Local Type Inference for Polarised System F with Existentials

ANONYMOUS AUTHOR(S)

We study type inference for the calculus $F^{\pm}_{\exists}$, which is a version of call-by-push-value extended with support for fully impredicative existential and universal quantifiers. We give a local type inference algorithm for this calculus as well as a declarative type system acting as a specification for the algorithm and show that the algorithm is sound and complete with respect to the declarative specification. The inclusion of existential quantifiers is unusual and supporting it required us to use a number of novel techniques including the combination of unification and anti-unification as part of the inference algorithm.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Additional Key Words and Phrases: Type Inference, System F, Call-by-Push-Value, Polarized Typing, Focalisation, Subtyping

ACM Reference Format:
Anonymous Author(s). 2018. Local Type Inference for Polarised System F with Existentials. *J. ACM* 37, 4, Article 111 (August 2018), 44 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Over the last half-century, there has been considerable work on developing type inference algorithms for programming languages, mostly aimed at solving the problem of *type polymorphism*.

That is, in pure polymorphic lambda calculus—System F [Girard 1971; Reynolds 1974]—the polymorphic type $\forall\alpha.\, A$ has a big lambda $\Lambda\alpha.\, e$ as an introduction form, and an explicit type application $e\,\{A\}$ as an elimination form. This is an extremely simple and compact type system, whose rules fit on a single page, but whose semantics are advanced enough to accurately represent concepts such as parametricity and representation independence. However, System F by itself is unwieldy as a programming language. The fact that the universal type $\forall\alpha.\, A$ has explicit eliminations means that programs written using polymorphic types will need to be stuffed to the gills with type annotations explaining how and when to instantiate the quantifiers.

Therefore, most work on type inference has been aimed at handling type instantiations implicitly—we want to be able to use a polymorphic function like $\texttt{len} : \forall\alpha.\, [\alpha] \to \texttt{Int}$ at any concrete list type without explicitly instantiating the quantifier in `len`'s type. That is, we want to write $\texttt{len}\,[1, 2, 3]$ instead of writing $\texttt{len}\,\{\texttt{Int}\}\,[1, 2, 3]$.

The way this is typically done is by using a *subtyping* relation, induced by the polymorphic instantiation: the type $\forall\alpha.\, A$ is a *subtype* of all of its instantiations. So we wish to be free to use the same polymorphic function `len` at many different types such as $[\texttt{Int}] \to \texttt{Int}$, $[\texttt{Bool}] \to \texttt{Int}$, $[\texttt{Int}] \times \texttt{Bool} \to \texttt{Int}$, and so on. However, the subtyping can be used nondeterministically: whenever we see a polymorphic type $\forall\alpha.\, A$, we know it is a subtype of *any* of its instantiations. To turn this into an algorithm, we have to actually make some choices.

0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

The most famous of the algorithms for handling this is the Damas-Hindley-Milner (DHM) algorithm [Damas et al. 1982; Hindley 1969; Milner 1978]. DHM deals with this choice using *unification*. Whenever we would have had to guess a particular concrete type in the specification, the DHM algorithm introduces a *unification variable*, and incrementally instantiates this variable as more and more type constraints are observed.

However, the universal quantifier is not the only quantifier! Dual to the universal quantifier $\forall$ is the existential quantifier $\exists$. Even though existential types have an equal logical status to universal types, they have been studied much less frequently in the literature. The most widely-used algorithm for handling existentials is the algorithm of Odersky and Läufer [Läufer et al. 1994]. This algorithm treats existentials in a second-class fashion: they are not explicit connectives of the logic but rather are tied to datatype declarations. As a result, packing and unpacking existential types is tied to constructor introductions and constructor eliminations in pattern matches. This allows Damas-Milner inference to continue to work almost unchanged, but it does come at the cost of losing first-class existentials and also of restricting the witness types to monomorphic types (i.e., types with no quantifiers inside of them).

There has been a limited amount of work on support for existential types as a first-class concept. In an unpublished draft, Leijen [2006] studied an extension of Damas-Milner inference in which type schemes contain alternating universal and existential quantifiers. Quantifiers still range over monotypes, so higher-rank polymorphism is not permitted: one cannot instantiate a quantifier with a type that itself contains quantifiers. More recently, Dunfield et al. [2016] studied type inference for existential types in the context of GADT type inference, which, while still predicative, supported higher-rank types (i.e., quantifiers can occur anywhere inside of a type scheme). Eisenberg et al. [2021] propose a system which only permits types of the form forall-exists, but which permits projective elimination in the style of ML modules.

All of these papers are restricted to *predicative* quantification, where quantifiers can only be instantiated with monotypes. However, existential types in full System F are *impredicative*—that is, quantifiers can be instantiated with arbitrary types, specifically including types containing quantifiers.

Historically, inference for impredicative quantification has been neglected, due to results by Tiuryn et al. [1996] and Chrzaszcz [1998] which show that not only is full type inference for System F undecidable, but even that the subtyping relation induced by type instantiation is undecidable. However, in recent years, interest in impredicative inference has revived (for example, the work of Serrano et al. [2020]), with a focus on avoiding the undecidability barriers by doing *partial* type inference. That is, we no longer try to infer the type of any term in the language, but rather accept that the programmer will need to write annotations in cases where inference would be too difficult. Unfortunately, it is often difficult to give a specification for when the partial algorithm succeeds—for example, Eisenberg et al. [2021] observe that their algorithm lacks a declarative specification, and explain why supporting existential types makes giving a specification particularly difficult.

One especially well-behaved form of partial type inference is *local type inference*, introduced by Pierce et al. [2000]. In this approach, quantifiers in a polymorphic function are instantiated using only the type information available in the arguments at each call site. While this infers fewer quantifiers than global algorithms such as Damas-Milner can, it has the twin benefits that it is easy to give a mathematical specification to, and that failures of type inference are easily explained in terms of local features of the call site. In fact, many production programming languages such as C# and Java use a form of local type inference, due to the ease of extending this style of inference to support object-oriented features.

In this paper, we extend the local type inference algorithm to a language with both universal and existential quantifiers, which can both be instantiated impredicatively. This extended system

is referred to as $F^{\pm}_{\exists}$. The combination of features in $F^{\pm}_{\exists}$ broke a number of the invariants which traditional type inference algorithms depend on and required us to invent new algorithms which combine both unification and (surprisingly) anti-unification.

We make the following ***contributions***:

Declarative Type System Specification (Sections 2 and 3). We present $F^{\pm}_{\exists}$—a second-order polymorphic type system that supports first-class existential and universal quantifiers, both of which can be instantiated impredicatively. To evade the undecidability results surrounding subtyping for System F, we base the system on call-by-push-value [Levy 2006]: we use *polarization* of types and terms to formulate the restrictions that make typing decidable.

We give a *declarative* specification of local type inference in this system (i.e., the typing information is only available at a limited scope) based on a *subtyping* relation. Our specification makes it easy to see what is and what is not inferrable in the system.

To show that our restrictions do not unduly limit the expressiveness of the language, we prove that every $F^{\pm}_{\exists}$ term can be embedded into System F and vice versa.

Type Inference Algorithm (Section 4). We give a type inference algorithm implementing the declarative specification. Because our system supports both existential and universal quantifiers, our implementation of the subtyping relation requires the use of *anti-unification*—a dual to unification that finds the most specific generalization rather than the most general unifier. To our knowledge, anti-unification has never been used in the context of polymorphic type inference.

The type inference algorithm itself is a purely local one, in the style of Pierce et al. [2000]. The locality can be seen from the fact that none of the unification variables introduced as part of type inference escape the scope of a single function application.

Correctness of the algorithm (Section 5). We prove that our algorithm is terminating and both sound and complete with respect to the declarative specification. This proof of soundness and completeness of the algorithm required us to introduce an intermediate system in-between the declarative and the algorithmic systems, and furthermore, the soundness and completeness proofs are mutually recursive with each other. The central theorems and the key proof ideas are formulated in Section 5, while the full definitions and formal proofs are deferred to the appendix (**??**).

Acknowledgments. We would like to thank Meven Lennon-Bertrand for his extremely helpful comments and suggestions.

2 OVERVIEW

In the presence of impredicative polymorphism, System F has undecidable subtyping and type inference. It means that any inference algorithm is incomplete: it only infers the types for a *fragment* of System F.

To tackle this issue, we restrict System F by employing the ideas from the call-by-push-value system [Levy 2006]. We introduce a new language, $F^{\pm}_{\exists}$, which stratifies the syntax of System F into positive and negative parts. It provides the structure that allows us to restrict the typing specification to make it decidable.

Every term and type of $F^{\pm}_{\exists}$ has either *positive* or *negative* polarity. The type variables are annotated with their polarities (e.g. α^{+} or β^{-}), and the types can change polarity via the shift operators $\uparrow$ (positive to negative) and $\downarrow$ (negative to positive).

Positive expressions correspond to *values* in the call-by-push-value system, and negative expressions correspond to (potentially effectful) *computations*. The difference between the computations and values is intuitively described in the motto of call-by-push-value: "a value is, a computation

does". In particular, a function type always takes a value and returns a computation. Similarly, the polymorphic ∀ quantifies a computation over positive types.

Whenever a computation is used in a value position (e.g. as an argument to a function), it must be *suspended* by a thunk constructor, which at the type level corresponds to the downshift operator $\downarrow N$. Symmetrically, to use a value as a computation, one can *return* it, which at the type level is reflected as an up-shift $\uparrow P$.

2.1 Examples

Generally, any term and type of System F can be embedded into $\mathrm{F}^{\pm}_{\exists}$ in multiple ways corresponding to different evaluation strategies of System F, which we will cover in more detail in Section 3.4.

Figure 1 illustrates how standard System F types are polarized when call-by-value evaluation is presumed.

Observe the consistent polarization pattern of types. Consider the type for `double`, which in System F would be `Int` → `Int`. In polarized form, it becomes $\downarrow(\texttt{Int} \to \uparrow\texttt{Int})$—a *suspended* function (indicated by the downshift) that accepts an integer `Int` and yields an integer *computation* (indicated by the upshift) $\uparrow\texttt{Int}$.

