$\mathsf{ML^F}$
Raising ML to the Power of System F

Didier Le Botlan and Didier Rémy
INRIA-Rocquencourt
78153 Le Chesnay Cedex, France
{Didier.Le_Botlan,Didier.Remy}@inria.fr

Abstract

We propose a type system $\mathsf{ML^F}$ that generalizes ML with first-class polymorphism as in System F. Expressions may contain second-order type annotations. Every typable expression admits a principal type, which however depends on type annotations. Principal types capture all other types that can be obtained by implicit type instantiation and they can be inferred. All expressions of ML are well-typed without any annotations. All expressions of System F can be mechanically encoded into $\mathsf{ML^F}$ by dropping all type abstractions and type applications, and injecting types of lambda-abstractions into $\mathsf{ML^F}$ types. Moreover, only parameters of lambda-abstractions that are used polymorphically need to remain annotated.

Categories and Subject Descriptors: D.3.3 Language Constructs and Features.

General Terms: Theory, Languages.

Keywords: Type Inference, First-Class Polymorphism, Second-Order Polymorphism, System F, ML, Type Annotations.

The quest for type inference with first-class polymorphic types

Programming languages considerably benefit from static type-checking. In practice however, types may sometimes trammel programmers, for two opposite reasons. On the one hand, type annotations may quickly become a burden to write; while they usefully serve as documentation for toplevel functions, they also obfuscate the code when every local function must be decorated. On the other hand, since types are only approximations, any type system will reject programs that are perfectly well-behaved and that could be accepted by another more expressive one; hence, sharp programmers may be irritated in such situations.

Fortunately, solutions have been proposed to both of these problems. Type inference allows to elide most type annotations, which relieves the programmer from writing such details and simultaneously lightens programs. In parallel, more expressive type systems have been developed, so that programmers are less often exposed to their limitations.

Unfortunately, those two situations are often conflicting. Expressive type systems tend to require an unbearable amount of type decorations, thus many of them only remained at the status of prototypes. Indeed, full type inference for System F is undecidable [26]. Conversely, languages with simple type inference are still limited in expressiveness; more sophisticated type inference engines, such as those with subtyping constraints or higher-order unification have not yet been proved to work well in practice.

The ML language [4] appears to be a surprisingly stable point of equilibrium between those two forces: it combines a reasonably powerful yet simple type system and comes with an effective type inference engine. Besides, the ML experience made it clear that expressiveness of the type system and a significant amount of type inference are equally important.

Despite its success, ML could still be improved: indeed, there are real examples that require first-class polymorphic types [25, 20, 7] and, even though these may not occur too frequently, ML does not offer any reasonable alternative. (The inconvenience is often underestimated, since the lack of a full-fledged language to experiment with first-class polymorphism insidiously keeps programmers thinking in terms of ML polymorphism.)

A first approach is to extend ML with first-class second-order polymorphism [15, 25, 20, 7]. However, the existing solutions are still limited in expressiveness and the amount of necessary type declarations keeps first-class polymorphism uneasy to use.

An alternative approach, initiated by Cardelli [2], is to start with an expressive but explicitly typed language, say $F^{\omega}_{<:}$, and perform a sufficient amount of type inference, so that simple programs—ideally including all ML programs—would not need any type annotation at all. This lead to *local type inference* [24], recently improved to *colored* local type inference [21]. These solutions are quite impressive. In particular, they include subtyping in combination with higher-order polymorphism. However, they fail to type all ML programs. Moreover, they also fail to provide an intuitive and simple specification of where type annotations are mandatory.

In this work, we follow the first approach. At least, by being conservative over ML, we are guaranteed to please programmers who are already quite happy with ML[1]. We build on some previous work [7], which has been used to add polymorphic methods to OCaml [16]. Here, we retain the same primary goal, that is to type all expressions of System-F, providing explicit annotations when needed, and to keep all expressions of ML unannotated. In addition, we aim at the elimination of all backward coercions from polymorphic types to ML-types. In particular, our goal is not to guess polymorphic types.

Our track

Church's style System-F and ML are quite different in nature. In ML, the elimination of polymorphism is implicitly performed at every use occurrence of a variable bound with a polymorphic type $\forall\bar{\alpha}.\tau$, which can then be given any instance, of the form $\tau[\bar{\tau}/\bar{\alpha}]$. Indeed, a polymorphic type somehow represents the set of its instances. This induces an instance relation between polymorphic types themselves. For example, the (polymorphic) type $\forall(\alpha)\ \alpha \to \alpha$ is said to be more general than $\forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to (\alpha \to \beta)$ and we write $\forall(\alpha)\ \alpha \to \alpha \preccurlyeq \forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to (\alpha \to \beta)$ because all instances of the latter are also instances of the former.

Conversely, in Church's style System F, a type $\forall(\alpha)\ \alpha \to \alpha$ only stands for itself (modulo renaming of bound variables) and the elimination of polymorphism must be performed explicitly by type application (and abstraction) at the source level. A counter-part in System F is that bound type variables may be instantiated by polymorphic types, allowing for expressive impredicative second-order types. For example, an expression of type $\forall(\alpha)\ \alpha \to \alpha$ can be given type $(\forall(\beta)\ \beta \to \beta) \to (\forall(\beta)\ \beta \to \beta)$ by an explicit type-application to $\forall(\beta)\ \beta \to \beta$.

Unfortunately, combining implicit instantiation of polymorphic types with second-order types raises conflicts almost immediately. For illustration, consider the application of the function `choose`, defined as $\lambda(x)\ \lambda(y)$ `if` *true* `then` x `else` y, to the identity function `id`. In ML, `choose` and `id` have principal types $\forall(\alpha)\ \alpha \to \alpha \to \alpha$ and $\forall(\alpha)\ \alpha \to \alpha$, respectively. For conciseness, we shall write $\mathtt{id}_\alpha$ for $\alpha \to \alpha$ and $\sigma_{\mathtt{id}}$ for $\forall(\alpha)\ \mathtt{id}_\alpha$. Should `choose id` have type σ_1 equal to $\forall(\alpha)\ \mathtt{id}_\alpha \to \forall(\alpha)\ \mathtt{id}_\alpha$, obtained by keeping the type of `id` uninstantiated? Or, should it have type σ_2 equal to $\forall(\alpha)\ (\mathtt{id}_\alpha \to \mathtt{id}_\alpha)$, obtained by instantiating the type of `id` to the monomorphic type $\mathtt{id}_\alpha$ and generalizing α only at the end? Indeed, both σ_1 and σ_2 are correct types for `choose id`. However, neither one is more general than the other in System F. Indeed, the function `auto` defined as $\lambda(x : \sigma_{\mathtt{id}})\ x\ x$ can be typed with σ_1, as `choose id`, but not with σ_2; otherwise `auto` could be applied, for instance, to the successor function, which would lead to a runtime error. Hence, σ_1 cannot be safely coerced to σ_2. Conversely, however, there is a retyping function—a function whose type erasure η-reduces to the identity [19]—from type σ_2 to type σ_1, namely, $\lambda(g : \sigma_2)\ \lambda(x : \sigma_{\mathtt{id}})\ \lambda(\alpha)\ g\ \alpha\ (x\ \alpha)$. Actually, σ_2 is a principal type for `choose id` in $F^{\eta *}$ (System F closed by η-expansion) [19].

While the argument of `auto` must be at least as polymorphic as $\sigma_{\mathtt{id}}$, the argument of the function `choose id` need not be polymorphic: it may be any instance τ of $\sigma_{\mathtt{id}}$ and the type of the return value is then τ. We could summarize these constraints by saying that:

$$\begin{array}{rl} \mathtt{auto}: & \forall(\alpha = \sigma_{\mathtt{id}})\ \alpha \to \alpha \\ \mathtt{choose\ id}: & \forall(\alpha \geq \sigma_{\mathtt{id}})\ \alpha \to \alpha \end{array}$$

The type given to `choose id` captures the intuition that this application has type $\tau \to \tau$ for any instance τ of $\sigma_{\mathtt{id}}$. This form of quantification allows to postpone the decision of whether $\sigma_{\mathtt{id}}$ should be instantiated as soon as possible or kept polymorphic as long as possible. The bound of α in $\forall(\alpha \geq \sigma_{\mathtt{id}})\ \alpha \to \alpha$, which is said to be *flexible*, can be weakened either by instantiating $\sigma_{\mathtt{id}}$ or by replacing $\geq$ by $=$. Both forms of weakening can be captured by an appropriate instance relation $\preccurlyeq$ between types. In a binder of the form $(\alpha = \sigma)$ the bound σ, which is said to be *rigid*, cannot be instantiated any longer. Intuitively, the type $\forall(\alpha = \sigma)\ \sigma'$ stands for the System-F type $\sigma'[\sigma/\alpha]$.

Finally, both `choose id succ` and `choose id auto` are well-typed, taking `int` $\to$ `int` or $\sigma_{\mathtt{id}}$ for the type of α, respectively. In fact, the type $\forall(\alpha \geq \sigma_{\mathtt{id}})\ \alpha \to \alpha$ happens to be a principal type for `choose id` in ML^{F}. This type summarizes in a compact way the part of typechecking `choose id` that depends on the context in which it will be used: some typing constraints have been resolved definitely and forgotten; others, such as "α is any instance of $\sigma_{\mathtt{id}}$", are kept unresolved. In short, ML^{F} provides richer types with constraints on the bounds of variables so that instantiation of these variables can be delayed until there is a principal way of instantiating them.

A technical road-map

The instantiations between types used above remain to be captured formally within an appropriate relation $\preccurlyeq$. Indeed, the instance relation plays a crucial role in type inference, via a typing rule INST stating that any expression a of type σ in a context Γ has also type σ' in the same context whenever $\sigma \preccurlyeq \sigma'$. Intuitively, the larger the relation $\preccurlyeq$ is, the more flexibility is left for inference, and, usually, the harder the inference algorithm is.

Unsurprisingly, the "smallest reasonable relation" $\preccurlyeq$ that validates all the instantiations used above leads to undecidable type inference, for full type inference in System-F is undecidable. Still, the relation $\preccurlyeq$ induces an interesting variant $\mathrm{UML}^{\mathrm{F}}$ that has the same expressiveness as ML^{F} but requires no type annotations at all. Fortunately, the relation $\preccurlyeq$ can be split into a composition of relations $\sqsupseteq\sqsubseteq\sqsupseteq$ where uses of the relation $\sqsubseteq$ can be inferred as long as all applications of $\sqsupseteq$ are fully explicit: this sets a clear distinction between explicit and inferred type information, which is the essence of ML^{F}.

Unfortunately, subject reduction does not hold in ML^{F} for a simple notion of reduction (non local computation of annotations would be required during reduction). Thus, we introduce an intermediate variant $\mathrm{ML}^{\mathrm{F}}_{\star}$ where only place holders for $\sqsupseteq$ are indicated. For example, using the symbol $\star$ in place of polytypes, $\lambda(x : \star)\ x\ x$ belongs to $\mathrm{ML}^{\mathrm{F}}_{\star}$ since $\lambda(x : \sigma_{\mathtt{id}})\ x\ x$ belongs to ML^{F} (and of course, $\lambda(x)\ x\ x$ belongs to $\mathrm{UML}^{\mathrm{F}}$). Subject reduction and progress are proved for $\mathrm{ML}^{\mathrm{F}}_{\star}$ and type soundness follows for $\mathrm{ML}^{\mathrm{F}}_{\star}$ and, indirectly, for ML^{F}.

In fact, we abstract the presentation of $\mathrm{ML}^{\mathrm{F}}_{\star}$ over a collection of primitives so that ML^{F} can then be embedded into $\mathrm{ML}^{\mathrm{F}}_{\star}$ by treating type-annotations as an appropriate choice of primitives and disallowing annotation place holders in source terms. Thus, although our practical interest is the system ML^{F}, most of the technical developments are pursued in $\mathrm{ML}^{\mathrm{F}}_{\star}$.

Unsurprisingly, neither $\mathrm{UML}^{\mathrm{F}}$ nor $\mathrm{ML}^{\mathrm{F}}_{\star}$ admits principal types. Conversely, *every expression typable in* ML^{F} *admits a principal type*. Of course, principal types depend on type annotations in

[1] On a practical level, this would also ensure upward compatibility of existing code, although translating tools could always be provided.

Figure 1. Syntax of Types

$\tau ::= \alpha \mid g^n\ \tau_1\ ..\ \tau_n$	Monotypes
$\sigma ::= \tau \mid \bot \mid \forall\,(\alpha \geq \sigma)\ \sigma \mid \forall\,(\alpha = \sigma)\ \sigma$	Polytypes

the source term. More precisely, if an expression is not typable in ML^{F}, it may sometimes be typable by adding extra type annotations. Moreover, two different type annotations may lead to two incomparable principal types. As an example, the expression $\lambda(x)\ x\ x$ is not typable in ML^{F}, while both expressions $\lambda(x : \forall\alpha.\alpha)\ x\ x$ and $\lambda(x : \forall\alpha.\alpha \to \alpha)\ x\ x$ are typable, with incomparable types. Adding a (polymorphic) type annotation to a typable expression may also lead to a new type that is not comparable with the previous one. This property should not be surprising since it is inherent to second-order polymorphism, which we keep explicit—remember that we only infer first-order polymorphism in the presence of second-order types. Still, the gain is the elusion of most type annotations, via the instance relation $\sqsubseteq$.

The paper is organized as follows. In Section 1, we describe types and instance relations $\sqsubseteq$ and $\sqsubseteq$. The syntax and the static and dynamic semantics of $\mathrm{ML}^{\mathrm{F}}_{\star}$ are described in Section 2. Section 3 presents formal properties, including type soundness for $\mathrm{ML}^{\mathrm{F}}_{\star}$ and type inference for ML^{F}. Section 4 introduces explicit type annotations. A comparison with System-F is drawn is Section 5. In Section 6, we discuss expressiveness, language extensions, and related works. For the sake of readability, unification and type inference algorithms have been moved to the appendices. Due to lack of space, all proofs are omitted.

"Monomorphic abstraction of polymorphic types"

In our proposal, ML-style polymorphism, as in the type of `choose` or `id`, can be fully inferred. (We will show that all ML programs remain typable without type annotations.) Unsurprisingly, some polymorphic functions cannot be typed without annotations. For instance, $\lambda(x)\ x\ x$ cannot be typed in ML^{F}. In particular, we do not infer types for function arguments that are used polymorphically. Fortunately, such arguments can be annotated with a polymorphic type, as illustrated in the definition of `auto` given above. Once defined, a polymorphic function can be manipulated by another unannotated function, as long as the latter does not *use* polymorphism, which is then retained. This is what we qualify "*monomorphic abstraction of polymorphic types*". For instance, both `id auto` and `choose id auto` remain of type $\forall\,(\alpha = \sigma_{\mathtt{id}})\ \alpha \to \alpha$ (the type $\sigma_{\mathtt{id}}$ of `auto` is never instantiated) and neither `choose` nor `id` require any type annotation. Finally, polymorphic functions can be used by implicit instantiation, much as in ML.

To summarize, a key feature of ML^{F} is that type variables can *always* be implicitly instantiated by polymorphic types. This can be illustrated by the killer-app(lication) $(\lambda(x)\ x\ \mathtt{id})\ \mathtt{auto}$. This expression is typable in ML^{F} as such, that is without any type application nor any type annotation—except, of course, in the definition of `auto` itself. In fact, a generalization of this example is the `app` function $\lambda(f)\ \lambda(x)\ f\ x$, whose ML^{F} principal type is $\forall\,(\alpha,\beta)\ (\alpha \to \beta) \to \alpha \to \beta$. It is remarkable that whenever $a_1\ a_2$ is typable in ML^{F}, so is $\mathtt{app}\ a_1\ a_2$, without any type annotation nor any type application. This includes, of course, cases where a_1 expects a polymorphic value as argument, such as in `app auto id`. We find such examples quite important in practice, since they model iterators (*e.g.* `app`) applied to polymorphic functions (*e.g.* `auto`) over structures holding polymorphic values (*e.g.* `id`).

1 Types

1.1 Syntax of types

The syntax of types is given in Figure 1. The syntax is parameterized by an enumerable set of type variables $\alpha \in \vartheta$ and a family of type symbols $g \in \mathcal{G}$ given with their arity $|g|$. To avoid degenerated cases, we assume that $\mathcal{G}$ contains at least a symbol of arity two (the infix arrow $\to$). We write g^n if g is of arity n. We also write $\overline{\tau}$ for tuples of types. The polytype $\bot$ corresponds to $(\forall\alpha.\alpha)$, intuitively. More precisely, it will be made equivalent to $\forall\,(\alpha \geq \bot)\ \alpha$.

We distinguish between *monotypes* and *polytypes*. By default, types refer to the more general form, *i.e.* to polytypes. As in ML, monotypes do not contain quantifiers. Polytypes generalize ML type schemes. Actually, ML type schemes can be seen as polytypes of the form $\forall\,(\alpha_1 \geq \bot) \ldots \forall\,(\alpha_n \geq \bot)\ \tau$ with outer quantifiers. Inner quantifiers as in System F cannot be written directly inside monotypes. However, they can be simulated with types of the form $\forall\,(\alpha = \sigma)\ \sigma'$, which stands, intuitively, for the polytype σ' where all occurrences of α would have been replaced by the polytype σ. However, our notation contains additional meaningful sharing information. Finally, the general form $\forall\,(\alpha \geq \sigma)\ \sigma'$ intuitively stands for the collection of *all polytypes* σ' *where* α *is an instance of* σ.

Notation. We say that α has a *rigid bound* in $(\alpha = \sigma)$ and a *flexible bound* in $(\alpha \geq \sigma)$. A particular case of flexible bound is the *unconstrained bound* $(\alpha \geq \bot)$, which we abbreviate as (α). For convenience, we write $(\alpha \diamond \sigma)$ for either $(\alpha = \sigma)$ or $(\alpha \geq \sigma)$. The symbol $\diamond$ acts as a meta-variable and two occurrences of $\diamond$ in the same context mean that they all stand for $=$ or all stand for $\geq$. To allow independent choices we use indices $\diamond_1$ and $\diamond_2$ for unrelated occurrences.

Conversion and free variables. Polytypes are considered equal modulo α-conversion where $\forall\,(\alpha \diamond \sigma)\ \sigma'$ binds α in σ', but not in σ. The set of free variables of a polytype σ is written $\mathsf{ftv}(\sigma)$ and defined inductively as follows:

$$\mathsf{ftv}(\alpha) = \{\alpha\} \qquad \mathsf{ftv}(g^n\ \tau_1 \ldots \tau_n) = \bigcup_{i=1..n} \mathsf{ftv}(\tau_i) \qquad \mathsf{ftv}(\bot) = \emptyset$$

$$\mathsf{ftv}(\forall\,(\alpha \diamond \sigma)\ \sigma') = \begin{cases} \mathsf{ftv}(\sigma') & \text{if } \alpha \notin \mathsf{ftv}(\sigma') \\ \mathsf{ftv}(\sigma') \setminus \{\alpha\} \cup \mathsf{ftv}(\sigma) & \text{otherwise} \end{cases}$$

The capture-avoiding substitution of α by τ in σ is written $\sigma[\tau/\alpha]$.

EXAMPLE 1. The syntax of types only allows quantifiers to be outermost, as in ML, or in the bound of other bindings. Therefore, the type $\forall\alpha \cdot (\forall\beta \cdot (\tau[\beta] \to \alpha)) \to \alpha$ of System F^2 cannot be written directly. (Here, $\tau[\beta]$ means a type τ in which the variable β occurs.) However, it could be represented by the type $\forall\,(\alpha)\ \forall\,(\beta' = \forall\,(\beta)\ \tau[\beta] \to \alpha)\ \beta' \to \alpha$. In fact, all types of System F can easily be represented as polytypes by recursively binding all occurrences of inner polymorphic types to fresh variables beforehand— an encoding from System F into ML^{F} is given in Section 5.1.

Types may be instantiated implicitly as in ML along an instance relation $\preccurlyeq$. As explained above, we decompose $\preccurlyeq$ into $\exists\sqsubseteq\exists$. In

[2] We write $\forall\alpha \cdot \tau$ for types of System F, so as to avoid confusion.

Section 1.2, we first define an equivalence relation between types, which is the kernel of both $\mathrel{-\!\!\!\sqsubset}$ and $\sqsubseteq$. In Section 1.3, we define the relation $\mathrel{-\!\!\!\sqsubset}$ that is the inverse of $\mathrel{\sqsupset\!\!\!-}$. The instance relation $\sqsubseteq$, which contains $\mathrel{-\!\!\!\sqsubset}$, is defined in Section 1.4.

1.2 Type equivalence

The order of quantifiers and some other syntactical notations are not always meaningful. Such syntactic artifacts are captured by a notion of type equivalence. Type equivalence and all other relations between types are relative to a prefix that specifies the bounds of free type variables.

DEFINITION 1 (PREFIXES). A *prefix* Q is a sequence of bindings $(\alpha_1 \diamond_1 \sigma_1)\ldots(\alpha_n \diamond_n \sigma_n)$ where variables $\alpha_1,\ldots\alpha_n$ are pairwise distinct and form the domain of Q, which we write $\mathsf{dom}(Q)$. The order of bindings in a prefix is significant: bindings are meant to be read from left to right; furthermore, we require that variables α_j do not occur free in σ_i whenever $i \leq j$. Since $\alpha_1,\ldots\alpha_n$ are pairwise distinct, we can unambiguously write $(\alpha\diamond\sigma) \in Q$ to mean that Q is of the form $(Q_1, \alpha\diamond\sigma, Q_2)$. We also write $\forall(Q)\,\sigma$ for the type $\forall(\alpha_1\diamond_1\sigma_1)\ldots\forall(\alpha_n\diamond_n\sigma_n)\,\sigma$. (Note that α_i's can be renamed in the type $\forall(Q)\,\sigma$, but not in the prefix Q.) □

DEFINITION 2 (EQUIVALENCE). The *equivalence under prefix* is a relation on triples composed of a prefix Q and two types σ_1 and σ_2, written $(Q)\;\sigma_1 \equiv \sigma_2$. It is defined as the smallest relation that satisfies the rules of Figure 2. We write $\sigma_1 \equiv \sigma_2$ for $(\emptyset)\;\sigma_1 \equiv \sigma_2$. □

Rule EQ-COMM allows the reordering of independent binders; Rule EQ-FREE eliminates unused bound variables. Rules EQ-CONTEXT-L and EQ-CONTEXT-R tell that $\equiv$ is a congruence; Reasoning under prefixes allows to break up a polytype $\forall(Q)\,\sigma$ and "look inside under prefix Q". For instance, it follows from iterations of Rule EQ-CONTEXT-R that $(Q)\;\sigma \equiv \sigma'$ suffices to show $(\emptyset)\;\forall(Q)\,\sigma \equiv \forall(Q)\,\sigma'$.

Rule EQ-MONO allows to *read* the bound of a variable from the prefix when it is (equivalent to) a monotype. An example of use of EQ-MONO is $(Q, \alpha = \tau_0, Q')\;\alpha \to \alpha \equiv \tau_0 \to \tau_0$. Rule EQ-MONO makes no difference between $\geq$ and $=$ whenever the bound is (equivalent to) a monotype. The restriction of Rule EQ-MONO to the case where σ_0 is (equivalent to) a monotype is required for the well-formedness of the conclusion. Moreover, it also disallows $(Q, \alpha = \sigma_0, Q')\;\alpha \equiv \sigma_0$ when τ is a variable α: variables with non trivial bounds must be treated abstractly and cannot be silently expanded. In particular, $(Q)\;\forall(\alpha = \sigma_0, \alpha' = \sigma_0)\,\alpha \to \alpha' \equiv \forall(\alpha = \sigma_0)\,\alpha \to \alpha$ does not hold.

Rule EQ-VAR expands into both $\forall(\alpha = \sigma)\,\alpha \equiv \sigma$ and $\forall(\alpha \geq \sigma)\,\alpha \equiv \sigma$. The former captures the intuition that $\forall(\alpha = \sigma)\,\sigma'$ stands for $\sigma'[\sigma/\alpha]$, which however, is not always well-formed. The latter may be surprising, since one could expect $\forall(\alpha \geq \sigma)\,\alpha \sqsubseteq \sigma$ to hold, but not the converse. The inverse part of the equivalence could be removed without changing the set of typable terms. However, it is harmless and allows for a more uniform presentation.

The equivalence $(Q)\;\forall(\alpha\diamond\tau)\,\sigma \equiv \sigma[\tau/\alpha]$ follows from Rules EQ-MONO, context rules, transitivity, and EQ-FREE, which we further refer to as the derived rule EQ-MONO*.

The equivalence under (a given) prefix is a symmetric operation. In other words, it captures reversible transformations. Irreversible transformations are captured by an *instance* relation $\sqsubseteq$. Moreover, we distinguish a subrelation $\mathrel{-\!\!\!\sqsubset}$ of $\sqsubseteq$ called *abstraction*. Inverse of abstractions are sound relations for $\preccurlyeq$ but made explicit so as to preserve type inference, while inverse of instance relations would, in general, be unsound for $\preccurlyeq$.

Figure 2. Type equivalence under prefix

All rules are considered symmetrically.

EQ-REFL
$$(Q)\;\sigma \equiv \sigma$$

EQ-TRANS
$$\frac{(Q)\;\sigma_1 \equiv \sigma_2 \qquad (Q)\;\sigma_2 \equiv \sigma_3}{(Q)\;\sigma_1 \equiv \sigma_3}$$

EQ-CONTEXT-R
$$\frac{(Q, \alpha\diamond\sigma)\;\sigma_1 \equiv \sigma_2}{(Q)\;\forall(\alpha\diamond\sigma)\,\sigma_1 \equiv \forall(\alpha\diamond\sigma)\,\sigma_2}$$

EQ-CONTEXT-L
$$\frac{(Q)\;\sigma_1 \equiv \sigma_2}{(Q)\;\forall(\alpha\diamond\sigma_1)\,\sigma \equiv \forall(\alpha\diamond\sigma_2)\,\sigma}$$

EQ-FREE
$$\frac{\alpha \notin \mathsf{ftv}(\sigma_1)}{(Q)\;\forall(\alpha\diamond\sigma)\,\sigma_1 \equiv \sigma_1}$$

EQ-COMM
$$\frac{\alpha_1 \notin \mathsf{ftv}(\sigma_2) \qquad \alpha_2 \notin \mathsf{ftv}(\sigma_1)}{(Q)\;\forall(\alpha_1\diamond_1\sigma_1)\,\forall(\alpha_2\diamond_2\sigma_2)\,\sigma \equiv \forall(\alpha_2\diamond_2\sigma_2)\,\forall(\alpha_1\diamond_1\sigma_1)\,\sigma}$$

EQ-VAR
$$(Q)\;\forall(\alpha\diamond\sigma)\,\alpha \equiv \sigma$$

EQ-MONO
$$\frac{(\alpha\diamond\sigma_0) \in Q \qquad (Q)\;\sigma_0 \equiv \tau_0}{(Q)\;\tau \equiv \tau[\tau_0/\alpha]}$$

Figure 3. The abstraction relation

A-EQUIV
$$\frac{(Q)\;\sigma_1 \equiv \sigma_2}{(Q)\;\sigma_1 \mathrel{-\!\!\!\sqsubset} \sigma_2}$$

A-TRANS
$$\frac{(Q)\;\sigma_1 \mathrel{-\!\!\!\sqsubset} \sigma_2 \qquad (Q)\;\sigma_2 \mathrel{-\!\!\!\sqsubset} \sigma_3}{(Q)\;\sigma_1 \mathrel{-\!\!\!\sqsubset} \sigma_3}$$

A-CONTEXT-R
$$\frac{(Q, \alpha\diamond\sigma)\;\sigma_1 \mathrel{-\!\!\!\sqsubset} \sigma_2}{(Q)\;\forall(\alpha\diamond\sigma)\,\sigma_1 \mathrel{-\!\!\!\sqsubset} \forall(\alpha\diamond\sigma)\,\sigma_2}$$

A-HYP
$$\frac{(\alpha_1 = \sigma_1) \in Q}{(Q)\;\sigma_1 \mathrel{-\!\!\!\sqsubset} \alpha_1}$$

A-CONTEXT-L
$$\frac{(Q)\;\sigma_1 \mathrel{-\!\!\!\sqsubset} \sigma_2}{(Q)\;\forall(\alpha = \sigma_1)\,\sigma \mathrel{-\!\!\!\sqsubset} \forall(\alpha = \sigma_2)\,\sigma}$$

1.3 The abstraction relation

DEFINITION 3. The *abstraction under prefix*, is a relation on triples composed of a prefix Q and two types σ_1 and σ_2, written[3] $(Q)\;\sigma_1 \mathrel{-\!\!\!\sqsubset} \sigma_2$, and defined as the smallest relation that satisfies the rules of Figure 3. We write $\sigma_1 \mathrel{-\!\!\!\sqsubset} \sigma_2$ for $(\emptyset)\;\sigma_1 \mathrel{-\!\!\!\sqsubset} \sigma_2$. □

Rules A-CONTEXT-L and A-CONTEXT-R are context rules; note that Rule A-CONTEXT-L does not allow abstraction under flexible bounds. The interesting rule is A-HYP, which replaces a polytype σ_1 by a variable α_1, provided α_1 is rigidly bound to σ_1 in Q.

