bridling ML^F to restricted derivations is that all polymorphic arguments must be annotated regardless of whether they are treated parametrically or used polymorphically. Consequently, typing judgments are not themselves stable by arbitrary substitutions (but only by monotype ones). In particular, preservation of typings by β-reduction is lost. In practice, this also means that polymorphic types are not quite first class citizens. For instance, replacing a ground datatype by its Church encoding, e.g., replacing int by nat where nat is an abbreviation for $\forall(\alpha)\ (\alpha \to \alpha) \to \alpha \to \alpha$, does not preserve typability. This also means that data-structures with inner but *passive* polymorphism (that is just stored in data-structures or passed around to functions) cannot be used as conveniently as similar data-structures where polymorphism has been abstracted away as a new ground or simple type.

In addition, this version requires additional type annotations, which may sometimes be surprising and unpleasant. For example, $(\lambda(x)\ \lambda(y)\ x\ y)\ \omega^\dagger\ (\mathbf{1})$ is typable but its reduct $\lambda(y)\ \omega^\dagger\ y$ is not. That is, extra-type annotations need to be inserted during reduction. The drawback is that the user may be tempted to write programs with extra, obfuscating abstractions that also have a runtime cost, just to avoid extra type annotations. The example above also illustrate the lost of stability by both η-expansion and β-reduction. Consequently, although HML type inference engine is in essence more powerful and more practical than that of Poly-ML, it is not strictly speaking an extension of Poly-ML.

Nevertheless, this remains an interesting variant of ML^F as it preserves principal types and the logical flavor of typing rules, as well as interesting program transformation such as abstraction of application, as shown in (1), or let-conversion. Moreover, the use of simpler types that moreover coincide with those of *i*ML^F is unquestionably a clear gain.

Depriving ML^F from flexible bindings. This has also been proposed by Leijen [18] and called HMF. However, this is a drastic restriction that does not interact well with type inference. In particular, the two typings of choose id cannot be captured by a principal type without flexible typings. Hence, principal types *per se* are lost. They may be recovered by an *ad hoc* non-logical side condition that disallows one of the two typings—in fact, the choice is always made in favor of more external polymorphism, so as to remain compatible with ML. Unsurprisingly, the robustness of the type system to program transformation is really lost. For example, one may no longer always abstract over type applications or share common subexpressions in let-bindings.

Back to System F-types. The two restrictions of $\mathsf{ML^F}$ to HMF or HML are independent of one another and can be combined together [20]. The result is attractive because types then look just as regular System-F types. Unfortunately, the drawbacks accumulate and are so important that the author of the proposal does not himself recommend dropping flexible bindings [20].

Yet, another proposal by Vytiniotis et al. [48], called FPH, which looks quite different in presentation, using boxes to keep track of impredicative type instantiations, is of comparable expressive power, and interestingly, can also be explained using the $\mathsf{ML^F}$ framework.

As for HMF, rigid types are disallowed in the prefix and therefore they can always be maximally shared. Alternatively, types may be considered up to abstraction, as in $i\mathsf{ML^F}$. This may actually be made explicit by introducing an additional bound "=" that stands for equivalence and behaves as "⇒" except for the two following differences: first, under a prefix containing $\alpha = \sigma$, we have $\alpha \equiv \sigma$, i.e., both $\sigma \sqsubseteq \alpha$ and $\alpha \sqsubseteq \sigma$; second, turning a rigid binding into an equivalence binding is part of revelation but not of equivalence—except for monotype bounds which can always be inlined. The second fact is the key to FPH that requires explicit type annotations to *confirm* impredicative type instantiations, and so disambiguate between predicative and impredicative type instantiations.

Interestingly, FPH can now be explained by restricting typing judgments $Q, \Gamma \vdash a : \sigma$ of $\mathsf{ML^F}$ so that Q is always the empty prefix, flexible bindings are never used in Γ nor in σ, and rigid bindings are only used at the toplevel of σ. Equivalence bounds can be printed inlined in FPH. Rigid bindings represent boxed types. As they can always be maximally shared, as in HMF, they can also be printed inlined.

While FPH has a modular type inference algorithm, in the sense that typechecking of let-bound expressions need not be delayed, it does not enjoy principal types, because flexible bindings may only be used internally, for type inference, but may not appear in the result of type inference. For this reason, choose id does not have a principal type.

Of course, despite the benefit that the user need not (in principle) be exposed to boxed types but only to System-F types, this proposal loses many modularity properties of $\mathsf{ML^F}$, much as the combination of HML and HMF. Moreover, rigid types (i.e., boxes) cannot be really hidden from the user when displaying type errors. In fact, even flexible types, which are used internally, would help explaining type errors. The absence of principal types in FPH also implies that interfaces of compilation units must always be given explicitly.

Viewing FPH in the framework of $\mathsf{ML^F}$ has some advantages. It first shows, even though we did not make all the details explicit, that FPH is a subset of $\mathsf{ML^F}$, as already noticed by its authors. It also emphasizes the strong relations between FPH and HML, originally remarked by Leijen. Additionally, it suggests a simpler presentation of FPH where the subtyping relation allows extra boxing so that equality-up-to-unboxing side conditions on the application and annotation typing rules can be removed, leading to a more traditional presentation of FPH in the style of our generic type system.

Despite their drawbacks and limitations when compared with $\mathsf{ML^F}$, both FPH and HML (with or without flexible bindings), may be interesting in several ways: they show that rigid bindings are not necessary for the sole purpose of type inference, but only for getting *more* type inference and more robustness to program transformations. They may also allow a faster migration of existing languages based on Hindley–Milner type systems to type systems with first class-polymorphism, in particular in the case of Haskell which uses System F as an internal language.

$\mathsf{ML^F}$ with graphical types. In parallel work, graph-based definitions of types and of the type-instance relation have been introduced to simplify their syntactic defintion and explore the possible variations of these definitions more systematically. Hopefully, graphic types provide better intuitions on both flexible and rigid bindings and will help for their acceptance. Simple notational conventions can also be used to display $e\mathsf{ML^F}$ as System-F types in many cases [43].

The graph representation has already enabled a new, efficient unification algorithm for $\mathsf{ML^F}$ types. It is currently used to revisit, improve and modularize type inference for $\mathsf{ML^F}$. As mentioned earlier (Section 2.3.4), this work validates *a posteriori* the design choices made in the definition of the instance relation of $e\mathsf{ML^F}$.

Graphs have also helped find an internal language $x\mathsf{ML^F}$ [44].

Finally, graphs bring syntactic and semantic types closer. Interestingly, the graph representation applies indifferently to the shallow or full versions, and might be a means to also give a semantics to Full $\mathsf{ML^F}$.

5.5. Future works

$\mathsf{ML^F}$ 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, a type application $F(\tau)$ where F is a type operator is treated as an application $@(F, \tau)$ where @ is an application operator and F can be treated as an ordinary type, but of a higher-order kind. This extension is minor and only differs in the introduction of kinds, which are easy to keep track of. Interestingly, instantiation of type operators can then be inferred as all other type-instantiations in $\mathsf{ML^F}$.

Another solution that would better integrate higher-order types and reach all of F^ω is to use a distinct universal quantifier at higher kinds with type introduction and type elimination constructs that would always be explicit at higher kinds.