Using the examples provided in Fig. 1, we now showcase the type inference capabilities and restrictions of $\mathrm{F}^{\pm}_{\exists}$

$$
\begin{aligned}
\texttt{double} &: \downarrow(\texttt{Int} \to \uparrow\texttt{Int}) \\
\texttt{map} &: \downarrow(\forall\alpha^+.\, \forall\beta^+.\, \downarrow(\alpha^+ \to \uparrow\beta^+) \to [\alpha^+] \to \uparrow[\beta^+]) \\
\texttt{len} &: \downarrow(\forall\alpha^+.\, [\alpha^+] \to \uparrow\texttt{Int}) \\
\texttt{choose} &: \downarrow(\forall\alpha^+.\, \alpha^+ \to \alpha^+ \to \uparrow\alpha^+) \\
\texttt{id} &: \downarrow(\forall\alpha^+.\, \alpha^+ \to \uparrow\alpha^+) \\
\texttt{auto} &: \downarrow(\downarrow(\forall\alpha^+.\, \alpha^+ \to \uparrow\alpha^+) \to (\forall\alpha^+.\, \alpha^+ \to \uparrow\alpha^+))
\end{aligned}
$$

Fig. 1. Polarization of System F types

Application of a Polymorphic Function. A polymorphic function can be applied to a value of a concrete type. In this case, the explicit type application is not required: the inference algorithm automatically instantiates the quantified variables. For instance, the polymorphic `len` can be applied to a list of any positive type, and the resulting type will be inferred by the algorithm:

$$\cdot; \Gamma \vdash \texttt{let}\, x = \texttt{len}([1, 2, 3]);\, \texttt{return}\, x : \uparrow\texttt{Int}$$

Polymorphic Self-application. Unlike predicative systems, $\mathrm{F}^{\pm}_{\exists}$ permits self-application of polymorphic functions. For instance, `id` can be applied to itself resulting in a suspended polymorphic computation of type $\downarrow(\forall\alpha^+.\, \alpha^+ \to \uparrow\alpha^+)$, which, in turn, can be applied to itself:

$$\cdot; \Gamma \vdash \texttt{let}\, x = \texttt{id}(\texttt{id});\, \texttt{let}\, y = x(x);\, \texttt{return}\, y : \uparrow\downarrow(\forall\alpha^+.\, \alpha^+ \to \uparrow\alpha^+)$$

Non-example: Polymorphic Arguments. The focus of our system is the *local* type inference [Pierce et al. 2000], which has certain limitations. In particular, the instantiation does not happen when the polymorphic term is in the argument position. For example, one could expect the following to be inferrable: $\cdot; \Gamma \vdash \texttt{let}\, x = \texttt{map}(\texttt{id}, [2, 3, 9]);\, \texttt{return}\, x : \uparrow[\texttt{Int}]$. However, to infer this, `id` must be instantiated to `Int` → $\uparrow$`Int` which requires the information to be propagated from the neighboring branch of the syntax tree ($[2, 3, 9]$), i.e., bypass the locality.

Negative declarative types

$$N, M, K \quad ::= \quad \alpha^- \mid \uparrow P \mid P \to N \mid \forall \overrightarrow{\alpha^+}. N$$

Positive declarative types

$$P, Q, R \quad ::= \quad \alpha^+ \mid \downarrow N \mid \exists \overrightarrow{\alpha^-}. P$$

Fig. 2. Declarative Types of $\mathrm{F}^{\pm}_{\exists}$

Inferring the Common Supertype. The function `choose` takes two arguments of the same polymorphic type. To apply it to two values of different (concrete) types, the inference algorithm must find the minimal type they can be coerced to—the least common supertype.

Let us assume that in Γ, we have two identity-like functions $\mathtt{id}_M : \downarrow(\downarrow M \to M)$ and $\mathtt{id}_N : \downarrow(\downarrow N \to N)$. Then their least common supertype—the existential $\exists \gamma^-. \downarrow(\downarrow \gamma^- \to \gamma^-)$ is inferrable:

$$\cdot; \Gamma \vdash \mathtt{let}\ x = \mathtt{choose}(\mathtt{id}_N, \mathtt{id}_M);\ \mathtt{return}\ x : \uparrow \exists \gamma^-. \downarrow(\downarrow \gamma^- \to \gamma^-)$$

Inferring a general type. As mentioned before, the local inference does not instantiate the polymorphic types in the argument position. Because of that, the application $\mathtt{let}\ x = \mathtt{choose}(\mathtt{id}, \mathtt{auto});\ \mathtt{return}\ x$ cannot infer $\uparrow\downarrow(\downarrow(\forall \alpha^+. \alpha^+ \to \uparrow \alpha^+) \to (\forall \alpha^+. \alpha^+ \to \uparrow \alpha^+))$. However, the inference succeeds and infers a less informative type:

$$\cdot; \Gamma \vdash \mathtt{let}\ x = \mathtt{choose}(\mathtt{id}, \mathtt{auto});\ \mathtt{return}\ x : \uparrow \exists \gamma^-. \downarrow \gamma^-$$

2.2 The Language of Types

The types of $\mathrm{F}^{\pm}_{\exists}$ are given in Fig. 2. They are stratified into two syntactic categories (polarities): positive and negative, similar to the values and computations in the call-by-push-value system.

- ± α^- and α^+ denote negative and positive type variables, respectively. The variables are either introduced by the corresponding quantifiers ($\forall$ and $\exists$, respectively) or declared in the type context;
- – a function $P \to N$ takes a *positive* input P and returns a *negative* output N; the function type itself is *negative*, so the 'arrow' operator is right-associative: $P \to Q \to N$ is equivalent to $P \to (Q \to N)$;
- – similarly to functions, a polymorphic abstraction $\forall \overrightarrow{\alpha^+}. N$ quantifies a negative type N over a list of positive type variables $\overrightarrow{\alpha^+}$;
- + dually, $\exists \overrightarrow{\alpha^-}. P$ binds negative variables in a positive type P;
- ± an upshift $\uparrow P$ and a downshift $\downarrow N$ change the polarity of a type from positive to negative and vice versa. At the level of terms, the constructors for shift types are `return` v and $\{c\}$ (thunked computation), respectively.

Syntactic Conventions. We always consider terms and types up to alpha-equivalence, and capture-avoiding substitution. Besides, we assume the quantification is associative, for example, $\forall \overrightarrow{\alpha^+}. \forall \overrightarrow{\beta^+}. N$ and $\forall \overrightarrow{\alpha^+}, \overrightarrow{\beta^+}. N$ represent the same type.

Type Context and Type Well-formedness. In the construction of $\mathrm{F}^{\pm}_{\exists}$, a type context (denoted as Θ) is represented as a *set* of positive and negative type variables and it is used to assert the well-formedness of types. The well-formedness (or well-scopeness) of a type is denoted as $\Theta \vdash P$ and $\Theta \vdash N$ and it asserts that all type variables are either bound by a quantifier ($\forall$ and $\exists$) or declared in the context Θ. The well-formedness checking is an *algorithmic* procedure. We represent it as a system of inference rules, that correspond to a recursive algorithm taking the context and the type as input. For brevity, we omit these rules, as they are standard binder scoping rules.

Computation Terms

$$c,\ d \quad ::= \quad (c : N) \mid \lambda x : P.\, c \mid \Lambda\alpha^{+}.\, c \mid \texttt{return}\, v \mid \texttt{let}\, x = v;\ c \mid \texttt{let}\, x : P = c;\ c' \mid \texttt{let}\, x : P = v(\overrightarrow{v});\ c \mid \texttt{let}\, x = v(\overrightarrow{v});\ c \mid \texttt{let}^{\exists}(\overrightarrow{\alpha}, x) = v;\ c$$

Value Terms

$$v,\ w \quad ::= \quad x \mid \{c\} \mid (v : P)$$

Fig. 3. Declarative Terms of $\mathrm{F}^{\pm}_{\exists}$

2.3 The Language of Terms

In Fig. 3, we define the language of terms of $\mathrm{F}^{\pm}_{\exists}$. The language is a combination of System F and call-by-push-value constructs: the terms are stratified into values and computations; there are return and thunk constructors, as well as several forms of let bindings to sequentialize the computation/application with or without type annotation;

- + x denotes a term variable. Following the call-by-push-value stratification, we only have *positive* (value) term variables;
- + $\{c\}$ is a value corresponding to a thunked or suspended computation;
- ± $(c : N)$ and $(v : P)$ allow one to annotate positive and negative terms;
- − $\texttt{return}\, v$ is a pure computation, returning a value;
- − $\lambda x : P.\, c$ and $\Lambda\alpha^{+}.\, c$ are standard lambda abstractions. Notice that we require the type annotation for the argument of λ;
- − $\texttt{let}\, x = v;\ c$ is a standard let, binding a value v to a variable x in a computation c;
- − computational let form $\texttt{let}\, x : P = c;\ c'$ operates similarly to a monadic bind. It binds the result of a computation $c : \uparrow P$ to a variable $x : P$, and continues with a computation c';
- − applicative let forms $\texttt{let}\, x : P = v(\overrightarrow{v});\ c$ and $\texttt{let}\, x = v(\overrightarrow{v});\ c$ can be viewed as a special case of the computational let, when the first computation is a function application $v(\overrightarrow{v})$. They take a suspended computation v, apply it to a list of arguments, bind the (pure) result to a variable x, and continues with a computation c. If the resulting type of the application is unique, one can omit the type annotation, as in the second form: it will be inferred by the algorithm;
- − $\texttt{let}^{\exists}(\overrightarrow{\alpha}, x) = v;\ c$ is the standard eliminator of an existential type (unpack): expecting v to be of an existential type, it binds the packed negative types to a list of variables $\overrightarrow{\alpha}$, binds the body of the existential to x, and continues with a computation c.

Missing constructors. Notice that the language does not have first-class applications: their role is played by the applicative let forms, binding the result of a *fully applied* function to a variable. Also notice that the language does not have a type application (i.e. the eliminator of $\forall$) and dually, it does not have pack (i.e. the constructor of $\exists$). This is because the instantiation of polymorphic and existential types is inferred by the algorithm. In Section 6, we discuss the way to modify the system to introduce *explicit* type applications.