Remarkably, rule A-HYP is not reversible. In particular, $(\alpha = \sigma) \in Q$ does not imply $(Q)\;\alpha \mathrel{-\!\!\!\sqsubset} \sigma$, unless σ is (equivalent to) a monotype. This asymmetry is essential, since uses of $\mathrel{-\!\!\!\sqsubset}$ will be inferred, but uses of $\mathrel{\sqsupset\!\!\!-}$ will not. Intuitively, the former consists in *abstracting* the polytype σ as the name α (after checking that α is declared as an alias for σ in Q). The latter consists in *revealing* the polytype abstracted by the name α. An abstract polytype, *i.e.* a variable bound to a polytype in Q, can only be manipulated by its name, *i.e.* abstractly. The polytype must be revealed *explicitly* (by using the relation $\mathrel{\sqsupset\!\!\!-}$) before it can be further instantiated (along the relation $\mathrel{-\!\!\!\sqsubset}$ or $\sqsubseteq$). (See also examples 6 and 7.)

[3] Read σ_2 is an abstraction of σ_1—or σ_1 is a revelation of σ_2—under prefix Q.

Figure 4. Type instance

$$\text{I-ABSTRACT}\quad \frac{(Q)\ \sigma_1 \mathrel{\sqsubseteq\!\!\!\!\!-} \sigma_2}{(Q)\ \sigma_1 \sqsubseteq \sigma_2} \qquad \text{I-TRANS}\quad \frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2 \quad (Q)\ \sigma_2 \sqsubseteq \sigma_3}{(Q)\ \sigma_1 \sqsubseteq \sigma_3} \qquad \text{I-CONTEXT-R}\quad \frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall(\alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \forall(\alpha \diamond \sigma)\ \sigma_2}$$

$$\text{I-HYP}\quad \frac{(\alpha_1 \geq \sigma_1) \in Q}{(Q)\ \sigma_1 \sqsubseteq \alpha_1} \qquad \text{I-CONTEXT-L}\quad \frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall(\alpha \geq \sigma_1)\ \sigma \sqsubseteq \forall(\alpha \geq \sigma_2)\ \sigma}$$

$$\text{I-BOT}\quad (Q)\ \bot \sqsubseteq \sigma \qquad \text{I-RIGID}\quad \frac{}{(Q)\ \forall(\alpha \geq \sigma_1)\ \sigma \sqsubseteq \forall(\alpha = \sigma_1)\ \sigma}$$

EXAMPLE 2. The abstraction $(\alpha = \sigma)\ \forall(\alpha = \sigma)\ \sigma' \mathrel{\sqsubseteq\!\!\!\!\!-} \sigma'$ is derivable: on the one hand, $(\alpha = \sigma)\ \sigma \mathrel{\sqsubseteq\!\!\!\!\!-} \alpha$ holds by A-HYP, leading to $(\alpha = \sigma)\ \forall(\alpha = \sigma)\ \sigma' \mathrel{\sqsubseteq\!\!\!\!\!-} \forall(\alpha = \alpha)\ \sigma'$ by A-CONTEXT-L; on the other hand, $(\alpha = \sigma)\ \forall(\alpha = \alpha)\ \sigma' \equiv \sigma'$ holds by EQ-MONO*. Hence, we conclude by A-EQUIV and A-TRANS.

1.4 The instance relation

DEFINITION 4. The *instance under prefix*, is a relation on triples composed of a prefix Q and two types σ_1 and σ_2, written[4] (Q) $\sigma_1 \sqsubseteq \sigma_2$. It is defined as the smallest relation that satisfies the rules of Figure 4. We write $\sigma_1 \sqsubseteq \sigma_2$ for $(\emptyset)\ \sigma_1 \sqsubseteq \sigma_2$. □

Rule I-BOT means that $\bot$ behaves as a least element for the instance relation. Rules I-CONTEXT-L and I-RIGID mean that flexible bounds can be instantiated and changed into rigid bounds. Conversely, instantiation cannot occur under rigid bounds, except when it is an abstraction, as described by Rule A-CONTEXT-L.

The interesting rule is I-HYP—the counter-part of rule A-HYP, which replaces a polytype σ_1 by a variable α_1, provided σ_1 is a flexible bound of α_1 in Q.

EXAMPLE 3. The instance relation $(\alpha \geq \sigma)\ \forall(\alpha \geq \sigma)\ \sigma' \sqsubseteq \sigma'$ holds. The derivation follows the one of Example 2 but uses I-HYP and I-CONTEXT-L instead of A-HYP and A-CONTEXT-L. More generally, $(QQ')\ \forall(Q')\ \sigma \sqsubseteq \sigma$ holds for any Q, Q', and σ, which we refer to as Rule I-DROP.

The relation $(Q)\ \forall(\alpha_1 \geq \forall(\alpha_2 \diamond \sigma_2)\ \sigma_1)\ \sigma \sqsubseteq \forall(\alpha_2 \diamond \sigma_2)\ \forall(\alpha_1 \geq \sigma_1)$ σ holds whenever $\alpha_2 \notin \mathsf{ftv}(\sigma)$, which we further refer to as the derived rule I-UP.

As expected, the equivalence is the kernel of the instance relation:

LEMMA 1 (EQUIVALENCE). *For any prefixes Q and types σ and σ', we have $(Q)\ \sigma \equiv \sigma'$ if and only if both $(Q)\ \sigma \sqsubseteq \sigma'$ and $(Q)\ \sigma' \sqsubseteq \sigma$ hold.*

The instance relation coincide with equivalence on monotypes, which captures the intuition that "monotypes are really monomorphic".

LEMMA 2. *For all prefixes Q and monotypes τ and τ', we have $(Q)\ \tau \sqsubseteq \tau'$ if and only if $(Q)\ \tau \equiv \tau'$.*

[4] Read σ_2 *is an instance of* σ_1—or σ_1 *is more general than* σ_2—*under prefix Q.*

The instance relation also coincides with the one of ML on ML-types. In particular, $\forall(\bar\alpha)\ \tau_0 \sqsubseteq \tau_1$ if and only if τ_1 is of the form $\tau_0[\bar\tau/\bar\alpha]$.

EXAMPLE 4. The instance relation covers an interesting case of type isomorphism [3]. In System F, type $\forall\alpha \cdot \tau' \to \tau$ is isomorphic[5] to $\tau' \to \forall\alpha \cdot \tau$ whenever α is not free in τ'. In ML$^{\mathrm{F}}$, the two corresponding polytypes are not equivalent but in an instance relation. Precisely, $\forall(\alpha' \geq \forall(\alpha)\ \tau)\ \tau' \to \alpha'$ is more general than $\forall(\alpha)\ \tau' \to \tau$, as shown by the following derivation:

$$\begin{array}{ll} \forall(\alpha' \geq \forall(\alpha)\ \tau)\ \tau' \to \alpha' & \\ \sqsubseteq \forall(\alpha)\ \forall(\alpha' \geq \tau)\ \tau' \to \alpha' & \text{by I-UP} \\ \equiv \forall(\alpha)\ \tau' \to \tau & \text{by EQ-MONO}^\star \end{array}$$

(However, as opposed to type containment [19], the instance relation cannot express any form of contravariance.)

1.5 Operation on prefixes and unification

Rules A-CONTEXT-L and I-CONTEXT-L show that two types $\forall(Q)\ \sigma$ and $\forall(Q')\ \sigma$ with the same suffix can be in an instance relation, for any suffix σ. This suggests a notion of inequality between prefixes alone. However, because prefixes are "open" this relation must be defined relatively to a set of variables that lists (a superset of) the free type variables of σ. In this context, a set of type variables is called an *interface* and is written with letter I.

DEFINITION 5 (PREFIX INSTANCE). A prefix Q is an *instance* of a prefix Q' under the interface I, and we write $Q \sqsubseteq^I Q'$, if and only if $\forall(Q)\ \sigma \sqsubseteq \forall(Q')\ \sigma$ holds for all types σ whose free variables are included in I. We omit I in the notation when it is equal to $\mathsf{dom}(Q)$. We define $Q \equiv^I Q'$ and $Q \mathrel{\sqsubseteq\!\!\!\!\!-}^I Q'$ similarly. □

Prefixes can be seen as a generalization of the notion of substitutions to polytypes. Then, $Q \sqsubseteq Q'$ captures the usual notion of (a substitution) Q being more general than (a substitution) Q'.

DEFINITION 6 (UNIFICATION). A prefix Q' *unifies* monotypes τ_1 and τ_2 under Q if and only if $Q \sqsubseteq Q'$ and $(Q)\ \tau_1 \equiv \tau_2$. □

The unification algorithm, called `unify`, is defined in Appendix A.

THEOREM 1. *For any prefix Q and monotypes τ_1 and τ_2,* `unify` *(Q, τ_1, τ_2) returns the smallest prefix (for the relation $\sqsubseteq^{\mathsf{dom}(Q)}$) that unifies τ_1 and τ_2 under Q, or fails if there exists no prefix Q' that unifies τ_1 and τ_2 under Q.*

The following lemma shows that first-order unification lies under ML$^{\mathrm{F}}$ unification.

LEMMA 3. *If $(Q)\ \tau_1 \equiv \tau_2$, then there exists a substitution $\widehat{Q}$ (depending only on Q) that unifies τ_1 and τ_2.*

2 The core language

As explained in the introduction we formalize the language ML$^{\mathrm{F}}$ as a restriction to the more permissive language ML$^{\mathrm{F}}_{\star}$. We assume given a countable set of variables, written with letter x, and a countable set of constants $c \in \mathcal{C}$. Every constant c has an arity $|c|$. A constant is either a primitive f or a constructor C. The distinction

[5] That is, there exists a function (η, β)-reducible to the identity that transforms one into the other, and conversely.

Figure 5. Expressions of $\mathrm{ML}^{\mathsf{F}}_{\star}$

$a ::= x \mid c \mid \lambda(x)\ a \mid a\ a \mid \mathtt{let}\ x = a\ \mathtt{in}\ a$	Terms
$\quad \mid (a : \star)$	Oracles
$c ::= f \mid C$	Constants
$z ::= x \mid c$	Identifiers

between constructors and primitives lies in their dynamic semantics: primitives (such as $+$) are reduced when fully applied, while constructors (such as `cons`) represent data structures, and are not reduced. We use letter z to refer to identifiers, *i.e.* either variables or constants.

Expressions of $\mathrm{ML}^{\mathsf{F}}_{\star}$, written with letter a, are described in Figure 5. Expressions are those of ML extended with *oracles*. An oracle, written $(a : \star)$ is simply a place holder for an implicit type annotation around the expression a. Intuitively, oracles are places where the type inference algorithm must call an "oracle" to fill the hole with a type annotation. Equivalently, the oracles can be replaced by explicit type annotations before type inference. Explicit annotations $(a : \sigma)$, which are described in Section 4, are actually syntactic sugar for applications $(\sigma)\ a$ where (σ) are constants. Examples in the introduction also use the notation $\lambda(x : \sigma)\ a$, which do not appear in Figure 5, because this is, again, syntactic sugar for $\lambda(x)\ \mathtt{let}\ x = (x : \sigma)\ \mathtt{in}\ a$. Similarly, $\lambda(x : \star)\ a$ means $\lambda(x)\ \mathtt{let}\ x = (x : \star)\ \mathtt{in}\ a$.

The language ML^{F} is the restriction of $\mathrm{ML}^{\mathsf{F}}_{\star}$ to expressions that do not contain oracles.

2.1 Static semantics

Typing contexts, written with letter Γ are lists of assertions of the form $z : \sigma$. We write $z : \sigma \in \Gamma$ to mean that z is bound in Γ and $z : \sigma$ is its rightmost binding in Γ. We assume given an initial typing context Γ_0 mapping constants to closed polytypes.

Typing judgments are of the form $(Q)\ \Gamma \vdash a : \sigma$. A tiny difference with ML is the presence of the prefix Q that assigns bounds to type variables appearing free in Γ or σ. By comparison, this prefix is left implicit in ML because all free type variables have the same (implicit) bound $\bot$. In ML^{F}, we require that σ and all polytypes of Γ be closed with respect to Q, that is, $\mathsf{ftv}(\Gamma) \cup \mathsf{ftv}(\sigma) \subseteq \mathsf{dom}(Q)$.

Typing rules. The typing rules of $\mathrm{ML}^{\mathsf{F}}_{\star}$ and ML^{F} are described in Figure 6. They correspond to the typing rules of ML modulo the richer types, the richer instance relation, and the explicit binding of free type variables in judgments. In addition, Rule ORACLE allows for the *revelation* of polytypes, that is, the transformation of types along the inverse of the abstraction relation. (This rule would has no effect in ML where abstraction is the same as equivalence.) For $\mathrm{UML}^{\mathsf{F}}$, it suffices to replace Rule ORACLE by U-ORACLE given below or, equivalently, combine ORACLE with INST into U-INST.

$$\textsc{U-Oracle}\ \dfrac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \mathrel{\exists} \sigma'}{(Q)\ \Gamma \vdash a : \sigma'} \qquad \textsc{U-Inst}\ \dfrac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \mathrel{\exists\sqsubseteq\exists} \sigma'}{(Q)\ \Gamma \vdash a : \sigma'}$$

As in ML, there is an important difference between rules FUN and LET: while typechecking their bodies, a let-bound variable can be assigned a polytype, but a λ-bound variable can only be assigned a monotype in Γ. Indeed, the latter must be guessed while the former can be inferred from the type of the bound expression. This restriction is essential to enable type inference. Notice that a λ-bound variable can refer to a polytype *abstractly* via a type variable α bound to a polytype σ in Q. However, this will not allow to take different instances of σ while typing the body of the abstraction, unless the polytype bound σ of α is first *revealed* by an oracle. Indeed, the only possible instances of α under a prefix Q that contains the binding $(\alpha = \sigma)$ are types equivalent to α under Q. However, $(Q)\ \alpha \equiv \sigma$ does not hold. Thus, if $x : \alpha$ is in the typing context Γ, the only way of typing x (modulo equivalence) is $(Q)\ \Gamma \vdash x : \alpha$, whereas $(Q)\ \Gamma \vdash x : \sigma$ is not derivable. Conversely, $(Q)\ \Gamma \vdash (x : \star) : \sigma$ is derivable, since $(Q)\ \alpha \mathrel{\exists} \sigma$.

Figure 6. Typing rules for ML^{F} and $\mathrm{ML}^{\mathsf{F}}_{\star}$

$$\textsc{Var}\ \dfrac{z : \sigma \in \Gamma}{(Q)\ \Gamma \vdash z : \sigma} \qquad \textsc{App}\ \dfrac{(Q)\ \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad (Q)\ \Gamma \vdash a_2 : \tau_2}{(Q)\ \Gamma \vdash a_1\ a_2 : \tau_1}$$

$$\textsc{Fun}\ \dfrac{(Q)\ \Gamma, x : \tau_0 \vdash a : \tau}{(Q)\ \Gamma \vdash \lambda(x)\ a : \tau_0 \to \tau} \qquad \textsc{Let}\ \dfrac{\begin{array}{c}(Q)\ \Gamma \vdash a_1 : \sigma \\ (Q)\ \Gamma, x : \sigma \vdash a_2 : \tau\end{array}}{(Q)\ \Gamma \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau}$$

$$\textsc{Gen}\ \dfrac{(Q, \alpha \diamond \sigma)\ \Gamma \vdash a : \sigma' \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{(Q)\ \Gamma \vdash a : \forall (\alpha \diamond \sigma)\ \sigma'}$$

$$\textsc{Inst}\ \dfrac{\begin{array}{c}(Q)\ \Gamma \vdash a : \sigma \\ (Q)\ \sigma \sqsubseteq \sigma'\end{array}}{(Q)\ \Gamma \vdash a : \sigma'} \qquad \textsc{Oracle}\ \dfrac{\begin{array}{c}(Q)\ \Gamma \vdash a : \sigma \\ (Q)\ \sigma \mathrel{\exists} \sigma'\end{array}}{(Q)\ \Gamma \vdash (a : \star) : \sigma'}$$

ML as a subset of ML^{F}. ML can be embedded into ML^{F} by restricting all bounds in the prefix Q to be unconstrained. Rules GEN and INST are then exactly those of ML. Hence, any closed program typable in ML is also typable in ML^{F}.

EXAMPLE 5. This first example of typing illustrates the use of polytypes in typing derivations: we consider the simple expression K' defined by $\lambda(x)\ \lambda(y)\ y$. Following ML, one possible typing derivation is (we recall that (α, β) stands for $(\alpha \geq \bot, \beta \geq \bot)$):

$$\textsc{Gen}\ \dfrac{\textsc{Fun}\ \dfrac{\textsc{Fun}\ \dfrac{(\alpha, \beta)\ x : \alpha, y : \beta \vdash y : \beta}{(\alpha, \beta)\ x : \alpha \vdash \lambda(y)\ y : \beta \to \beta}}{(\alpha, \beta)\ \vdash K' : \alpha \to (\beta \to \beta)}}{\vdash K' : \forall (\alpha, \beta)\ \alpha \to (\beta \to \beta)}$$

There is, however, another typing derivation that infers a more general type for K' in ML^{F} (for conciseness we write Q for $\alpha, \beta \geq \sigma_{\mathtt{id}}$):

$$\textsc{Gen}\ \dfrac{\textsc{Fun}\ \dfrac{\textsc{Inst}\ \dfrac{\textsc{Gen}\ \dfrac{\textsc{Fun}\ \dfrac{(Q, \gamma)\ x : \alpha, y : \gamma \vdash y : \gamma}{(Q, \gamma)\ x : \alpha \vdash \lambda(y)\ y : \gamma \to \gamma}}{(Q)\ x : \alpha \vdash \lambda(y)\ y : \sigma_{\mathtt{id}}} \qquad (Q)\ \sigma_{\mathtt{id}} \sqsubseteq \beta}{(Q)\ x : \alpha \vdash \lambda(y)\ y : \beta}}{(Q)\ \vdash K' : \alpha \to \beta}}{\vdash K' : \forall (Q)\ \alpha \to \beta}$$

Notice that the polytype $\forall (\alpha, \beta \geq \sigma_{\mathtt{id}})\ \alpha \to \beta$ is more general than $\forall (\alpha, \beta)\ \alpha \to (\beta \to \beta)$, which follows from Example 4.

Figure 7. Syntax directed typing rules

$$\textsc{Var}^{\triangledown}\quad \frac{z:\sigma\in\Gamma}{(Q)\,\Gamma\vdash^{\triangledown} z:\sigma} \qquad \textsc{Fun}^{\triangledown}\quad \frac{(QQ')\,\Gamma,x:\tau_0\vdash^{\triangledown} a:\sigma \qquad \mathsf{dom}(Q')\cap\Gamma=\emptyset}{(Q)\,\Gamma\vdash^{\triangledown} \lambda(x)\,a:\forall(Q',\alpha\geq\sigma)\,\tau_0\to\alpha}$$

$$\textsc{App}^{\triangledown}\quad \frac{\begin{array}{cc}(Q)\,\Gamma\vdash^{\triangledown} a_1:\sigma_1 & (Q)\,\Gamma\vdash^{\triangledown} a_2:\sigma_2\\ (Q)\,\sigma_1\sqsubseteq\forall(Q')\,\tau_2\to\tau_1 & (Q)\,\sigma_2\sqsubseteq\forall(Q')\,\tau_2\end{array}}{(Q)\,\Gamma\vdash^{\triangledown} a_1\,a_2:\forall(Q')\,\tau_1}$$

$$\textsc{Let}^{\triangledown}\quad \frac{\begin{array}{c}(Q)\,\Gamma\vdash^{\triangledown} a_1:\sigma_1\\ (Q)\,\Gamma,x:\sigma_1\vdash^{\triangledown} a_2:\sigma_2\end{array}}{(Q)\,\Gamma\vdash^{\triangledown} \mathtt{let}\;x=a_1\;\mathtt{in}\;a_2:\sigma_2} \qquad \textsc{Oracle}^{\triangledown}\quad \frac{\begin{array}{c}(Q)\,\Gamma\vdash^{\triangledown} a:\sigma\\ (Q)\,\sigma\exists\sigma'\end{array}}{(Q)\,\Gamma\vdash^{\triangledown} (a:\star):\sigma'}$$

2.2 Syntax directed presentation

As in ML, we can replace the typing rules of $\mathrm{ML}^{\mathrm{F}}_{\star}$ by a set of equivalent syntax-directed typing rules, which are given in Figure 7. Naively, a sequence of non-syntax-directed typing Rules GEN and INST should be placed around any other rule. However, many of these occurrences can be proved unnecessary by following an appropriate strategy. For instance, in ML, judgments are maintained instantiated as much as possible and are only generalized on the left-hand side of Rule LET. In $\mathrm{ML}^{\mathrm{F}}_{\star}$, this strategy would require more occurrences of generalization. Instead, we prefer to maintain typing judgments generalized as much as possible. Then, it suffices to allow Rule GEN right after Rule FUN and to allow Rule INST right before Rule APP (see Rules $\textsc{Fun}^{\triangledown}$ and $\textsc{App}^{\triangledown}$).

EXAMPLE 6. As we claimed in the introduction, a λ-bound variable that is used polymorphically must be annotated. Let us check that $\lambda(x)\;x\,x$ is not typable in ML^{F} by means of contradiction. A syntax-directed type derivation of this expression would be of the form:

$$\textsc{Fun}^{\triangledown}\;\dfrac{\textsc{App}^{\triangledown}\;\dfrac{\begin{array}{cc}\textsc{Var}^{\triangledown}\;(Q)\,x:\tau_0\vdash^{\triangledown} x:\tau_0 & (Q)\,x:\tau_0\vdash^{\triangledown} x:\tau_0\;\textsc{Var}^{\triangledown}\\ (Q)\,\tau_0\sqsubseteq\forall(Q')\,\tau_2\to\tau_1\;(\mathbf{2}) & (Q)\,\tau_0\sqsubseteq\forall(Q')\,\tau_2\;(\mathbf{1})\end{array}}{(Q)\,x:\tau_0\vdash^{\triangledown} x\,x:\forall(Q')\,\tau_1}}{(Q)\,\emptyset\vdash\lambda(x)\,x\,x:\forall(\alpha\geq\forall(Q')\,\tau_1)\,\tau_0\to\alpha}$$

Applying Rule I-DROP to (2) and (1), we get respectively $(QQ')\;\tau_0\sqsubseteq\tau_2\to\tau_1$ and $(QQ')\;\tau_0\sqsubseteq\tau_2$. Then $(QQ')\;\tau_2\to\tau_1\equiv\tau_2$ follows by Lemma 2 and EQ-TRANS. Thus, by Lemma 3, there exists a substitution θ such that $\theta(\tau_2)=\theta(\tau_2\to\tau_1)$, that is, $\theta(\tau_2)=\theta(\tau_2)\to\theta(\tau_1)$, which cannot be the case.

This example shows the limit of type inference, which is actually the strength of our system! That is, to maintain principal types by rejecting examples where type inference would need to guess second-order types.

EXAMPLE 7. Let us recover typability by introducing an oracle and build a derivation for $\lambda(x)\;(x:\star)\;x$. Taking $(\alpha=\sigma_{\mathtt{id}})$ for Q and α for τ_0, we obtain:

$$\textsc{Fun}^{\triangledown}\;\dfrac{\textsc{App}^{\triangledown}\;\dfrac{\textsc{Oracle}^{\triangledown}\;\dfrac{\begin{array}{c}\textsc{Var}^{\triangledown}\;(Q)\,x:\alpha\vdash^{\triangledown} x:\alpha\\ (Q)\,\alpha\exists\sigma_{\mathtt{id}}\;(\mathbf{3})\end{array}}{(Q)\,x:\alpha\vdash^{\triangledown}(x:\star):\sigma_{\mathtt{id}}} \qquad \begin{array}{c}(Q)\,\sigma_{\mathtt{id}}\sqsubseteq\alpha\to\alpha\\ (Q)\,x:\alpha\vdash^{\triangledown} x:\alpha\;\textsc{Var}^{\triangledown}\end{array}}{(Q)\,x:\alpha\vdash^{\triangledown}(x:\star)\,x:\alpha}}{\vdash\lambda(x)\;(x:\star)\,x:\forall(\alpha=\sigma_{\mathtt{id}})\,\alpha\to\alpha}$$

The oracle plays a crucial role in (3)—the revelation of the type scheme $\sigma_{\mathtt{id}}$ that is the bound of the type variable α used in the type of x. We have $(Q)\;\sigma_{\mathtt{id}}\sqsubseteq\alpha$, indeed, but the converse relation does not hold, so rule INST cannot be used here to replace α by its bound $\sigma_{\mathtt{id}}$.