Although, inferring instantiation of type operators at higher-order kinds seems possible in some interesting cases, type-level computation resulting from instantiation of higher-order types seems to be conflicting with the precise sharing between polymorphic types that has to be maintained in ML^F. Studying this interaction remains an interesting research direction.

Extending ML^F with recursive types is another interesting investigation to pursue. In the presence of type inference, one would expect implicit equi-recursive types to be used, as in ML. While we anticipate 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.

6. Conclusions

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

The Curry-style version $i\mathsf{ML}^\mathsf{F}$ 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 $i\mathsf{ML}^\mathsf{F}$ is a subset of $\mathsf{F}^{\mathrm{let}}$, 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-style version $e\mathsf{ML}^\mathsf{F}$ that permits type inference. Expressions of $e\mathsf{ML}^\mathsf{F}$, given with some explicit type annotations, always have principal types.[13] Technically, $e\mathsf{ML}^\mathsf{F}$ introduces rigid bindings written $\forall(\alpha \Rightarrow \sigma)\ \sigma'$ to mediate between explicit type information σ and its implicit view α within σ'. Interestingly, $e\mathsf{ML}^\mathsf{F}$ is a conservative extension of ML as fully unannotated programs are typable in $e\mathsf{ML}^\mathsf{F}$ if and only if they are typable in ML. Moreover, all System F programs can be turned into $e\mathsf{ML}^\mathsf{F}$ programs by simply dropping type abstractions and type applications and by a simple translation of type annotations. Besides, only function arguments that are used polymorphically need a type annotation in $e\mathsf{ML}^\mathsf{F}$. This provides a clear specification of *when* and *how* to annotate type parameters.

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 $i\mathsf{ML}^\mathsf{F}$ have *more-expressive* types (and *more* principal types) than in System F and are therefore more modular and more robust to program transformations.

Although there are still a few options in the design space, $e\mathsf{ML}^\mathsf{F}$ seems to be a local optimal, where any restriction looses some of its interesting properties and any extension in expressiveness would very likely require a form of higher-order unification. The restriction of ML^F to flexible-only bindings [20] is another interesting point in the design space that might be another alternative, a less expressive but simpler surface language for $i\mathsf{ML}^\mathsf{F}$.

Acknowledgments

The authors are grateful to Boris Yakobowski for many technical discussions and his expertise on the graphic presentation of ML^F. We express our gratitude to Daan Leijen and Dimitrios Vytiniotis whose interesting insight on ML^F helped us improve the technical content of this article. We thank the anonymous referees for their precious comments.

Appendix

Proofs of main results

Proof of Lemma 3.3.8

By structural induction on σ.

∘ Case α: Obviously, α is exposed in σ, therefore t must be inert. 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 $t_2' \in \{\!\{\sigma_2\}\!\}$.

[13] This has only been shown formally for a small variant of $e\mathsf{ML}^\mathsf{F}$ [17,16].

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 *i*Equ-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 $\alpha_1 \notin \mathrm{ftv}(\sigma_2)$ (**1**) and $\alpha_2 \notin \mathrm{ftv}(\sigma_1)$ (**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 $\bar{\beta} \# \mathrm{ftv}(\sigma_a)$, $t' \in \{\!\{\forall(\alpha_2 \geq \sigma_2)\ \sigma\}\!\}$, 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 $\bar{\beta}' \# \mathrm{ftv}(\forall(\alpha_2 \geq \sigma_2)\ \sigma)$, $t \in \{\!\{\sigma\}\!\}$, and $t_2 \in \{\!\{\sigma_2\}\!\}$ (**6**). By α-conversion, we may assume, *w.l.o.g.*, $\alpha_2 \notin \mathrm{ftv}(t_1) \cup \{\alpha_1\}$ (**7**) and $\bar{\beta}' \# \mathrm{ftv}(t_1) \cup \{\alpha_1\} \cup \mathrm{ftv}(\sigma_1)$. 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 $\bar{\beta}\bar{\beta}' \# \mathrm{ftv}(\sigma_b)$, which implies (3) by Definition 3.2.1.

∘ Case *i*Equ-Free: Let σ with $\alpha \notin \mathrm{ftv}(\sigma)$. We show $\{\!\{\sigma\}\!\} = \{\!\{\forall(\alpha \geq \sigma')\ \sigma\}\!\}$ by considering both inclusions separately. Assume $t \in \{\!\{\sigma\}\!\}$. We may as well assume $\alpha \notin \mathrm{ftv}(t)$, *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 *i*Equ-Var is by definition and Lemma 3.3.4.

∘ Case *i*Equ-Inert: by hypothesis, $(\alpha \geq \sigma') \in Q$. Thus, Q is of the form $(Q_1, \alpha \geq \sigma', Q_2)$ 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 $t_a \in \{\!\{\sigma'\}\!\}$. Since by hypothesis $(Q)\ \sigma' \boxminus \tau$, where τ is an inert type, this implies $t_a = \lceil \tau \rceil$ (**9**) by induction hypothesis. 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, $\alpha \notin \mathrm{ftv}(t'')$, 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) holds by structural induction on the derivation. The proof of (ii) is by structural induction on σ. □

Proof of Lemma 3.3.12

Each rule is considered separately. Rule *i*Ins-Equiv is by Lemma 3.3.9. Rule *i*Ins-Bot is by definition. For Rule *i*Ins-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 $\alpha \notin \mathrm{ftv}(t)$, *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 *i*Ins-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. □

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 $\bar{\beta} \# \mathrm{ftv}(t)$. If τ' is some variable α with $\alpha \in \bar{\alpha}$, then $\lfloor t \rfloor$ is equivalent to $\bot$ by *i*Equ-Var, and we conclude directly by *i*Ins-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

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

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 (Inst and *i*Equ-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 *i*F-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 *i*F-Gen, which can be shown as follows. We reuse the notations of rule *i*F-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 *i*F-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 4.2.11

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 *e*Con-AllLeft and *e*Con-AllRight), 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 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 $\sigma_b \notin \mathcal{V}$: 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 *i*Con-AllLeft, *i*Con-AllRight 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 $\sigma_a \notin \mathcal{V}$, this is the expected result. Otherwise, $\sigma_a \equiv \beta$, which implies $(Q)\ \sigma_b \equiv \beta$, and then by Lemma 4.2.7, $\sigma_b \equiv \tau'$ (**5**) for some type τ'. Since $\sigma_b \notin \mathcal{V}$, this implies $\tau' \notin \mathcal{V}$ (**6**). By Lemma 3.3.11.ii and (5), we have $(Q')\ \forall(\alpha \geq \theta(\llbracket\sigma_a\rrbracket))\ \theta\llbracket\sigma'_a\rrbracket \boxminus \forall(\alpha \geq \theta(\tau'))\ \theta\llbracket\sigma'_a\rrbracket$ (**7**). By (6), we have $\theta(\tau') \in \mathcal{I}_i$. Hence, by *i*Equ-Inert applied to (5), we have $(Q')\ \forall(\alpha \geq \theta(\tau'))\ \theta\llbracket\sigma'_a\rrbracket \boxminus (\theta(\llbracket\sigma'_a\rrbracket)[\theta(\tau')/\alpha]$, i.e., $(Q')\ \forall(\alpha \geq \theta(\tau'))\ \theta\llbracket\sigma'_a\rrbracket \boxminus \theta(\llbracket\sigma'_a\rrbracket[\tau'/\alpha])$ (**8**). We conclude by (8), (7), (4) and transitivity.