2.4 The Key Ideas of the Algorithm

The inference algorithm for $\mathrm{F}^{\pm}_{\exists}$ is *local*: it has a limited scope of the inference information and does not propagate it between far-apart branches of the syntax tree. Nevertheless, many difficulties appear in the inference of the combination of polymorphic and existential types. We now discuss the most challenging of these difficulties and how they are addressed in the algorithm.

Subtyping. The inference algorithm's complexity mainly lies in the subtyping relation, which is, in particular, responsible for polymorphic instantiation. It's this subtyping relation that enables us to apply polymorphic terms in situations where their specific instantiations are anticipated.

To check whether one type is a subtype of another, the algorithm introduces *algorithmic* type variables ($\widehat{\alpha}^{\pm}, \widehat{\beta}^{\pm}, \dots$)—the 'placeholders' representing the polymorphic variables whose instantiation is postponed. This way, the problem of $\forall\alpha^{+}.\uparrow\alpha^{+} \leqslant \uparrow\mathtt{Int}$ becomes the problem of finding $\widehat{\alpha}^{+}$ such that $\uparrow\widehat{\alpha}^{+} \leqslant \uparrow\mathtt{Int}$.

Equivalence. The system introduces another complexity: the equivalence induced by subtyping ($\Theta \vdash N \simeq^{\leqslant} M \overset{\text{def}}{\iff} \Theta \vdash N \leqslant M$ and $\Theta \vdash M \leqslant N$) is not straightforward. This relation extends beyond just alpha-equivalent types. As the neighboring polymorphic quantifiers can be instantiated in arbitrary order. the equivalence relation must permit the *permutations* of the quantifiers. For example, we have $\cdot \vdash \forall\alpha^{+}.\forall\beta^{+}.N \simeq^{\leqslant} \forall\beta^{+}.\forall\alpha^{+}.N$. Similarly, the equivalence permits the *removal/addition* of the unused quantifiers: $\cdot \vdash \forall\alpha^{+}.\uparrow\mathtt{Int} \simeq^{\leqslant} \uparrow\mathtt{Int}$.

To handle the complexity of non-trivial equivalence, we employ a type normalization algorithm. This approach simplifies mutual subtyping to a more manageable alpha-equivalence by eliminating unused quantifiers and reordering the rest into a canonical permutation, effectively replacing subtyping-induced equivalence on terms with alpha-equivalence on their normal forms.

Least Upper Bound. Eventually, the subtyping algorithm reduces the initial subtyping problem to a set of constraints that need to be resolved. At this moment, another problem arises: as one variable can occur in multiple constraints, the resolution becomes non-trivial. In particular, to resolve the conjunction of $\widehat{\alpha}^{+} \geqslant P$ and $\widehat{\alpha}^{+} \geqslant Q$, we would need to find the least upper bound of P and Q.

In many type systems, the least upper bound boils down to the trivial syntactic equality. Because of that, the set of constraints is typically resolved using *unification*. However, this problem is trickier in the presence of existential types. For example, in $\mathrm{F}^{\pm}_{\exists}$, the least upper bound of $\downarrow\uparrow\mathtt{Int}$ and $\downarrow\uparrow\mathtt{Bool}$ is $\exists\alpha^{-}.\downarrow\alpha^{-}$, because the quantified α^{-} can be instantiated to either $\uparrow\mathtt{Int}$ or $\uparrow\mathtt{Bool}$.

To find this least upper bound, we use *anti-unification*—the dual of unification. While unification finds *the most general instance* of two given patterns, the anti-unification finds *the most detailed* (under the restrictions of the system) *pattern* that matches two given types. In the example above, the anti-unifier of $\downarrow\uparrow\mathtt{Int}$ and $\downarrow\uparrow\mathtt{Bool}$ is $\downarrow\widehat{\alpha}^{-}$, and in $\mathrm{F}^{\pm}_{\exists}$, this unifier is the most specific (in particular because $\downarrow\uparrow\widehat{\alpha}^{+}$ is not considered as a candidate since only *negative* variables are existentially quantified).

Impredicativity. Another difficulty that hinders the constraint resolution arises from the impredicativity of the system—the ability to quantify over the types that are themselves polymorphic. In the impredicative polymorphic system (even with only $\forall$-quantifiers), the naturally formulated subtyping is undecidable, which follows from the undecidability of second-order unification. All the more the undecidability is evident in the presence of existential types and subtyping constraints. For instance, in $\mathrm{F}^{\pm}_{\exists}$ the constraint $\widehat{\alpha}^{+} :\geqslant \downarrow\uparrow\widehat{\alpha}^{+}$ would have a surprising impredicative solution $\widehat{\alpha}^{+} = \exists\alpha^{-}.\downarrow\alpha^{-}$.

To maintain decidability in the presence of impredicativity, we carefully restrict the system using the polarity stratification. One of the main constraints that we put on subtyping is the invariance of the shift operators: the subtyping $\uparrow P \leqslant \uparrow Q$ is only allowed if $Q \geqslant P$ *and* $P \geqslant Q$. These constraints do not trivialize the system: we still have non-trivial upper bounds, and thus, need to use anti-unification to find them. However, it prevents such examples as $\widehat{\alpha}^{+} :\geqslant \downarrow\uparrow\widehat{\alpha}^{+}$, and as we prove, makes the constraint resolution decidable.

$\boxed{\Theta \vdash N \leqslant M}$ Negative subtyping

$$\frac{}{\Theta \vdash \alpha^- \leqslant \alpha^-}\ (\textsc{Var}^{\leqslant}_{-}) \qquad \frac{\Theta \vdash P \simeq^{\leqslant} Q}{\Theta \vdash \uparrow P \leqslant \uparrow Q}\ (\uparrow^{\leqslant})$$

$$\frac{\Theta \vdash P \geqslant Q \quad \Theta \vdash N \leqslant M}{\Theta \vdash P \to N \leqslant Q \to M}\ (\to^{\leqslant}) \qquad \frac{\Theta, \overrightarrow{\beta^-} \vdash \sigma : \overrightarrow{\alpha^-} \quad \Theta, \overrightarrow{\beta^-} \vdash [\sigma]P \geqslant Q}{\Theta \vdash \exists \overrightarrow{\alpha^-}.\, P \geqslant \exists \overrightarrow{\beta^-}.\, Q}\ (\exists^{\geqslant})$$

$$\frac{\Theta, \overrightarrow{\beta^+} \vdash \sigma : \overrightarrow{\alpha^+} \quad \Theta, \overrightarrow{\beta^+} \vdash [\sigma]N \leqslant M}{\Theta \vdash \forall \overrightarrow{\alpha^+}.\, N \leqslant \forall \overrightarrow{\beta^+}.\, M}\ (\forall^{\leqslant})$$

$\boxed{\Theta \vdash N \simeq^{\leqslant} M}$ Negative equivalence

$$\frac{\Theta \vdash N \leqslant M \quad \Theta \vdash M \leqslant N}{\Theta \vdash N \simeq^{\leqslant} M}\ (\simeq^{\leqslant}_{-})$$

$\boxed{\Theta \vdash P \geqslant Q}$ Positive supertyping

$$\frac{}{\Theta \vdash \alpha^+ \geqslant \alpha^+}\ (\textsc{Var}^{\geqslant}_{+}) \qquad \frac{\Theta \vdash N \simeq^{\leqslant} M}{\Theta \vdash \downarrow N \geqslant \downarrow M}\ (\downarrow^{\geqslant})$$

$\boxed{\Theta \vdash P \simeq^{\leqslant} Q}$ Positive equivalence

$$\frac{\Theta \vdash P \geqslant Q \quad \Theta \vdash Q \geqslant P}{\Theta \vdash P \simeq^{\leqslant} Q}\ (\simeq^{\leqslant}_{+})$$

Fig. 4. Declarative Subtyping

3 DECLARATIVE SYSTEM

In this section, we present the declarative system of $F^{\pm}_{\exists}$, which serves as a specification of the type inference algorithm. The declarative system is represented as a set of inference rules and consists of two main subsystems: subtyping and type inference. First, we present the declarative subtyping rules specifying when one type is a subtype of another; next, we discuss the equivalence relation induced by mutual subtyping; finally, we present the type inference rules, that refer to the subtyping and equivalence relation. We conclude this section by discussing the relation between the proposed type system and the standard System F.

3.1 Subtyping

The inference rules representing declarative subtyping are shown in Fig. 4. Let us discuss them in more detail.

Quantifiers. Symmetric rules $(\forall^{\leqslant})$ and $(\exists^{\geqslant})$ specify subtyping between top-level quantified types. Usually, the polymorphic subtyping is represented by two rules introducing quantifiers to the left and to the right-hand side of subtyping. For conciseness of representation, we compose these rules into one. First, our rule extends context Θ with the quantified variables from the right-hand side ($\overrightarrow{\beta^+}$ or $\overrightarrow{\beta^-}$), as these variables must remain abstract. Second, it verifies that the left-hand side quantifiers ($\overrightarrow{\alpha^+}$ or $\overrightarrow{\alpha^-}$) can be instantiated to continue subtyping recursively. This is the point of non-determinism as the instantiation can be chosen arbitrarily

The instantiation of quantifiers is modeled by substitution σ. The notation $\Theta_2 \vdash \sigma : \Theta_1$ specifies its domain and range. For instance, $\Theta, \overrightarrow{\beta^+} \vdash \sigma : \overrightarrow{\alpha^+}$ means that σ maps the variables from $\overrightarrow{\alpha^+}$ to (positive) types well-formed in $\Theta, \overrightarrow{\beta^+}$. This way, application $[\sigma]N$ instantiates (replaces) every α_i^- in N with $\sigma(\alpha_i^-)$.