2.3 Dynamic semantics

The semantics of $\mathrm{ML}^{\mathrm{F}}_{\star}$ is the standard call-by-value semantics of ML. We present it as a small-step reduction semantics. Values and call-by-value evaluation contexts are described below.

$$\begin{array}{rll} v ::= & \lambda(x)\,a & \\ & \mid f\,v_1\,\ldots v_n & n<|f|\\ & \mid C\,v_1\,\ldots v_n & n\leq|C|\\ & \mid (v:\star) & \\ E ::= & [\,]\mid E\;a\mid v\;E\mid(E:\star)\mid\mathtt{let}\;x=E\;\mathtt{in}\;a & \end{array}$$

The reduction relation $\longrightarrow$ is parameterized by a set of δ-rules of the form (δ) below:

$$\begin{array}{rll} f\,v_1\,\ldots v_n\longrightarrow a & \text{when } |f|=n & (\delta)\\ (\lambda(x)\,a)\;v\longrightarrow a[v/x] & & (\beta_v)\\ \mathtt{let}\;x=v\;\mathtt{in}\;a\longrightarrow a[v/x] & & (\beta_{let})\\ (v_1:\star)\;v_2\longrightarrow(v_1\,v_2:\star) & & (\star)\end{array}$$

The main reduction is the β-reduction that takes two forms Rule (β_v) and Rule (β_{let}). Oracles are maintained during reduction to which they do not contribute: they are simply pushed out of applications by rule $(\star)$. Finally, the reduction is the smallest relation containing (δ), (β_v), (β_{let}), and $(\star)$ rules that is closed by E-congruence:

$$E[a]\longrightarrow E[a']\text{ if }a\longrightarrow a' \qquad (\textsc{Context})$$

3 Formal properties

We verify type soundness for $\mathrm{ML}^{\mathrm{F}}_{\star}$ and address type inference in ML^{F}.

3.1 Type soundness

Type soundness for $\mathrm{ML}^{\mathrm{F}}_{\star}$ is shown as usual by a combination of *subject reduction*, which ensures that typings are preserved by reduction, and *progress*, which ensures that well-typed programs that are not values can be further reduced.

To ease the presentation, we introduce a relation $\subseteq$ between programs: we write $a\subseteq a'$ if and only if every typing of a, *i.e.* a triple (Q,Γ,σ) such that $(Q)\;\Gamma\vdash a:\sigma$ holds, is also a typing of a'. A relation $\mathcal{R}$ on programs preserves typings whenever it is a sub-relation of $\subseteq$.

Of course, type soundness cannot hold without some assumptions relating the static semantics of constants described by the initial typing context Γ_0 and their dynamic semantics.

DEFINITION 7 (HYPOTHESES). We assume that the following three properties hold for constants.

(H0) (Arity) Each constant $c \in \mathsf{dom}(\Gamma_0)$ has a closed type $\Gamma_0(c)$ of the form $\forall (Q)\ \tau_1 \to \dots \tau_{|c|} \to \tau$ and such that the top symbol of $\forall (Q)\ \tau$ is not in $\{\to, \bot\}$ whenever c is a constructor.

(H1) (Subject-Reduction) All δ-rules preserve typings.

(H2) (Progress) Any expression a of the form $f\ v_1 \dots v_{|f|}$, such that $(Q)\ \Gamma_0 \vdash a : \sigma$ is in the domain of (δ). □

THEOREM 2 (SUBJECT REDUCTION). *Reduction preserves typings.*

THEOREM 3 (PROGRESS). *Any expression a such that $(Q)\ \Gamma_0 \vdash a : \sigma$ is a value or can be further reduced.*

Combining theorems 2 and 3 ensures that the reduction of well-typed programs either proceeds for ever or ends up with a value. This holds for programs in $\mathrm{ML}^{\mathrm{F}}_{\star}$ but also for programs in ML^{F}, since ML^{F} is a subset of $\mathrm{ML}^{\mathrm{F}}_{\star}$. Hence ML^{F} is also sound. However, ML^{F} does not enjoy subject reduction, since reduction may create oracles. Notice, however, that oracles can only be introduced by δ-rules.

3.2 Type inference

A *type inference problem* is a triple (Q, Γ, a), where all free type variables in Γ are bound in Q. A pair (Q', σ) is a *solution* to this problem if $Q \sqsubseteq Q'$ and $(Q')\ \Gamma \vdash a : \sigma$. A pair (Q', σ') is an *instance* of a pair (Q, σ) if $Q \sqsubseteq Q'$ and $(Q')\ \sigma \sqsubseteq \sigma'$. A solution of a type inference problem is *principal* if all other solutions are instances of the given one.

Figure 9 in the Appendix B defines a type inference algorithm $\mathcal{W}^{\mathrm{F}}$ for ML^{F}. This algorithm proceeds much as type inference for ML: the algorithm follows the syntax-directed typing rules and reduces type inference to unification under prefixes.

THEOREM 4 (TYPE INFERENCE). *The set of solutions of a solvable type inference problem admits a principal solution. Given any type inference problem, the algorithm $\mathcal{W}^{\mathrm{F}}$ either returns a principal solution or fails if no solution exists.*

4 Type annotations

In this section, we restrict our attention to ML^{F}, *i.e.* to expressions that do not contain oracles. Since expressions of ML^{F} are exactly those of ML, its expressiveness may only come from richer types and typing rules. However, the following lemma shows that this is not sufficient:

LEMMA 4. *If the judgment $(Q)\ \Gamma \vdash a : \sigma$ holds in ML^{F} where the typing context Γ contains only ML types and Q contains only type variables with unconstrained bounds, then there exists a derivation of $\Gamma \vdash a : \forall (\bar\alpha)\ \tau$ in ML where $\forall (\bar\alpha)\ \tau$ is obtained from σ by moving all inner quantifiers ahead.*

The inverse inclusion has already been stated in Section 2.1. In the particular case where the initial typing context Γ_0 contains only ML types, a closed expression can be typed in ML^{F} under Γ_0 if and only if it can be typed in ML under Γ_0. This is not true for $\mathrm{ML}^{\mathrm{F}}_{\star}$ in which the expression $\lambda(x)\ (x : \star)\ x$ is typable. Indeed, as shown below, all terms of System F can be typed in $\mathrm{ML}^{\mathrm{F}}_{\star}$. Fortunately, there is an interesting choice of constants that provides ML^{F} with the same expressiveness as $\mathrm{ML}^{\mathrm{F}}_{\star}$ while retaining type inference. Precisely, we provide type annotations as a collection of coercion primitives, *i.e.* functions that change the type of expressions without changing their meaning. The following example, which describes a single annotation, should provide intuition for the general case.

EXAMPLE 8. Let f be a constant of type σ equal to $\forall (\alpha = \sigma_{\mathtt{id}}, \alpha' \geq \sigma_{\mathtt{id}})\ \alpha \to \alpha'$ with the δ-reduction $f\ v \longrightarrow (v : \star)$. Then, the expression a defined as $\lambda(x)\ (f\ x)\ x$ behaves as $\lambda(x)\ x\ x$ and is well-typed, of type $\forall (\alpha = \sigma_{\mathtt{id}})\ \alpha \to \alpha$. To see this, let Q and Γ stand for $(\alpha = \sigma_{\mathtt{id}}, \alpha' = \sigma_{\mathtt{id}})$ and $x : \alpha$. By Rules INST, VAR, and APP $(Q)\ \Gamma \vdash f\ x : \alpha'$; hence by rule GEN, $(\alpha = \sigma_{\mathtt{id}})\ \Gamma \vdash f\ x : \forall (\alpha' = \sigma_{\mathtt{id}})\ \alpha'$ since α' is not free in the Γ. By rule EQ-VAR, we have $\forall (\alpha' = \sigma_{\mathtt{id}})\ \alpha' \equiv \sigma_{\mathtt{id}}$ (under any prefix); besides, $\sigma_{\mathtt{id}} \sqsubseteq \alpha \to \alpha$ under any prefix that binds α. Thus, we get $(\alpha = \sigma_{\mathtt{id}})\ \Gamma \vdash f\ x : \alpha \to \alpha$ by Rule INST. The result follows by Rules APP, FUN, and GEN.

Observe that the static effect of f in $f\ x$ is (i) to enforce the type of x to be abstracted by a variable α bound to $\sigma_{\mathtt{id}}$ in Q and (ii) to give $f\ x$, that is x, the type $\sigma_{\mathtt{id}}$, exactly as the oracle $(x : \star)$ would. Notice that the bound of α in σ is rigid: the function f expects a value v that must have type $\sigma_{\mathtt{id}}$ (and not a strict instance of $\sigma_{\mathtt{id}}$). Conversely, the bound of α' is flexible: the type of $f\ v$ is $\sigma_{\mathtt{id}}$ but may also be any instance of $\sigma_{\mathtt{id}}$.

DEFINITION 8. We call *annotations* the denumerable collection of primitives $(\exists (Q)\ \sigma)$, of arity 1, defined for all prefixes Q and polytypes σ closed under Q. The initial typing environment Γ_0 contains these primitives with their associated type:

$$(\exists (Q)\ \sigma) : \forall (Q)\ \forall (\alpha = \sigma)\ \forall (\beta \geq \sigma)\ \alpha \to \beta \qquad \in \Gamma_0$$

We may identify annotation primitives up to the equivalence of their types. □

Besides, we write $(a : \exists (Q)\ \sigma)$ for the application $(\exists (Q)\ \sigma)\ a$. We also abbreviate $(\exists (Q)\ \sigma)$ as (σ) when all bounds in Q are unconstrained. Actually, replacing an annotation $(\exists (Q)\ \sigma)$ by (σ) preserves typability and, more precisely, preserves typings.

While annotations are introduced as primitives for simplicity of presentation, they are obviously meant to be applied. Notice that the type of an annotation may be instantiated before the annotation is applied. However, the annotation keeps exactly the same "revealing power" after instantiation. This is described by the following technical lemma (the reader may take $\emptyset$ for Q_0 at first).

LEMMA 5. *The judgment $(Q_0)\ \Gamma \vdash (a : \exists (Q)\ \sigma) : \sigma_0$ is valid if and only if there exists a type $\forall (Q')\ \sigma'_1$ such that the judgment $(Q_0)\ \Gamma \vdash a : \forall (Q')\ \sigma'_1$ holds together with the following relations: $Q_0 Q \sqsubseteq Q_0 Q'$, $(Q_0 Q')\ \sigma'_1 \sqsupseteq \sigma$, and $(Q_0)\ \forall (Q')\ \sigma \sqsubseteq \sigma_0$.*

The prefix Q of the annotation $\exists (Q)\ \sigma$ may be instantiated into Q'. However, Q' guards $\sigma'_1 \sqsupseteq \sigma$ in $(Q_0 Q')\ \sigma'_1 \sqsupseteq \sigma$. In particular, the lemma would not hold with $(Q_0)\ \forall (Q')\ \sigma'_1 \sqsupseteq \forall (Q'')\ \sigma$ and $(Q_0)\ \forall (Q'')\ \sigma'_1 \sqsubseteq \sigma_0$. Lemma 5 has similarities with Rule ANNOT of Poly-ML [7].

COROLLARY 6. *The judgment $(Q)\ \Gamma \vdash (a : \star) : \sigma_0$ holds if and only if there exists an annotation (σ) such that $(Q)\ \Gamma \vdash (a : \sigma) : \sigma_0$ holds.*

Hence, all expressions typable in $\text{ML}^{\text{F}}_{\star}$ are typable in ML^{F} as long as all annotation primitives are in the initial typing context Γ_0.

Reduction of annotations. The δ-reduction for annotations just replaces explicit type information by oracles.

$$(v : \exists (Q)\ \sigma) \longrightarrow (v : \star)$$

LEMMA 7 (SOUNDNESS OF TYPE ANNOTATIONS). *All three hypotheses (H0, arity), (H1, subject-reduction), and (H2, progress) hold when primitives are the set of annotations, alone.*

The annotation $(\exists (Q)\ \sigma)$ can be simulated by $\lambda(x)\ (x : \star)$ in $\text{ML}^{\text{F}}_{\star}$, both statically and dynamically. Hence annotations primitives are unnecessary in $\text{ML}^{\text{F}}_{\star}$.

Syntactic sugar. As mentioned in Section 2, $\lambda(x{:}\sigma)\ a$ is syntactic sugar for $\lambda(x)\ \texttt{let}\ x = (x : \sigma)\ \texttt{in}\ a$. The derived typing rule is:

$$\text{FUN}^{\star}\quad \frac{(Q)\ \Gamma, x : \sigma \vdash a : \sigma' \qquad Q' \sqsubseteq Q}{(Q)\ \Gamma \vdash \lambda(x{:}\exists (Q')\ \sigma)\ a : \forall (\alpha{=}\sigma)\ \forall (\alpha' \geq \sigma')\ \alpha \to \alpha'}$$

This rule is actually simpler than the derived annotation rule suggested by lemma 5, because instantiation is here left to each occurrence of the annotated program variable x in a.

The derived reduction rule is $(\lambda(x{:}\exists (Q)\ \sigma)\ a)\ v \xrightarrow{\beta_\star} \texttt{let}\ x = (v : \exists (Q)\ \sigma)\ \texttt{in}\ a$. Values must then be extended with expressions of the form $\lambda(x{:}\exists (Q)\ \sigma)\ a$, indeed.

5 Comparison with System-F

We have already seen that all ML programs can be written in ML^{F} without annotations. In Section 5.1, we provide a straightforward compositional translation of terms of System-F into ML^{F}. In Section 5.2, we then identify a subsystem of ML^{F}, called Shallow-$\text{ML}^{\text{F}}_{\star}$, whose let-binding free version is exactly the target of the encoding of System-F.

5.1 Encoding System-F into ML^{F}

The types, terms, and typing contexts of system F are given below:

$$\begin{aligned} t &::= \alpha \mid t \to t \mid \forall \alpha \cdot t \\ M &::= x \mid M\ M \mid \lambda(x : t)\ M \mid \Lambda(\alpha) M \mid M\ t \\ A &::= \emptyset \mid A, x : t \mid A, \alpha \end{aligned}$$

The translation of types of System-F into ML^{F} types uses auxiliary rigid bindings for arrow types. This ensures that there are no inner polytypes left in the result of the translation, which would otherwise be ill-formed. Quantifiers that are present in the original types are translated to unconstrained bounds.

$$[\![\alpha]\!] = \alpha \qquad [\![\forall \alpha \cdot t]\!] = \forall (\alpha)\ [\![t]\!]$$
$$[\![t_1 \to t_2]\!] = \forall (\alpha_1 = [\![t_1]\!])\ \forall (\alpha_2 = [\![t_2]\!])\ \alpha_1 \to \alpha_2$$

In order to state the correspondence between typing judgments, we must also translate typing contexts. We write $A \vdash M : t$ to mean that M has type t in typing context A in System F. The translation of A, written $[\![A]\!]$, returns a pair $(Q)\ \Gamma$ of a prefix and a typing context and is defined inductively as follows:

$$[\![\emptyset]\!] = ()\ \emptyset \qquad \frac{[\![A]\!] = (Q)\ \Gamma}{[\![A, x : t]\!] = (Q)\ \Gamma, x : [\![t]\!]}$$
$$\frac{[\![A]\!] = (Q)\ \Gamma \qquad \alpha \notin \mathsf{dom}(Q)}{[\![A, \alpha]\!] = (Q, \alpha)\ \Gamma}$$

The translation of System F terms into ML^{F} terms forgets type abstraction and type applications, and translates types in term-abstractions.

$$[\![\Lambda(\alpha) M]\!] = [\![M]\!] \qquad [\![M\ t]\!] = [\![M]\!] \qquad [\![x]\!] = x$$
$$[\![M\ M']\!] = [\![M]\!]\ [\![M']\!] \qquad [\![\lambda(x : t)\ M]\!] = \lambda(x : [\![t]\!])\ [\![M]\!]$$

Finally, we can state the following lemma:

LEMMA 8. *For any closed typing context A (that does not bind the same type variable twice), term M, and type t of system F such that $A \vdash M : t$, there exists a derivation $(Q)\ \Gamma \vdash [\![M]\!] : \tau$ such that $(Q)\ \Gamma = [\![A]\!]$ and $[\![t]\!] \sqsubseteq \tau$.*

Remarkably, translated terms contain strictly fewer annotations than original terms—a property that was not true in Poly-ML. In particular, all type Λ-abstractions and type applications are dropped and only annotations of λ-bound variables remain. Moreover, some of these annotations are still superfluous.

5.2 Shallow-$\text{ML}^{\text{F}}_{\star}$

Types whose flexible bounds are always $\bot$ are called F-types (they are the translation of types of System F). Types of the form $\forall (\alpha \geq \sigma)\ \tau$, where σ is not equivalent to a monotype nor to $\bot$, have been introduced to factor out choices during type inference. Such types are indeed used in a derivation of $\texttt{let}\ f = \texttt{choose id in}\ (f\ \texttt{auto})\ (f\ \texttt{succ})$. However, they are never used in the encoding of System-F. Are they needed as annotations?

Let a type be *shallow* if and only if all its rigid bounds are F-types. More generally, prefixes, typing contexts, and judgments are shallow if and only if they contain only shallow types. A derivation is shallow if all its judgments are shallow and Rule ORACLE is only applied to F-types. Notice that the explicit annotation (σ) has a shallow type if and only if σ is an F-type. We call Shallow-$\text{ML}^{\text{F}}_{\star}$ the set of terms that have shallow derivations. Interestingly, subject reduction also holds for Shallow-$\text{ML}^{\text{F}}_{\star}$.

Let-bindings do not increase expressiveness in $\text{ML}^{\text{F}}_{\star}$, since they can always be replaced by λ-bindings with oracles or explicit annotations. This is not true for Shallow-$\text{ML}^{\text{F}}_{\star}$, since shallow-types that are not F-types cannot be used as annotations. Therefore, we also consider the restriction Shallow-F of Shallow-$\text{ML}^{\text{F}}_{\star}$ to programs without let-bindings.

The encoding of System-F into ML^{F} given in Section 5.1 is actually an encoding into Shallow-F. Conversely, all programs typable into Shallow-F are also typable in System-F. Hence, Shallow-F and System-F have the same expressiveness.

6 Discussion

6.1 Expressiveness of ML^{F}

By construction, we have the chain of inclusions Shallow-F $\subseteq$ Shallow-$\mathrm{ML}^{\mathsf{F}}_{\star} \subseteq \mathrm{ML}^{\mathsf{F}}_{\star}$. We may wonder whether these languages have strictly increasing power. That is, ignoring annotations and the difference in notation between let-bindings and λ-redexes, do they still form a strict chain of inclusions? We conjecture that this is true.

Still, $\mathrm{ML}^{\mathsf{F}}_{\star}$ remains a second-order system and in that sense should not be *significantly* more expressive than System F. In particular, we conjecture that the term $(\lambda(y)\, y\, I\, ;\, y\, K)\ (\lambda(x)\, x\, x)$ that is typable in F^{ω} but not in F [9] is not typable in ML^{F} either. Conversely, we do not know whether there exists a term of ML^{F} that is not typable in F^{ω}.

Reducing all let-bindings in a term of Shallow-$\mathrm{ML}^{\mathsf{F}}_{\star}$ produces a term in Shallow-F. Hence, terms of Shallow-$\mathrm{ML}^{\mathsf{F}}_{\star}$ are strongly normalizable. We conjecture that so are all terms of ML^{F}.

6.2 Simple language extensions

Because the language is parameterized by constants, which can be used either as constructors or primitive operations, the language can import foreign functions defined via appropriate δ-rules. These could include primitive types (such as integers, strings, *etc.*) and operations over them. Sums and products, as well as predefined datatypes, can also be treated in this manner, but some extension is required to declare new data-types within the language itself.

The value restriction of polymorphism [28] that allows for safe mutable data-structures in ML should carry over to ML^{F} by allowing only rigid bounds that appear in the type of expansive expressions to be generalized. However, this solution is likely to be disappointing in ML^{F}, as it is in Poly-ML, which uses polymorphism extensively. An interesting relaxation of the value-only restriction has been recently proposed [6] and allows to always generalize type variables that never appears on the left hand-side of an arrow type; this gave quite satisfactory results in the context of Poly-ML and we can expect similar benefits for ML^{F}.

6.3 Related works

Our work is related to all other works that aim at some form of type inference in the presence of higher-order types.The closest of them is unquestionably Poly-ML [7], with which close connections have already been made. Poly-ML also subsumes previous proposals that encapsulate first-class polymorphic values within datatypes [25]. Odersky and Laufer's proposal [20] also falls into this category; however, a side mechanism simultaneously allows a form of toplevel rank-2 quantification, which is not covered by Poly-ML but is, we think, subsumed by ML^{F}.

Rank-2 polymorphism actually allows for full type inference [13, 11]. However, the algorithm proceeds by reduction on source terms and is not very intuitive. Rank-2 polymorphism has also been incorporated in the Hugs implementation of Haskell [17], but with explicit type annotations. The GHC implementation of Haskell has recently been released with second-order polymorphism at arbitrary ranks [8]; however, types at rank 2 or higher must be given explicitly and the interaction of annotations with implicit types remains unclear. Furthermore, to the best of our knowledge, this has not yet been formalized. Indeed, type inference is undecidable as soon as universal quantifiers may appear at rank 3 [14].

Although our proposal relies on the let-binding mechanism to introduce implicit polymorphismand flexible bounds in types to factorize all ways of obtaining type instances, there may still be some connection with intersection types [12], which we would like to explore. Our treatment of annotations as type-revealing primitives also resembles retyping functions (functions whose type-erasure η-reduces to the identity) [19]. However, our annotations are explicit and contain only certain forms of retyping functions. Type inference for System F modulo η-expansion is known to be undecidable as well [27].

Several people have considered partial type inference for System F [10, 1, 23] and stated undecidability results for some particular variants that in all cases amount—directly or indirectly—to permitting (and so forcing) inference of the type of at least one variable that can be used in a polymorphic manner, which we avoid.

Second-order unification, although known to be undecidable, has been used to explore the practical effectiveness of type inference for System F by Pfenning [22]. Despite our opposite choice, that is not to support second-order unification, there are at least two comparisons to be made. Firstly, Pfenning's work does not cover the language ML *per se*, but only the λ-calculus, since let-bindings are expanded prior to type inference. Indeed, ML is not the simply-typed λ-calculus and type inference in ML cannot, *in practice*, be reduced to type inference in the simply-typed λ-calculus after expansion of let-bindings. Secondly, one proposal seems to require annotations exactly where the other can skip them: in [22], markers (but no type) annotations must replace type-abstraction and type-application nodes; conversely, this information is omitted in ML^{F}, but instead, explicit type information must remain for (some) arguments of λ-abstractions.

Our proposal is implicitly parameterized by the type instance relation and its corresponding unification algorithm. Thus, most of the technical details can be encapsulated within the instance relation. We would like to understand our notion of unification as a particular case of second-order unification. One step in this direction would be to consider a modular constraint-based presentation of second-order unification such as [5]. Flexible bounds might partly capture, within principal types, what constraint-based algorithms capture as partially unresolved multi-sets of unification constraints. Another example of restricted unification within second-order terms is unification under a mixed prefix [18]. However, our notion of prefix and its role in abstracting polytypes is quite different.

Actually, none of the above works did consider subtyping at all. This is a significant difference with proposals based on local type inference [2, 24, 21] where subtyping is a prerequisite. The addition of subtyping to our framework remains to be explored.

Furthermore, 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, which, at least intuitively, could be understood as a preprocessing of the source term. Such a mechanism is being used in the GHC Haskell compiler, and could in principle be added on top of ML^{F} as well.

Conclusions

We have proposed an integration of ML and System F that combines the convenience of type inference as present in ML and the expressiveness of second-order polymorphism. Type information is only required for arguments of functions that are used polymorphically in their bodies. This specification should be intuitive to the user. Besides, it is modular, since annotations depend more on the behavior of the code than on the context in which the code is placed; in particular, functions that only carry polymorphism without using it can be left unannotated.

The obvious potential application of our work is to extend ML-like languages with second-order polymorphism while keeping full type inference for a large subset of the language, containing at least all ML programs. Indeed, we implemented a prototype of ML^{F} and verified on a variety of examples that few annotations are actually required and always at predictable places. However, further investigations are still needed regarding the syntactic-value polymorphism restriction and its possible relaxation.

Furthermore, on the theoretical side, we wish to better understand the concept of "first-order unification of second-order terms", and, if possible, to confine it to an instance of second-order unification. We would also like to give logical meaning to our types and to the abstraction and instance relations.

7 References

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

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

[3] R. D. Cosmo. *Isomorphisms of Types: from lambda-calculus to information retrieval and language design*. Birkhauser, 1995.

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

[5] G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Higher-order unification via explicit substitutions: the case of higher-order patterns. In M. Maher, editor, *Joint international conference and symposium on logic programming*, pages 259–273, 1996.

[6] J. Garrigue. Relaxing the value-restriction. Presented at the third Asian workshop on Programmaming Languages and Systems (APLAS), 2002.

[7] J. Garrigue and D. Rémy. Extending ML with semi-explicit higher-order polymorphism. *Journal of Functional Programming*, 155(1/2):134–169, 1999. A preliminary version appeared in TACS'97.

[8] The GHC Team. *The Glasgow Haskell Compiler User's Guide, Version 5.04*, 2002. Chapter *Arbitrary-rank polymorphism*.

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

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

[11] T. Jim. Rank-2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, Massachusetts Institute of Technology, Laboratory for Computer Science, Nov. 1995.

[12] T. Jim. What are principal typings and what are they good for? In ACM, editor, *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 42–53, 1996.

[13] A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order lambda -calculus. In *ACM Conference on LISP and Functional Programming*, 1994.

[14] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 161–174. ACM, Jan. 1999.

[15] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, Sept. 1994.

[16] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, documentation and user's manual - release 3.05. Technical report, INRIA, July 2002. Documentation distributed with the Objective Caml system.

[17] Mark P Jones, Alastair Reid, the Yale Haskell Group, and the OGI School of Science & Engineering at OHSU. An overview of hugs extensions. Available electronically, 1994-2002.

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

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

[20] M. Odersky and K. Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM Conference on Principles of Programming Languages*, pages 54–67, Jan. 1996.

[21] M. Odersky, C. Zenger, and M. Zenger. Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41–53, Mar. 2001.

[22] F. Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 153–163. ACM Press, July 1988.

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

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

[25] D. Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346. Springer-Verlag, April 1994.

[26] J. B. Wells. Typability and type checking in the second order λ-calculus are equivalent and undecidable. In *Ninth annual IEEE Symposium on Logic in Computer Science*, pages 176–185, July 1994.

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

[28] A. K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

A Unification algorithm

The algorithm `unify` is specified in Section 1.5; it takes a prefix Q and two types τ and τ' and returns a prefix that unifies τ and τ' under Q (as described in Theorem 1) or fails. In fact, the algorithm `unify` is recursively defined by an auxiliary unification algorithm for polytypes: `polyunify` takes a prefix Q and two type schemes σ_1 and σ_2 and returns a pair (Q', σ') such that $Q \sqsubseteq Q'$ and $(Q')\ \sigma_1 \sqsubseteq \sigma'$ and $(Q')\ \sigma_2 \sqsubseteq \sigma'$.

The algorithms `unify` and `polyunify` are described in Figure 8. For the sake of comparison with ML, think of the input prefix Q

Semantic Subtyping for Imperative Object-Oriented Languages

Davide Ancona Andrea Corradi

DIBRIS - Università di Genova
via Dodecaneso, 35 - 16146 Genova, Italy
davide.ancona@unige.it andrea.corradi@dibris.unige.it

Abstract

Semantic subtyping is an approach for defining sound and complete procedures to decide subtyping for expressive types, including union and intersection types; although it has been exploited especially in functional languages for XML based programming, recently it has been partially investigated in the context of object-oriented languages, and a sound and complete subtyping algorithm has been proposed for record types, but restricted to immutable fields, with union and recursive types interpreted coinductively to support cyclic objects.