Subcase $\sigma_b \in \mathcal{V}$: we assume $\sigma_b \equiv \beta$. By induction hypothesis, we have $(Q')\ \theta(\llbracket\sigma'_a\rrbracket)[\theta(\beta)/\alpha] \boxminus \theta(\llbracket\sigma'_b\rrbracket)[\theta(\beta)/\alpha]$. From (3), we have $(Q')\ \theta\llbracket\sigma_a\rrbracket \boxminus \theta(\beta)$ (**9**). If $\sigma_a \equiv \gamma$, then $(Q')\ \theta\llbracket\sigma_a\rrbracket \boxminus \theta(\gamma)$, and we get $(Q')\ \theta(\gamma) \equiv \theta(\beta)$ from (9). Then, we conclude by Lemma 3.3.11.ii. Otherwise, $\llbracket \sigma_a \rrbracket \notin \mathcal{V}$. We conclude by (9), *i*CON-ALLLEFT, *i*CON-ALLRIGHT, and *i*EQU-INERT, using Lemma 4.2.7.

∘ CASE *e*EQU-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 *i*EQU-COMM. The last subcase is by reflexivity.

∘ CASE *e*EQU-VAR: We distinguish two subcases, depending on the binding being flexible or rigid. If it is flexible, we use *i*EQU-VAR. Otherwise, we use reflexivity.

∘ CASE *e*EQU-FREE: Similarly, we use *i*EQU-FREE if the binding is flexible. Otherwise, we use reflexivity.

∘ CASE *e*EQU-MONO: We use *i*EQU-INERT if the binding is flexible. Otherwise, we use reflexivity.

As for the abstraction relation (statement ii), we show the property for both relations $\sqsubseteq$ and $\sqsubseteq^\sharp$, simultaneously by induction on the derivation. That is, if $(Q)\ \sigma_1 \sqsubseteq \sigma_2$ or $(Q)\ \sigma_1 \sqsubseteq^\sharp \sigma_2$, then $(Q)'\ \theta \llbracket \sigma_1 \rrbracket \boxminus \theta \llbracket \sigma_2 \rrbracket$. Transitivity is by induction hypothesis. Rule *e*ABS-SHARPLEFT : shown as above, like for equivalence. Rule *e*ABS-EQUIV : by consequence of property i. Rule *e*ABS-HYP : by reflexivity. Rule *e*ABS-SHARPDROP : by induction hypothesis.

∘ CASE *e*ABS-LEFT : σ_1 is of the form $\forall(\alpha \geq \sigma_1')\ \sigma'$ and σ_2 is $\forall(\alpha \geq \sigma_2')\ \sigma'$. By hypothesis, $(Q)\ \sigma_1' \sqsubseteq^\sharp \sigma_2'$ (**10**) holds. If σ_1' or σ_2' is in $\mathcal{V}$, then $(Q)\ \sigma_1' \equiv \sigma_2'$ holds by Lemma 4.2.5, so we get $(Q)\ \sigma_1 \equiv \sigma_2$ by congruence, and we conclude by property i. We assume now that neither σ_1' nor σ_2' are in $\mathcal{V}$. Then, $\llbracket \sigma_1 \rrbracket$ is $\forall(\alpha \geq \llbracket \sigma_1 \rrbracket)\ \llbracket \sigma' \rrbracket$ and $\llbracket \sigma_2 \rrbracket$ is $\forall(\alpha \geq \llbracket \sigma_2 \rrbracket)\ \llbracket \sigma' \rrbracket$. We conclude by induction hypothesis and (10).

∘ CASE *e*EQU-INERT : σ_1 is $\forall(\alpha \geq \sigma)\ \sigma'$ and σ_2 is $\forall(\alpha \Rightarrow \sigma)\ \sigma'$. By hypothesis, σ is inert in *e*ML$^\mathsf{F}$. As a consequence, $\llbracket \sigma \rrbracket$ is inert in *i*ML$^\mathsf{F}$(this is an easy consequence of the definition). Hence, we get the expected result by *i*EQU-INERT.

The instance relation (statement iii) is shown similarly. Transitivity is by induction hypothesis. Flexible-congruence is shown as above (see the equivalence case), using *i*INS-SUBST if the right-hand side is in $\mathcal{V}$. *e*INS-ABSTRACT is a consequence of ii and *i*INS-EQUIV. *e*INS-BOT is shown with *i*INS-BOT. *e*INS-HYP is shown with *i*INS-HYP. Finally, *e*INS-RIGID is shown with *i*INS-SUBST, using Lemma 4.2.9. □

Proof of Lemma 4.2.12

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 $(\alpha \geq \sigma') \in Q$ with $(\emptyset)\ \lceil \sigma' \rceil\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \lceil \tau \rceil$ (**1**) and $\tau \in \mathcal{I}_i$. We have to show $(\lceil Q \rceil)\ \alpha\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \lceil \tau \rceil$ (**2**). By *e*ABS-HYP, we get $(\lceil Q \rceil)\ \alpha \sqsupseteq \lceil \sigma' \rceil$. We conclude by (1).

∘ CASE $\sigma = \tau_1 \to \tau_2$ is by induction hypothesis and *e*ABS-SHARPLEFT.

∘ CASE $\sigma = \forall(\alpha \geq \sigma_1)\ \sigma_2$ is by induction hypothesis and *e*ABS-LEFT. □

Proof of Lemma 4.2.14

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

∘ CASE *i*EQU-COMM: by Rule *e*EQU-COMM.

∘ CASE *i*EQU-FREE: by Rule *e*EQU-FREE.

∘ CASE *i*EQU-INERT: by Lemma 4.2.12 and induction hypothesis.

∘ CASE *i*EQU-VAR: by Rule *e*EQU-VAR.

Congruence of $\boxminus$ is defined by rules *i*CON-ALLLEFT, *i*CON-ALLRIGHT, and *i*CON-ARROW:

∘ CASE *i*CON-ALLLEFT: by induction hypothesis and Rule *e*ABS-LEFT.

∘ CASE *i*CON-ALLRIGHT: σ_1 is of the form $\forall(\alpha \geq \sigma)\ \sigma_1'$ and σ_2 is of the form $\forall(\alpha \geq \sigma)\ \sigma_2'$. The premise is $(Q, \alpha \geq \sigma)\ \sigma_1' \boxminus \sigma_2'$. By induction hypothesis, we get $(\lceil Q \rceil, \alpha \diamond \lceil \sigma \rceil)\ \lceil \sigma_1' \rceil\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \lceil \sigma_2' \rceil$, where the binding of α is rigid if σ is equivalent to an inert type τ, and flexible otherwise. By Rule *e*CON-ALLRIGHT, we get $(\lceil Q \rceil)\ \forall(\alpha \diamond \lceil \sigma \rceil)\ \lceil \sigma_1' \rceil\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \forall(\alpha \diamond \lceil \sigma \rceil)\ \lceil \sigma_2' \rceil$ (**1**). If σ is not equivalent to a monotype, this is the expected result. Otherwise, $(\emptyset)\ \sigma \boxminus \tau$ holds and we derive $(\lceil Q \rceil)\ \forall(\alpha \Rightarrow \lceil \sigma \rceil)\ \lceil \sigma_1' \rceil\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \forall(\alpha \Rightarrow \lceil \tau \rceil)\ \lceil \sigma_1' \rceil$ by induction hypothesis and *e*ABS-SHARPLEFT. Additionally, $(\lceil Q \rceil)\ \forall(\alpha \Rightarrow \lceil \tau \rceil)\ \lceil \sigma_1' \rceil\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \forall(\alpha \geq \lceil \tau \rceil)\ \lceil \sigma_1' \rceil$ hold by *e*EQU-INERT. We conclude by doing the same derivation for the right-hand side of (1) and transitivity.