Invariant Shifts. As mentioned above, an important restriction that we put on the subtyping system is that subtyping of shifted types requires their equivalence, as shown in $(\downarrow^{\geqslant})$ and $(\uparrow^{\leqslant})$. The reason for this is that if both of these rules were relaxed to the covariant form, the subtyping relation

would become equivalent to the standard subtyping of System F, which is undecidable [Tiuryn et al. 1996]. However, it is suggested after certain changes ($\uparrow^{\leqslant}$) can be relaxed to the covariant form, thereby increasing the expressiveness of the system. These changes are discussed in Section 6.2.

Functions. Standardly, subtyping of function types is covariant in the return type and contravariant in the argument type.

Variables. Subtyping of variables is defined reflexively, which is enough to ensure the reflexivity of subtyping in general. The algorithm—specifically the least upper bound procedure—will use the fact that the subtypes of a variable coincide with its supertypes (Property 2), which however does not hold for arbitrary types.

3.1.1 Properties of Declarative Subtyping. A property that is important for the subtyping algorithm, in particular for the type *upgrade* procedure (Section 4.6), is the preservation of free variables by subtyping. Informally, it says that the free variables of a positive type cannot disappear in its subtypes, and the free variables of a negative type cannot disappear in its supertypes.

Property 1 (Subtyping preserves free variables)**.** *Let us assume that all the mentioned types are well-formed in* Θ*. Then* $\Theta \vdash N_1 \leqslant N_2$ *implies* $\mathtt{fv}(N_1) \subseteq \mathtt{fv}(N_2)$*, and* $\Theta \vdash P_1 \geqslant P_2$ *implies* $\mathtt{fv}(P_1) \subseteq \mathtt{fv}(P_2)$*.*

Property 2 (Variable subtyping is trivial)**.** *A subtype or a supertype of a variable is equivalent to the variable itself:*

$$- \quad \Theta \vdash \alpha^- \leqslant N \iff \Theta \vdash N \leqslant \alpha^- \iff N = \forall \overrightarrow{\beta^+}.\, \alpha^-$$
$$+ \quad \Theta \vdash \alpha^+ \geqslant P \iff \Theta \vdash P \geqslant \alpha^+ \iff P = \exists \overrightarrow{\beta^-}.\, \alpha^+$$

Another property that we extensively use is that subtyping is reflexive and transitive, and agrees with substitution.

Property 3 (Subtyping forms a preorder)**.** *For a fixed context* Θ*, the negative subtyping relation* $\Theta \vdash N_1 \leqslant N_2$ *and the positive subtyping relation* $\Theta \vdash P_1 \geqslant P_2$ *are reflexive and transitive on types well-formed in* Θ*.*

Property 4 (Subtyping agrees with substitution)**.** *Suppose that* σ *is a substitution such that* $\Theta_2 \vdash \sigma : \Theta_1$*. Then*

- $\Theta_1 \vdash N \leqslant M$ *implies* $\Theta_2 \vdash [\sigma]N \leqslant [\sigma]M$*, and*
- $\Theta_1 \vdash P \geqslant Q$ *implies* $\Theta_2 \vdash [\sigma]P \geqslant [\sigma]Q$*.*

Moreover, any two *positive* types have a least upper bound, which makes positive subtyping a semilattice. The positive least upper bound can be found algorithmically, which we will discuss in the next section.

Property 5 (Positive least upper bound exists)**.** *Suppose that* P_1 *and* P_2 *are positive types well-formed in* Θ*. Then there exists a least common supertype— a type* P *such that*

- $\Theta \vdash P \geqslant P_1$ *and* $\Theta \vdash P \geqslant P_2$*, and*
- *for any* Q *such that* $\Theta \vdash Q \geqslant P_1$ *and* $\Theta \vdash Q \geqslant P_2$*,* $\Theta \vdash Q \geqslant P$*.*

Negative greatest lower bound might not exist. The symmetric construction—the greatest lower bound of two negative types—does not always exist, as the following counterexample shows. Consider the following types:

- N and Q are arbitrary closed types,
- P, P_1, and P_2 are non-equivalent closed types such that $\cdot \vdash P_1 \geqslant P$ and $\cdot \vdash P_2 \geqslant P$, and none of the types is equivalent to Q.

What is the greatest common subtype of $Q \to \downarrow\uparrow Q \to \downarrow\uparrow Q \to N$ and $P \to \downarrow\uparrow P_1 \to \downarrow\uparrow P_2 \to N$? The type $\forall \alpha^+, \beta^+, \gamma^+.\ \alpha^+ \to \downarrow\uparrow \beta^+ \to \downarrow\uparrow \gamma^+ \to N$ is a common subtype, however, it is not the greatest one, as it is too general.

One can find two greater candidates: $M_1 = \forall \alpha^+, \beta^+.\ \alpha^+ \to \downarrow\uparrow \alpha^+ \to \downarrow\uparrow \beta^+ \to N$ and $M_2 = \forall \alpha^+, \beta^+.\ \beta^+ \to \downarrow\uparrow \alpha^+ \to \downarrow\uparrow \beta^+ \to N$. Instantiating α^+ and β^+ with Q ensures that both of these types are subtypes of $Q \to \downarrow\uparrow Q \to \downarrow\uparrow Q \to N$; instantiating α^+ with P_1 and β^+ with P_2 demonstrates the subtyping with $P \to \downarrow\uparrow P_1 \to \downarrow\uparrow P_2 \to N$, as P is a subtype of both P_1 and P_2.

By analyzing the inference rules, we can prove that both M_1 and M_2 are *maximal* common subtypes, i.e., there is no common subtype that is greater than them. However, M_1 and M_2 are not equivalent, which means that none of them is the greatest.

3.2 Equivalence and Normalization

The subtyping relation forms a preorder on types, and thus, it induces an equivalence relation also known as bicoercibility [Tiuryn 1995]. The declarative specification of subtyping must be defined up to this equivalence. Moreover, the algorithms we use must withstand changes in input types within the equivalence class. To deal with non-trivial equivalence, we use normalization—a function that uniformly selects a representative of the equivalence class.

Using normalization gives us two benefits: (i) we do not need to modify significantly standard operations such as unification to withstand non-trivial equivalence, and (ii) if the subtyping (and thus, the equivalence) changes, we only need to modify the normalization function, while the rest of the algorithm remains the same.

In our system, equivalence is richer than equality. Specifically, while staying within one equivalence class, one can change the type:

(i) introduce and remove redundant quantifiers. For example, $\forall \alpha^+, \beta^+.\ \uparrow \alpha^+$ is equivalent but not equal to $\forall \alpha^+.\ \uparrow \alpha^+$;
(ii) reorder adjacent quantifiers. For example, $\forall \alpha^+, \beta^+.\ \alpha^+ \to \beta^+ \to \gamma^-$ is equivalent but not equal to $\forall \alpha^+, \beta^+.\ \beta^+ \to \alpha^+ \to \gamma^-$;
(iii) make the transformations (i) and (ii) at any position in the type.

It turns out that the transformations (i-iii) are complete, in the sense that they generate the whole equivalence class. This way, to normalize the type, one must

(i) remove the redundant quantifiers,
(ii) reorder the quantifiers to the canonical order,
(iii) do the procedures (i) and (ii) recursively on the subterms.

The normalization algorithm is shown in Fig. 5. The steps (i-ii) are implemented by the *ordering* function '$\texttt{ord}\ \mathit{vars}\ \texttt{in}\ N$' and '$\texttt{ord}\ \mathit{vars}\ \texttt{in}\ P$'. For a set of variables *vars*, and a type, it returns a list of variables from *vars* that occur in the type in the *order of their first occurrence*. For brevity, we omit the formal definition of ordering referring the reader to the appendix (**??**).

For the normalization procedure, we prove soundness and completeness w.r.t. the equivalence relation.

Property 6 (Correctness of normalization)**.** *Assuming all types are well-formed in* Θ*,*

$-$ $\Theta \vdash N \simeq^{\leqslant} M$ *is equivalent to* $\texttt{nf}\ (N) = \texttt{nf}\ (M)$*, and*
$+$ $\Theta \vdash P \simeq^{\leqslant} Q$ *is equivalent to* $\texttt{nf}\ (P) = \texttt{nf}\ (Q)$.

3.3 Typing

The declarative specification of the type inference is given in Fig. 6. The positive typing judgment $\Theta; \Gamma \vdash v\colon P$ is read as "under the type context Θ and variable context Γ, the term v is allowed to

$$\boxed{\mathtt{nf}\ (N) = M} \qquad\qquad \boxed{\mathtt{nf}\ (P) = Q}$$

$$\frac{}{\mathtt{nf}\ (\alpha^-) = \alpha^-}\ (\textsc{Var}^{\textsc{nf}}_-) \qquad \frac{}{\mathtt{nf}\ (\alpha^+) = \alpha^+}\ (\textsc{Var}^{\textsc{nf}}_+)$$

$$\frac{\mathtt{nf}\ (P) = Q}{\mathtt{nf}\ (\uparrow P) = \uparrow Q}\ (\uparrow^{\textsc{nf}}) \qquad \frac{\mathtt{nf}\ (N) = M}{\mathtt{nf}\ (\downarrow N) = \downarrow M}\ (\downarrow^{\textsc{nf}})$$

$$\frac{\mathtt{nf}\ (P) = Q \quad \mathtt{nf}\ (N) = M}{\mathtt{nf}\ (P \to N) = Q \to M}\ (\to^{\textsc{nf}})$$

$$\frac{\mathtt{nf}\ (N) = N' \quad \mathtt{ord}\ \overrightarrow{\alpha^+}\ \mathtt{in}\ N' = \overrightarrow{\alpha^{+\prime}}}{\mathtt{nf}\ (\forall \overrightarrow{\alpha^+}.\ N) = \forall \overrightarrow{\alpha^{+\prime}}.\ N'}\ (\forall^{\textsc{nf}}) \qquad \frac{\mathtt{nf}\ (P) = P' \quad \mathtt{ord}\ \overrightarrow{\alpha^-}\ \mathtt{in}\ P' = \overrightarrow{\alpha^{-\prime}}}{\mathtt{nf}\ (\exists \overrightarrow{\alpha^-}.\ P) = \exists \overrightarrow{\alpha^-}.\ P'}\ (\exists^{\textsc{nf}})$$

$\mathtt{ord}\ \mathit{vars}\ \mathtt{in}\ N$ returns a list of variables from $\mathit{vars} \cap \mathtt{fv}\ (N)$ in the order of their first occurrence in N

$\mathtt{ord}\ \mathit{vars}\ \mathtt{in}\ P$ returns a list of variables from $\mathit{vars} \cap \mathtt{fv}\ (P)$ in the order of their first occurrence in P

Fig. 5. Type Normalization Procedure

infer type P", where Γ—the variable context—is defined standardly as a set of pairs of the form $x : P$. The negative typing judgment is read similarly.