In this work we address the problem of studying semantic subtyping for imperative object-oriented languages, where fields can be mutable; in particular, we add read/write field annotations to record types, and, besides union, we consider intersection types as well, while maintaining coinductive interpretation of recursive types. In this way, we get a richer notion of type with a flexible subtyping relation, able to express a variety of type invariants useful for enforcing static guarantees for mutable objects.

The addition of these features radically changes the definition of subtyping, and, hence, the corresponding decision procedure, and surprisingly invalidates some subtyping laws that hold in the functional setting.

We propose an intuitive model where mutable record values contain type information to specify the values that can be correctly stored in fields. Such a model, and the corresponding subtyping rules, require particular care to avoid circularity between coinductive judgments and their negations which, by duality, have to be interpreted inductively.

A sound and complete subtyping algorithm is provided, together with a prototype implementation.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; F.3.1 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

Keywords Structural Types for Objects, Semantic Subtyping, Read/Write Field Annotations

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4444-9/16/11...
http://dx.doi.org/10.1145/2983990.2983992

1. Introduction

Subtyping and structural types are essential notions for precise type analysis. They can be employed for typing dynamic languages as JavaScript [7, 19, 20, 24, 25], or to integrate nominal type systems for more flexible and accurate type analysis [16, 18, 23, 27, 30].

In simple type systems the subtyping relation can be defined axiomatically by a set of inference rules; however, when more complex types are considered, the axiomatic approach does not provide a model for formal reasoning, and, thus, may fail to convey the right intuition behind subtyping, and, in general, it is not simple to prove that the subtyping rules are complete.

Semantic subtyping has been proposed as a possible solution to these problems, especially in the context of functional languages for XML based programming, as XDuce [21], and ℂDuce [17]. Semantic subtyping has been investigated also for the π-calculus [11], for coinductive object types with unions [4], and for ML-like languages with polymorphic variants [14]. More recently, in the context of ℂDuce, semantic subtyping has been extended with parametric polymorphism [12, 13].

In semantic subtyping types are interpreted as sets of values and the subtyping relation corresponds to set inclusion between type interpretations. In this way, the definition of subtyping is more intuitive, and several properties can be easily deduced (for instance, transitivity always holds trivially). Semantic subtyping naturally supports boolean type constructors; for instance, the interpretation of union and intersection types coincide with set-theoretic union and intersection, respectively. Furthermore, semantic subtyping helps reasoning on recursive types. Let us consider, for instance, the following recursive record type: $\tau = \langle next{:}\tau \rangle$. In the semantic subtyping approach, the syntactic equation above is turned into a semantic equation, which, in general, can be interpreted either inductively or coinductively. In functional languages as ℂDuce where values are inductive, types are interpreted inductively, therefore the least solution of the semantic equation $[\![\tau]\!] = [\![\langle next{:}\tau\rangle]\!]$ is considered, and the type τ denotes the empty set.

In object-oriented languages objects are allowed to contain cycles, therefore recursive types are interpreted coinductively [4]; the interpretation of τ corresponds to the greatest solution of $[\![\tau]\!] = [\![\langle next{:}\tau\rangle]\!]$, hence τ denotes a non-empty set of cyclic objects.

Recently, semantic subtyping for record and union types has been studied with recursive types interpreted coinductively, and a practical sound and complete top-down algorithm for deciding it has been provided [4, 9]. However, this result is limited to immutable records, where record subtyping is allowed to be covariant in field types, whereas it is well-known that record subtyping with mutable fields must be invariant to avoid unsoundness. Although invariant subtyping is sound, it severely restricts the flexibility of the type system; to avoid this problem, read/write field annotations [1] can be introduced, to allow subtyping to be covariant for read-only fields, and contravariant for write-only fields.

This paper provides two main contributions.

- A coinductive semantic model is defined for record types with read/write field annotations, supporting union, intersection, and recursive types; as it is customary in the semantic subtyping approach, such a semantic model naturally induces a subtyping relation.

  However, the semantic model used to interpret records with mutable fields substantially deviates from previous models adopted for immutable records [4, 9], and requires more complex definitions and challenging proofs of consistency.

- A set of sound and complete rules for such a semantic subtyping relation is defined, and a corresponding algorithm is devised to decide subtyping. The algorithm has been implemented in SWI Prolog.

  The difference between the semantic model used to interpret mutable records and that adopted for immutable records is reflected in the sets of inequations that can be derived: laws that hold for immutable records [4, 9] are no longer valid for mutable records. Furthermore, in our work new laws have to be introduced because a richer type language is considered: fields are annotated, and intersection types are introduced. Consequently, the algorithm presented here to decide subtyping significantly departs from previously proposed algorithms for immutable records.

The proposed model is language independent, in the sense that it supports reasoning on subtyping in the presence of field annotations, without requiring values of the underlying language to support read/write-only fields. In other words, field annotation is a static notion that must not necessarily be reflected at runtime.

In comparison with semantic subtyping for immutable records, the main challenge consists in field annotation, and record mutability. While the definition of a semantic model in a purely functional setting is rather straightforward, here particular care is required to ensure that the model is well-defined, since fields of record values are associated with types to specify which values can be stored in them.

As a consequence, a circularity is introduced since types are interpreted in terms of themselves, and this needs to be properly managed; in particular, the definitions of the coinductive judgment for typing values, and of its negation (which, by duality, is inductive), are mutually recursive. These issues propagate to the definition of the subtyping rules which involves four mutually recursive judgments: the two coinductive judgments for subtyping, and type emptiness, and their corresponding (inductive) negations.

To our knowledge, no standard approach can be found in literature to properly manage coinductive definitions involving negation, or, equivalently, mutually recursive coinductive and inductive definitions. Our simple but effective solution to break circularity consists in equipping judgments with sets of assumptions that have to be verified, and that, in practice, are abducted by the subtyping algorithm we have implemented.

A previous attempt to investigate semantic subtyping for mutable records can be found in literature [5]; however, the considered model is different, and neither a proof of soundness and completeness is provided, nor an algorithm to decide subtyping is presented.

The paper is organized as follows: Section 2 introduces and motivates semantic subtyping, read/write-only field annotations, and union and intersection types, in the context of mutable records, and, hence, of object-oriented programming. After some preliminary definitions given in Section 3, the novel contributions of this paper span the next three sections: Section 4 tackles the problem of providing a consistent model for types supporting mutable records; Section 5 defines a set of subtyping rules which are sound and complete w.r.t. the model presented in the previous section; Section 6 presents an algorithm driven by the specification provided by the subtyping rules, and shows its implementation in SWI Prolog. Conclusion and directions for future work are discussed in Section 7.

2. Motivating Examples

In this section we provide some examples introducing and motivating the main features of the structural types that will be used throughout the paper: record types with read/write field annotations, union, and intersection types, recursion and coinductive interpretation to deal with circular objects.

2.1 Why Semantic Subtyping for Mutable Records?

A sound and complete procedure to decide semantic subtyping for recursive record and union types has been recently proposed [4, 9]; while this result has paved the way to the investigation of semantic subtyping relations in the context of object-oriented languages by interpreting types coinductively, its usefulness for static type analysis of object-oriented languages is limited, since the definition of subtyping and the corresponding soundness result strongly rely on the assumption that record fields are immutable.

Such an assumption turns out to be unrealistic for mainstream object-oriented languages where objects are allowed to be mutable.

According to the definition provided in the above mentioned papers, semantic subtyping between record types is covariant in the type of their fields; for instance, semantic subtyping holds for the following pair of types: $\langle f{:}int\rangle \leq \langle f{:}int \vee bool\rangle$; intuitively, this means that if a record has field f of type int, then it has also field f of type int or $bool$. This is sound as long as field f cannot be modified, but if f is in fact the field of a modifiable object, then the judgment $\langle f{:}int\rangle \leq \langle f{:}int \vee bool\rangle$ is no longer sound, as shown by the following example:

```
⟨f:int⟩ o1 = ...;
⟨f:int∨bool⟩ o2 = o1;
o2.f = true;
int i = o1.f;
```

If $\langle f{:}int\rangle \leq \langle f{:}int \vee bool\rangle$ holds, then the assignment `o2=o1` should be considered type safe, however if object assignments are by reference, as usually happens in object-oriented languages, then the line `int i = o1.f;` is unsound.

A simple solution to this problem consists in restricting subtyping between record types to make it invariant in the type of fields: $\langle f{:}\tau\rangle \leq \langle f{:}\tau'\rangle$ iff $\tau \equiv \tau'$, hence $\langle f{:}int\rangle \not\leq \langle f{:}int \vee bool\rangle$.

Even though such a solution is technically sound, it also severely restricts subtyping and, hence, makes static type analysis less precise. The examples that follow in the next subsections show how semantic subtyping allows much more precise static typing of object-oriented languages when record types are considered with read and write field annotations; furthermore, together with union types [4, 9], intersection types are introduced as well. As union types, intersection types naturally arise in the semantic subtyping approach, and allow a further increase of the expressive power of the underlying type system.

2.2 Record Types with Read/Write Annotations

We have shown that record subtyping with mutable fields must be invariant in field types to avoid unsoundness, but this severely restricts the flexibility of the type system; to avoid this problem, two different directions can be followed.

- Covariant subtyping is adopted, despite its unsoundness; this happens in the Java, and C# rule for array type subtyping: T_1`[]`$\leq$ T_2`[]` iff $T_1 \leq T_2 \leq$ `Object`. Let us consider the following simplified utility method for copying arrays:

```
static void copyarray(Object[] src, Object[] dst) {
    // omitted checks
    int i = 0;
    for (Object el : src) dst[i++] = el;
}
```

  Thanks to the covariant rule for array type subtyping, this method can work with arrays of any reference type, but the Java type system cannot prevent[1] invocation of `copyarray` to throw `ArrayStoreException` as it would happen for `copyarray(new String[]{ "one", "two"},new Integer[2])`.

- A more flexible, but sound subtyping relation is introduced, based on the idea that covariant/contravariant subtyping is sound when restricted to contexts which limit the operations available on objects. Let us consider the following example involving Java generic types.

```
class Ref<T> {
    T cont;
    Ref(T cont) { super(); this.cont = cont; }
}
static <T> void copyref(Ref<T> src, Ref<T> dst) {
    dst.cont = src.cont;
}
```

  Because subtyping for generic types is invariant, the following method invocations fail to be compiled.

```
Ref<Double> src = new Ref<>(1.0);
Ref<Integer> dst = new Ref<>(1);
Ref<Number> dst2 = new Ref<>(1);
copyref(src,dst); // type error
copyref(src,dst2); // type error
```

  Unfortunately, invariant subtyping is too rigid; for instance, `copyref(src,dst2)` does not compile, although this case causes no harm.

  To overcome this problem one can note that in the body of `copyref`, field `cont` of `src` is read, but not updated, whereas field `cont` of `dst` is updated, but not read. Therefore, in the context of method `copyref`, field `cont` of `src` can be considered read-only, although outside it might be updatable as well, while field `cont` of `dst` can be considered write-only, although outside it might be readable as well. Therefore, it is type safe to consider covariant subtyping for `src`, and contravariant subtyping for `dst`.

  In Java this can be achieved with wildcards [10, 29].

```
static <T>
void copyref2(Ref<? extends T> src, Ref<? super T> dst)
{ dst.cont = src.cont; }
copyref(src,dst); // type error
copyref(src,dst2); // statically correct
```

In Java the wildcard `Ref<? extends T>` supports covariant subtyping, whereas `Ref<? super T>` supports contravariant subtyping, therefore

```
Ref<Double>≤Ref<? extends Double>≤Ref<? extends Number>
Ref<Number>≤Ref<? super Number>≤Ref<? super Double>.
```

A similar example can be reproduced in C#, although more involved, because Java generics support call site variance annotations with wildcards, whereas C# generic interfaces and delegates support declaration site variance annotations [3].

With record types and read/write field annotations [1] the example above can be recast as follows: parameter `src` has

[1] Analogous considerations apply to C#.

type $\langle cont^{+}{:}T\rangle$, that is, a record with a read-only field `cont` (hence, allowed to be covariant) of type T: field *cont* can be always accessed, with a result of type T; parameter `dst` has type $\langle cont^{-}{:}T\rangle$, that is, a record with write-only field `cont` (hence, allowed to be contravariant) of type T: field *cont* can be always updated with a value of type T. We stress that the fact that `src` has type $\langle cont^{+}{:}T\rangle$ means that field `cont` of the object denoted by `src` can be read, but not updated inside the method, which is different from assuming that field `cont` of the object denoted by `src` must be constant; it could be updatable, but only outside the method. A dual consideration applies to parameter `dst` as well.

The subtyping rules for read-only and write-only record types are pretty intuitive: $\langle f^{+}{:}T_1\rangle \leq \langle f^{+}{:}T_2\rangle$ iff $T_1 \leq T_2$, and $\langle f^{-}{:}T_1\rangle \leq \langle f^{-}{:}T_2\rangle$ iff $T_2 \leq T_1$.

2.3 Read/Write Fields and Monotonic Initialization

Besides read-only and write-only annotations, read-write annotations are usually introduced [1] for dealing with fields that must be both readable and writable in a certain context; however, with intersection types this third kind of annotation is redundant. The record type with read-write field f of type T is represented by $\langle f^{+}{:}T\rangle\wedge\langle f^{-}{:}T\rangle$: all record where field f can be accessed, with a result of type T, **and** can be updated with a value of type T. The model defined in Section 4 ensures that subtyping is invariant as expected: $\langle f^{+}{:}T_1\rangle\wedge\langle f^{-}{:}T_1\rangle \leq \langle f^{+}{:}T_2\rangle\wedge\langle f^{-}{:}T_2\rangle$ iff $T_1 \leq T_2$ and $T_2 \leq T_1$, that is, $T_1 \equiv T_2$.

Intersection types, not only make read-write annotations superfluous, but also increase the expressive power of types because the two types assigned to f^{+} and f^{-} need not to be the same; in terms of Java wildcards that would roughly correspond in allowing parameterized types as `Ref<? super Double extends Number>`, not supported by the current Java type system, where both a lower and an upper bound for a wildcard can be declared. By combining field annotations, and intersection and union types, it is possible to enforce *monotonic initialization* [16]: objects monotonically evolve from an uninitialized to a full initialized state; for object values this guarantees that a field, once initialized with a non-null value, never becomes uninitialized again with `null`.

Let *null* be the singleton type denoting the null value, and T be a non-null type (for instance, any record type is non-null); then the type $\langle f^{+}{:}null\vee T\rangle\wedge\langle f^{-}{:}T\rangle$ forces monotonic initialization for field f: access to f may evaluate to null **or** to a value of type T, but only non-null values of type T can be assigned to f. A more detailed example of monotonic initialization in presence of circular objects can be found below.

2.4 Multiple Fields and Recursive Types

With intersection types, record types need not contain multiple fields as it is usually required for types in structural type systems for object-oriented languages; all we need is singleton record types, because a type as $\langle elem^{+}{:}int, next^{+}{:}null\rangle$ simply reads as an abbreviation for $\langle elem^{+}{:}int\rangle\wedge\langle next^{+}{:}null\rangle$.

We can now consider a more complex example involving recursive types specifying node objects in linked lists. In the Java standard API a unique interface is used for defining three different kinds of lists:

1. *unmodifiable* lists: their elements cannot be replaced nor their size can be modified;
2. *structurally unmodifiable* lists: their elements can be replaced, but their size cannot be modified;
3. *structurally modifiable* lists: their elements can be replaced, and their size can be modified.

Accordingly, the node objects composing an unmodifiable/structurally unmodifiable/modifiable linked list of elements of type T will have the following types, respectively.

1. $\tau_1 = null\vee(\langle elem^{+}{:}T\rangle\wedge\langle next^{+}{:}\tau_1\rangle)$;
2. $\tau_2 = null\vee(\langle elem^{+}{:}T\rangle\wedge\langle elem^{-}{:}T\rangle\wedge\langle next^{+}{:}\tau_2\rangle)$;
3. $\tau_3 = null\vee(\langle elem^{+}{:}T\rangle\wedge\langle elem^{-}{:}T\rangle\wedge \langle next^{+}{:}\tau_3\rangle\wedge\langle next^{-}{:}\tau_3\rangle)$.

All types are recursively defined by means of syntactic equations, hence they correspond to *regular trees* (a.k.a. as rational trees or cyclic terms, see Section 3 for further details); there are two main advantages in managing recursion with regular trees, instead of introducing a μ recursive binder:

- subtyping rules are simpler, since they can be presented at a more abstract level;
- cyclic terms are naturally supported by SWI Prolog, hence, there is also a practical advantage; for instance, the unification[2] `T=rp(f:T)` succeeds, and the logical variable `T` is instantiated with the unique cyclic term satisfying the equation above. As expected, the following SWI Prolog query (where `==` denotes syntactic equality) succeeds: `T1=rp(f:T1),T2=rp(f:rp(f:T2)),T1==T2.`

Let us comment on type τ_1; if linked lists are not circular, and no dummy node is employed, then the first node can be null if the list is empty, or (union type) it can be an object with a read-only field *elem* of type T, and (intersection type) a read-only field *next* of type τ_1; both fields are read-only because nodes belong to unmodifiable lists. In τ_2 field *elem* is read-write, while *next* is still read-only, whereas in τ_3 both fields are read-write. As expected, in our model $\tau_3 \leq \tau_2 \leq \tau_1$, and $\tau_1 \not\leq \tau_2 \not\leq \tau_3$ (and $\tau_1 \not\leq \tau_3$ as well), capturing the intuition that, for instance, a structurally modifiable list can be safely considered as an unmodifiable list, but not the other way round.

2.5 Coinductive Interpretation of Types

Coinduction is employed at two different levels that must not be confused.

[2] `rp(f:T)` is the Prolog term corresponding to the type $\langle f^{+}{:}T\rangle$.

At the *syntactic* level, types are regular trees, which include also infinite trees, hence they cannot be defined inductively. Regular trees allow a convenient treatment of recursive types, as previously motivated, and the same approach has been taken for ℂDuce [17].

At the *semantic* level, types are interpreted as set of values, and when types are recursive their interpretation is recursive as well, and admits both a least and a greatest fixed-point, corresponding to an inductive and coinductive interpretation, respectively.

Let us consider the recursive type specifying nodes of circular linked lists with a dummy node: $\tau = \langle elem^{+}{:}T\rangle \wedge \langle next^{+}{:}\tau\rangle$. Because of circularity, and the dummy node, in this case nodes cannot be null, as opposed to the previous examples.

The interpretation of τ, denoted by $[\![\tau]\!]$, has to satisfy the equation $[\![\tau]\!] = [\![\langle elem^{+}{:}T\rangle \wedge \langle next^{+}{:}\tau\rangle]\!] = [\![\langle elem^{+}{:}T\rangle]\!] \cap [\![\langle next^{+}{:}\tau\rangle]\!]$. The inductive interpretation (least fixed-point) is the empty set, because no well-founded object can have such a type, whereas the coinductive interpretation (greatest fixed-point) is not empty; indeed, it contains non-well-founded objects constituting circular lists.

In our model, $\tau \leq \tau_1$ (with τ_1 as defined in Section 2.4), whereas $\tau_1 \not\leq \tau$; this corresponds to the intuition that a circular linked list can be safely considered as a linked list, but not the other way round. To grasp better such an intuition, let us consider the following code[3] snippet.

```
Node<T> getNode(int index) {
    Node<T> res = this;
    for (int i = 0; i < index; i++)
        res = res.next;
    return res;
}
```

If we assume that `this` (and, hence, variable `res`) has type τ_1, then it is not possible to deduce that the execution of `getNode` cannot throw `NullPointerException`; indeed, `res` could contain null, and, hence, `res.next` could throw `NullPointerException`. However, if we assume that `this` has the more specific type τ, then it is possible to statically guarantee that the method never throws `NullPointerException`.

2.6 Empty Type

With intersection it is possible to define types equivalent to the empty (a.k.a. bottom) type $\mathbf{0}$, whose interpretation is, by definition, the empty set (hence, such types are not inhabited); a simple example is given by $bool \wedge int$, or $\langle f^{+}{:}bool \wedge int\rangle$, but arbitrary complex types which are not inhabited can be constructed.

As it will be more evident in Section 5, type emptiness is necessary to ensure a sound and complete subtyping decision procedure; indeed, the judgment $\sigma_1 \leq \sigma_2$ holds whenever it is possible to derive that σ_1 is empty, but deciding type emptiness is not trivial. Hence, it is not possible to devise a sound and complete subtyping decision procedure, without a procedure to decide whether a type is empty; as shown in Section 5, checking type emptiness plays a fundamental role also in defining a sound and complete decision procedure for type emptiness, and it is quite challenging, because of the interplay between read-only and write-only fields.

Let us consider the type $\langle f^{-}{:}bool \vee int\rangle \wedge \langle f^{+}{:}int\rangle$; this type cannot be inhabited, because, otherwise, the following code would typecheck.

```
⟨f⁻:bool∨int⟩∧⟨f⁺:int⟩ r = ...;
r.f = true;
int i = r.f;
```

The second line of the code above typechecks because, according to type $\langle f^{-}{:}bool \vee int\rangle$, it is allowed to assign boolean or integer values to field `f` of `r`; the subsequent line typechecks as well, because, according to type $\langle f^{+}{:}int\rangle$, it is allowed to access field `f` of `r` to get an integer value. Hence, the code must not typecheck because of the declaration of `r`: its type is empty.

More generally, $\langle f^{-}{:}\tau_1\rangle \wedge \langle f^{+}{:}\tau_2\rangle \neq \mathbf{0}$ only if $\tau_1 \leq \tau_2$; this dependency between non-empty types and subtyping makes the problem of devising a decision procedure for subtyping more challenging.

The interplay between read-only and write-only fields has also some subtle effect on subtyping. In a functional setting where fields are always immutable, the two types $\langle f{:}\tau_1 \vee \tau_2\rangle$ and $\langle f{:}\tau_1\rangle \vee \langle f{:}\tau_2\rangle$ are equivalent [4]. Surprisingly, in an imperative setting where fields are allowed to be mutable $\langle f^{+}{:}\tau_1\rangle \vee \langle f^{+}{:}\tau_2\rangle \leq \langle f^{+}{:}\tau_1 \vee \tau_2\rangle$ always holds, but $\langle f^{+}{:}\tau_1 \vee \tau_2\rangle \leq \langle f^{+}{:}\tau_1\rangle \vee \langle f^{+}{:}\tau_2\rangle$ holds only if $\tau_1 \leq \tau_2$ or $\tau_2 \leq \tau_1$.

For instance, $\langle f^{+}{:}bool \vee int\rangle \not\leq \langle f^{+}{:}bool\rangle \vee \langle f^{+}{:}int\rangle$; indeed, $\langle f^{+}{:}bool \vee int\rangle \leq \langle f^{+}{:}bool\rangle \vee \langle f^{+}{:}int\rangle$ is unsound, because we could deduce from it the following inequalities, where the last one is clearly unsound because $\langle f^{+}{:}bool \vee int\rangle \not\leq \langle f^{+}{:}bool\rangle$. Let set $\tau = \langle f^{+}{:}bool \vee int\rangle \wedge \langle f^{-}{:}bool\rangle$; then we have the following derivation:

$$\begin{gathered}
\tau \leq (\langle f^{+}{:}bool\rangle \vee \langle f^{+}{:}int\rangle) \wedge \langle f^{-}{:}bool\rangle \\
\Leftrightarrow \\
\tau \leq (\langle f^{+}{:}bool\rangle \wedge \langle f^{-}{:}bool\rangle) \vee (\langle f^{+}{:}int\rangle \wedge \langle f^{-}{:}bool\rangle) \\
\Leftrightarrow \\
\tau \leq (\langle f^{+}{:}bool\rangle \wedge \langle f^{-}{:}bool\rangle) \vee \mathbf{0} \\
\Leftrightarrow \\
\tau \leq \langle f^{+}{:}bool\rangle \wedge \langle f^{-}{:}bool\rangle
\end{gathered}$$

Interestingly enough, while union does not distribute over immutable records, in Section 4 we show that distributivity of intersection over immutable records is sound.

Finally, type emptiness checks can be useful also for warning users, as already happens in the ℂDuce implementation. While an empty return type could be reasonable for a non terminating function, declaring/inferring an empty type for parameters or local variables is more questionable: parameters and variables that are not allowed to carry values are symptoms of latent bugs; in particular, a function with a pa-

[3] We use Java, but any other object-oriented language would equally work for the purpose.

rameter with an empty type is unusable, because it can never be applied to any value.

3. Technical Background

In this section we overview the main technical notions employed in our formal treatment, and motivate them.

In the rest of the paper, values are modeled by finitely branching trees which are allowed to contain infinite paths, where nodes correspond to value constructors, and the number of children of a node corresponds to its arity. Infinite paths can correspond either to cyclic values (for instance, a record containing itself as a member), or to non-well founded values that can be obtained by a diverging computation.

Analogously, type expressions are modeled by finitely branching trees which are allowed to contain infinite paths, where nodes correspond to type constructors, and the number of children of a node corresponds to its arity; paths can be infinite because types can be recursive. However, since types always correspond to finite sets of definitions, differently from values, type expressions are modeled as *regular trees* (see below).

Trees with infinite paths arise also when considering proof trees of coinductive judgments (see the next subsection).

A formalization of trees with infinite paths and their main properties has been given by Courcelle [15].

3.1 Types and Trees

Recursive types are modeled by infinite but regular trees.

Definition 1. *A regular tree is a possibly infinite tree containing a finite set of subtrees.*

Trees representing type expressions are regular because each (possibly recursive) type is defined by a finite set of definitions (that is, syntactic equations).

The following proposition states a well-known property of regular terms [15].

A system of guarded equations is a finite set of syntactic equations of shape $X = e$, where X is a variable, and e may contain variables, such that there exist no subsets of equations having shape $X_0 = X_1, \ldots, X_n = X_0$.

A solution to a set of guarded equations is a substitution to all variables contained in the equations that satisfies all syntactic equations.

Propostion 1. *Every regular tree t can be represented by a system of guarded equations.*

We define types as all regular trees built on top of the type constructors whose syntax is defined as follows.

$$\begin{array}{lcl} \tau, \rho & ::= & \mathbf{0} \mid \mathbf{1} \mid \mathit{int} \mid \mathit{null} \mid \langle\rangle \mid \langle f^{\nu}{:}\tau\rangle \mid \tau_1 \vee \tau_2 \mid \tau_1 \wedge \tau_2 \\ \nu & ::= & + \mid - \end{array}$$

Since trees are allowed to be infinite (although regular), this is not the standard inductive definition that would generate just finite trees.

The type $\langle\rangle$ represents all record values. The singleton record types $\langle f^{+}{:}\tau\rangle$ and $\langle f^{-}{:}\tau\rangle$ represent all record values containing the read-only, and the write-only field f, respectively.

Union types $\tau_1 \vee \tau_2$ and intersection types $\tau_1 \wedge \tau_2$ [8, 22] correspond to logic disjunction and conjunction, respectively. Types $\mathbf{0}$ and $\mathbf{1}$ are the empty, and the universe (a.k.a. top) type, int represents the set $\mathbb{Z}$, and null denotes the singleton set containing the null reference; the technical treatment can be easily extended to include other primitive types.