∘ CASE *i*CON-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 $(\lceil Q \rceil)\ \lceil \tau_1 \rceil\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \lceil \tau_2 \rceil$ (**2**) and $(\lceil Q \rceil)\ \lceil \tau_1' \rceil\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \lceil \tau_2' \rceil$ (**3**). by definition, $\lceil \tau_1 \to \tau_1' \rceil$ is $\forall(\alpha_1 \Rightarrow \lceil \tau_1 \rceil, \alpha_1' \Rightarrow \lceil \tau_1' \rceil)\ \alpha_1 \to \alpha_1'$. by Rule *e*ABS-LEFT, (2) and (3), we get $(\lceil Q \rceil)\ \lceil \tau_1 \to \tau_1' \rceil\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \forall(\alpha_1 \Rightarrow \lceil \tau_2 \rceil, \alpha_1' \Rightarrow \lceil \tau_2' \rceil)\ \alpha_1 \to \alpha_1'$, that is, $(\lceil Q \rceil)\ \lceil \tau_1 \to \tau_1' \rceil\ (\sqsubseteq^\sharp \cup \sqsupseteq^\sharp)^*\ \lceil \tau_2 \to \tau_2' \rceil$, 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 $(\sqsubseteq \cup \sqsupseteq)^*$.

∘ CASE *i*INS-EQUIV: by property i), and rules *e*ABS-SHARPDROP and *e*INS-ABSTRACT.

∘ CASE *i*INS-BOT: by *e*INS-BOT

∘ CASE *i*INS-HYP: by *e*INS-HYP, *e*ABS-HYP and *e*ABS-SHARPLEFT.

- Case *i*Ins-Subst: by *e*Ins-Rigid and Lemma 4.2.13.
  The $\geq$-congruence of $\leq$ is defined by the rules *i*Con-AllLeft and *i*Con-AllRight.
- Case *i*Con-AllLeft: by induction hypothesis and Rule *e*Con-FlexLeft.
- Case *i*Con-AllRight: similar to case *i*Con-AllRight above. □

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 4.2.11.ii and 4.2.11.iii. □

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 *i*ML$^{\mathsf{F}}$ :: $(Q)\ \Gamma \vdash a : \sigma$.

- Case Var: necessarily, a is a variable x such that $x : \sigma$ belongs to Γ, and so $x : \lceil\sigma\rceil$ belongs to $\lceil\Gamma\rceil$. We conclude by Rule Var, taking $a' = x$.
- Case Fun: a is of the form $\lambda(x)\ b, \sigma$ is $\tau \to \tau'$, and the premise is *i*ML$^{\mathsf{F}}$ :: $(Q)\ \Gamma, x : \tau \vdash b : \tau'$. By induction hypothesis, there exists b' (with type erasure b) such that *e*ML$^{\mathsf{F}}$ :: $(\lceil Q\rceil)\ \lceil\Gamma\rceil, x : \lceil\tau\rceil \vdash b' : \lceil\tau'\rceil$ (**4**) holds. Let $\bar\alpha$ be $\mathrm{ftv}(\lceil\tau\rceil)$ and a' be $\lambda(x : \exists(\bar\alpha)\ \lceil\tau\rceil)\ b'$. We have

$$\begin{array}{lll}
(\mathbf{5}) & (\lceil Q\rceil) & \lceil\Gamma\rceil, x : \lceil\tau\rceil \vdash b' : \forall(\alpha_2 \Rightarrow \lceil\tau'\rceil)\ \alpha_2 \\
(\mathbf{6}) & (\lceil Q\rceil, \alpha_2 \Rightarrow \lceil\tau'\rceil) & \lceil\Gamma\rceil, x : \lceil\tau\rceil \vdash b' : \alpha_2 \\
(\mathbf{7}) & (\lceil Q\rceil, \alpha_2 \Rightarrow \lceil\tau'\rceil) & \lceil\Gamma\rceil \vdash a' : \forall(\alpha_1 \Rightarrow \lceil\tau\rceil)\ \alpha_1 \to \alpha_2 \\
(\mathbf{8}) & (\lceil Q\rceil) & \lceil\Gamma\rceil \vdash a' : \forall(\alpha_1 \Rightarrow \lceil\tau\rceil, \alpha_2 \Rightarrow \lceil\tau'\rceil)\ \alpha_1 \to \alpha_2 \\
(\mathbf{9}) & (\lceil Q\rceil) & \lceil\Gamma\rceil \vdash a' : \lceil\tau \to \tau'\rceil
\end{array}$$

We get (5) by equivalence from (4) (Rule Inst and Rule *e*Equ-Var). We obtain (6) by Rule UnGen*, and (7) by Rule Fun*. By Rule Gen, we get (8), which is equal to the expected result (9).