The *Application typing* judgment infers the type of the application of a function to a list of arguments. It has form $\Theta\,;\Gamma \vdash N \bullet \vec{v} \Rrightarrow M$, which reads "under the type context Θ and variable context Γ, the application of a function of type N to the list of arguments $\vec{v}$ is allowed to infer type M".

Let us discuss the rules of the declarative system in more detail.

Variables Rule ($\textsc{Var}^{\textsc{inf}}$) allows us to infer the type of a variable from the context. In literature, one can find another version of this rule, that enables inferring a type *equivalent* to the type from the context. In our case, the inference of equivalent types is admissible by ($\simeq^{\textsc{inf}}_+$).

Annotations The annotation rules ($\textsc{ann}^{\textsc{inf}}_-$) and ($\textsc{ann}^{\textsc{inf}}_+$) use subtyping. The annotation is only valid if the inferred type is a subtype of the annotation type.

Abstractions The typing of lambda abstraction is standard. Rule ($\lambda^{\textsc{inf}}$) first checks that the given type annotating the argument is well-formed, and then infers the type of the body in the extended context. As a result, it returns an arrow type of function from the annotated type of the argument to the type of the body. Rule ($\Lambda^{\textsc{inf}}$) infers polymorphic $\forall$-type. It extends the type context with the quantifying variable α^+ and infers the type of the body. As a result, it returns a polymorphic type quantifying the abstracted variable α^+ over the type of the body.

Return and thunk Rules ($\textsc{ret}^{\textsc{inf}}$) and ($\{\}^{\textsc{inf}}$) simply add the corresponding shift constructors to the type of the body.

Unpack Rule ($\textsc{let}^{\textsc{inf}}_\exists$) types elimination of $\exists$. First, it infers the normalized type of the existential package. The normalization is required to fix the order of the quantifying variables to bind them. Alternative approaches that do not require normalization will be discussed in Section 6.1. After the bind, the rule infers the type of the body and checks that it does not use the bound variables so that they do not escape the scope.

Let binders Rule ($\textsc{let}^{\textsc{inf}}$) is a standard rule for typing let binders: we infer the type of the bound value and continue the typing of the computation in the extended context.

$\boxed{\Theta;\Gamma \vdash c : N}$ Negative typing

$$\frac{\Theta \vdash P \quad \Theta;\Gamma, x : P \vdash c : N}{\Theta;\Gamma \vdash \lambda x : P.\, c : P \to N}\ (\lambda^{\text{INF}}) \qquad \frac{\Theta;\Gamma \vdash v : \downarrow M \quad \Theta ; \Gamma \vdash M \bullet \vec{v} \Longrightarrow \uparrow Q \text{ principal} \quad \Theta;\Gamma, x : Q \vdash c : N}{\Theta;\Gamma \vdash \mathtt{let}\ x = v(\vec{v});\ c : N}\ (\text{LET}^{\text{INF}}_{@})$$

$$\frac{\Theta, \alpha^{+};\Gamma \vdash c : N}{\Theta;\Gamma \vdash \Lambda\alpha^{+}.\, c : \forall\alpha^{+}.\, N}\ (\Lambda^{\text{INF}}) \qquad \frac{\Theta \vdash P \quad \Theta;\Gamma \vdash v : \downarrow M \quad \Theta ; \Gamma \vdash M \bullet \vec{v} \Longrightarrow M' \quad \Theta \vdash M' \leqslant \uparrow P \quad \Theta;\Gamma, x : P \vdash c : N}{\Theta;\Gamma \vdash \mathtt{let}\ x : P = v(\vec{v});\ c : N}\ (\text{LET}^{\text{INF}}_{:@})$$

$$\frac{\Theta;\Gamma \vdash v : P}{\Theta;\Gamma \vdash \mathtt{return}\ v : \uparrow P}\ (\text{RET}^{\text{INF}}) \qquad \frac{\Theta;\Gamma \vdash v : \exists\overrightarrow{\alpha^{-}}.\, P \quad \mathsf{nf}\,(\exists\overrightarrow{\alpha^{-}}.\, P) = \exists\overrightarrow{\alpha^{-}}.\, P \quad \Theta, \overrightarrow{\alpha^{-}};\Gamma, x : P \vdash c : N \quad \Theta \vdash N}{\Theta;\Gamma \vdash \mathtt{let}^{\exists}(\overrightarrow{\alpha^{-}}, x) = v;\ c : N}\ (\text{LET}^{\text{INF}}_{\exists})$$

$$\frac{\Theta;\Gamma \vdash v : P \quad \Theta;\Gamma, x : P \vdash c : N}{\Theta;\Gamma \vdash \mathtt{let}\ x = v;\ c : N}\ (\text{LET}^{\text{INF}}) \qquad \frac{\Theta \vdash P \quad \Theta;\Gamma \vdash c : M \quad \Theta \vdash M \leqslant \uparrow P \quad \Theta;\Gamma, x : P \vdash c' : N}{\Theta;\Gamma \vdash \mathtt{let}\ x : P = c;\ c' : N}\ (\text{LET}^{\text{INF}}_{\text{C}})$$

$$\frac{\Theta;\Gamma \vdash c : N \quad \Theta \vdash N \simeq^{\leqslant} N'}{\Theta;\Gamma \vdash c : N'}\ (\simeq^{\text{INF}}_{-}) \qquad \frac{\Theta \vdash M \quad \Theta;\Gamma \vdash c : N \quad \Theta \vdash N \leqslant M}{\Theta;\Gamma \vdash (c : M) : M}\ (\text{ANN}^{\text{INF}}_{-})$$

$\boxed{\Theta;\Gamma \vdash v : P}$ Positive typing

$$\frac{x : P \in \Gamma}{\Theta;\Gamma \vdash x : P}\ (\text{VAR}^{\text{INF}}) \qquad \frac{\Theta \vdash Q \quad \Theta;\Gamma \vdash v : P \quad \Theta \vdash Q \geqslant P}{\Theta;\Gamma \vdash (v : Q) : Q}\ (\text{ANN}^{\text{INF}}_{+})$$

$$\frac{\Theta;\Gamma \vdash c : N}{\Theta;\Gamma \vdash \{c\} : \downarrow N}\ (\{\}^{\text{INF}}) \qquad \frac{\Theta;\Gamma \vdash v : P \quad \Theta \vdash P \simeq^{\leqslant} P'}{\Theta;\Gamma \vdash v : P'}\ (\simeq^{\text{INF}}_{+})$$

$\boxed{\Theta ; \Gamma \vdash N \bullet \vec{v} \Longrightarrow M}$ Application typing

$$\frac{\Theta \vdash N \simeq^{\leqslant} N'}{\Theta ; \Gamma \vdash N \bullet \cdot \Longrightarrow N'}\ (\emptyset^{\text{INF}}_{\bullet\Rightarrow}) \qquad \frac{\vec{v} \neq \quad \overrightarrow{\alpha^{+}} \neq \cdot \quad \Theta \vdash \sigma : \overrightarrow{\alpha^{+}} \quad \Theta ; \Gamma \vdash [\sigma]N \bullet \vec{v} \Longrightarrow M}{\Theta ; \Gamma \vdash \forall\overrightarrow{\alpha^{+}}.\, N \bullet \vec{v} \Longrightarrow M}\ (\forall^{\text{INF}}_{\bullet\Rightarrow})$$

$$\frac{\Theta;\Gamma \vdash v : P \quad \Theta \vdash Q \geqslant P \quad \Theta ; \Gamma \vdash N \bullet \vec{v} \Longrightarrow M}{\Theta ; \Gamma \vdash Q \to N \bullet v, \vec{v} \Longrightarrow M}\ (\to^{\text{INF}}_{\bullet\Rightarrow})$$

Fig. 6. Declarative Inference

Rule ($\text{LET}^{\text{INF}}_{\text{C}}$) associates a variable with a computation. The inferred type for this computation (M) must be castable to a shifted positive type $\uparrow P$, with P then assigned to the bound variable x to type the let binder's body. Like all annotated constructors, we also verify the annotation type P's well-formedness.

Applicative let binders Rules ($\text{LET}^{\text{INF}}_{@}$) and ($\text{LET}^{\text{INF}}_{:@}$) infer the type of the applicative let binders. Both of them infer the type of the head v and invoke the application typing to infer the type of the application before recursing on the body of the let binder.

The former rule infers the type of an *unannotated* let binder, and thus it requires the resulting type of application to be *the principal type*, so that the type we assign to the bound variable x is determined. In this context, principality means minimality. In other words, $\Theta ; \Gamma \vdash M \bullet \vec{v} \Longrightarrow \uparrow Q$ principal

means that any other type Q' inferrable for the application (i.e., $\Theta\,;\Gamma \vdash M \bullet \vec{v} \Longrightarrow \uparrow Q'$) is greater than the principal type Q, i.e., $\Theta \vdash Q' \geqslant Q$.

The latter rule is for the *annotated* binder, and thus, the type of the bound x is given, however, the rule must check that this type is a a supertype of the inferred type of the application. This check is done by invoking the subtyping judgment $\Theta \vdash M' \leqslant \uparrow P$.

Typing up to equivalence. As discussed in Section 3.2, the subtyping, as a preorder, induces a non-trivial equivalence relation on types. The system must not distinguish between equivalent types, and thus, type inference must be defined up to equivalence. For this purpose, we use rules $(\simeq^{\text{INF}}_{+})$ and $(\simeq^{\text{INF}}_{-})$. They allow one to replace the inferred type with an equivalent one.