Example 1. *Let us consider the type τ_1 introduced in Section 2, such that $\tau_1 = \mathit{null} \vee (\langle \mathit{elem}^{+}{:}T\rangle \wedge \langle \mathit{next}^{+}{:}\tau_1\rangle)$, and let us fix T to be int for simplicity. Then τ_1 is infinite but regular and has only the following six subterms (in infinite regular types a type can be subterm of itself as it happens in this example):*

$$\tau_1 \quad \mathit{null} \quad \langle \mathit{elem}^{+}{:}\mathit{int}\rangle \wedge \langle \mathit{next}^{+}{:}\tau_1\rangle$$

$$\langle \mathit{elem}^{+}{:}\mathit{int}\rangle \quad \langle \mathit{next}^{+}{:}\tau_1\rangle \quad \mathit{int}$$

Let us consider the types τ'_i such that the following equations hold for all natural numbers i.

$$\begin{array}{lcl} \tau'_0 & = & \mathit{null} \\ \tau'_{i+1} & = & \langle \mathit{pred}^{+}{:}\tau'_i\rangle \end{array}$$

The type $\tau'_0 \vee \tau'_1 \vee \ldots \vee \tau'_n \vee \tau'_{n+1} \ldots$ is infinite, and not regular.

We now introduce the notion of *contractive* type, which allows us to rule out all those types whose intended interpretation is not coinductive.

Definition 2. *A type is contractive if all its infinite paths contain a record type constructor.*

The definition above requires all recursive types to be guarded by a record constructor.

Example 2. *The type τ s.t. $\tau = \tau \vee \mathit{int}$ is not contractive, because there exists an infinite path whose nodes are all labeled by the union type constructor.*

$$\begin{tikzcd} & \mathit{int} & & \mathit{int} & & \mathit{int} \\ \vee \arrow[ur] \arrow[rr] & & \vee \arrow[ur] \arrow[rr] & & \vee \arrow[ur] \arrow[rr, dashed] & & {} \end{tikzcd}$$

The type τ s.t. $\tau = \langle f^{+}{:}\tau \vee \mathit{int}\rangle$ is contractive because all infinite paths have nodes that are alternatively labeled by a record and a union type.

$$\begin{tikzcd} & & \mathit{int} & & & \mathit{int} \\ \langle f^{+}{:}\_\rangle \arrow[r] & \vee \arrow[ur] \arrow[r] & \langle f^{+}{:}\_\rangle \arrow[r] & \vee \arrow[ur] \arrow[r] & \langle f^{+}{:}\_\rangle \arrow[r, dashed] & {} \end{tikzcd}$$

In the non contractive type τ s.t. $\tau = \tau \vee \mathit{int}$ the intended interpretation is the inductive one: $[\![\tau]\!] = \mathbb{Z}$ (the coinductive interpretation is the top type $\mathbf{1}$); in each proof tree showing that a value has type τ the right operand of the union constructor has to be necessarily selected to actually check something and "consume" the value [6]. Hence, in this case proof trees are required to be finite.

By contrast, in the contractive type τ s.t. $\tau = \langle f^{+}{:}\tau \vee int\rangle$ the proof tree showing that a value has type τ can be infinite, because each time the rule for record is applied, a check is performed and the value is "consumed". In case the value is not-well-founded (that is, it is a circular object), the rule is applied for an infinite number of times.

Non contractive types do not increase the expressive power of the types, since a non-contractive type can be always turned into an equivalent contractive one [6]. For instance τ s.t. $\tau = \tau \vee int$ is just equivalent to int. Contractive types simplify the formalization, indeed, several definitions and proofs of the claims stated in this paper exploit the assumption that types are contractive; furthermore, contractive types allow optimizations when collecting the set of coinductive hypotheses required to support coinduction in the implementation (Section 6).

For this reason, in the rest of the paper all types are restricted to be regular and contractive.

3.2 Inference Systems and Proof Trees

In the technical sections of the paper several judgments are defined by means of inference systems [2], and, depending on the specific judgment, such inference systems are interpreted either inductively, or coinductively. For all considered inference systems, the associated one step inference operator is monotone (hence, negation is not used in the inference rules), hence the existence of both the least and the greatest fixed point, corresponding to the inductive and coinductive interpretation of the system, respectively, is guaranteed by the Knaster-Tarski theorem.

While the proof trees of inductive judgments have always a finite depth, for coinductive judgments their corresponding proof trees are allowed to contain infinite paths [26].

Since we consider both inductive and coinductive inference systems, we adopt a standard notation [26] for distinguishing inference systems whose intended interpretation is inductive from those having a coinductive intended interpretation: in the latter case, the horizontal lines of the inference rules are thicker.

4. Semantic Subtyping

In the semantic subtyping approach subtyping is not defined in terms of axioms, usually expressed with an inference system, but coincides with set theoretic inclusion between type interpretations:

$$\tau_1 \leq \tau_2 \text{ iff } [\![\tau_1]\!] \subseteq [\![\tau_2]\!] .$$

The interpretation $[\![\tau]\!]$ of a type τ is the set of values that have type τ, hence, the definition of $[\![\tau]\!]$ directly depends on the judgment, denoted in this paper by $v \in \tau$, assigning types to values.

$$[\![\tau]\!] = \{v \mid v \in \tau\}.$$

As a consequence, besides types, in the semantic subtyping approach there are other two basic notions that have to be necessarily provided:

- the set of values, and
- the judgment $v \in \tau$ has to be defined.

Since type interpretation does not depend on the notion of term, but uniquely relies on values, semantics subtyping allows a certain degree of language independence; such an independence is reinforced by the fact that an abstract notion of value is employed: values in type interpretations do not coincide with concrete values manipulated by programs. For instance, in our model fields are annotated with type information usually not present in the runtime model of languages. This abstraction allows the same definition of subtyping to be successfully adopted for a family of programming languages sharing some features.

Except for the basic types $null$ and int for which the interpretation is straightforward, the other values are records. In the functional setting, immutable record values are maps from field labels to values, and in the coinductive interpretation they are allowed to be non-well-founded, that is, they can be infinite trees.

For instance, the value uniquely defined by the equation $v = \langle f \mapsto v\rangle$ corresponds to the immutable record with one field f associated with the value itself; that is, v corresponds to a cyclic object.

Mutable record values are more complex, since they have also to carry the information specifying which values can be safely stored in the fields of the record; for instance, the interpretation of type $\langle f^{-}{:}int \vee null\rangle$ must contain all record values where updating f with either an integer or null is type safe, while $\langle f^{-}{:}int\rangle$ must contain all record values where updating f with an integer is type safe, and, by contravariance, the type $\langle f^{-}{:}int \vee null\rangle$ must contain less values than those contained in $\langle f^{-}{:}int\rangle$: a record value where updating f with an integer is type safe cannot be contained in type $\langle f^{-}{:}int \vee null\rangle$ which can be safely used also in contexts where f is updated with null.

Therefore the information "field f can be safely updated with values of type τ" must be contained in record values; a simple way to do that is associating field f with type τ, thus introducing a circularity that requires to be properly managed as explained in the rest of this section, since types are interpreted in terms of types.

Values are all finite and infinite trees built on top of the type constructors whose syntax is defined as follows (where $i \in \mathbb{Z}$).

$$\begin{array}{lcl} v & ::= & i \mid \text{null} \mid \langle f_1 \mapsto (\kappa_1, \rho_1), \ldots, f_n \mapsto (\kappa_n, \rho_n)\rangle \\ \kappa & ::= & \emptyset \mid \{v\} \end{array}$$

Record values are finite domain maps from field labels to pairs, hence field names are implicitly assumed to be distinct, and their order is immaterial. The first component of the pair, κ, is a set of cardinality ≤ 1; when empty, it means that its corresponding field is not readable; when it has shape $\{v\}$, then the field is readable, and associated with value v. The second component of the pair is a type that specifies the values that can be stored in the corresponding field. We denote with $\mathbb{R}\mathrm{c}$ the set of all record values.

As an example, the record value

$$\langle f \mapsto (\emptyset, \mathbf{1}), g \mapsto (\{42\}, \mathbf{0}), h \mapsto (\{\text{null}\}, \mathit{int})\rangle$$

is the record value with three fields f, g, and h; field f is write-only: the empty set $\emptyset$ means that read access for f is forbidden (hence, we do not know the value associated with f), while the top type $\mathbf{1}$ specifies that f can be safely updated with any value; field g is read-only: it is associated with 42, and its value can be read, while the bottom type $\mathbf{0}$ specifies that g can be safely updated with no values; field h is read-write: it is associated with the value null, which is allowed to be read, and can be safely updated with values of type int.

There are three reasons that make the definition of the judgment $v \in \tau$ non trivial.

1. Values can be non-well-founded and types can be infinite regular trees, therefore the judgment has to be defined coinductively; for instance, it should be possible to derive $v \in \tau$ when $v = \langle f \mapsto (\{v\}, \mathbf{0})\rangle$, and $\tau = \langle f^{+}{:}\tau\rangle$, that is, the read-only cyclic record value v has the recursive type $\tau = \langle f^{+}{:}\tau\rangle$.
2. If $\tau = \langle f^{-}{:}\tau'\rangle$, or $\tau = \langle f^{+}{:}\tau'\rangle$, then the validity of the judgment $v \in \tau$ depends on the validity of subtyping between τ' and the type contained in the record value. For $\tau = \langle f^{-}{:}\tau'\rangle$ this dependency is rather intuitive; for instance, by contravariance, $\langle f \mapsto (\kappa, \rho)\rangle \in \langle f^{-}{:}\tau'\rangle$ only if $\tau' \leq \rho$. For $\tau = \langle f^{+}{:}\tau'\rangle$ the situation is more involved: $\langle f \mapsto (\kappa, \rho)\rangle \in \langle f^{+}{:}\tau'\rangle$ requires $\kappa = \{v'\}$, and $v' \in \tau'$, but these requirements are not sufficient to guarantee that $\langle f \mapsto (\kappa, \rho)\rangle \in \langle f^{+}{:}\tau'\rangle$ holds; the condition $\tau' \leq \rho$ is needed as well, otherwise $\langle f^{+}{:}\tau'\rangle \wedge \langle f^{-}{:}\rho\rangle$ would be always non empty, even when $\tau' \not\leq \rho$. Indeed, if $v' \in \tau'$, then we would get that both $\langle f \mapsto (\{v'\}, \rho)\rangle \in \langle f^{+}{:}\tau'\rangle$ and $\langle f \mapsto (\{v'\}, \rho)\rangle \in \langle f^{-}{:}\rho\rangle$ hold, hence $\langle f \mapsto (\{v'\}, \rho)\rangle \in \langle f^{+}{:}\tau'\rangle \wedge \langle f^{-}{:}\rho\rangle$ would hold, and, therefore, $\langle f^{+}{:}\tau'\rangle \wedge \langle f^{-}{:}\rho\rangle$ would be non-empty. But assuming that $\langle f^{+}{:}\tau'\rangle \wedge \langle f^{-}{:}\rho\rangle$ is non-empty, even when $\tau' \not\leq \rho$, leads to unsoundness. Let us consider, for instance, the following function declaration:

```
int foo(⟨f+:int⟩∧⟨f−:null⟩ x) {
   x.f=null;
   return x.f+1;
}
```

The first line of the body of the function is type safe because `x` has type $\langle f^{+}{:}\mathit{int}\rangle \wedge \langle f^{-}{:}\mathit{null}\rangle$, and $\langle f^{+}{:}\mathit{int}\rangle \wedge \langle f^{-}{:}\mathit{null}\rangle \leq \langle f^{-}{:}\mathit{null}\rangle$, while the second line is type safe because `x` has type $\langle f^{+}{:}\mathit{int}\rangle \wedge \langle f^{-}{:}\mathit{null}\rangle$, and $\langle f^{+}{:}\mathit{int}\rangle \wedge \langle f^{-}{:}\mathit{null}\rangle \leq \langle f^{+}{:}\mathit{int}\rangle$; but if the type $\langle f^{+}{:}\mathit{int}\rangle \wedge \langle f^{-}{:}\mathit{null}\rangle$ is not empty, then it would be possible to call function `foo` with a value in $\langle f^{+}{:}\mathit{int}\rangle \wedge \langle f^{-}{:}\mathit{null}\rangle$, but the execution of the body of the function would lead to a type error.
3. There exists a circularity between the definition of $v \in \tau$ and $\tau \leq \tau'$ that has to be properly managed, to show that the two judgments are well-defined.

$$\begin{array}{l} \tau_1 \leq \tau_2 \Leftrightarrow (\forall v. v \in \tau_1 \Rightarrow v \in \tau_2) \\ \langle f \mapsto (\{v\}, \rho)\rangle \in \langle f^{+}{:}\tau\rangle \Leftrightarrow (v \in \tau, \rho \leq \tau) \\ \langle f \mapsto (\kappa, \rho)\rangle \in \langle f^{-}{:}\tau\rangle \Leftrightarrow \tau \leq \rho \end{array}$$

Such a circularity can be removed by replacing $\rho \leq \tau$, and $\tau \leq \rho$ on the second and third line with the definition of semantic subtyping given on the first line:

$$\begin{array}{l} \langle f \mapsto (\{v\}, \rho)\rangle \in \langle f^{+}{:}\tau\rangle \Leftrightarrow (v \in \tau, \forall v'. v' \in \rho \Rightarrow v' \in \tau) \\ \langle f \mapsto (\kappa, \rho)\rangle \in \langle f^{-}{:}\tau\rangle \Leftrightarrow (\forall v'. v' \in \tau \Rightarrow v' \in \rho) \end{array}$$

In this way we obtain a recursive definition for the judgment $v \in \tau$ which does not depend on the definition of $\tau \leq \tau'$; such a definition has to be interpreted coinductively, since $v \in \tau$ is expected to hold when $v = \langle f \mapsto (\{v\}, \mathbf{0})\rangle$ and $\tau = \langle f^{+}{:}\tau\rangle$. Unfortunately, the occurrence of $\Rightarrow$ on the right hand side of $\Leftrightarrow$ makes the recursive definition of $v \in \tau$ problematic, because both $v' \in \rho$ and $v' \in \tau$ occur negatively: $v' \in \rho \Rightarrow v' \in \tau$ is equivalent to "not $v' \in \rho$ or $v' \in \tau$", and $v' \in \tau \Rightarrow v' \in \rho$ is equivalent to "not $v' \in \tau$ or $v' \in \rho$". For this reason, the existence of the greatest fixed point is not guaranteed, because the Knaster-Tarski fixed point theorem cannot be applied. To avoid this problem, the negative judgment $v \notin \tau$ is explicitly introduced and defined; in this way, the definitions above can be rewritten as follows:

$$\begin{array}{l} \langle f \mapsto (\{v\}, \rho)\rangle \in \langle f^{+}{:}\tau\rangle \Leftrightarrow (v \in \tau, \forall v'. v' \notin \rho \text{ or } v' \in \tau) \\ \langle f \mapsto (\kappa, \rho)\rangle \in \langle f^{-}{:}\tau\rangle \Leftrightarrow (\forall v'. v' \notin \tau \text{ or } v' \in \rho) \end{array}$$

The definitions of the two judgments $v \in \tau$, and $v \notin \tau$ are mutually recursive; for instance, $\langle f \mapsto (\kappa, \rho)\rangle \notin \langle f^{-}{:}\tau\rangle$ holds if and only if there exists v s.t. $v \in \tau$, and $v \notin \rho$ (that is, $\tau \not\leq \rho$). Unfortunately, such a circularity is problematic because the judgment $v \in \tau$ is interpreted coinductively, whereas, by duality, $v \notin \tau$ requires an inductive definition; for instance, $v \notin \tau$ must not hold when $v = \langle f \mapsto (\{v\}, \mathbf{0})\rangle$, and $\tau = \langle f^{+}{:}\tau\rangle$. Mutual dependencies between coinductive and inductive judgments is critical [28], since neither the least fixed point, nor the greatest fixed point semantics works properly for mutually recursive definitions involving both coinductive and inductive judgments; to our knowledge, no solution has been proposed in literature for dealing with mutual recursion between inductive and coinductive definitions. To break such a circularity we propose a simple approach which consists in transforming the judgment $v \notin \tau$ into $\Gamma \vdash v \notin \tau$, where Γ is a set of pairs (v, τ), each one corresponding to the hypothesis

$v \in \tau$; accordingly, the meaning of $\Gamma \vdash v \notin \tau$ is as follows: value v does not have type τ, under the assumption that v' has type τ' for all $(v', \tau') \in \Gamma$.

In this way, $v \in \tau$ is defined in terms of both itself, and the judgment $\Gamma \vdash v \notin \tau$, whereas $\Gamma \vdash v \notin \tau$ is defined only in terms of itself; therefore, the definition of the two judgments can be stratified: first, the least fixed point of the recursive definition of $\Gamma \vdash v \notin \tau$ is considered, and its existence is guaranteed by monotonicity, and the Knaster-Tarski theorem. Then, on top of the judgment $\Gamma \vdash v \notin \tau$, the judgment $v \in \tau$ can be defined coinductively; also in this case, the existence of the greatest fixed point is guaranteed by monotonicity, and the Knaster-Tarski theorem.

The complete definitions for the two judgments $v \in \tau$ and $\Gamma \vdash v \notin \tau$ is given in Figure 1 and Figure 2, respectively. We recall that both $v \in \tau$ and $\Gamma \vdash v \notin \tau$ are semantic judgments used to interpret types, and not intended to be computable (indeed, they are not, because values are allowed to be non regular).

Except for the cases on record types, all rules of Figure 1 are straightforward. In rule (rec $\in$) $\langle \ldots \rangle$ denotes any possible record value. The premises of rule (rec$^-$ $\in$) are equivalent to the condition that for all v either v does not have type τ, or v has type ρ (that is, if v has type τ, then it has also type ρ, and, hence, $\tau \leq \rho$); the negative version of the judgment $v \in \tau$ requires a set Γ (which is existentially quantified) of abducted assumptions that have to be verified (hence, $\forall (v_\Gamma, \tau_\Gamma) \in \Gamma . v_\Gamma \in \tau_\Gamma$).

Similar comments apply for rule (rec$^+$ $\in$), but here ρ is expected to be a subtype of τ (recall the explanations given in item 2 on page 8); furthermore, $v \in \tau$ must hold.

Also for the rules in Figure 2 defining $\Gamma \vdash v \notin \tau$, we comment the less trivial cases, that is, when record types are considered.

The judgment $\Gamma \vdash v \notin \langle f^-{:}\tau \rangle$ is derivable in two cases;

- rule (rec$^-$1 $\notin$): v is not a record value, or is a record value with no field f (condition $v \neq \langle f \mapsto \_, \ldots \rangle$);
- rule (rec$^-$2 $\notin$): $v = \langle f \mapsto (\_, \rho), \ldots \rangle$ and $\tau \not\leq \rho$, hence, there exists v' that has type τ (condition $(v', \tau) \in \Gamma$) but does not have type ρ (premise $\Gamma \vdash v' \notin \rho$).

The judgment $\Gamma \vdash v \notin \langle f^+{:}\tau \rangle$ is derivable in three cases;

- rule (rec$^+$1 $\notin$): v is not a record value, or is a record value with no field f (condition $v \neq \langle f \mapsto \_, \ldots \rangle$);
- rule (rec$^+$2 $\notin$): $v = \langle f \mapsto (\_, \rho), \ldots \rangle$ and $\rho \not\leq \tau$, hence, there exists v' that has type ρ (condition $(v', \rho) \in \Gamma$) but does not have type τ (premise $\Gamma \vdash v' \notin \tau$);
- rule (rec$^+$3 $\notin$): $v = \langle f \mapsto (\kappa, \_), \ldots \rangle$, where either $\kappa = \emptyset$, or $\kappa = \{v'\}$, but v' does not have type τ (premise $\Gamma \vdash v' \notin \tau$).

We have arranged the recursive definitions of $v \in \tau$ and $\Gamma \vdash v \notin \tau$ in such a way that we know that there exist the greatest and the least fixed point for the former and the latter judgment, respectively. However, we need to show that one is the negation of the other. This is guaranteed by the following two lemmas.

In the following we denote by OK^Γ the set of all correct assumptions Γ, that is, those containing only derivable pairs:

$$\Gamma \in OK^\Gamma \text{ iff } v \in \tau \text{ is derivable for all } (v, \tau) \in \Gamma.$$

Lemma 1. *If there exists $\Gamma \in OK^\Gamma$ s.t. $\Gamma \vdash v \notin \tau$ is derivable, then $v \in \tau$ is not derivable.*

Proof. The proof is based on a main lemma proved by arithmetic induction and based on the approximation of $\Gamma \vdash v \notin \tau$, and $v \in \tau$. $\square$

Lemma 2. *If for all $\Gamma \in OK^\Gamma$ $\Gamma \vdash v \notin \tau$ is not derivable, then $v \in \tau$ is derivable.*

Proof. The proof is based on a main lemma proved by arithmetic induction and based on the approximation of $\Gamma \vdash v \notin \tau$, and $v \in \tau$. $\square$

4.1 Laws

We conclude this section by showing some of the main laws satisfied by the semantic model, and exploited in the next section for defining the subtyping rules. Most of the proofs of these laws are tacitly based on Lemma 1 and Lemma 2.

As discussed in Section 2 the intersection between read-only and write-only record types, $\langle f^+{:}\tau_1 \rangle \wedge \langle f^-{:}\tau_2 \rangle$, is non-empty only when τ_2 is a subtype of τ_1.

Law 1. $[\![\langle f^+{:}\tau_1 \rangle \wedge \langle f^-{:}\tau_2 \rangle]\!] \neq \emptyset$ *iff* $[\![\tau_2]\!] \subseteq [\![\tau_1]\!]$.

Some (but not all) of the laws that hold in a functional setting where record fields are immutable [4] can be proved for mutable fields as well.

A first example of law that holds also for mutable records concerns empty read-only records.

Law 2. $[\![\langle f^+{:}\tau \rangle]\!] = \emptyset$ *iff* $[\![\tau]\!] = \emptyset$

As opposed to records with read-only fields, records with write-only fields can never be empty.

Law 3. $[\![\langle f^-{:}\tau \rangle]\!] \neq \emptyset$

In particular, the type $\langle f^-{:}\tau \rangle$, where $[\![\tau]\!] = \emptyset$, represents all records having field f.

Law 4. $[\![\langle f^-{:}\tau' \rangle]\!] \subseteq [\![\langle f^-{:}\tau \rangle]\!]$ *and* $[\![\langle f^+{:}\tau' \rangle]\!] \subseteq [\![\langle f^-{:}\tau \rangle]\!]$ *if* $[\![\tau]\!] = \emptyset$

Intersection and record types behave as expected.

Law 5.

1. $[\![\langle f^+{:}\tau_1 \rangle \wedge \langle f^+{:}\tau_2 \rangle]\!] = [\![\langle f^+{:}\tau_1 \wedge \tau_2 \rangle]\!]$
2. $[\![\langle f^-{:}\tau_1 \rangle \wedge \langle f^-{:}\tau_2 \rangle]\!] = [\![\langle f^-{:}\tau_1 \vee \tau_2 \rangle]\!]$

Surprisingly, union types do not behave similarly, as revealed in Section 2.6.

$$
\text{(any }\epsilon)\frac{}{v \mathrel{\epsilon} \mathbf{1}}
\qquad
\text{(null }\epsilon)\frac{}{\text{null} \mathrel{\epsilon} \mathit{null}}
\qquad
\text{(int }\epsilon)\frac{}{i \mathrel{\epsilon} \mathit{int}}
\qquad
\text{(rec }\epsilon)\frac{}{\langle \ldots \rangle \mathrel{\epsilon} \langle\rangle}
$$

$$
\text{(and }\epsilon)\frac{v \mathrel{\epsilon} \tau_1 \text{ and } v \mathrel{\epsilon} \tau_2}{v \mathrel{\epsilon} \tau_1 \wedge \tau_2}
\qquad
\text{(rec}^- \ \epsilon)\frac{\forall v.(\exists \Gamma.\Gamma \vdash v \notin \tau \text{ and } \forall (v_\Gamma, \tau_\Gamma) \in \Gamma. v_\Gamma \mathrel{\epsilon} \tau_\Gamma) \text{ or } v \mathrel{\epsilon} \rho}{\langle f \mapsto (\_, \rho), \ldots \rangle \mathrel{\epsilon} \langle f^-{:}\tau \rangle}
$$

$$
\text{(or }\epsilon)\frac{v \mathrel{\epsilon} \tau_1 \text{ or } v \mathrel{\epsilon} \tau_2}{v \mathrel{\epsilon} \tau_1 \vee \tau_2}
\qquad
\text{(rec}^+ \ \epsilon)\frac{v \mathrel{\epsilon} \tau \text{ and } \forall v'.(\exists \Gamma.\Gamma \vdash v' \notin \rho \text{ and } \forall (v_\Gamma, \tau_\Gamma) \in \Gamma. v_\Gamma \mathrel{\epsilon} \tau_\Gamma) \text{ or } v' \mathrel{\epsilon} \tau}{\langle f \mapsto (\{v\}, \rho), \ldots \rangle \mathrel{\epsilon} \langle f^+{:}\tau \rangle}
$$

Figure 1. Definition of $v \mathrel{\epsilon} \tau$

$$
\text{(empty }\notin)\frac{}{\Gamma \vdash v \notin \mathbf{0}}
\qquad
\text{(null }\notin)\frac{v \neq \text{null}}{\Gamma \vdash v \notin \mathit{null}}
\qquad
\text{(or }\notin)\frac{\Gamma \vdash v \notin \tau_1 \text{ and } \Gamma \vdash v \notin \tau_2}{\Gamma \vdash v \notin \tau_1 \vee \tau_2}
\qquad
\text{(and }\notin)\frac{\Gamma \vdash v \notin \tau_1 \text{ or } \Gamma \vdash v \notin \tau_2}{\Gamma \vdash v \notin \tau_1 \wedge \tau_2}
$$

$$
\text{(int }\notin)\frac{v \notin \mathbb{Z}}{\Gamma \vdash v \notin \mathit{int}}
\qquad
\text{(rec }\notin)\frac{v \notin \mathbb{R}\mathrm{c}}{\Gamma \vdash v \notin \langle\rangle}
\qquad
\text{(rec}^-1 \notin)\frac{v \neq \langle f \mapsto \_, \ldots \rangle}{\Gamma \vdash v \notin \langle f^-{:}\tau \rangle}
\qquad
\text{(rec}^-2 \notin)\frac{\exists v.\Gamma \vdash v \notin \rho \text{ and } (v, \tau) \in \Gamma}{\Gamma \vdash \langle f \mapsto (\_, \rho), \ldots \rangle \notin \langle f^-{:}\tau \rangle}
$$

$$
\text{(rec}^+1 \notin)\frac{v \neq \langle f \mapsto \_, \ldots \rangle}{\Gamma \vdash v \notin \langle f^+{:}\tau \rangle}
\qquad
\text{(rec}^+2 \notin)\frac{\exists v.\Gamma \vdash v \notin \tau \text{ and } (v, \rho) \in \Gamma}{\Gamma \vdash \langle f \mapsto (\_, \rho), \ldots \rangle \notin \langle f^+{:}\tau \rangle}
\qquad
\text{(rec}^+3 \notin)\frac{\kappa = \emptyset \text{ or } (\kappa = \{v\} \text{ and } \Gamma \vdash v \notin \tau)}{\Gamma \vdash \langle f \mapsto (\kappa, \_), \ldots \rangle \notin \langle f^+{:}\tau \rangle}
$$

Figure 2. Definition of $\Gamma \vdash v \notin \tau$

Law 6.