- Case App: a is $a_1\ a_2$, and the premises are *i*ML$^{\mathsf{F}}$ :: $(Q)\ \Gamma \vdash a_1 : \tau_2 \to \tau_1$ (**10**) and *i*ML$^{\mathsf{F}}$ :: $(Q)\ \Gamma \vdash a_2 : \tau_2$ (**11**), where τ_1 is σ. By induction hypothesis, there exist both a_1' and a_2' (with respective type erasures a_1 and a_2) such that *e*ML$^{\mathsf{F}}$ :: $(\lceil Q\rceil)\ \lceil\Gamma\rceil \vdash a_1' : \forall(\alpha_2 \Rightarrow \lceil\tau_2\rceil)\ \forall(\alpha_1 \Rightarrow \lceil\tau_1\rceil)\ \alpha_2 \to \alpha_1$ (**12**) and *e*ML$^{\mathsf{F}}$ :: $(\lceil Q\rceil)\ \lceil\Gamma\rceil \vdash a_2' : \lceil\tau_2\rceil$. The latter can as well be written $(\lceil Q\rceil)\ \lceil\Gamma\rceil \vdash a_2' : \forall(\alpha_2 \Rightarrow \lceil\tau_2\rceil)\ \alpha_2$ by equivalence (Rule Inst and Rule *e*Equ-Var). We conclude by Rule App* take $a_1'\ a_2'$ for a'.
- Case Inst: The premises are *i*ML$^{\mathsf{F}}$ :: $(Q)\ \Gamma \vdash a : \sigma'$ (**13**) and $(Q)\ \sigma' \leq \sigma$ (**14**). By induction hypothesis, there exists a_0 such that *e*ML$^{\mathsf{F}}$ :: $(\lceil Q\rceil)\ \lceil\Gamma\rceil \vdash a_0 : \lceil\sigma'\rceil$. By Lemma 4.2.14.ii and (14), $(\lceil Q\rceil)\ \lceil\sigma'\rceil\ (\sqsubseteq \cup \exists)^*\ \lceil\sigma\rceil$ holds, that is, there exist a sequence $\sigma_0, \sigma_1, \ldots, \sigma_n$, with $\sigma_0 = \lceil\sigma'\rceil$ and $\sigma_n = \lceil\sigma\rceil$, such that $(\lceil Q\rceil)\ \sigma_i \sqsubseteq \sigma_{i+1}$ if i is even, and $(\lceil Q\rceil)\ \sigma_i \exists \sigma_{i+1}$ if i is odd. We show by induction on i, for $i \leqslant n$, that there exists a_i with type erasure a, such that $(\lceil Q\rceil)\ \lceil\Gamma\rceil \vdash a_i : \sigma_i$ (**15**). Then, the expected result $(\lceil Q\rceil)\ \lceil\Gamma\rceil \vdash a' : \lceil\sigma\rceil$ is obtained with $i = n - 1$ and $a' = a_{n-1}$. The judgment (15) holds for $i = 0$, as shown above. By induction hypothesis, we assume it holds for some $i < n$. If i is even, we get judgment (15) for $i + 1$ using Rule Inst. If i is odd, we get judgment (15) 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* $(\leqslant \cup \exists)^* \subset (\leqslant \circ \exists)$. *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 *i*ML$^{\mathsf{F}}$ :: $(Q, \alpha \geq \sigma_1)\ \Gamma \vdash a : \sigma_2$, with $\alpha \notin \mathrm{ftv}(\Gamma)$, and σ is $\forall(\alpha \geq \sigma_1)\ \sigma_2$. By induction hypothesis, there exists a'' such that *e*ML$^{\mathsf{F}}$ :: $(\lceil Q, \alpha \geq \sigma_1\rceil)\ \lceil\Gamma\rceil \vdash a'' : \lceil\sigma_2\rceil$ (**16**). Let $\forall(Q_1)\ \sigma_1'$ be $\lceil\sigma_1\rceil$, with Q_1 rigid and as large as possible. Then (16) is $(\lceil Q\rceil, Q_1, \alpha \geq \sigma_1')\ \lceil\Gamma\rceil \vdash a'' : \lceil\sigma_2\rceil$. We notice that the domain of Q_1 can be chosen disjoint from $\mathrm{ftv}(\Gamma)$. By repeated uses of Rule Gen, we get $(\lceil Q\rceil)\ \lceil\Gamma\rceil \vdash a'' : \forall(Q_1, \alpha \geq \sigma_1')\ \lceil\sigma_2\rceil$. Since $(\lceil Q\rceil)\ \forall(Q_1, \alpha \geq \sigma_1')\ \lceil\sigma_2\rceil \exists \forall(\alpha \geq \forall(Q_1)\ \sigma_1')\ \lceil\sigma_2\rceil$ holds by *e*Con-AllRight, *e*Abs-Left, and repeated uses of *e*Abs-Hyp, we conclude by Rule Annot, taking $a' = (a'' : \lceil\sigma\rceil)$.
- Case Let: by induction hypothesis and Rule Let. □

Proof of Lemma 4.4.7

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 4.4.9

To lighten the notation in this proof and subsequent ones, 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 $\{\alpha\} \cap \mathsf{dom}(Q)$.

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 4.4.9 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*

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

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 $\alpha_i \notin \mathsf{dom}(Q_iP_i) \Longrightarrow \alpha_i \notin I$.

∘ 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 $\alpha_2 \notin \mathsf{dom}(Q_2P_2) \Longrightarrow \alpha_2 \notin I$. 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

$$\mathsf{dom}(\phi_1) \# I \cup \alpha P\ (\mathbf{7}) \qquad I \cup (\alpha_1 \cap Q'_1) \subseteq J_1\ (\mathbf{8}) \qquad Q'_1 \equiv^{J_1} \phi_1(Q'_2)\ (\mathbf{9})$$
$$\phi_1(\alpha_2) = \alpha_1\ (\mathbf{10}) \qquad \phi_1(P'_2) = P'_1\ (\mathbf{11}) \qquad \mathsf{ftv}(P'_1) \subseteq I \cup \alpha\ (\mathbf{12})$$

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:

$$\phi_1(Q_2)\alpha P \otimes_{\phi_1(\alpha_2)} \sigma = (\phi_1(Q'_2)\alpha\phi_1(P'_2), \phi_1(\alpha_2))$$
$$\phi_1(Q_3)\alpha P \otimes_{\phi_1(\alpha_3)} \sigma = (\phi_1(Q'_3)\alpha\phi_1(P'_3), \phi_1(\alpha_3))$$

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

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

$$\phi_1(Q_2') \mathrel{\overset{J_2}{\equiv}} \phi_2 \circ \phi_1(Q_3') \ (\mathbf{15}) \qquad \phi_2 \circ \phi_1(\alpha_3) = \alpha_1 \ (\mathbf{16}) \qquad \phi_2 \circ \phi_1(P_3') = P_1' \ (\mathbf{17})$$

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

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

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

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 *e*Sha-Comm: Similar to reflexivity.

◦ Case *e*Sha-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 *e*Sha-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, $Q_1' \mathrel{\overset{J}{\equiv}} \phi(Q_2')$ 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

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

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

$$(Q_2'\alpha P_2', \alpha_2) = Q_2^b \alpha P_2^b \otimes_{\alpha_2} \alpha_2^a \to \alpha_2^b \ (\mathbf{30})$$

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

$$\mathsf{dom}(\phi_1) \# I \cup \alpha P \ (\mathbf{31}) \qquad I \cup (\alpha_1^a \cap Q_1^a) \subseteq J_1 \qquad Q_1^a \mathrel{\overset{J_1}{\equiv}} \phi_1(Q_2^a) \ (\mathbf{32})$$

$$\phi_1(\alpha_2^a) = \alpha_1^a \qquad \phi_1(P_2^a) = P_1^a \qquad \mathsf{ftv}(P_1^a) \subseteq J_1 \cup \alpha$$

Note that (31) and (24) imply $\phi_1(t) = t$. By Lemma 4.4.5 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

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

$$\phi_2 \circ \phi_1(\alpha_2^b) = \alpha_1^b \qquad \phi_2 \circ \phi_1(P_2^b) = P_1^b \qquad \mathsf{ftv}(P_1^b) \subseteq J_2 \cup \alpha$$

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

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

$$\phi'(\alpha_2^a) = \alpha_1^a \qquad \phi'(P_2^b) = P_1^b \qquad \mathsf{ftv}(P_1^b) \subseteq J_2 \cup \alpha$$

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

$$\mathsf{dom}(\psi) \# J_2 \cup \alpha P_1^b \qquad J_2 \cup (\alpha_1 \cap Q_1') \subseteq J \qquad Q_1' \equiv^J \psi \circ \phi'(Q_2')$$
$$\psi \circ \phi'(\alpha_2) = \alpha_1 \qquad \psi \circ \phi'(P_2') = P_1' \qquad \mathsf{ftv}(P_1') \subseteq J \cup \alpha$$

Let ϕ be $\psi \circ \phi'$. We get

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

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

$$(Q_1\alpha P\beta)\ \langle\langle t_0\rangle\rangle : (Q_1^a\alpha P_1^a\beta P_1^b, \beta_1)\ (\mathbf{33})$$
$$(Q_2\alpha P\beta)\ \langle\langle t_0\rangle\rangle : (Q_2^a\alpha P_2^a\beta P_2^b, \beta_2)\ (\mathbf{34})$$
$$(Q_1'\alpha P_1', \alpha_1) = Q_1^a\alpha P_1^a \otimes_{\alpha_1} \forall(\beta P_1^b)\ \beta_1\ (\mathbf{35})$$
$$(Q_2'\alpha P_2', \alpha_2) = Q_2^a\alpha P_2^a \otimes_{\alpha_2} \forall(\beta P_2^b)\ \beta_2\ (\mathbf{36})$$

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

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

Let ϕ be $\psi \circ \phi'$. We have

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

This is the expected result. □

Proof of Corollary 4.4.10

In order to lighten the presentation, we use the notations defined in the proof of Lemma 4.4.9.