Application to an empty list of arguments. The base case of the application type inference is represented by rule $(\emptyset^{\text{INF}}_{\bullet\Longrightarrow})$. If the head of the type N is applied to no arguments, the type of the result is allowed to be N or any equivalent type. We need to relax this rule up to equivalence to ensure the corresponding property globally: the inferred application type can be replaced with an equivalent one. Alternatively, we could have added a separate rule similar to $(\simeq^{\text{INF}}_{+})$, however, the local relaxation is sufficient to prove the global property.

Application of a polymorphic type $\forall$. The complexity of the system lives in the rules whose output type is not immediately defined by their input and the output of their premises (also known as not mode-correct [Dunfield et al. 2020]). In our typing system, $(\forall^{\text{INF}}_{\bullet\Longrightarrow})$ is such a rule: the instantiation of the quantifying variables is not known a priori. The algorithm we present in Section 4 delays this instantiation until more information about it (in particular, typing constraints) is collected.

To ensure the priority of application between this rule and $(\emptyset^{\text{INF}}_{\bullet\Longrightarrow})$, we also check that the list of arguments is not empty.

Application of an arrow type. Another application rule is $(\rightarrow^{\text{INF}}_{\bullet\Longrightarrow})$. This is where the subtyping is used to check that the type of the argument is convertible to (a subtype of) the type of the function parameter. In the algorithm (Section 4), this subtyping check will provide the constraints we need to resolve the delayed instantiations of the quantifying variables.

An important property that the declarative system has is that the declarative specification is correctly defined for equivalence classes.

Property 7 (Declarative typing is defined up to equivalence)**.** *Let us assume that* $\Theta \vdash \Gamma_1 \simeq^{\leqslant} \Gamma_2$, *i.e., the corresponding types assigned by* Γ_1 *and* Γ_2 *are equivalent in* Θ. *Also, let us assume that* $\Theta \vdash N_1 \simeq^{\leqslant} N_2$, $\Theta \vdash P_1 \simeq^{\leqslant} P_2$, *and* $\Theta \vdash M_1 \simeq^{\leqslant} M_2$. *Then*

- $-$ $\Theta;\Gamma_1 \vdash c\colon N_1$ *holds if and only if* $\Theta;\Gamma_2 \vdash c\colon N_2$,
- $+$ $\Theta;\Gamma_1 \vdash v\colon P_1$ *holds if and only if* $\Theta;\Gamma_2 \vdash v\colon P_2$, *and*
- $\bullet$ $\Theta\,;\Gamma_1 \vdash N_1 \bullet \vec{v} \Longrightarrow M_1$ *holds if and only if* $\Theta\,;\Gamma_2 \vdash N_2 \bullet \vec{v} \Longrightarrow M_2$.

3.4 Relation to System F

To establish a correspondence between $\mathrm{F}^{\pm}_{\exists}$ and standard unpolarized System F, we present a translation in both ways: the polarization from System F to $\mathrm{F}^{\pm}_{\exists}$ and the depolarization from $\mathrm{F}^{\pm}_{\exists}$ to System F.

Type-level Translation. The type depolarization (Fig. 7) simply forgets the polarization structure of types: it removes the shift operators and the polarities of the free type variables.

The type polarization (Fig. 8) is more involved, as there are multiple ways to polarize a type: for instance, a variable α can be polarized to α^{+} or α^{-}. The choice of polarization affects the execution strategy of the program. In particular, a functional type can be represented in either positive (thunked, call-by-name) way: $\downarrow(P \to \uparrow Q)$ or negative (call-by-value): $\downarrow N \to M$. We chose

$\boxed{|P|}$

$$|\alpha^+| \equiv \alpha$$
$$|\downarrow N| \equiv |N|$$
$$|\exists \overrightarrow{\alpha^-}.\, P| \equiv \exists \overrightarrow{\alpha}.\, |P|$$

$\boxed{|N|}$

$$|\alpha^-| \equiv \alpha$$
$$|\uparrow P| \equiv |P|$$
$$|\forall \alpha^+.\, N| \equiv \forall \alpha.\, |N|$$
$$|P \to N| \equiv |P| \to |N|$$

Fig. 7. Type Depolarization [1]

$\boxed{\lfloor T \rfloor}$

$$\lfloor \alpha \rfloor \equiv \alpha^+$$
$$\lfloor \forall \alpha.\, T \rfloor \equiv \downarrow(\forall \alpha^+.\, \uparrow \lfloor T \rfloor)$$
$$\lfloor A \to B \rfloor \equiv \downarrow(\lfloor A \rfloor \to \uparrow \lfloor B \rfloor)$$

Fig. 8. Type Polarization

the positive polarization: every System F type is translated to a *positive* type of $\mathrm{F}^{\pm}_{\exists}$. This choice makes it smoother to lift the translation to the term level, which will be discussed next.

The type-level translation is naturally lifted to the contexts: $|\Theta|$ is the context Θ with every type depolarized, and symmetrically, $\lfloor \Theta \rfloor$ is the context Θ with every type polarized. The context translation is used to formulate the soundness of the term-level translation, which we discuss next.

Term-level Translation. The term-level translations between the systems are more complex. They are defined as *elaborations* of the typing derivations. In other words, the input of the translation is not a term, but a whole typing derivation in the corresponding system. For example, the subtyping elaboration $\Theta \vdash N \leqslant M \rightsquigarrow t$ constructs a System F function t of type $|N| \to |M|$ that witnesses the given $\mathrm{F}^{\pm}_{\exists}$-subtyping $\Theta \vdash N \leqslant M$; a typing elaboration $\Theta; \Gamma \vdash v\colon P \rightsquigarrow t$ builds a term t—a System F counterpart of v—of type $|P|$.

For brevity, we omit the formal definition of the term-level elaboration, referring the reader to the appendix (**??**). We define the elaboration in such a way that it preserves the soundness invariants given in Table 1.

Elaboration Judgment		The Soundness Property
Negative Subtyping	$\Theta \vdash N \leqslant M \rightsquigarrow t$	$\lvert\Theta\rvert; \cdot \vdash t\colon \lvert N\rvert \to \lvert M\rvert$
Positive Subtyping	$\Theta \vdash P \geqslant Q \rightsquigarrow t$	$\lvert\Theta\rvert; \cdot \vdash t\colon \lvert Q\rvert \to \lvert P\rvert$
Negative Typing	$\Theta; \Gamma \vdash c\colon N \rightsquigarrow t$	$\lvert\Theta\rvert; \lvert\Gamma\rvert \vdash t\colon \lvert N\rvert$
Positive Typing	$\Theta; \Gamma \vdash v\colon P \rightsquigarrow t$	$\lvert\Theta\rvert; \lvert\Gamma\rvert \vdash t\colon \lvert P\rvert$
Application Typing	$\Theta; \Gamma \vdash N \bullet \vec{v} \Rrightarrow M \rightsquigarrow e; \vec{t}$	$\lvert\Theta\rvert; \lvert\Gamma\rvert, x : \lvert N\rvert \vdash e\,(x\,\vec{t})\colon \lvert M\rvert$
System F Typing	$\Theta; \Gamma \vdash t\colon T \rightsquigarrow^{\pm} c$	$\lfloor\Theta\rfloor; \lfloor\Gamma\rfloor \vdash c\colon \uparrow\lfloor T\rfloor$

Table 1. Soundness of the Elaboration: the elaboration judgment on the left implies the typing judgment on the right

The least obvious aspect of the elaboration soundness is the application typing. The elaboration judgment $\Theta; \Gamma \vdash N \bullet \vec{v} \Rrightarrow M \rightsquigarrow e; \vec{t}$ outputs two things: (i) $\vec{t}$—the System F counterpart of the argument list $\vec{v}$, and (ii) e—a System F term that is used for the final cast of the application $(((x\,t_1)\cdots)\,t_n)$ to the resulting type $|M|$. This final cast is needed to reflect the $\mathrm{F}^{\pm}_{\exists}$ feature of $(\emptyset^{\mathrm{INF}}_{\bullet\Rightarrow})$ that permits the equivalent conversion of the resulting type $\Theta \vdash N \simeq^{\leqslant} N'$, as $\mathrm{F}^{\pm}_{\exists}$-equivalent types might have different System F representations (i.e., $|N| \neq |N'|$).

4 THE ALGORITHM

In this section, we present the algorithmization of the declarative system described above. The algorithmic system follows the structure of the declarative specification closely. First, it is also given by a set of inference rules, which, however, are mode-correct ([Dunfield et al. 2020]), i.e., the

[1] Here the System F existential $\exists \vec{\alpha}.\, T$ is a syntax sugar for its standard encoding $\forall \beta.\, (\forall \vec{\alpha}.\, (T \to \beta)) \to \beta$

Negative Algorithmic Variables
$\widehat{\alpha}^-, \widehat{\beta}^-, \widehat{\gamma}^-, \ldots$

Positive Algorithmic Variables
$\widehat{\alpha}^+, \widehat{\beta}^+, \widehat{\gamma}^+, \ldots$

Negative Algorithmic Types
$N, M ::= \cdots \mid \widehat{\alpha}^-$

Positive Algorithmic Types
$P, Q ::= \cdots \mid \widehat{\alpha}^+$

Algorithmic Type Context
$\widehat{\Theta} ::= \{\widehat{\alpha}_1^{\pm}, \ldots, \widehat{\alpha}_n^{\pm}\}$ where $\widehat{\alpha}_1^{\pm}, \ldots, \widehat{\alpha}_n^{\pm}$ are pairwise distinct

Instantiation Context
$\Xi ::= \{\widehat{\alpha}_1^{\pm}\{\Theta_1\}, \ldots, \widehat{\alpha}_n^{\pm}\{\Theta_n\}\}$ where $\widehat{\alpha}_1^{\pm}, \ldots, \widehat{\alpha}_n^{\pm}$ are pairwise distinct

Fig. 9. Algorithmic Syntax

output of each rule is always uniquely defined by its input. And second, most of the declarative rules (except for the rules $(\simeq_+^{\text{INF}})$ and $(\simeq_-^{\text{INF}})$) have a unique algorithmic counterpart, which simplifies reasoning about the algorithm and its correctness proofs.