1. $[\![\langle f^+{:}\tau_1\rangle \vee \langle f^+{:}\tau_2\rangle]\!] \subseteq [\![\langle f^+{:}\tau_1 \vee \tau_2\rangle]\!]$
2. $[\![\langle f^+{:}\tau_1 \vee \tau_2\rangle]\!] \not\subseteq [\![\langle f^+{:}\tau_1\rangle \vee \langle f^+{:}\tau_2\rangle]\!]$
3. $[\![\langle f^-{:}\tau_1\rangle \vee \langle f^-{:}\tau_2\rangle]\!] \subseteq [\![\langle f^-{:}\tau_1 \wedge \tau_2\rangle]\!]$
4. $[\![\langle f^-{:}\tau_1 \wedge \tau_2\rangle]\!] \not\subseteq [\![\langle f^-{:}\tau_1\rangle \vee \langle f^-{:}\tau_2\rangle]\!]$

We have already shown in Section 2.6 why $\langle f^+{:}\tau_1 \vee \tau_2\rangle \leq \langle f^+{:}\tau_1\rangle \vee \langle f^+{:}\tau_2\rangle$ is unsound; an analogous example shows why $\langle f^-{:}\tau_1 \wedge \tau_2\rangle \leq \langle f^-{:}\tau_1\rangle \vee \langle f^-{:}\tau_2\rangle$ is unsound. If $\tau_1 = \mathit{null} \vee \mathit{bool}$ and $\tau_2 = \mathit{null} \vee \mathit{int}$, then $\tau_1 \wedge \tau_2 = \mathit{null}$, and the following inequalities can be deduced.

$$
\begin{aligned}
&\langle f^-{:}\mathit{null}\rangle \wedge \langle f^+{:}\mathit{null} \vee \mathit{int} \vee \mathit{string}\rangle \leq \\
&\qquad (\langle f^-{:}\mathit{null} \vee \mathit{bool}\rangle \vee \langle f^-{:}\mathit{null} \vee \mathit{int}\rangle) \wedge \\
&\qquad \langle f^+{:}\mathit{null} \vee \mathit{int} \vee \mathit{string}\rangle \\
&\langle f^-{:}\mathit{null}\rangle \wedge \langle f^+{:}\mathit{null} \vee \mathit{int} \vee \mathit{string}\rangle \leq \\
&\qquad \langle f^-{:}\mathit{null} \vee \mathit{int}\rangle \wedge \langle f^+{:}\mathit{null} \vee \mathit{int} \vee \mathit{string}\rangle
\end{aligned}
$$

The last deduced inequality is unsound, because field f of the rhs type can be updated with integer values, but not field f of the lhs type.

Finally, the following law concerns the relationship between read-only and write-only records with the same field.

Law 7. *For all* τ_1, τ_2, $[\![\langle f^+{:}\tau_1\rangle]\!] \subseteq [\![\langle f^-{:}\tau_2\rangle]\!]$ *iff* $[\![\tau_1]\!] = \emptyset$ *or* $[\![\tau_2]\!] = \emptyset$.

5. Subtyping

In this section we show how subtyping can be defined by a system of subtyping rules which are sound and complete w.r.t. the semantic subtyping induced by the model defined in the previous section. Such a system drives the Prolog implementation detailed in Section 6.

5.1 Or- and And-sets

To simplify the subtyping rules and make the subtyping algorithm more effective, besides considering only regular and contractive types (we recall the comments in Section 3), union and intersection types are generalized to allow union and intersection over finite non-empty sets of types.

$$
\begin{array}{rcll}
\pi & ::= & \mathbf{0} \mid \mathbf{1} \mid \mathit{int} \mid \mathit{null} \mid \langle\rangle \mid \langle f^\nu{:}\pi\rangle & \\
& & \mid \vee\{\pi_1, \ldots, \pi_n\} \mid \wedge\{\pi_1, \ldots, \pi_n\} & \text{(n>0)} \\
\sigma & ::= & \vee\{\varsigma_1, \ldots, \varsigma_n\} \mid \varsigma & \text{(n>0)} \\
\varsigma & ::= & \wedge\{\iota_1, \ldots, \iota_n\} \mid \iota & \text{(n>0)} \\
\iota & ::= & \mathbf{0} \mid \mathbf{1} \mid \mathit{int} \mid \mathit{null} \mid \langle\rangle \mid \langle f^\nu{:}\pi\rangle &
\end{array}
$$

Figure 3. Types with or- and and-sets

The type $\vee\{\pi_1, \ldots, \pi_n\}$ (called or-set) represents the union of the types $\pi_1, \ldots, \pi_n$ and $\wedge\{\pi_1, \ldots, \pi_n\}$ (called and-set) represents the intersection of the types $\pi_1, \ldots, \pi_n$.

The meta-variables ι, ς, and σ range over subsets of types that will be used in the subtyping rules. In particular, in a σ type each type which is not guarded by a record type is guaranteed to be in disjunctive normal form (see the next subsection).

For the subtyping rules we assume that types adhere to the more abstract syntax defined above, types expressed with the syntax defined in Section 3 (that is, where binary union and intersection constructors are employed instead of or- and and-sets) can be easily transformed to match the new syntax.

The judgment $\tau \rightsquigarrow \pi$, coinductively defined in Figure 4, is used to transform all binary boolean operators into the corresponding and-/or-sets (where repetitions are removed).

For instance $int \vee null \vee int \rightsquigarrow \vee\{int, null\}$, and $(int \vee int) \wedge (int \vee int) \rightsquigarrow \wedge\{\vee\{int\}\}$; clearly, $\vee\{\pi\}$ and $\wedge\{\pi\}$ are both equivalent to π, but for keeping definitions simpler we avoid this further simplification of singleton or-sets and and-sets.

$$\frac{\tau \in \{null, int, \langle\rangle, \mathbf{0}, \mathbf{1}\}}{\tau \rightsquigarrow \tau} \qquad \frac{\tau \rightsquigarrow \pi}{\langle f^{\nu}{:}\tau\rangle \rightsquigarrow \langle f^{\nu}{:}\pi\rangle}$$

$$\frac{\tau_1 \rightsquigarrow \pi_1 \quad \tau_2 \rightsquigarrow \pi_2}{\tau_1 \vee \tau_2 \rightsquigarrow \pi_1 \uplus_{\vee} \pi_2} \qquad \frac{\tau_1 \rightsquigarrow \pi_1 \quad \tau_2 \rightsquigarrow \pi_2}{\tau_1 \wedge \tau_2 \rightsquigarrow \pi_1 \uplus_{\wedge} \pi_2}$$

Figure 4. Definition of $\tau \rightsquigarrow \pi$

The definition uses two auxiliary operators: $\uplus_{\vee}$ and $\uplus_{\wedge}$, their definitions are given in Figure 5 in terms of the parametric operator $\uplus_{\alpha}$, where α can be instantiated with either $\vee$ or $\wedge$.

$$\pi_1 \uplus_{\alpha} \pi_2 = \alpha S$$

$$\text{with } S = \begin{cases} S_1 \cup S_2 & \text{if } \pi_1 = \alpha S_1 \text{ and } \pi_2 = \alpha S_2 \\ \{\pi_1\} \cup S_2 & \text{if } \pi_1 \neq \alpha\{\ldots\} \text{ and } \pi_2 = \alpha S_2 \\ S_1 \cup \{\pi_2\} & \text{if } \pi_1 = \alpha S_1 \text{ and } \pi_2 \neq \alpha\{\ldots\} \\ \{\pi_1\} \cup \{\pi_2\} & \text{if } \pi_1 \neq \alpha\{\ldots\} \text{ and } \pi_2 \neq \alpha\{\ldots\} \end{cases}$$

$$\biguplus_{\alpha\, i=1}^{n>1} \pi_i = \pi_1 \uplus_{\alpha} \ldots \uplus_{\alpha} \pi_{n-1} \uplus_{\alpha} \pi_n \qquad \biguplus_{\alpha\, i=1}^{1} \pi_i = \pi_1$$

Figure 5. Definition of auxiliary operators

The operator $\pi_1 \uplus_{\vee} \pi_2$ ($\pi_1 \uplus_{\wedge} \pi_2$ respectively) returns a flattened or-set (and-set) that represents the union (intersection) of π_1 and π_2, and, therefore, enjoys all the property of set-theoretic union (intersection).

For simplicity, in the examples we keep the syntax defined in Section 3.

5.2 Type Normalization

Besides employing or- and and-sets, we assume that types are initially normalized (see function $norm$ defined below), that is, they are put in disjunctive normal form (DNF), and simplifications are applied to record types (see function $simp$ defined in Figure 8).

Since type normalization does not enter record types, it is performed lazily, therefore, while we assume that types are initially normalized before checking subtyping (hence, function $norm$ is tacitly applied), function $norm$ is also explicitly applied to types in the subtyping rules whenever the type of a record field is accessed.

Function dnf, defined in Figure 7 along with $dstr$, puts a type in DNF; it is the identity function when restricted to basic types ι. Function $dstr$ takes as input an and-set of types that are already in DNF, and returns an equivalent type in DNF by applying the distribution property of intersection over union.

For instance: $dstr(\wedge\{\vee\{\langle f^{+}{:}int\rangle, int\}, null\}) = \vee\{\wedge\{\langle f^{+}{:}int\rangle, null\}, \wedge\{int, null\}\}$

Function $simp$ is applied to types σ in DNF to simplify record types inside or/and-sets types according to Law 5 proved in Section 4: $[\![\langle f^{+}{:}\pi_1\rangle \wedge \langle f^{+}{:}\pi_2\rangle]\!] = [\![\langle f^{+}{:}\pi_1 \wedge \pi_2\rangle]\!]$, and $[\![\langle f^{-}{:}\pi_1\rangle \wedge \langle f^{-}{:}\pi_2\rangle]\!] = [\![\langle f^{-}{:}\pi_1 \vee \pi_2\rangle]\!]$.

Function $norm$ corresponds to the composition of the two functions $simp$, and dnf:

$$norm(\pi) = simp(dnf(\pi)).$$

Lemma 3. $[\![\pi]\!] = [\![norm(\pi)]\!]$.

Proof. The dnf and $simp$ functions apply the distributive law and Law 5, respectively. □

5.3 Subtyping Rules

We define a subtyping judgment with a system of subtyping rules which are sound and complete w.r.t. the definition of semantic subtyping.

According to Law 7, $[\![\langle f^{+}{:}\pi_1\rangle]\!] \subseteq [\![\langle f^{-}{:}\pi_2\rangle]\!]$ iff $[\![\pi_1]\!] = \emptyset$ or $[\![\pi_2]\!] = \emptyset$, therefore the definition of the subtyping judgment depends on the definition of the judgment for checking whether types are empty. These two judgments cannot be merged because subtyping is coinductively defined, whereas emptiness is inductive. By Law 1, $[\![\langle f^{+}{:}\pi_1\rangle \wedge \langle f^{-}{:}\pi_2\rangle]\!] = \emptyset$ iff $[\![\pi_2]\!] \not\subseteq [\![\pi_1]\!]$, therefore the judgment for type emptiness is defined in terms of the negation of the subtyping judgment, which, in turn, is defined in terms of the judgment for non emptiness of types, because $[\![\pi]\!] \neq \emptyset$ iff $[\![\pi]\!] \not\subseteq \emptyset$; again, these two judgments cannot be merged, since non-emptiness is coinductive, whereas negation of subtyping is inductive. Finally, again by Law 1, the definition of the non-emptiness judgment depends on the definition of the subtyping judgment.

The dependency graph between these four judgments is depicted in Figure 9.

As happens in Section 4 for the definition of the coinductive judgment $v \in \tau$, and its negation, there is a circular definition involving coinductive and inductive judgments that has to be broken; the dashed edge in the picture corresponds to the dependency which is removed as similarly done in Section 4: the judgments for emptiness, non-emptiness and negation of subtyping use a set Π of abducted pairs (π_1, π_2) corresponding to the assumptions that $norm(\pi_1) \leq norm(\pi_2)$ holds. In this way, there is no circularity involving coinductive and inductive judgments, the coinductive judgment for non-emptiness is defined only in terms of itself, and is used by the judgment for negation of subtyping, which is used by the inductive judgment for emptiness, and, finally, the coinductive judgment for subtyping depends on the judgment for emptiness. For these reasons, the definitions of the judgments are stratified and fixed points are computed as follows: first, the greatest fixed point of the recursive definition of non-emptiness; then, the least fixed point of emptiness, and the

$$dstr(\wedge\{\pi,\pi_1,\ldots,\pi_n\}) = \begin{cases} \displaystyle\mathop{\sqcup\hspace{-0.62em}\vee}_{i=1}^{n}\ \mathop{\sqcup\hspace{-0.62em}\vee}_{j=1}^{h} \pi'_i \mathbin{\sqcup\hspace{-0.62em}\wedge} \pi''_j \\ \qquad \text{if } dstr(\wedge\{\pi_1,\ldots,\pi_n\}) = \vee\{\pi''_1,\ldots,\pi''_h\} \\ \displaystyle\mathop{\sqcup\hspace{-0.62em}\vee}_{i=1}^{n} \pi'_i \mathbin{\sqcup\hspace{-0.62em}\wedge} dstr(\wedge\{\pi_1,\ldots,\pi_n\}) \text{ otherwise} \end{cases}$$
$$\text{if } \pi = \vee\{\pi'_1,\ldots,\pi'_n\}$$
$$dstr(\wedge\{\pi_1,\ldots,\pi_n\}) = \mathop{\sqcup\hspace{-0.62em}\wedge}_{i=1}^{n} \pi_i \text{ if } \pi_i \neq \vee\{\ldots\}\ \forall i \in [1,n]$$

Figure 6. Definition of auxiliary operator $dstr$

$$dnf(\vee\{\pi_1,\ldots,\pi_n\}) = \mathop{\sqcup\hspace{-0.62em}\vee}_{i=1}^{n} dnf(\pi_i)$$
$$dnf(\wedge\{\pi_1,\ldots,\pi_n\}) = dstr(\mathop{\sqcup\hspace{-0.62em}\wedge}_{i=1}^{n} dnf(\pi_i))$$
$$dnf(\iota) = \iota$$

Figure 7. Definition of dnf

$$simp(\vee\{\varsigma_1,\ldots,\varsigma_n\}) = \mathop{\sqcup\hspace{-0.62em}\vee}_{i=1}^{n} simp(\varsigma_i)$$

$$simp(\wedge\{\langle f^+{:}\pi_1\rangle,\ldots,\langle f^+{:}\pi_n\rangle,\iota'_1,\ldots,\iota'_m\}) = \langle f^+{:}\mathop{\sqcup\hspace{-0.62em}\wedge}_{i=1}^{n}\pi_i\rangle \mathbin{\sqcup\hspace{-0.62em}\wedge} simp(\wedge\{\iota'_1,\ldots,\iota'_m\}) \text{ if } n \geq 2 \text{ and } \iota'_i \neq \langle f^+{:}\_\rangle \forall i \in [1,m]$$

$$simp(\wedge\{\langle f^-{:}\pi_1\rangle,\ldots,\langle f^-{:}\pi_n\rangle,\iota'_1,\ldots,\iota'_m\}) = \langle f^-{:}\mathop{\sqcup\hspace{-0.62em}\vee}_{i=1}^{n}\pi_i\rangle \mathbin{\sqcup\hspace{-0.62em}\wedge} simp(\wedge\{\iota'_1,\ldots,\iota'_m\}) \text{ if } n \geq 2 \text{ and } \iota'_i \neq \langle f^-{:}\_\rangle \forall i \in [1,m]$$

$$simp(\sigma) = \sigma \text{ otherwise}$$

Figure 8. Simplification of types

$$\begin{tikzcd} \pi_1 \leq \pi_2 \arrow[r] & \pi \simeq \emptyset \arrow[r] & \pi_1 \not\leq \pi_2 \arrow[r] & \pi \not\simeq \emptyset \arrow[ll, bend right, dashed] \end{tikzcd}$$

Figure 9. Dependency graph between the four judgments

least fixed point of negation of subtyping; finally, the greatest fixed point of the recursive definition of subtyping.

Subtyping has to be interpreted coinductively because if $\tau_1 = \langle f^+{:}null\rangle \vee \langle f^+{:}\tau_1\rangle$, and $\tau_2 = \langle f^+{:}null \vee \tau_2\rangle$, then $\tau_1 \leq \tau_2$ must hold; therefore, with a syntax directed definition of the subtyping judgment the derivation for $\tau_1 \leq \tau_2$ must contain $\tau_1 \leq \tau_2$ itself in the descendant nodes of the derivation tree. By duality, negation of subtyping has to be interpreted inductively.

Non emptiness has to be interpreted coinductively because if $\tau = \langle f^+{:}\tau\rangle$, then τ must be non-empty; therefore, with a syntax directed definition of the non emptiness judgment the derivation for $\emptyset \vdash \tau \not\simeq \emptyset$ must contain $\emptyset \vdash \tau \not\simeq \emptyset$ itself in the descendant nodes of the derivation tree. By duality, emptiness has to be interpreted inductively.

All rules in this section are deliberately non algorithmic in order to provide a high level specification of the subtyping algorithm: in particular, several rules overlap, and some of them have premises with existential quantification. In Section 6 we show how these rules can be made algorithmic and effectively implemented.

The non emptiness judgment $\Pi \vdash \sigma \not\simeq \emptyset$ is defined in Figure 10.

Rule ($\vee \not\simeq \emptyset$) states that a union type is not empty if at least one of its operands is not empty.

Rules (any $\not\simeq \emptyset$), (simple $\not\simeq \emptyset$), and (rec $\not\simeq \emptyset$) state that the types $\mathbf{1}$, int, $null$, $\langle\rangle$ are non empty, as expected.

Rule (rec-r $\not\simeq \emptyset$) is driven by Law 2 which states that $[\![\langle f^+{:}\tau\rangle]\!] = \emptyset$ iff $[\![\tau]\!] = \emptyset$; since normalization does not propagate inside record types, the type of field f has to be normalized with function $norm$.

Rule (rec-w $\not\simeq \emptyset$) is driven by Law 3 which states that $[\![\langle f^-{:}\tau\rangle]\!] \neq \emptyset$.

All the remaining rules deal with intersection types.

Rule ($\wedge$single$\not\simeq \emptyset$) corresponds to the base case consisting of singleton and-sets.

Rule ($\wedge$any$\not\simeq \emptyset$) deals with the simple case when an element of the and-set is $\mathbf{1}$; the judgment $\Pi \vdash \wedge\{\mathbf{1}\} \not\simeq \emptyset$ can be correctly derived thanks to rule ($\wedge$single$\not\simeq \emptyset$).

Rule ($\wedge$rec $\not\simeq \emptyset$) deals with the simple case when an element of the and-set is $\langle\rangle$; all other elements in the and-set must not be primitive types; the judgment $\Pi \vdash \wedge\{\langle\rangle\} \not\simeq \emptyset$ can be correctly derived thanks to rule ($\wedge$single$\not\simeq \emptyset$).

Rule (r $\wedge$ w $\not\simeq \emptyset$) is the only one that requires the use of Π; if both $\langle f^+{:}\pi\rangle$ and $\langle f^-{:}\pi'\rangle$ are contained in the same and-set then by Law 1, their intersection is not empty only if $\pi' \leq \pi$. To break circularity the rule requires the assumption $\pi' \leq \pi$ with the side condition $(\pi', \pi) \in \Pi$. Furthermore, the intersections of the record types with the rest of the and-set must be non empty.

Rules (r$\wedge \not\simeq \emptyset$) and (w$\wedge \not\simeq \emptyset$) deal with the simpler cases where the and-set does not contain two record types with the same field, and different annotations (recall that type normalization exploits function $simp$ which merges together record types in and-sets with the same field and annotation); this covers also the case where there are record types with different fields, since they cannot interfere. Rules (r$\wedge \not\simeq \emptyset$) requires also the premise $\Pi \vdash norm(\pi) \not\simeq \emptyset$ (as for rule (rec-r $\not\simeq \emptyset$), the type of the field has to be normalized with function $norm$, since normalization does not propagate inside record types) because, by Law 2, the type of the field of a non empty covariant record type must be non empty.

The inductive rules for checking emptiness of a type are defined in Figure 11.

$$(\vee\not\simeq\emptyset)\,\frac{\Pi \vdash \varsigma \not\simeq \emptyset}{\Pi \vdash \vee\{\ldots,\varsigma,\ldots\} \not\simeq \emptyset} \qquad (\text{any}\not\simeq\emptyset)\,\frac{}{\Pi \vdash \mathbf{1} \not\simeq \emptyset} \qquad (\text{simple}\not\simeq\emptyset)\,\frac{\iota \in \{\mathit{int}, \mathit{null}, \langle\rangle\}}{\Pi \vdash \iota \not\simeq \emptyset}$$

$$(\text{rec-r}\not\simeq\emptyset)\,\frac{\Pi \vdash \mathit{norm}(\pi) \not\simeq \emptyset}{\Pi \vdash \langle f^{+}{:}\pi\rangle \not\simeq \emptyset} \qquad (\text{rec-w}\not\simeq\emptyset)\,\frac{}{\Pi \vdash \langle f^{-}{:}\_\rangle \not\simeq \emptyset} \qquad (\wedge\text{any}\not\simeq\emptyset)\,\frac{\Pi \vdash \wedge\{\iota_1,\ldots,\iota_n\} \not\simeq \emptyset}{\Pi \vdash \wedge\{\mathbf{1},\iota_1,\ldots,\iota_n\} \not\simeq \emptyset}$$

$$(\wedge\text{single}\not\simeq\emptyset)\,\frac{\Pi \vdash \iota \not\simeq \emptyset}{\Pi \vdash \wedge\{\iota\} \not\simeq \emptyset} \qquad (\wedge\text{rec}\not\simeq\emptyset)\,\frac{\Pi \vdash \wedge\{\iota_1,\ldots,\iota_n\} \not\simeq \emptyset \text{ and } \forall i \in [1,n]\ \iota_i \notin \{\mathit{null}, \mathit{int}\}}{\Pi \vdash \wedge\{\langle\rangle,\iota_1,\ldots,\iota_n\} \not\simeq \emptyset}$$

$$(\text{r}\wedge\text{w}\not\simeq\emptyset)\,\frac{\Pi \vdash \wedge\{\langle f^{+}{:}\pi\rangle,\iota_1,\ldots,\iota_n\} \not\simeq \emptyset \quad \Pi \vdash \wedge\{\langle f^{-}{:}\pi'\rangle,\iota_1,\ldots,\iota_n\} \not\simeq \emptyset \quad (\pi',\pi) \in \Pi}{\Pi \vdash \wedge\{\langle f^{+}{:}\pi\rangle,\langle f^{-}{:}\pi'\rangle,\iota_1,\ldots,\iota_n\} \not\simeq \emptyset}$$

$$(\wedge\text{r}\not\simeq\emptyset)\,\frac{\Pi \vdash \mathit{norm}(\pi) \not\simeq \emptyset \text{ and } \Pi \vdash \wedge\{\iota_1,\ldots,\iota_n\} \not\simeq \emptyset \text{ and } \forall i \in [1,n]\ \iota_i \notin \{\mathit{null}, \mathit{int}, \langle f^{-}{:}\_\rangle\}}{\Pi \vdash \wedge\{\langle f^{+}{:}\pi\rangle,\iota_1,\ldots,\iota_n\} \not\simeq \emptyset}$$

$$(\wedge\text{w}\not\simeq\emptyset)\,\frac{\Pi \vdash \wedge\{\iota_1,\ldots,\iota_n\} \not\simeq \emptyset \text{ and } \forall i \in [1,n]\ \iota_i \notin \{\mathit{null}, \mathit{int}, \langle f^{+}{:}\_\rangle\}}{\Pi \vdash \wedge\{\langle f^{-}{:}\_\rangle,\iota_1,\ldots,\iota_n\} \not\simeq \emptyset}$$

Figure 10. Non-emptiness of types σ

$$(\vee\simeq\emptyset)\,\frac{\Pi \vdash \varsigma_i \simeq \emptyset\ \forall i \in [1,n]}{\Pi \vdash \vee\{\varsigma_1,\ldots,\varsigma_n\} \simeq \emptyset} \qquad (\wedge\simeq\emptyset)\,\frac{\Pi \vdash \iota \simeq \emptyset}{\Pi \vdash \wedge\{\ldots,\iota,\ldots\} \simeq \emptyset} \qquad (\text{rec-r}\simeq\emptyset)\,\frac{\Pi \vdash \mathit{norm}(\pi) \simeq \emptyset}{\Pi \vdash \langle f^{+}{:}\pi\rangle \simeq \emptyset}$$

$$(\text{empty}\simeq\emptyset)\,\frac{}{\Pi \vdash \mathbf{0} \simeq \emptyset} \qquad (\wedge\text{prim}\simeq\emptyset)\,\frac{\iota \in \{\mathit{int}, \mathit{null}\} \quad \iota' \neq \mathbf{1}}{\Pi \vdash \wedge\{\ldots,\iota,\iota',\ldots\} \simeq \emptyset} \qquad (\text{r}\wedge\text{w}\simeq\emptyset)\,\frac{\Pi \vdash \mathit{norm}(\pi') \not\leq \mathit{norm}(\pi)}{\Pi \vdash \wedge\{\ldots,\langle f^{+}{:}\pi\rangle,\langle f^{-}{:}\pi'\rangle,\ldots\} \simeq \emptyset}$$

Figure 11. Emptiness of types σ

As depicted in Figure 9, the judgment $\Pi \vdash \sigma \simeq \emptyset$ is defined in terms of the judgment $\Pi \vdash \sigma_1 \not\leq \sigma_2$, which is interpreted inductively as well.

Rule $(\vee \simeq \emptyset)$ states that an or-set is empty if all of its elements are empty.

Rule $(\wedge\simeq\emptyset)$ states that an and-set is empty if at least one of its elements is empty.

Rule (rec-r$\simeq\emptyset$) is based on Law 2: a covariant record type is empty if the type of its field is empty; as in other cases, type normalization has to be propagated to the type of the field by applying function norm.

Rule (empty$\simeq\emptyset$) is immediate.

Rule $(\wedge\text{prim} \simeq \emptyset)$ states that an and-set is empty if it contains at least two different types ι and ι' when one is either null or int, and the other is not $\mathbf{1}$.

Finally, rule $(\text{r} \wedge \text{w} \simeq \emptyset)$ is the only one which requires the use of the judgment $\Pi \vdash \sigma_1 \not\leq \sigma_2$; by Law 1, an and-set containing both $\langle f^{+}{:}\pi\rangle$ and $\langle f^{-}{:}\pi'\rangle$ is empty if π' is not a subtype of π.

The inductive rules defining the judgment $\Pi \vdash \sigma_1 \not\leq \sigma_2$ are defined in Figure 12.

Rules (any$\not\leq$) and (simple$\not\leq$) deal with the cases involving two basic types in $\{\mathit{null}, \mathit{int}, \langle\rangle, \mathbf{1}\}$.