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:

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

Using (5) and *e*SHA-FREE repeatedly, one may derive $\emptyset \equiv^I Q_1$. By Lemma 4.4.9, there exist a set J and a renaming ϕ such that

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

From (8), (7), and Lemma 4.48, 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 4.4.11

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

Proof of Lemma 4.4.12

Each property is shown separately.

<u>P-I</u>: 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 $Q_0\alpha_1Q_1..\alpha_nQ_n$ with $Q_1, .., Q_n$ rigid. From (1), we know that Q' equals $Q_0'\alpha_1Q_1'..\alpha_nQ_n'$ 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 $Q_0\alpha_1Q_1..\alpha_iQ_i, (\alpha' \Rightarrow \sigma), ..\alpha_nQ_n$.

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 $Q_0'\alpha_1Q_1'..\alpha_iQ_i', (\alpha' \Rightarrow \sigma'), ..\alpha_nQ_n'$. 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 $\phi' \circ \phi(\alpha')$.

<u>P-II</u>: 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 $(Q_1)\ \langle\!\langle t_1\rangle\!\rangle : (Q_1^a, \alpha_1^a)$ (**3**) and $(Q_1^a)\ \langle\!\langle t_2\rangle\!\rangle : (Q_1^b, \alpha_1^b)$ (**4**), and Q_1' is $Q_1^b \otimes_{\alpha_1} \alpha_1^a \to \alpha_1^b$. By induction hypothesis and (3), there exist Q_2^a, α_2^a and ϕ^a such that

$$(Q_2)\ \langle\!\langle \psi(t_1)\rangle\!\rangle : (Q_2^a, \alpha_2^a)\ \mathbf{(5)} \qquad \psi(Q_1^a) \Rrightarrow_{\phi^a \circ \phi} Q_2^a \qquad \phi^a \circ \phi \circ \psi(\alpha_1^a) = \alpha_2^a$$

By induction hypothesis and (4), there exist Q_2^b, α_2^b and ϕ^b such that

$$(Q_2^a)\ \langle\!\langle \psi(t_2)\rangle\!\rangle : (Q_2^b, \alpha_2^b)\ \mathbf{(6)} \qquad \psi(Q_1^b) \Rrightarrow_{\phi^b \circ \phi^a \circ \phi} Q_2^b\ \mathbf{(7)}$$

$$\phi^b \circ \phi^a \circ \phi \circ \psi(\alpha_1^b) = \alpha_2^b$$

We note that $\phi^b \circ \phi^a \circ \phi \circ \psi(\alpha_1^a \to \alpha_1^b)$ equals $\alpha_2^a \to \alpha_2^b$ and so $\phi^b \circ \phi^a \circ \phi \circ \psi(\alpha_1^a \to \alpha_1^b) \Rrightarrow \alpha_2^a \to \alpha_2^b$ holds. Thus, Property P-I and (7), imply that there exists ϕ^c such that $\mathsf{dom}(\phi^c) \subseteq \{\alpha_1\}$ (**8**) and $\psi(Q_1') \Rrightarrow_{\phi^c \circ \phi^b \circ \phi^a \circ \phi} Q_2^b \otimes_{\phi^c \circ \phi^b \circ \phi^a \circ \phi(\alpha_1)} \alpha_2^a \to \alpha_2^b$ (**9**). Let ϕ' be $\phi^c \circ \phi^b \circ \phi^a$. Let Q_2' be $Q_2^b \otimes_{\phi' \circ \phi(\alpha_1)} \alpha_2^a \to \alpha_2^b$. 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 $(Q_1\gamma)\ \langle\!\langle t'\rangle\!\rangle : (Q_1^a\gamma Q_1^b, \delta)$ and Q_1' is $Q_1^a \otimes_{\alpha_1} \forall(\gamma Q_1^b)\ \delta$. We observe that $\psi(Q_1\gamma) \Rrightarrow_\phi Q_2\gamma$ holds. Thus, by induction hypothesis, there exist Q_2^a, Q_2^b, δ' and ϕ^a such that

$$(Q_2\gamma)\ \langle\!\langle \psi(t')\rangle\!\rangle : (Q_2^a\gamma Q_2^b, \delta')\ \mathbf{(11)} \qquad \psi(Q_1^a\gamma Q_1^b) \Rrightarrow_{\phi^a \circ \phi} Q_2^a\gamma Q_2^b\ \mathbf{(12)}$$

$$\phi^a \circ \phi \circ \psi(\delta) = \delta'\ \mathbf{(13)}$$

We note that (12) implies that there exist ϕ^b and ϕ^c such that $\psi(Q_1^a) \Rrightarrow_{\phi^b \circ \phi} Q_2^a$ (**14**) and $\phi^b \circ \phi \circ \psi(Q_1^b) \Rrightarrow_{\phi^c} Q_2^b$ with $\phi^a = \phi^c \circ \phi^b$ (**15**). As a consequence, $\phi^b \circ \phi \circ \psi(\forall(\gamma Q_1^b)\ \delta) \Rrightarrow \forall(\gamma Q_2^b)\ \phi^c \circ \phi^b \circ \phi \circ \psi(\delta)$ 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 $\psi(Q_1')\Rrightarrow_{\phi^d\circ\phi^b\circ\phi} Q_2^a \otimes_{\phi^d\circ\phi^b\circ\phi(\alpha_1)} \forall(\gamma Q_2^b)\ \delta'$. Let ϕ' be $\phi^d \circ \phi^b$, α_2 be $\phi' \circ \phi(\alpha_1)$ (**16**) and Q_2' be $Q_2^a \otimes_{\alpha_2} \forall(\gamma Q_2^b)\ \delta'$ (**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 $\psi(\alpha_1) = \alpha_1$).
<u>P-III</u>: 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 4.4.7, 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 4.4.13

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

$$(Q_1\beta Q_2 P)\ \langle\langle t_a\rangle\rangle : (Q_1^a\beta Q_2^a P^a, \alpha_a)\ (\mathbf{2}) \qquad (Q_1^a\beta Q_2^a P^a)\ \langle\langle t_b\rangle\rangle : (Q_1^b\beta Q_2^b P^b, \alpha_b)\ (\mathbf{3})$$
$$(Q_1'\beta Q_2' P', \alpha) = Q_1^b\beta Q_2^b P^b \otimes_\alpha \alpha_a \to \alpha_b\ (\mathbf{4})$$

By induction hypothesis applied to (2) and (3), we get $(Q_3P)\ \langle\langle t_a\rangle\rangle : (Q_3^aP^a, \alpha_a)$ with $Q_1^aQ_2^a \approx Q_3^a$ and $(Q_3^aP^a)\ \langle\langle t_b\rangle\rangle : (Q_3^bP^b, \alpha_b)$ with $Q_1^bQ_2^b \approx Q_3^b$. Let Q_3'' be $Q_3^bP^b \oplus_\alpha \alpha_a \to \alpha_b$ (**5**). We have $(Q_3P)\ \langle\langle t\rangle\rangle : (Q_3'', \alpha)$.