4.1 Algorithmic Syntax

First, let us discuss the syntax of the algorithmic system.

Algorithmic Variables. To design a mode-correct inference system, we slightly modify the language we operate on. The entities (terms, types, contexts) that the algorithm manipulates we call *algorithmic*. They extend the previously defined declarative terms and types by adding *algorithmic type variables* (also known as unification variables). The algorithmic variables represent unknown types, which cannot be inferred immediately but are promised to be instantiated as the algorithm proceeds.

We denote algorithmic variables as $\widehat{\alpha}^+, \widehat{\beta}^-, \ldots$ to distinguish them from normal variables α^+, β^-. In a few places, we replace the quantified variables $\overrightarrow{\alpha^+}$ with their algorithmic counterpart $\overrightarrow{\widehat{\alpha}^+}$. The procedure of replacing declarative variables with algorithmic ones we call *algorithmization* and denote as $\overrightarrow{\widehat{\alpha}^-}/\overrightarrow{\alpha^-}$ and $\overrightarrow{\widehat{\alpha}^+}/\overrightarrow{\alpha^+}$. The converse operation—*dealgorithmization*—is denoted as $\overrightarrow{\alpha^-}/\overrightarrow{\widehat{\alpha}^-}$ and is used in the least upper bound procedure (Section 4.6).

Algorithmic Types. The syntax of algorithmic types extends the declarative syntax by adding algorithmic variables as new terminals. We add positive algorithmic variables $\widehat{\alpha}^+$ to the syntax of positive types, and negative algorithmic variables $\widehat{\alpha}^-$ to the syntax of negative types. All the constructors of the system can be applied to *algorithmic* types, however, algorithmic variables cannot be abstracted by the quantifiers $\forall$ and $\exists$.

Algorithmic Contexts $\widehat{\Theta}$ and Well-formedness. To specify when algorithmic types are well-formed, we define algorithmic contexts $\widehat{\Theta}$ as sets of algorithmic variables. Then $\Theta\,;\widehat{\Theta} \vdash P$ and $\Theta\,;\widehat{\Theta} \vdash N$ represent the well-formedness judgment of algorithmic terms defined as expected. Informally, they check that all free declarative variables are in Θ, and all free algorithmic variables are in $\widehat{\Theta}$. Most of the rules are inherited from the well-formedness of *declarative* types: the declarative variables are checked to belong to the context Θ, the quantifiers extend the context, type constructors are well-formed congruently. As we extend the syntax with algorithmic variables, we also add two base-case rules for them: $(\text{UVAR}_+^{\text{WF}})$ and $(\text{UVAR}_-^{\text{WF}})$ (see Fig. 10).

$$\frac{\vdots}{\widehat{\alpha}^{+} \in \widehat{\Theta}} \quad (\textsc{UVar}_{+}^{\textsc{WF}}) \qquad \frac{\vdots}{\widehat{\alpha}^{-} \in \widehat{\Theta}} \quad (\textsc{UVar}_{-}^{\textsc{WF}})$$

Fig. 10. Well-formedness of Algorithmic Types

$$\frac{\vdots}{\mathsf{nf}\,(\widehat{\alpha}^{+}) = \widehat{\alpha}^{+}} \quad (\textsc{UVar}_{+}^{\textsc{NF}}) \qquad \frac{\vdots}{\mathsf{nf}\,(\widehat{\alpha}^{-}) = \widehat{\alpha}^{-}} \quad (\textsc{UVar}_{-}^{\textsc{NF}})$$

Fig. 11. Normalization of Algorithmic Types

Instantiation Context Ξ. When one instantiates an algorithmic variable, one may only use type variables *available in its scope*. As such, each algorithmic variable must remember the context at the moment when it was introduced. In our algorithm, this information is represented by an *instantiation context* Ξ—a set of pairs associating algorithmic variables and declarative contexts.

Algorithmic Substitution. We define the algorithmic substitution $\widehat{\sigma}$ as a mapping from algorithmic variables to *declarative* types. The signature $\Xi \vdash \widehat{\sigma} : \widehat{\Theta}$ specifies the domain and the range of $\widehat{\sigma}$: for each variable $\widehat{\alpha}^{\pm}$ in $\widehat{\Theta}$, there exists an corresponding entry in Ξ associating $\widehat{\alpha}^{\pm}$ with a declarative context Θ such that $[\widehat{\sigma}]\widehat{\alpha}^{\pm}$ is well-formed in Θ. In addition, we assume that $\widehat{\sigma}$ acts as the identity on the variables not in $\widehat{\Theta}$.

Algorithmic Normalization. Similarly to well-formedness, the normalization of algorithmic types is defined by extending the declarative definition (Fig. 5) with the algorithmic variables. To the rules repeating the declarative normalization, we add rules saying that normalization is trivial on algorithmic variables.

4.2 Type Constraints

Throughout the algorithm's operation, it gathers information about the algorithmic type variables, represented as *constraints*. These constraints in our system can be either *subtyping constraints* or *unification constraints*. As preserved by the algorithm, each subtyping constraint has a positive shape $\widehat{\alpha}^{+} :\geqslant P$, meaning it confines a positive algorithmic variable to be the supertype of a positive declarative type. Unification constraints can take a positive $\widehat{\alpha}^{+} :\simeq P$ or negative $\widehat{\alpha}^{-} :\simeq N$ form, but algorithmic type variables cannot occur on their right-hand side. The constraints *set* is denoted as C, and we presume that each algorithmic variable can be restricted by at most one constraint.

We define UC separately as a set solely containing *unification* constraints for simpler algorithm representation. The unification algorithm, which we use as a subroutine of the subtyping algorithm, can only produce unification constraints. A resolution of unification constraints is simpler than that of a general constraint set. This way, this separation allows us to better decompose the algorithm's structure, thus simplifying the inductive proofs.

Fig. 12. Constraint Entries and Sets

Auxiliary Functions.

- We define $\mathtt{dom}(C)$—the domain of constraint set C—as a set of algorithmic variables that it restricts. Similarly, we define $\mathtt{dom}(\Xi)$—the domain of instantiation context—as a set of algorithmic variables that Ξ associates with their contexts.
- We write $\Xi(\widehat{\alpha}^{\pm})$ to denote the declarative context associated with $\widehat{\alpha}^{\pm}$ in Ξ.
- $\mathtt{fav}(N)$ and $\mathtt{fav}(P)$ denote the set of free algorithmic variables in N and P respectively.
- We write $\Theta \vdash^{\supseteq} \Xi$ to denote that each declarative context associated with an algorithmic variable in Ξ is a subcontext (subset) of Θ.

Equivalent Substitutions. In the proofs of the algorithm correctness, we often state or require that two substitutions are equivalent on a given set of algorithmic variables. We denote it as $\Xi \vdash \widehat{\sigma}' \simeq^{\leqslant} \widehat{\sigma} : \widehat{\Theta}$ meaning that for each $\widehat{\alpha}^{\pm}$ in $\widehat{\Theta}$, substitutions $\widehat{\sigma}$ and $\widehat{\sigma}'$ map $\widehat{\alpha}^{\pm}$ to types equivalent in the corresponding context: $\Xi(\widehat{\alpha}^{\pm}) \vdash [\widehat{\sigma}']\widehat{\alpha}^{\pm} \simeq^{\leqslant} [\widehat{\sigma}]\widehat{\alpha}^{\pm}$.

Constraint well-formedness and satisfaction. Suppose that Ξ is an instantiation context. We say that constraint set C is well-formed in Ξ (denoted as $\Xi \vdash C$) if for every every constrain entry $e \in C$ associating a variable $\widehat{\alpha}^{\pm}$ with a type P, this type is well-formed in the corresponding declarative context $\Xi(\widehat{\alpha}^{\pm})$.

Substitution $\widehat{\sigma}$ satisfies a constraint *entry* e restricting algorithmic variable $\widehat{\alpha}^{\pm}$ if $[\sigma]\widehat{\alpha}^{\pm}$ can be validly substituted for $\widehat{\alpha}^{\pm}$ in e (so that the corresponding equivalence or subtyping holds).

Substitution $\widehat{\sigma}$ satisfies a constraint *set* C if $\Xi \vdash C$ and *each entry* of C is satisfied by $\widehat{\sigma}$.

4.3 Subtyping Algorithm

For convenience and scalability, we decompose the subtyping algorithm into several procedures. Figure 13 shows these procedures and the dependencies between them: arrows denote the invocation of one procedure from another.

Some of the procedures (in particular, the unification and the anti-unification) assume that the input types are normalized. Therefore, we call the normalization procedure before invoking them, indicating this by the '$\mathtt{nf}$' annotation on the arrows in Fig. 13. Alternatively, one could normalize the input types in the very beginning and preserve the normalization throughout the algorithm. However, we delay the normalization to the places where it is required to show that normalization is needed only at these stages to maintain consistent invariants.

In the remainder of this section, we will delve into each of these procedures in detail, following the top-down order of the dependency graph. First, we present the subtyping algorithm itself.

As an input, the subtyping algorithm takes a type context Θ, an instantiation context Ξ, and two types of the corresponding polarity: N and M for the negative subtyping, and P and Q for the positive subtyping. We assume the second type (M and Q) to be declarative (with no algorithmic variables) and well-formed in Θ, but the first type (N and P) may contain algorithmic variables, whose instantiation contexts are specified by Ξ.

Notice that the shape of the input types uniquely determines the applied subtyping rule. If the subtyping is successful, it returns a set of constraints C restricting the algorithmic variables of the first type. If the subtyping does not hold, there will be no inference tree with such inputs.

The rules of the subtyping algorithm bijectively correspond to the rules of the declarative system. Let us discuss them in detail.

Variables. Rules ($\textsc{Var}^{\leqslant}_{-}$) and ($\textsc{Var}^{\geqslant}_{+}$) say that if both of the input types are equal declarative variables, they are subtypes of each other, with no constraints (as there are no algorithmic variables).