Rules (l-or$\not\leq$) and (r-or$\not\leq$) deal with or-sets; the former corresponds to the set-theoretic property $A \cup B \subseteq C \Rightarrow A \subseteq C$, that is, $A \not\subseteq C \Rightarrow A \cup B \not\subseteq C$, while the latter corresponds to a property which is peculiar of this type system, and does not hold in general for sets.

Similarly, rules (l-and$\not\leq$) and (r-and$\not\leq$) deal with and-sets; the former corresponds to a property which is peculiar of this type system, and does not hold in general for sets, while the latter corresponds to the set-theoretic property $A \subseteq B \cap C \Rightarrow A \subseteq B$, that is, $A \not\subseteq B \Rightarrow A \not\subseteq B \cap C$. In particular, rule (l-and$\not\leq$) requires the additional premise $\Pi \vdash \wedge\{\iota_1,\ldots,\iota_n\} \not\simeq \emptyset$, otherwise it would be possible to derive invalid judgments, as $\Pi \vdash \wedge\{\mathit{null}, \mathit{int}\} \not\leq \langle\rangle$.

Rule $(\mathit{empty} \not\leq)$ uses the judgment for non emptiness, and deals with cases when the type on the right-hand side is empty, but the type on the left-hand side is not; in this way it is possible to cover situations that are not considered by other rules; for instance, rule $(\mathit{empty} \not\leq)$ is required to derive $\Pi \vdash \mathit{int} \not\leq \vee\{\wedge\{\mathbf{0}, \mathit{int}\}, \mathbf{0}\}$.

Rule (r$\backslash$w$\not\leq$) is driven by Law 7 stating that $[\![\langle f^{+}{:}\tau_1\rangle]\!] \subseteq [\![\langle f^{-}{:}\tau_2\rangle]\!]$ iff $[\![\tau_1]\!] = \emptyset$ (and hence, by Law 2, $[\![\langle f^{+}{:}\tau_1\rangle]\!] = \emptyset$) or $[\![\tau_2]\!] = \emptyset$, while rule (w$\backslash$r$\not\leq$) states that $\langle f^{-}{:}\sigma_1\rangle$ can never be a subtype of $\langle f^{+}{:}\sigma_2\rangle$ for any σ_1, σ_2; indeed,

$$\text{(any}\not\leq)\frac{\iota \neq \mathbf{1}}{\Pi \vdash \mathbf{1} \not\leq \iota} \quad \text{(empty}\not\leq)\frac{\Pi \vdash \varsigma \not\simeq \emptyset}{\Pi \vdash \varsigma \not\leq \mathbf{0}} \quad \text{(l-or}\not\leq)\frac{\exists i \in [1,n]\ \Pi \vdash \varsigma_i \not\leq \sigma}{\Pi \vdash \vee\{\varsigma_1,\ldots,\varsigma_n\} \not\leq \sigma} \quad \text{(r-or}\not\leq)\frac{\forall i \in [1,n]\ \Pi \vdash \varsigma \not\leq \varsigma_i}{\Pi \vdash \varsigma \not\leq \vee\{\varsigma_1,\ldots,\varsigma_n\}}$$

$$\text{(l-and}\not\leq)\frac{\forall i \in [1,n]\ \Pi \vdash \iota_i \not\leq \iota \quad \Pi \vdash \wedge\{\iota_1,\ldots,\iota_n\} \not\simeq \emptyset}{\Pi \vdash \wedge\{\iota_1,\ldots,\iota_n\} \not\leq \iota} \quad \text{(r-and}\not\leq)\frac{\exists i \in [1,n]\ \Pi \vdash \varsigma \not\leq \iota_i}{\Pi \vdash \varsigma \not\leq \wedge\{\iota_1,\ldots,\iota_n\}}$$

$$\text{(r}\backslash\text{w}\not\leq)\frac{\Pi \vdash \langle f^{+}{:}\pi\rangle \not\simeq \emptyset \quad \Pi \vdash norm(\pi') \not\simeq \emptyset}{\Pi \vdash \langle f^{+}{:}\pi\rangle \not\leq \langle f^{-}{:}\pi'\rangle} \quad \text{(w}\backslash\text{r}\not\leq)\frac{}{\Pi \vdash \langle f^{-}{:}\_\rangle \not\leq \langle f^{+}{:}\_\rangle} \quad \text{(rec}\not\leq)\frac{f_1 \neq f_2}{\Pi \vdash \langle {f_1}^{\nu_1}{:}\_\rangle \not\leq \langle {f_2}^{\nu_2}{:}\_\rangle}$$

$$\text{(simple}\not\leq)\frac{\iota \in \{null, int, \langle\rangle\} \quad \iota' \notin \{\iota, \mathbf{1}\}}{\Pi \vdash \iota \not\leq \iota'} \quad \text{(w}\backslash\text{w}\not\leq)\frac{\Pi \vdash norm(\pi') \not\leq norm(\pi)}{\Pi \vdash \langle f^{-}{:}\pi\rangle \not\leq \langle f^{-}{:}\pi'\rangle} \quad \text{(r}\backslash\text{r}\not\leq)\frac{\Pi \vdash norm(\pi) \not\leq norm(\pi')}{\Pi \vdash \langle f^{+}{:}\pi\rangle \not\leq \langle f^{+}{:}\pi'\rangle}$$

Figure 12. Rules for negation of subtyping

$\langle f \mapsto (\emptyset, \sigma_1)\rangle \in [\![\langle f^{-}{:}\sigma_1\rangle]\!]$, but $\langle f \mapsto (\emptyset, \sigma_1)\rangle$ never belongs to $\langle f^{+}{:}\sigma_2\rangle$.

Rule (rec$\not\leq$) states that two record types with different fields can never be in subtyping relation, independently of the variance annotations.

Finally, (w$\backslash$w$\not\leq$) and (r$\backslash$r$\not\leq$) are the counterparts of the standard rules for contravariant and covariant record subtyping, respectively; as in analogous cases, type normalization has to be propagated to the types of the fields by applying function $norm$.

The coinductive rules defining the subtyping relation can be found in Figure 13.

Subtyping is the only judgment that does not depend on a set of assumptions; we recall that the set Π of subtyping assumptions is required for breaking the cyclic dependency between the non emptiness and the subtyping judgment. Of course, such assumptions need to be verified by the subtyping judgment. In Section 6 the existential quantification over subtyping assumptions in the premises of the subtyping rules will be removed, and the set of assumptions will be abducted by the implemented algorithm. In the typing rules, which are expressed in a purely declarative way, Π can be considered as an input to the judgments, whereas in the implementation it is actually an output.

Rule (prim$\leq$) imposes reflexivity between the primitive types $null$ and int.

Rule (empty$\leq$) states that an empty type σ_1 is always in subtyping relation with any type; however, all subtyping assumptions in Π required to derive that σ_1 is empty need to be verified. Furthermore, types in Π have to be normalized.

Rules (l-or$\leq$) and (r-or$\leq$) deal with or-sets, whereas (r-and$\leq$) and (l-and$\leq$) deal with and-sets. While rules (l-or$\leq$) and (r-and$\leq$) are standard and can be correctly read in both directions (hence, set-theoretically, they are sound and complete), in set theory rules (r-or$\leq$) and (l-and$\leq$) are sound but not complete; nevertheless, they are also complete in this type system.

Rule (any$\leq$) states that type $\mathbf{1}$ is a supertype of any type.

Rule (r$\backslash$w$\leq$) is driven by Law 7 which states that $[\![\langle f^{+}{:}\tau_1\rangle]\!] \subseteq [\![\langle f^{-}{:}\tau_2\rangle]\!]$ iff $[\![\tau_1]\!] = \emptyset$ or $[\![\tau_2]\!] = \emptyset$. As for rule (empty$\leq$), all subtyping assumptions in Π required to derive that π is empty need to be verified, and types in Π have to be normalized. The case when the type on the left-hand side is empty is covered by rule (empty$\leq$).

Rules (r$\backslash$r$\leq$) and (w$\backslash$w$\leq$) are the standard ones for covariant and contravariant subtyping, respectively.

Finally, rule ($\langle\rangle$ $\leq$) states that type $\langle\rangle$ is the supertype of all record types.

Before stating and proving the main results on the defined judgments, we provide an example of derivation tree for the subtyping judgment $\tau_3 \leq \tau_1$ where τ_3 and τ_1 correspond to the normalization of types in the example in Section 2.4 on unmodifiable/modifiable linked lists; for simplicity, we assume that the type T of the elements of the lists is int.

$$\begin{array}{l}
\tau_1 = \vee\{null, \wedge\{S_1\}\} \\
\tau_3 = \vee\{null, \wedge\{S_2\}\} \\
S_1 = \{\langle elem^{+}{:}int\rangle, \langle next^{+}{:}\tau_1\rangle\} \\
S_2 = S_1 \cup \{\langle elem^{-}{:}int\rangle, \langle next^{-}{:}\tau_3\rangle\}
\end{array}$$

The infinite, but regular, proof tree for $\tau_3 \leq \tau_1$ is the following one

$$\text{(l-or}\leq)\frac{\text{(r-or}\leq)\dfrac{\text{(prim}\leq)\dfrac{}{null \leq null}}{null \leq \tau_1} \qquad \text{(r-or}\leq)\dfrac{\dfrac{\vdots}{\wedge\{S_2\} \leq \wedge\{S_1\}}}{\wedge\{S_2\} \leq \tau_1}}{\tau_3 \leq \tau_1}$$

where the proof tree for $\wedge\{S_2\} \leq \wedge\{S_1\}$ is as follows

$$\text{(r-and}\leq)\frac{\dfrac{\vdots}{\wedge\{S_2\} \leq \langle elem^{+}{:}int\rangle} \qquad \dfrac{\vdots}{\wedge\{S_2\} \leq \langle next^{+}{:}\tau_1\rangle}}{\wedge\{S_2\} \leq \wedge\{S_1\}}$$

$$\text{(prim}\leq\text{)}\ \frac{\iota \in \{int, null\}}{\iota \leq \iota} \qquad \text{(empty}\leq\text{)}\ \frac{\exists\Pi.\Pi \vdash \sigma_1 \simeq \emptyset \quad \forall(\pi_1,\pi_2) \in \Pi.norm(\pi_1) \leq norm(\pi_2)}{\sigma_1 \leq \sigma_2} \qquad \text{(any}\leq\text{)}\ \frac{}{\sigma \leq \mathbf{1}}$$

$$\text{(l-or}\leq\text{)}\ \frac{\forall i \in [1,n]\ \varsigma_i \leq \sigma}{\vee\{\varsigma_1,\dots,\varsigma_n\} \leq \sigma} \qquad \text{(r-or}\leq\text{)}\ \frac{\exists i \in [1,n]\ \varsigma \leq \varsigma_i}{\varsigma \leq \vee\{\varsigma_1,\dots,\varsigma_n\}} \qquad \text{(r-and}\leq\text{)}\ \frac{\forall i \in [1,n]\ \varsigma \leq \iota_i}{\varsigma \leq \wedge\{\iota_1,\dots,\iota_n\}} \qquad \text{(l-and}\leq\text{)}\ \frac{\exists i \in [1,n]\ \iota_i \leq \iota}{\wedge\{\iota_1,\dots,\iota_n\} \leq \iota}$$

$$\text{(r\textbackslash w}\leq\text{)}\ \frac{\exists\Pi.\Pi \vdash norm(\pi) \simeq \emptyset \quad \forall(\pi_1,\pi_2) \in \Pi.norm(\pi_1) \leq norm(\pi_2)}{\langle f^{+}{:}\_\rangle \leq \langle f^{-}{:}\pi\rangle}$$

$$\text{(w\textbackslash w}\leq\text{)}\ \frac{norm(\pi') \leq norm(\pi)}{\langle f^{-}{:}\pi\rangle \leq \langle f^{-}{:}\pi'\rangle} \qquad \text{(r\textbackslash r}\leq\text{)}\ \frac{norm(\pi) \leq norm(\pi')}{\langle f^{+}{:}\pi\rangle \leq \langle f^{+}{:}\pi'\rangle} \qquad \text{(}\langle\rangle\leq\text{)}\ \frac{\iota \in \{\langle f^{+}{:}\_\rangle, \langle f^{-}{:}\_\rangle, \langle\rangle\}}{\iota \leq \langle\rangle}$$

Figure 13. Subtyping rules

and the proof trees for $\wedge\{S_2\} \leq \langle elem^{+}{:}int\rangle$ and $\wedge\{S_2\} \leq \langle next^{+}{:}\tau_1\rangle$ are

$$\text{(l-and}\leq\text{)}\ \cfrac{\text{(r\textbackslash r}\leq\text{)}\ \cfrac{\text{(prim}\leq\text{)}\ \cfrac{}{int \leq int}}{\langle elem^{+}{:}int\rangle \leq \langle elem^{+}{:}int\rangle}}{\wedge\{S_2\} \leq \langle elem^{+}{:}int\rangle}$$

$$\text{(l-and}\leq\text{)}\ \cfrac{\text{(r\textbackslash r}\leq\text{)}\ \cfrac{\cfrac{\vdots}{\tau_3 \leq \tau_1}}{\langle next^{+}{:}\tau_3\rangle \leq \langle next^{+}{:}\tau_1\rangle}}{\wedge\{S_2\} \leq \langle next^{+}{:}\tau_1\rangle}$$

The vertical dots in the premises of $\tau_3 \leq \tau_1$ mean that the tree is infinite (although regular) and the derivation continues as already specified above.

5.4 Main Results

We provide the main claims stating that the judgments are well-defined and the subtyping judgment is sound and complete w.r.t. semantic subtyping. All proofs are available in an extended version[4] of this paper.

Lemma 4. *If $\Pi \in OK^{\Pi}$ and $\Pi \vdash \sigma \not\simeq \emptyset$ is derivable, then $\Pi \vdash \sigma \simeq \emptyset$ is not derivable; if there exists $\Pi \in OK^{\Pi}$ s.t. $\Pi \vdash \sigma_1 \not\leq \sigma_2$ is derivable, then $\sigma_1 \leq \sigma_2$ is not derivable.*

Lemma 5. *If $\Pi \in OK^{\Pi}$ and $\Pi \vdash \sigma \not\simeq \emptyset$ is not derivable, then $\Pi \vdash \sigma \simeq \emptyset$ is derivable; if for all $\Pi \in OK^{\Pi}$ $\Pi \vdash \sigma_1 \not\leq \sigma_2$ is not derivable, then $\sigma_1 \leq \sigma_2$ is derivable.*

The proofs of the two lemmas above follow the same technique adopted for Lemma 1 and 2 in Section 4.

The next two theorems state soundness and completeness of the subtyping rules. All types are assumed to be in normal form.

Theorem 1 (Soundness). *If $\pi_1 \leq \pi_2$, then $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$.*

[4] Downloadable at http://www.disi.unige.it/person/AnconaD/papers/AnconaCorradiOOPSLA2016extended.pdf.

Proof. The proof exploits the approximations of the four judgments. □

Theorem 2 (Completeness). *If $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$, then $\pi_1 \leq \pi_2$.*

Proof. By Theorem 1, Lemma 4 and Lemma 5, and by contraposition. □

6. Algorithm and Implementation

From the inference rules defined in Section 5 it is not possible to directly derive an algorithm for deciding semantic subtyping between coinductive types. In particular, coinductive judgments, and abduction of sets Π of subtyping assumptions have to be implemented.

Once that Lemma 4 and 5 prove that the judgment defining $\not\leq$ is actually the complement of $\leq$, negation can be exploited in the pseudo-code defining the algorithm, hence subtyping depends on emptiness which in turns depends on the negation of subtyping, therefore only the two predicates `subtype` and `is_empty` need to be defined: the former is coinductive and depends on the latter which is inductive and depends on the negation of the former. To break circularity, the `is_empty` predicate abducts sets $\bar{\Pi}$ of pairs (π_1, π_2) corresponding to the assumptions $norm(\pi_1) \not\leq norm(\pi_2)$, as opposed to what happens in the inference rules in Section 5, where judgments refer to sets Π of positive hypotheses $norm(\pi_1) \leq norm(\pi_2)$.

Pseudo-code for predicate `is_empty` is defined in Listing 1; to be closer to the real prototype implementation in SWI Prolog, pseudo-code is expressed with high-level Horn clauses where some details have been abstracted away; furthermore, some auxiliary predicates have been omitted, and their behavior is only specified informally.

Atom `is_empty(`σ`,` $\bar{\Pi}$`)` succeeds if σ (a simplified type in DNF) is empty under the assumptions in set $\bar{\Pi}$; hence, σ and $\bar{\Pi}$ are treated as input and output, respectively. The predicate is defined in terms of another predicate with the same name, but four arguments:

- input σ: the simplified type in DNF that has to be checked;
- input Ψ: the set of coinductive hypotheses, corresponding to the types on which emptiness has been already checked; this is essential to guarantee termination in presence of recursive types (that is, cyclic Prolog terms); since types are contractive, only non basic record types (that is, $\langle f^{+}{:}ext\rangle$ or $\langle f^{-}{:}ext\rangle$, but not $\langle\rangle$) need to be added to Ψ, since, by contractivity, an infinite path in a recursive type must necessarily involve a record type. The same consideration applies for the `subtype` predicate defined below.
- input $\bar{\Pi}'$: the initial set of abducted assumptions; indeed, this argument is used to accumulate abducted assumptions;
- output $\bar{\Pi}$: the final set of abducted assumptions, returned if σ can be empty; the invariant $\bar{\Pi}' \subseteq \bar{\Pi}$ always holds, that is, the returned final set $\bar{\Pi}$ of abducted hypotheses is always a super-set of the initial set $\bar{\Pi}'$ of abducted hypotheses.

Except for the first straightforward rule dealing with the empty type $\mathbf{0}$, all other rules are applicable only if $\sigma \notin \Psi$, to ensure termination if a recursive type is processed more than once. In this case the predicate has to fail; for instance, this happens when $\sigma = \langle f^{+}{:}\sigma\rangle$.

The two rules dealing with union types are intuitive, whereas the rule for intersection types is more involved. The auxiliary predicate `partition_by_field`, whose definition has been omitted, partitions the set $\wedge\{\iota_1, \ldots, \iota_n\}$ in two collections, `RecsMap` and `Others`. The former is a map containing all basic types of shape $\langle f^{\nu}{:}\pi\rangle$, indexed by their fields; since types are normalized, each field f can be mapped at most to the two types $\langle f^{+}{:}\_\rangle$ and $\langle f^{-}{:}\_\rangle$. The latter is a set containing all other basic types $\mathbf{0}$, $\mathbf{1}$, int, $null$, or $\langle\rangle$.

The first part of the body of the clause covers rule ($\wedge\simeq\emptyset$), instantiated with $\iota = \mathbf{0}$, and rule ($\wedge$prim$\simeq\emptyset$).

The remaining part (`get_keys(RecsMap, Fs), ...`) covers rule ($\wedge\simeq\emptyset$), instantiated with $\iota = \langle f^{+}{:}\pi\rangle$, and rule (r$\wedge$w$\simeq\emptyset$). The auxiliary predicate `get_keys`, whose definition has been omitted, returns all fields which are mapped to some record type. Predicate `is_empty_and` tries to apply to some existing field either rule ($\wedge\simeq\emptyset$), instantiated with $\iota = \langle f^{+}{:}\pi\rangle$, or rule (r$\wedge$w$\simeq\emptyset$). This latter rule is applicable only when $\pi_2 \not\leq \pi_1$, hence the pair (π_2, π_1) is added to the set of abducted assumptions. The auxiliary predicate `lookup`, whose definition has been omitted, returns the set of records associated with a specific field in the map.

Pseudo-code for predicate `subtype` is defined in Listing 2; the main predicate is defined in terms of an auxiliary predicate with the same name but one more argument.

In `subtype(`Ψ`, `σ_1`, `σ_2`)` all arguments are considered as input; `subtype(`Ψ`, `σ_1`, `σ_2`)` succeeds if σ_1 is a subtype of σ_2 (where both types are simplified and in DNF), under the set Ψ of coinductive hypotheses, corresponding to the pairs of types on which subtyping has been already checked; again, this is essential to guarantee termination in presence of recursive types. As opposed to what happens for the inductive predicate `is_empty`, the coinductive predicate `subtype` succeeds if the pair consisting of the two types σ_1, and σ_2 belongs to Ψ.

The clauses defining `subtype(`Ψ`, `σ_1`, `σ_2`)` are very similar to the rules defined in Figure 13, except for some cases. A clause has been added to ensure termination in case of recursive types, for dealing with coinduction: if $(\sigma_1, \sigma_2) \in \Psi$ holds, then the predicate succeeds by virtue of the coinductive interpretation. Iteration over the elements of or-sets and and-sets in rules(l-or$\leq$) and (r-and$\leq$), respectively, is implemented through recursion, hence, two clauses have added to deal with the base cases. Finally, the clauses corresponding to rules (empty$\leq$) and (r$\backslash$w$\leq$) use negation, because, as already explained, this allows an implementation based on the definition of just two predicates, instead of four.

6.1 Prototype Implementation

We have developed a prototype implementation in Prolog; besides allowing rapid prototyping and conciseness, Prolog has the advantage of offering native support for cyclic terms, unification, and backtracking. In particular, backtracking is needed to properly deal with abducted assumptions. Consider for instance the type

$$\sigma = \vee\{\wedge\{\langle f^{+}{:}\mathbf{1}\rangle, \langle f^{-}{:}null\rangle\}, \wedge\{\langle f^{+}{:}null\rangle, \langle f^{-}{:}\mathbf{1}\rangle\}\}$$

whose semantics is the empty set. The atom `is_empty(`σ`, `$\bar{\Pi}$`)` succeeds for two different sets of abducted assumptions, $\bar{\Pi} = \{(null, \mathbf{1})\}$, or $\bar{\Pi} = \{(\mathbf{1}, null)\}$; in this case, only the second set of assumptions holds (that is, $\mathbf{1}$ is not a subtype of $null$).

Besides the implementation shown here, we have experimented another solution based on the implementation of the judgments $\Pi \vdash \sigma \simeq \emptyset$ and $\sigma_1 \not\leq \sigma_2$. Since both judgments are inductive, in this case stratification is not needed, and the benchmark shows that this last solution seems to be more efficient.

The benchmark consists of more than a hundred tests, including all examples presented in Section Section 2. Experiments have been performed with SWI Prolog 7.2.3, running on a i7-3610QM machine with GNU/Debian.

123 tests	$\not\leq$
Total time:	1.7e+00s
Average time:	1.4e-02s
Total number of inferences:	14 620 376
Average number of inferences:	118 865

Table 1. Benchmark results summary

The implementation based on $\not\leq$ performs two order of magnitude better than the one based on $\leq$.

We conjecture that this difference is due to the fact that the implementation based on $\not\leq$ employs two inductive predicates and hence does not rely on stratification.

Listing 1. Pseudo-code for is_empty

```
is_empty(σ, Π̄) ← is_empty(∅, σ, ∅, Π̄).

is_empty(_, 0, Π̄, Π̄) ← true. % rule (empty)
is_empty(Ψ, ⟨f⁺:π⟩, Π̄′, Π̄) ← ⟨f⁺:π⟩ ∉ Ψ, is_empty({⟨f⁺:π⟩} ∪ Ψ, norm(π), Π̄′, Π̄). % rule (rec-r)
is_empty(Ψ, ∨{}, Π̄′, Π̄) ← ∨{} ∉ Ψ. % rule (\/)
is_empty(Ψ, ∨{ς1,...,ςn}, Π̄′, Π̄) ← n > 0, ∨{ς1,...,ςn} ∉ Ψ, % rule (\/)
  is_empty(Ψ, ς1, Π̄′, Π̄″), is_empty(Ψ, ∨{ς2,...,ςn}, Π̄″, Π̄).
is_empty(Ψ, ∧{ι1,...,ιn}, Π̄′, Π̄) ← n > 0, ∧{ι1,...,ιn} ∉ Ψ, % rule (/\)
  partition_by_field(∧{ι1,...,ιn}, RecsMap, Others), % RecsMap map from fields to a set of record types
                                                       % Others contains all types that are not ⟨f⁺:π⟩ or ⟨f⁻:π⟩
  ((0 ∈Others, Π̄′=Π̄) or % rule (and)
   ((int ∈Others or null ∈Others), (size(Others) > 1 or not empty(RecsMap)), Π̄′=Π̄) or % rule (/\prim)
   (get_keys(RecsMap, Fs), is_empty_and(Ψ, Fs, RecsMap, Π̄′, Π̄))).

is_empty_and(Ψ, [F|_], RecsMap, Π̄′, Π̄) ← % F is a fieldname
    lookup(F, RecsMap, Recs), ⟨F⁺:π1⟩ ∈Recs,
    (is_empty(Ψ, ⟨F⁺:π1⟩, Π̄′, Π̄) or % rule (/\)
     ⟨F⁻:π2⟩ ∈Recs, Π̄={(π2,π1)} ∪ Π̄′). % rule (r/\w)
is_empty_and(Ψ, [_|Fs], RecsMap, Π̄′, Π̄) ← is_empty_and(Ψ, Fs, RecsMap, Π̄′, Π̄).
```

Listing 2. Pseudo-code for subtype

```
subtype(σ1, σ2) ← subtype(∅, σ1, σ2).

subtype(_, σ, σ) ← is_primitive(σ). % rule (prim)
subtype(_, _, any) ← true. % rule (any)
subtype(_, σ, ⟨⟩) ← (σ=⟨f⁺:_⟩ or σ=⟨f⁻:_⟩ or σ=⟨⟩). % rule (<>)
subtype(Ψ, σ1, σ2) ← (σ1,σ2) ∈ Ψ. % termination condition
subtype(Ψ, ∨{}, σ2) ← true. % rule (l-or)
subtype(Ψ, ∨{ς1,...,ςn}, σ2) ← n > 0, subtype(Ψ, ς1, σ2), subtype(Ψ, ∨{ς2,...,ςn}, σ2). % rule (l-or)
subtype(Ψ, σ1, ∨{ς1,...,ςn}) ← n > 0, subtype(Ψ, σ1, ς1)or subtype(Ψ, σ1, ∨{ς2,...,ςn}). % rule (r-or)
subtype(Ψ, σ1, σ2) ← is_empty(σ1, Π̄), % rule (empty)
  Ψ′ = {(σ1,σ2)} ∪ Ψ, ∀(π1,π2) ∈ Π̄ not subtype(Ψ′,norm(π1),norm(π2)).
subtype(Ψ, σ1, ∧{}) ← true. % rule (r-and)
subtype(Ψ, σ1, ∧{ι1,...,ιn}) ← n > 0, subtype(Ψ,σ1,ι1), subtype(Ψ, ∧{ι2,...,ιn}, σ1). % rule (r-and)
subtype(Ψ, ∧{ι1,...,ιn}, σ2) ← n > 0, subtype(Ψ, ι1, σ2) or subtype(Ψ,∧{ι2,...,ιn}, σ2). % rule (l-and)
subtype(Ψ, ⟨f⁺:π1⟩, ⟨f⁻:π2⟩) ← is_empty(π2, Π̄), % rule (r\w)
  Ψ′ = {(⟨f⁺:π1⟩,⟨f⁻:π2⟩)} ∪ Ψ, ∀(π1,π2) ∈ Π̄ not subtype(Ψ′,norm(π1),norm(π2)).
subtype(Ψ, ⟨f⁺:π1⟩, ⟨f⁺:π2⟩) ← % rule (r\r)
  Ψ′ = {(⟨f⁺:π1⟩,⟨f⁺:π2⟩)} ∪ Ψ, subtype(Ψ′,norm(π1),norm(π2)).
subtype(Ψ, ⟨f⁻:π1⟩, ⟨f⁻:π2⟩) ← % rule (w\w)
  Ψ′ = {(⟨f⁻:π1⟩,⟨f⁻:π2⟩)} ∪ Ψ, subtype(Ψ′,norm(π2),norm(π1)).
```

More details on the experimental results can be found in the documentation of the accompanying artifact.