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_1^b and Q_2' is Q_2^b and $\alpha \notin \mathsf{dom}(Q_1'Q_2')$. It follows from (4) that α was inserted in at most one of Q_1^b, Q_2^b, or P^b, and as left as possible. If α was inserted in Q_1^b or Q_2^b, leaving P' equal to P_b then (5) would also insert α in Q_3^b, 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 $Q_1^bQ_2^b$ and was inserted in P^b leading to P'. Thus (5) could not either insert α in Q_3^b 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 $Q_1^a\beta Q_2^aP^a \otimes_\alpha \forall(\gamma P^b)\ \alpha'$ and $(Q_1\beta Q_2 P\gamma)\ \langle\langle t_a\rangle\rangle :$ $(Q_1^a\beta Q_2^aP^a\gamma P^b, \alpha')$ (**6**) By induction hypothesis applied to (6), we get $(Q_3P\gamma)\ \langle\langle t_a\rangle\rangle : (Q_3^aP^a\gamma P^b, \alpha')$ with $Q_1^aQ_2^a \approx Q_3^a$. Let $Q_3'P'$ be $Q_3^aP^a \otimes_\alpha \forall(\gamma P^b)\ \alpha'$. 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 4.4.10 and Inst, it suffices to show $e\mathsf{ML^F} :: (Q)$ $\Gamma \vdash a' : \sigma$ for *one* σ in $\langle\langle t\rangle\rangle$ instead of *all* of them. By default, we let typing judgment be in $e\mathsf{ML^F}$.
∘ Case 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 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, α_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 eEqu-Var. Thus, by Inst and eEqu-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 eEqu-Free. Consequently, $(Q)\ \Gamma \vdash a_2' : \forall(Q_2)\ \alpha_1$ (**7**) holds by Inst.

From App* (Section 3.4.1), (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 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 $(QQ_2)\ \Gamma, x : \forall(Q_1)\ \alpha_1 \vdash a_1' :$

α_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 *e*EQU-FREE. Then, the following holds:

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

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

∘ CASE 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 4.4.13 and (12), there exists Q'_3 such that $(\emptyset)\ \langle\langle t'\rangle\rangle : (Q_3, \beta)$ holds with $Q_3 \approx Q_1Q_2$ (**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 $(Q\alpha)\ \forall(Q_3)\ \beta \equiv \forall(Q_1Q_2)\ \beta$. Thus, $(Q\alpha)\ \Gamma \vdash a' : \forall(Q_1Q_2)\ \beta$ holds by INST. By GEN, we get $(Q)\ \Gamma \vdash a' : \forall(\alpha Q_1Q_2)\ \beta$. By *e*EQU-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 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 *e*EQU-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:

$$\begin{array}{lll} (Q) & \forall(Q_1\beta Q_2)\ \gamma & \\ = & \forall(Q_1)\ \forall(\beta \geq \bot)\ \forall(Q_2)\ \gamma & \text{by notation} \\ \sqsubseteq & \forall(Q_1)\ \forall(\beta \geq \alpha)\ \forall(Q_2)\ \gamma & \text{by } e\text{INS-BOT and flexible congruence} \\ \equiv & \forall(Q_1)\ \psi(\forall(Q_2)\ \gamma) & \text{by } e\text{EQU-MONO and } e\text{EQU-FREE} \\ = & \forall(\psi(Q_1Q_2))\ \psi(\gamma) & \end{array}$$

Therefore, $(Q)\ \Gamma \vdash a' : \forall(\psi(Q_1Q_2))\ \psi(\gamma)$ (**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 4.4.13, there exists Q_3 such that $(Q')\ \langle\langle t_0\rangle\rangle : (Q'_3, \gamma)$ with $Q'_3 \approx Q_1Q_2$ (**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 4.4.11.ii, 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 (26).

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 4.5.2

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

Proof of Lemma 4.5.3

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 4.5.2.iii. 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 4.5.6

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 *e*Equ-Free: We may as well add or remove useless binders in an ML type scheme.

∘ Case *e*Equ-Comm: We may as well commute binders in an ML type scheme.

∘ Case *e*Equ-Var: By definition, $\langle\!\langle \forall(\alpha \geq \sigma)\ \alpha \rangle\!\rangle$ and $\langle\!\langle \sigma \rangle\!\rangle$ are identical.

∘ Case *e*Equ-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 *e*Ins-Rigid cannot occur since it mentions a rigid binding, whereas the derivation is assumed to be flexible.

∘ Case *e*Ins-Bot: $\forall(\alpha)\ \alpha \leq_{\mathsf{ML}} \sigma$ holds for any ML type σ.

∘ Case *e*Ins-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 *e*Ins-Abstract: The premise is $(Q)\ \sigma_1 \sqsubseteq\!\!\!\!- \sigma_2$. Necessarily, we have $(Q)\ \sigma_1 \equiv \sigma_2$ (Rule *e*Abs-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 4.5.4 and (7). The second step (9) is by induction hypothesis applied to (4). □

Proof of Lemma 4.5.7

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 4.5.6.ii. 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 4.5.7 and 4.5.3. □

References

[1] Henk P. Barendregt, The Lambda Calculus: Its Syntax and Semantics, 0-444-86748-1revised ed., North-Holland, 1984

[2] H.-J. Boehm, Partial polymorphic type inference is undecidable, in: 26th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, October 1985, pp. 339–345, ISBN: 0-8186-0644-4.

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

[4] Sébastien Carlier, Jeff Polakow, J.B. Wells, A.J. Kfoury, System e: expansion variables for flexible typing with linear and non-linear types and intersection types, in: David A. Schmidt (Ed.), Prooceedings of the 13th European Symposium on Programming, Lecture Notes in Computer Science, vol. 2986, Springer, 2004, pp. 294–309, ISBN: 978-3-540-21313-0.

[5] Gilles Dowek, Thérèse Hardin, Claude Kirchner, Frank Pfenning, Higher-order unification via explicit substitutions: the case of higher-order patterns, in: M. Maher (Ed.), Joint International Conference and Symposium on Logic Programming, 1996, pp. 259–273.

[6] Jacques Garrigue, Didier Rémy, Extending ML with Semi-Explicit higher-order polymorphism, Journal of Functional Programming 155 (1999) 134–169.A preliminary version appeared in TACS'97
[7] P. Giannini, S. Ronchi Della Rocca, Characterization of typings in polymorphic type discipline, in: Third Annual Symposium on Logic in Computer Science, IEEE, 1988, pp. 61–70.
[8] Jean-Yves Girard, Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur, Thèse d'état, University of Paris VII, 1972.
[9] Haruo Hosoya, Benjamin C. Pierce, How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.
[10] Trevor Jim, A polar type system, in: ICALP Satellite Workshops, 2000, pp. 323–338.
[11] 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.
[12] S. Peyton Jones, M. Shields, Lexically scoped type variables, March 2004. Available from: <http://research.microsoft.com/∼simonpj/papers/scoped-tyvars/>.
[13] S. Peyton Jones, M. Shields, Practical type inference for arbitrary-rank types, 2004.
[14] A.J. Kfoury, 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, June 1994, pp. 196–207.
[15] Konstantin Läufer, Martin Odersky, Polymorphic type inference and abstract data types, ACM Transactions on Programming Languages and Systems 16 (5) (1994) 1411–1430.
[16] Didier Le Botlan, MLF: Une extension de ML avec polymorphisme de second ordre et instanciation implicite, Ph.D. thesis, Ecole Polytechnique, May 2004, 326 pp.
[17] Didier Le Botlan, Didier Rémy, MLF: raising ML to the power of System-F, in: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, August 2003, pp. 27–38.
[18] Daan Leijen, A type directed translation of MLF to System F, in: The International Conference on Functional Programming (ICFP'07), ACM Press, October 2007.
[19] Daan Leijen, HMF: simple type inference for first-class polymorphism, in: The 13th ACM Symposium of the International Conference on Functional Programming (ICFP'08), Victoria, BC, Canada, September 2008, Extended version available as Microsoft Research Technical Report MSR-TR-2007-118, September 2007.
[20] Daan Leijen, Flexible types: robust type inference for first-class polymorphism, in: Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL'09), ACM, New York, NY, USA, 2009, pp. 66–77. ISBN:978-1-60558-379-2, doi:http://doi.acm.org/10.1145/1480881.1480891.
[21] Daan Leijen, Andres Löh, Qualified types for MLF, in: ICFP'05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming, ACM Press, New York, NY, USA, September 2005, pp. 144–155, ISBN: 1-59593-064-7.
[22] Daniel Leivant, Discrete polymorphism, in: Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP 90), ACM, New York, NY, USA, 1990, pp. 288–297, ISBN: 0-89791-368-X. doi: http://doi.acm.org/10.1145/91556.91675.
[23] David McAllester, A logical algorithm for ML type inference, in: Rewriting Techniques and Applications 14th International Conference (RTA 2003), Lecture Notes in Computer Science, vol. 2706, Springer-Verlag, Valencia, Spain, June 2003, pp. 436–451.
[24] Dale Miller, Unification under a mixed prefix, Journal of Symbolic Computation 14 (1992) 321–358.
[25] R. Milner, A theory of type polymorphism in programming, Journal of Computer and System Sciences 17 (1978) 348–375.
[26] John C. Mitchell, Polymorphic type inference and containment, Information and Computation 2/3 (76) (1988) 211–249.
[27] Martin Odersky, Konstantin Läufer, Putting type annotations to work, in: Proceedings of the 23rd ACM Conference on Principles of Programming Languages, January 1996, pp. 54–67.
[28] Martin Odersky, Christoph Zenger, Matthias Zenger, Colored local type inference, ACM SIGPLAN Notices 36 (3) (2001) 41–53.
[29] James William O'Toole Jr., David K. Gifford, Type reconstruction with first-class polymorphic values, in: SIGPLAN'89 Conference on Programming Language Design and Implementation, ACM, Portland, Oregon, June 1989.
[30] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, Mark Shields, Practical type inference for arbitrary-rank types, Journal of Functional Program 17 (1) (2007) 1–82.ISSN: 0956-7968
[31] Frank Pfenning, Partial polymorphic type inference and higher-order unification, in: Proceedings of the ACM Conference on Lisp and Functional Programming, ACM Press, July 1988, pp. 153–163.
[32] Frank Pfenning, On the undecidability of partial polymorphic type reconstruction, Fundamenta Informaticae 19(1–2) (1993) 185–199, Preliminary version available as Technical Report CMU-CS-92-105, School of Computer Science, Carnegie Mellon University, January 1992.
[33] Benjamin C. Pierce, Types and Programming Languages, The MIT Press, Massachusetts Institute of Technology Cambridge, Massachusetts 02142, 2002, ISBN: 0-262-16209-1.
[34] Benjamin C. Pierce, Bounded quantification is undecidable, Information and Computation 112 (1) (1994) 131–165.
[35] Benjamin C. Pierce, Programming with Intersection Types and Bounded Polymorphism, Ph.D. thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science Technical Report CMU-CS-91-205.
[36] Benjamin C. Pierce, David N. Turner, Local type inference, ACM Transactions on Programming Languages and Systems 22 (1) (2000) 1–44.
[37] Frantois Pottier, Didier Rémy, The essence of ML type inference, in: Benjamin C. Pierce (Ed.), Advanced Topics in Types and Programming Languages, MIT Press, 2005, pp. 389–489 (Chapter 10).
[38] Frantois Pottier, Yann RTgis-Gianas, Stratified type inference for generalized algebraic data types, in: Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL'06), Charleston, South Carolina, January 2006, pp. 232–244.
[39] Frantois Pottier, Didier Rémy, The essence of ML type inference, Extended preliminary version of [37], 2003.
[40] 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.
[41] Didier Rémy, Programming objects with ML-ART: an extension to ML with abstract and record types, in: Masami Hagiya, John C. Mitchell (Eds.), Theoretical Aspects of Computer Software, Lecture Notes in Computer Science, vol. 789, Springer-Verlag, April 1994, pp. 321–346.
[42] Didier Rémy, Boris Yakobowski, A graphical presentation of MLF types with a linear-time unification algorithm, in: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI'07), ACM Press, Nice, France, January 2007, pp. 27–38, ISBN: 1-59593-393-X.
[43] Didier Rémy, Boris Yakobowski, From ML to MLF: graphic type constraints with efficient type inference, in: The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08), Victoria, BC, Canada, September 2008, pp. 63–74.
[44] Didier Rémy, Boris Yakobowski, A church-style intermediate language for MLF, September 2008. Available from: <http://gallium.inria.fr/∼remy/mlf/xmlf.pdf/>.
[45] John C. Reynolds, Towards a theory of type structure, in: Proceedings of the Colloque sur la Programmation, LNCS, vol. 19, Springer-Verlag, New York, 1974, pp. 408–425.
[46] 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, New York, NY, 1998, pp. 279–288.
[47] Dimitrios Vytiniotis, Stephanie Weirich, Simon Peyton Jones, Boxy types: inference for higher-rank types and impredicativity, in: ICFP'06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming, ACM Press, New York, NY, USA, 2006, pp. 251–262, ISBN: 1-59593-309-3.
[48] Dimitrios Vytiniotis, Stephanie Weirich, Simon Peyton Jones, FPH: first-class Polymorphism for Haskell, in: The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08), Victoria, BC, Canada, September 2008.

[49] J.B. Wells, The essence of principal typings, in: Proceedings of the 29th International Colloque Automata, Languages, and Programming, LNCS, vol. 2380, Springer-Verlag, 2002, pp. 913–925, ISBN: 3-540-43864-5.
[50] J.B. Wells, Typability and type checking in System F are equivalent and undecidable, Annals of Pure Applied Logic 98 (1–3) (1999) 111–156.
[51] Joe B. Wells, Type Inference for System F with and without the Eta Rule, Ph.D. thesis, Boston University, 1996.
[52] Boris Yakobowski, Graphical Types and Constraints: Second-Order Polymorphism and Inference, Ph.D. thesis, University of Paris 7, December 2008.