6.2 Termination

We provide a proof sketch for termination of predicate `is_empty`.

The measure of the function `isEmpty` is defined by the function $\mathcal{D}(\Psi,\sigma)$.

$$\mathcal{D}(\Psi,\sigma) = \begin{cases} 1+\Sigma_{i=1}^{n}\mathcal{D}(\Psi,\varsigma_i) \text{ if } \sigma=\vee\{\varsigma_1,\ldots,\varsigma_n\} \\ 1+\Sigma_{i=1}^{n}\mathcal{D}(\Psi,\iota_i) \text{ if } \sigma=\wedge\{\iota_1,\ldots,\iota_n\} \\ 1+\mathcal{D}(\Psi\cup\{\sigma\},norm(\pi)) \text{ if } \sigma=\langle f^{+}{:}\pi\rangle\wedge\sigma\notin\Psi \\ 1 \text{ if } \sigma=\langle f^{+}{:}\pi\rangle\wedge\sigma\in\Psi \\ 1 \text{ if } \sigma\in\{\langle f^{-}{:}\_\rangle, int, null, \mathbf{0}, \mathbf{1}\} \end{cases}$$

For instance, let $T=\langle f^{+}{:}int\rangle\vee\langle g^{+}{:}T\rangle$, then we have that $norm(T)=\vee\{\langle f^{+}{:}int\rangle,\langle g^{+}{:}T\rangle\}$, and then:

$$\begin{aligned} &\mathcal{D}(\emptyset,\vee\{\langle f^{+}{:}int\rangle,\langle g^{+}{:}T\rangle\}) = \\ &1+\mathcal{D}(\emptyset,\langle f^{+}{:}int\rangle)+\mathcal{D}(\emptyset,\langle g^{+}{:}T\rangle) = \\ &1+(1+\mathcal{D}(\{\langle f^{+}{:}int\rangle\},norm(int)))+ \\ &(1+\mathcal{D}(\{\langle g^{+}{:}T\rangle\},norm(T)\})) = \\ &3+(1+\mathcal{D}(\{\langle g^{+}{:}T\rangle\},\vee\{\langle f^{+}{:}int\rangle,\langle g^{+}{:}T\rangle\}\})) = \\ &4+(1+\mathcal{D}(\{\langle g^{+}{:}T\rangle\},\langle f^{+}{:}int\rangle)+ \\ &\mathcal{D}(\{\langle g^{+}{:}T\rangle\},\langle g^{+}{:}T\rangle)) = \\ &5+(1+\mathcal{D}(\{\langle g^{+}{:}T\rangle,\langle f^{+}{:}int\rangle\},norm(int)))+1 = \\ &8 \end{aligned}$$

The function $\mathcal{D}(\Psi,\sigma)$ is well-defined, and returns positive numbers. The type σ is regular and contractive, hence the set of its subtrees is finite; $\vee\{\varsigma_1,\ldots,\varsigma_n\}$ and $\wedge\{\iota_1,\ldots,\iota_n\}$

are always bounded. The set Ψ can contain only subtrees (eventually normalized) of σ, hence is bounded.

The set Π can contain only pairs of types that are subtrees of σ, hence its size is bounded.

We now prove that the measure strictly decreases for each recursive occurrence of the predicate.

When $\sigma = \langle f^{+}{:}\pi\rangle$ let $\mathcal{D}(\Psi,\sigma)$ be the measure value, then the value for the recursive call is $\mathcal{D}(\Psi \cup \{\sigma\}, norm(\pi))$ and $\mathcal{D}(\Psi \cup \{\sigma\}, norm(\pi)) < \mathcal{D}(\Psi,\sigma)$.

When $\sigma = \vee\{\varsigma_1,\dots,\varsigma_n\}$ let $\mathcal{D}(\Psi,\sigma)$ be the measure value, then the value for the first recursive call is $\mathcal{D}(\Psi,\varsigma_1)$ and $\mathcal{D}(\Psi,\varsigma_1) < \mathcal{D}(\Psi,\sigma)$; the value for the second recursive call is $\mathcal{D}(\Psi,\vee\{\varsigma_2\dots\varsigma_n\})$ and $\mathcal{D}(\Psi,\vee\{\varsigma_2\dots\varsigma_n\}) < \mathcal{D}(\Psi,\sigma)$.

When $\sigma = \wedge\{\iota_1,\dots,\iota_n\}$ let $\mathcal{D}(\Psi,\sigma)$ be the measure value, then the value for the recursive call is $\mathcal{D}(\Psi,\iota_i)$ where $\iota_i = \langle f^{+}{:}\pi\rangle$ and $\mathcal{D}(\Psi,\iota_i) < \mathcal{D}(\Psi,\sigma)$.

As done for `is_empty`, we provide a proof sketch for termination of predicate `subtype`, based on the definition of a similar measure. We define $\delta_i(\Psi)$ as the set of the i-th component of the pairs in Ψ.

$$\mathcal{D}(\Psi,\sigma_1,\sigma_2) = \Sigma_{i=1}^{2}\Sigma_{j=1}^{2}\mathcal{D}_{aux}(\delta_i(\Psi),\sigma_j)$$

$$\mathcal{D}_{aux}(\Psi,\sigma) = \begin{cases} 1 + \Sigma_{i=1}^{n}\mathcal{D}_{aux}(\Psi,\varsigma_i) \text{ if } \sigma = \vee\{\varsigma_1,\dots,\varsigma_n\} \\ 1 + \Sigma_{i=1}^{n}\mathcal{D}_{aux}(\Psi,\iota_i) \text{ if } \sigma = \wedge\{\iota_1,\dots,\iota_n\} \\ 1 + \mathcal{D}_{aux}(\Psi\cup\{\sigma\}, norm(\pi)) \text{ if } \sigma = \langle f^{\nu}{:}\pi\rangle \wedge \sigma \notin \Psi \\ 1 \text{ if } \sigma = \langle f^{\nu}{:}\pi\rangle \wedge \sigma \in \Psi \\ 1 \text{ if } \sigma \in \{int, null, \mathbf{0}, \mathbf{1}\} \end{cases}$$

We now prove that the measure strictly decreases for each recursive occurrence of the predicate.

When $\sigma_1 = \vee\{\varsigma_1,\dots,\varsigma_n\}$ let $\mathcal{D}(\Psi,\sigma_1,\sigma_2)$ be the measure value, then the value for the first recursive call is $\mathcal{D}(\Psi,\varsigma,\sigma_2)$ and $\mathcal{D}_{aux}(\delta_1(\Psi),\varsigma) < \mathcal{D}_{aux}(\delta_1(\Psi),\sigma_1)$; the value for the second recursive call is $\mathcal{D}(\Psi,\vee\{\varsigma_2,\dots,\varsigma_n\},\sigma_2)$ and $\mathcal{D}_{aux}(\delta_1(\Psi),\vee\{\varsigma_2,\dots,\varsigma_n\}) < \mathcal{D}_{aux}(\delta_1(\Psi),\sigma_1)$.

When $\sigma_2 = \vee\{\varsigma_1,\dots,\varsigma_n\}$ let $\mathcal{D}(\Psi,\sigma_1,\sigma_2)$ be the measure value, then the value for the first recursive call is $\mathcal{D}(\Psi,\sigma_1,\varsigma_1)$ and $\mathcal{D}_{aux}(\delta_2(\Psi),\varsigma_1) < \mathcal{D}_{aux}(\delta_2(\Psi),\sigma_2)$; then, the value for the second recursive call in clause is $\mathcal{D}(\Psi,\sigma_1,\vee\{\varsigma_2,\dots,\varsigma_n\})$ and $\mathcal{D}_{aux}(\delta_2(\Psi),\vee\{\varsigma_2,\dots,\varsigma_n\}) < \mathcal{D}_{aux}(\delta_2(\Psi),\sigma_2)$.

In the case corresponding to rule $(empty)$ let $\mathcal{D}(\Psi,\sigma_1,\sigma_2)$ be the measure value, then the value for the first recursive call is $\mathcal{D}(\Psi, norm(\pi_1), norm(\pi_2))$, since $\bar{\Pi}$ contains only subterms of σ_2 we have that $\mathcal{D}(\Psi, norm(\pi_1), norm(\pi_2)) < \Sigma_{i=1}^{2}\mathcal{D}_{aux}(\delta_i(\Psi),\sigma_2)$.

When $\sigma_1 = \wedge\{\iota_1,\dots,\iota_n\}$ let $\mathcal{D}(\Psi,\sigma_1,\sigma_2)$ be the measure value, then the value for the first recursive call is $\mathcal{D}(\Psi,\iota_1,\sigma_2)$ and $\mathcal{D}_{aux}(\delta_1(\Psi),\iota_1) < \mathcal{D}_{aux}(\delta_1(\Psi),\sigma_1)$; the value for the second recursive call is $\mathcal{D}(\Psi,\wedge\{\iota_2,\dots,\iota_n\},\sigma_2)$ and $\mathcal{D}_{aux}(\delta_1(\Psi),\wedge\{\iota_2,\dots,\iota_n\}) < \mathcal{D}_{aux}(\delta_1(\Psi),\sigma_1)$.

When $\sigma_2 = \wedge\{\iota_1,\dots,\iota_n\}$ let $\mathcal{D}(\Psi,\sigma_1,\sigma_2)$ be the measure value, then the value for the first recursive call is $\mathcal{D}(\Psi,\sigma_1,\iota_1)$ and $\mathcal{D}_{aux}(\delta_2(\Psi),\iota_1) < \mathcal{D}_{aux}(\delta_2(\Psi),\sigma_2)$; the value for the second recursive call is $\mathcal{D}(\Psi,\sigma_1,\wedge\{\iota_2,\dots,\iota_n\})$ and $\mathcal{D}_{aux}(\delta_2(\Psi),\wedge\{\iota_2,\dots,\iota_n\}) < \mathcal{D}_{aux}(\delta_2(\Psi),\sigma_2)$.

When $\sigma_1 = \langle f^{+}{:}\_\rangle$ and $\sigma_2 = \langle f^{-}{:}\pi\rangle$, let $\mathcal{D}(\Psi,\sigma_1,\sigma_2)$ be the measure value, and $\Psi' = \{(\sigma_1,\sigma_2)\} \cup \Psi$; then the value for the recursive call is $\mathcal{D}(\Psi, norm(\pi_1), norm(\pi_2))$; note that π_1 and π_2 are subterms of $norm(\pi)$ and that $\mathcal{D}_{aux}(\delta_2(\Psi), norm(\pi_1)) + \mathcal{D}_{aux}(\delta_2(\Psi), norm(\pi_2)) < \mathcal{D}_{aux}(\delta_2(\Psi),\sigma_2)$.

When $\sigma_1 = \langle f^{+}{:}\pi_1\rangle$ and $\sigma_2 = \langle f^{+}{:}\pi_2\rangle$, let $\mathcal{D}(\Psi,\sigma_1,\sigma_2)$ be the measure value, and $\Psi' = \{(\sigma_1,\sigma_2)\} \cup \Psi$; then the value for the recursive call is $\mathcal{D}(\Psi', norm(\pi_1), norm(\pi_2))$ and $\mathcal{D}_{aux}(\delta_1(\Psi'), norm(\pi_1)) < \mathcal{D}_{aux}(\delta_1(\Psi),\sigma_1)$ and $\mathcal{D}_{aux}(\delta_2(\Psi'), norm(\pi_2)) < \mathcal{D}_{aux}(\delta_2(\Psi),\sigma_2)$.

When $\sigma_1 = \langle f^{-}{:}\pi_1\rangle$ and $\sigma_2 = \langle f^{-}{:}\pi_2\rangle$ let $\mathcal{D}(\Psi,\sigma_1,\sigma_2)$ be the measure value, and $\Psi' = \{(\sigma_1,\sigma_2)\} \cup \Psi$ then the value for the recursive call is $\mathcal{D}(\Psi', norm(\pi_2), norm(\pi_1))$ and $\mathcal{D}_{aux}(\delta_1(\Psi'), norm(\pi_1)) < \mathcal{D}_{aux}(\delta_1(\Psi),\sigma_1)$ and $\mathcal{D}_{aux}(\delta_2(\Psi'), norm(\pi_2)) < \mathcal{D}_{aux}(\delta_2(\Psi),\sigma_2)$.

7. Conclusion

In this paper we have investigated a coinductive semantic model for record types with read/write field annotations, supporting union, intersection, and recursive types.

We have provided an interpretation of types which accommodates read/write annotations with the semantic subtyping approach, so that subtyping corresponds to set inclusion between type interpretations. Although intuitive, the model poses some challenging issues, since subtyping is defined in terms of values, but values in turn contain type annotations to indicate what can be safely assigned to fields. This leads to a circular definition between the coinductive judgment for typing values, and its negation, which is inductive; to break this circularity, the inductive judgment depends on a set of type assumptions on values that must hold for the judgment to be derivable.

The semantic model has allowed us to study the main laws underlying subtyping for record types with read/write field annotations, supporting union, intersection, and recursive types.

Furthermore, we have tackled the challenging problem of defining a system of subtyping rules which are sound and complete w.r.t. the definition of semantic subtyping. Similar issues concerning circularity between coinductive judgments and their corresponding negations (which, by duality, are inductive) have been faced.

In Section 6 we show how such rules can be effectively implemented.

To our knowledge, this is the first implementation of a sound and complete procedure to decide subtyping for record types with read/write field annotations, supporting union, intersection, and recursive types.

There are several interesting directions for future extensions to types and the corresponding subtyping relation. In Section 2 we have shown that union and intersection types together with read/write annotations can be used to enforce monotonic initialization for fields; for instance, the type $\langle f^+{:}null{\vee}T\rangle{\wedge}\langle f^-{:}T\rangle$ allows field f to be associated with the initial null value, but forces any assignment to f to store a non-null value (if we assume that $null \not\leq T$). Once a value of type T has been assigned to field $x.f$, a system supporting strong updates would allow narrowing the type of x from $\langle f^+{:}null{\vee}T\rangle{\wedge}\langle f^-{:}T\rangle$ to $\langle f^+{:}T\rangle{\wedge}\langle f^-{:}T\rangle$; in general, to be sound, such a narrowing requires non trivial points-to analysis, because x could be aliased by a variable y having the subtype $\langle f^+{:}null{\vee}T\rangle{\wedge}\langle f^-{:}null{\vee}T\rangle$, hence the null value could be reassigned to field $x.f$ through y. While write-only fields allow safe contravariant subtyping, it would be also possible to introduce record types $\langle f^\ominus{:}T\rangle$ with write-only invariant fields, s.t. $\langle f^\ominus{:}T_1\rangle \leq \langle f^\ominus{:}T_2\rangle$ iff $T_1 \equiv T_2$. Invariant write-only fields support strong updates without requiring points-to analysis: for instance, after field $x.f$ is assigned to a value of type T, the type of x can be safely narrowed from $\langle f^+{:}null{\vee}T\rangle{\wedge}\langle f^\ominus{:}T\rangle$ to $\langle f^+{:}T\rangle{\wedge}\langle f^\ominus{:}T\rangle$, independently of aliasing, because x can be aliased by variables whose type only allows assignment of values of type T to field f.

Another interesting extension would consist in the introduction of record types with negative information to specify the absence of fields.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[2] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical logic*. North Holland, 1977.

[3] J. Altidor, C. Reichenbach, and Y. Smaragdakis. Java wildcards meet definition-site variance. In *ECOOP'12*, pages 509–534, 2012.

[4] D. Ancona and A. Corradi. Sound and complete subtyping between coinductive types for object-oriented languages. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference*, volume 8586 of *Lecture Notes in Computer Science*, pages 282–307. Springer, 2014.

[5] D. Ancona and A. Corradi. Semantic subtyping between coinductive mutable record types with unions and intersections. In *Italian Conference on Theoretical Computer Science (ICTCS 2015)*, 2015. On-line proceedings.

[6] D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In S. Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming*, volume 5653 of *LNCS*, pages 2–26. Springer, 2009. ISBN 978-3-642-03012-3.

[7] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *ECOOP 2005*, pages 428–452, 2005.

[8] F. Barbanera, M. Dezani-Ciancaglini, and U. De'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[9] M. Bonsangue, J. Rot, D. Ancona, F. de Boer, and J. Rutten. A coalgebraic foundation for coinductive union types. In *ICALP 2014 - 41st International Colloquium on Automata, Languages and Programming*, pages 62–73, 2014.

[10] N. Cameron, S. Drossopoulou, and E. Ernst. A model for Java with wildcards. In *ECOOP 2008*, pages 2–26, 2008.

[11] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the π-calculus. *Theoretical Computer Science*, 398 (1-3):217–242, 2008. doi: 10.1016/j.tcs.2008.01.049.

[12] G. Castagna, K. Nguyen, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *POPL '14*, pages 5–18, 2014.

[13] G. Castagna, K. Nguyen, Z. Xu, and P. Abate. Polymorphic functions with set-theoretic types: Part 2: Local type inference and type reconstruction. In *POPL 2015*, pages 289–302, 2015.

[14] G. Castagna, T. Petrucciani, and K. Nguyen. Set-theoretic types for polymorphic variants. In *ICFP 2016*, 2016. To appear.

[15] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[16] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA 2003*, pages 302–312, 2003.

[17] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4), 2008.

[18] J. Gil and I. Maman. Whiteoak: Introducing structural typing into Java. In *OOPSLA '08*, 2008.

[19] B. Hackett and S. Guo. Fast and precise hybrid type inference for javascript. In *PLDI '12*, pages 239–250, 2012.

[20] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *ECOOP 2010*, pages 200–224, 2010.

[21] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.

[22] A. Igarashi and H. Nagira. Union types for object-oriented programming. *Journ. of Object Technology*, 6(2):47–68, 2007.

[23] T. Jones, M. Homer, and J. Noble. Brand Objects for Nominal Typing. In *ECOOP 2015*, pages 198–221, 2015.

[24] V. Kashyap, J. Sarracino, J. Wagner, B. Wiedermann, and B. Hardekopf. Type refinement for static analysis of javascript. In *DLS'13*, pages 17–26, 2013.

[25] B. Lerner, J. Politz, A. Guha, and S. Krishnamurthi. Tejas: retrofitting type systems for javascript. In *DLS'13*, pages 1–16, 2013.

[26] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207:284–304, 2009.

[27] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *ECOOP 2008*, pages 260–284, 2008.

[28] L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007*, pages 472–483, 2007.

[29] M. Torgersen, E. Ernst, C. P. Hansen, P. von der Ahé, G. Bracha, and N. M. Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97–116, 2004.

[30] T. Wang and S. F. Smith. Precise constraint-based type inference for Java. In *ECOOP'01 - European Conference on Object-Oriented Programming*, volume 2072, pages 99–117. Springer, 2001.

Figure 8. Unification algorithm

`unify` (Q, τ'', τ''')

— *first rewrites all bounds of Q in normal form and proceeds by case analysis on (τ'', τ''') :*

Case (α, α): **return** Q.

Case $(g\ \tau_1^1\ ..\ \tau_1^n, g\ \tau_2^1\ ..\ \tau_2^n)$:

- **let** Q^1 be Q **in**
- **let** Q^{i+1} be `unify` $(Q^i, \tau_1^i, \tau_2^i)$ for $1 \leqslant i \leqslant n$ **in**
- **return** Q^{n+1}.

Case $(g_1\ \tau_1^1\ ..\ \tau_1^p, g_2\ \tau_2^1\ ..\ \tau_2^q)$ with $g_1 \neq g_2$: **fail**.

Case (α, τ) or (τ, α) when $(\alpha \diamond \tau') \in Q$:

- **return** `unify` (Q, τ, τ').

Case (α, τ) or (τ, α) when $(\alpha \diamond \sigma) \in Q$ and $\tau \notin \mathsf{dom}(Q)$ and $\sigma \notin \mathcal{T}$

- **let** $(Q', \_)$ be `polyunify` (Q, σ, τ) **in**
- **fail** if $(\alpha \diamond \sigma) \notin Q'$.
- **return** $(Q') \Leftarrow (\alpha = \tau)$

Case (α_1, α_2) when $(\alpha_1 \diamond_1 \sigma_1) \in Q$ and $(\alpha_2 \diamond_2 \sigma_2) \in Q$ and $\alpha_1 \neq \alpha_2$ and σ_1, σ_2 are not in $\mathcal{T}$.

- **let** (Q', σ_3) be `polyunify` (Q, σ_1, σ_2) **in**
- **fail** if $(\alpha_1 \diamond_1 \sigma_1) \notin Q'$ or if $(\alpha_2 \diamond_2 \sigma_2) \notin Q'$
- **return** $(Q') \Leftarrow (\alpha_1 \diamond_1 \sigma_3) \Leftarrow (\alpha_2 \diamond_2 \sigma_3) \Leftarrow \alpha_1 \wedge \alpha_2$.

`polyunify` (Q, σ_1, σ_2)

— *requires σ_1, σ_2, and all bounds in Q to be in normal form*[1] *:*

Case $(\bot, \sigma)$ or $(\sigma, \bot)$: **return** (Q, σ)

Case $(\forall (Q_1)\ \tau_1, \forall (Q_2)\ \tau_2)$ with Q_1, Q_2, and Q having disjoint domains (which usually requires renaming σ_1 and σ_2)

- **let** Q_0 be `unify` $(QQ_1Q_2, \tau_1, \tau_2)$ **in**
- **let** (Q_3, Q') be $Q_0 \uparrow \mathsf{dom}(Q)$ **in**
- **return** $(Q_3, \forall (Q')\ \tau_1)$

[1] Actually, only need to replace types of the form $\forall (\alpha \diamond \sigma)\ \alpha$ by σ, which can always be done lazily.

given to `unify` as a substitution and of the result prefix Q' as an instance of Q (*i.e.* a substitution of the form $Q'' \circ Q$) that unifies τ and τ'. First-order unification of polytypes essentially follows the general structure of first-order unification of monotypes. The main differences are that (i) the computation of the unifying substitution is replaced by the computation of a unifying prefix, (ii) additional work must be performed when a variable bound to a strict polytype (*i.e.* other than $\bot$ and not equivalent to a monotype) is being unified: bounds must be further unified (last case of `polyunify`) and the prefix updated accordingly. Auxiliary algorithms are used for this purpose.

Let a *rearrangement* of a prefix Q be a prefix equivalent to Q obtained by a permutation of bindings of Q.

DEFINITION 9. The *split* algorithm $Q \uparrow \bar{\alpha}$ takes a prefix Q and returns a pair of prefixes (Q_1, Q_2) such that (i) $Q_1 Q_2$ is a rearrangement of Q, (ii) $\bar{\alpha} \subseteq \mathsf{dom}(Q_1)$, and (iii) $\mathsf{dom}(Q_1)$ is minimal (in other words, Q_1 contains only bindings of Q useful for exporting the interface $\bar{\alpha}$, and Q_2 contains the rest). □

Figure 9. Algorithm W^{F}

infer (Q, Γ, a):

— *proceeds by case analysis on expression a :*

Case x: **return** $Q, \Gamma(x)$

Case $\lambda(x)\ a$:

- **let** $Q_1 = (Q, \alpha \geq \bot)$ with $\alpha \notin \mathsf{dom}(Q)$
- **let** $(Q_2, \sigma) =$ *infer* $(Q_1, \Gamma, x : \alpha, a)$
- **let** $\beta \notin \mathsf{dom}(Q_2)$ and $(Q_3, Q_4) = Q_2 \uparrow \mathsf{dom}(Q)$
- **return** $Q_3, \forall (Q_4)\ \forall (\beta \geq \sigma)\ \alpha \to \beta$

Case $a\ b$:

- **let** $(Q_1, \sigma_a) =$ *infer* (Q, Γ, a)
- **let** $(Q_2, \sigma_b) =$ *infer* (Q_1, Γ, b)
- **let** $\alpha_a, \alpha_b, \beta \notin \mathsf{dom}(Q_2)$
- **let** $Q_3 =$ `unify` $((Q_2, \alpha_a \geq \sigma_a, \alpha_b \geq \sigma_b, \beta \geq \bot), \alpha_a, \alpha_b \to \beta)$
- **let** $(Q_4, Q_5) = Q_3 \uparrow \mathsf{dom}(Q)$
- **return** $(Q_4, \forall (Q_5)\ \beta)$

Case **let** $x = a_1$ **in** a_2:

- **let** $(Q_1, \sigma_1) =$ *infer* (Q, Γ, a_1)
- **return** *infer* $(Q_1, (\Gamma, x : \sigma_1), a_2)$

DEFINITION 10. The *abstraction-check* algorithm $(Q)\ \sigma \mathrel{-\!\!\!\sqsubset}^? \sigma'$ takes a prefix Q and two polytypes σ and σ' such that $(Q)\ \sigma \sqsubseteq \sigma'$ and checks that $(Q)\ \sigma \mathrel{-\!\!\!\sqsubset} \sigma'$ or fails otherwise. □

DEFINITION 11. The *update* algorithm $Q \Leftarrow (\alpha \diamond \sigma)$ takes a prefix Q and a binding $(\alpha \diamond \sigma)$ such that α is in the domain of Q and returns a prefix $(Q_0, \alpha \diamond \sigma, Q_1)$ such that (i) $(Q_0, \alpha \diamond' \sigma', Q_1)$ is a rearrangement of Q and (ii) $\mathsf{dom}(Q_1) \cap \mathsf{ftv}(\sigma) = \emptyset$. The algorithm fails when there is not such decomposition (because of circular dependencies) or when $\diamond'$ is $=$ and $(Q)\ \sigma' \mathrel{-\!\!\!\sqsubset}^? \sigma$ fails. □

DEFINITION 12. The *merge* algorithm $Q \Leftarrow \alpha \wedge \alpha'$ takes two variables α and α' and a prefix of the form $(Q_0, \alpha \diamond \sigma, Q_1, \alpha' \diamond' \sigma', Q_2)$ and returns the prefix $(Q_0, \alpha \diamond'' \sigma, Q_1, \alpha' = \alpha, Q_2)$ where $\diamond''$ is $\geq$ if both $\diamond$ and $\diamond'$ are $\geq$, and $\diamond''$ is $=$ otherwise. □

The implementation of algorithms *split*, *update*, and *merge* is straightforward. The algorithm *abstraction-check* can be reduced to a simple check on the structure of paths, thanks to the assumption $(Q)\ \sigma \sqsubseteq \sigma'$. By lack of space, they are all omitted.

B Type inference algorithm

Figure 9 defines the type-inference algorithm W^{F} for ML^{F}. The algorithm follows the algorithm W for ML, with only two differences: first, the algorithm builds a prefix Q instead of a substitution; second, all free type variables not in Γ are quantified at each abstraction or application. Since free variables of Γ are in $\mathsf{dom}(Q)$, finding quantified variables consists in splitting the current prefix according to $\mathsf{dom}(Q)$, as described by Definition 9.