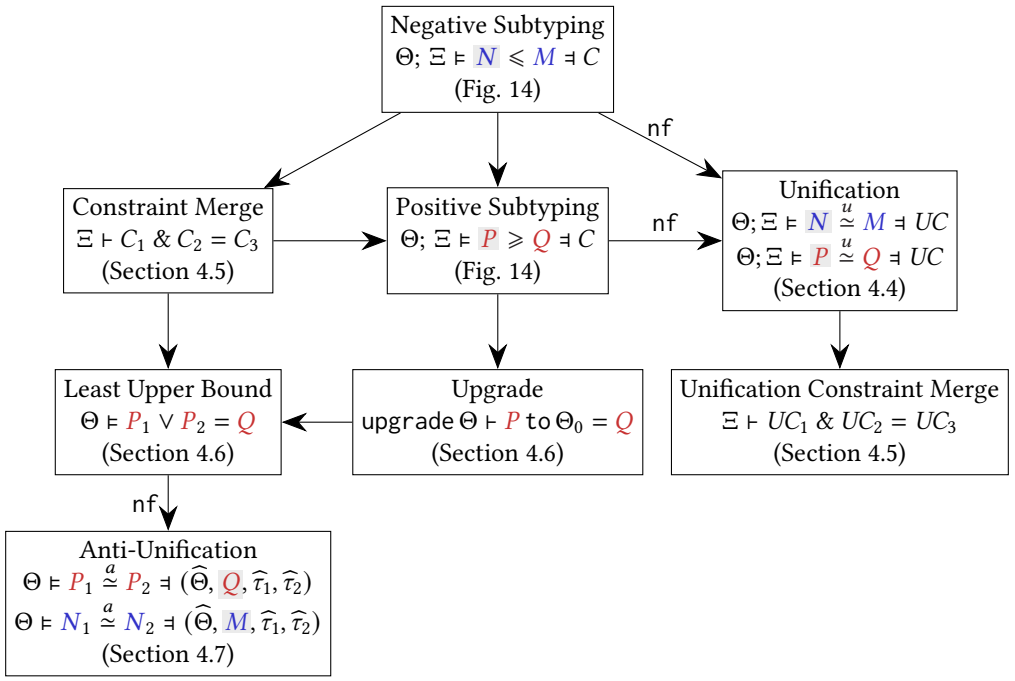

Fig. 13. Dependency graph of the subtyping algorithm

$\boxed{\Theta; \Xi \vDash N \leqslant M \dashv C}$ Negative subtyping

$$\frac{}{\Theta; \Xi \vDash \alpha^- \leqslant \alpha^- \dashv \cdot}\ (\text{VAR}^{\leqslant}_{-})$$

$$\frac{\Theta; \Xi \vDash \mathsf{nf}\,(P) \overset{u}{\simeq} \mathsf{nf}\,(Q) \dashv UC}{\Theta; \Xi \vDash \uparrow P \leqslant \uparrow Q \dashv UC}\ (\uparrow^{\leqslant})$$

$$\frac{\overrightarrow{\widehat{\alpha}^+}\ \mathsf{are\ fresh} \quad \Theta, \overrightarrow{\beta^+}; \Xi, \overrightarrow{\widehat{\alpha}^+}\{\Theta, \overrightarrow{\beta^+}\} \vDash [\overrightarrow{\widehat{\alpha}^+}/\overrightarrow{\alpha^+}]N \leqslant M \dashv C}{\Theta; \Xi \vDash \forall \overrightarrow{\alpha^+}.\, N \leqslant \forall \overrightarrow{\beta^+}.\, M \dashv C \setminus \overrightarrow{\widehat{\alpha}^+}}\ (\forall^{\leqslant})$$

$$\frac{\Theta; \Xi \vDash P \geqslant Q \dashv C_1 \quad \Theta; \Xi \vDash N \leqslant M \dashv C_2 \quad \Xi \vdash C_1 \,\&\, C_2 = C}{\Theta; \Xi \vDash P \to N \leqslant Q \to M \dashv C}\ (\to^{\leqslant})$$

$\boxed{\Theta; \Xi \vDash P \geqslant Q \dashv C}$ Positive supertyping

$$\frac{}{\Theta; \Xi \vDash \alpha^+ \geqslant \alpha^+ \dashv \cdot}\ (\text{VAR}^{\geqslant}_{+})$$

$$\frac{\overrightarrow{\widehat{\alpha}^-}\ \mathsf{are\ fresh} \quad \Theta, \overrightarrow{\beta^-}; \Xi, \overrightarrow{\widehat{\alpha}^-}\{\Theta, \overrightarrow{\beta^-}\} \vDash [\overrightarrow{\widehat{\alpha}^-}/\overrightarrow{\alpha^-}]P \geqslant Q \dashv C}{\Theta; \Xi \vDash \exists \overrightarrow{\alpha^-}.\, P \geqslant \exists \overrightarrow{\beta^-}.\, Q \dashv C \setminus \overrightarrow{\widehat{\alpha}^-}}\ (\exists^{\geqslant})$$

$$\frac{\Theta; \Xi \vDash \mathsf{nf}\,(N) \overset{u}{\simeq} \mathsf{nf}\,(M) \dashv UC}{\Theta; \Xi \vDash \downarrow N \geqslant \downarrow M \dashv UC}\ (\downarrow^{\geqslant})$$

$$\frac{\mathsf{upgrade}\,\Theta \vdash P\,\mathsf{to}\,\Xi(\widehat{\alpha}^+) = Q}{\Theta; \Xi \vDash \widehat{\alpha}^+ \geqslant P \dashv (\widehat{\alpha}^+ :\geqslant Q)}\ (\text{UVAR}^{\geqslant})$$

Fig. 14. Subtyping Algorithm

Shifts. Rules ($\downarrow^{\geqslant}$) and ($\uparrow^{\leqslant}$) cover the downshift and the upshift cases, respectively. If the input types are constructed by shifts, then the subtyping can only hold if they are equivalent. This way, the algorithm must find the instantiations of the algorithmic variables on the left-hand side such that these instantiations make the left-hand side and the ride-hand side equivalent. For this purpose,

the algorithm invokes the unification procedure (Section 4.4) preceded by the normalization of the input types. It returns the resulting constraints given by the unification algorithm.

Quantifiers. Rules ($\forall^{\leqslant}$) and ($\exists^{\geqslant}$) are symmetric. Declaratively, the quantified variables on the left-hand side must be instantiated with types, which are not known beforehand. We address this problem by algorithmization of the quantified variables (see Section 4.1). The rule introduces fresh algorithmic variables $\widehat{\overrightarrow{\alpha^+}}$ or $\widehat{\overrightarrow{\alpha^-}}$, puts them into the instantiation context Ξ (specifying that they must be instantiated in the extended context $\Theta, \overrightarrow{\beta^+}$ or $\Theta, \overrightarrow{\beta^-}$) and substitute the quantified variables for them in the input type.

After algorithmization of the quantified variables, the algorithm proceeds with the recursive call, returning constraints C. As the output, the algorithm removes the freshly introduced algorithmic variables from the instantiation context, This operation is sound: it is guaranteed that C always has a solution, but the specific instantiation of the freshly introduced algorithmic variables is not important, as they do not occur in the input types.

Functions. To infer the subtyping of the function types, the algorithm makes two calls: (i) a recursive call ensuring the subtyping of the result types, and (ii) a call to positive subtyping (or rather super-typing) on the argument types. The resulting constraints are merged using a special procedure defined in Section 4.5 and returned as the output.

Algorithmic variables. If one of the sides of the subtyping is a unification variable, the algorithm creates a new constraint. Because the right-hand side of the subtyping is always declarative, it is only the left-hand side that can be a unification variable. Moreover, another invariant we preserve prevents the negative algorithmic variables from occurring in types during the negative subtyping algorithm. It means that the only possible form of the subtyping here is $\widehat{\alpha}^+ \geqslant P$, which is covered by ($\textsc{UVar}^{\geqslant}$).

The potential problem here is that the type P might be not well-formed in the instantiation context required for $\widehat{\alpha}^+$ by Ξ because this context might be smaller than the current context Θ. As we wish the resulting constraint set to be sound w.r.t. Ξ, we cannot simply put $\widehat{\alpha}^+ :\geqslant P$ into the output. Prior to that, we update the type P to its lowest supertype Q well-formed in $\Xi(\widehat{\alpha}^+)$. It is done by the *upgrade* procedure, which we discuss in detail in Section 4.6.

To summarize, the subtyping algorithm uses the following additional subroutines: (i) rules ($\downarrow^{\geqslant}$) and ($\uparrow^{\leqslant}$) invoke the *unification* algorithm to equate the input types; (ii) rule ($\rightarrow^{\leqslant}$) *merges* the constraints produced by the recursive calls on the result and the argument types; and (iii) rule ($\textsc{UVar}^{\geqslant}$) *upgrades* the input type to its least supertype well-formed in the context required by the algorithmic variable. The following sections discuss these additional procedures in detail.

4.4 Unification

As an input, the unification context takes a type context Θ, an instantiation context Ξ, and two types of the required polarity: N and M for the negative unification, and P and Q for the positive unification. It is assumed that only the left-hand side type may contain algorithmic variables. This way, the left-hand side is well-formed as an algorithmic type in Θ and Ξ, whereas the right-hand side is well-formed declaratively in Θ.

Since only the left-hand side may contain algorithmic variables that the unification instantiates, we could have called this procedure *matching*. However, in Section 6.2, we will discuss several modifications of the type system, where this invariant is not preserved, and therefore, this procedure requires general first-order pattern unification [Miller 1991].

As the output, the unification algorithm returns *the weakest* set of unification constraints UC such that *any* instantiation satisfying these constraints unifies the input types.

$\boxed{\Theta; \Xi \vDash N \overset{u}{\simeq} M \dashv UC}$ Negative unification

$$\frac{}{\Theta; \Xi \vDash \alpha^- \overset{u}{\simeq} \alpha^- \dashv \cdot} \ (\textsc{Var}^{\overset{u}{\simeq}}_-)$$

$$\frac{\Theta; \Xi \vDash P \overset{u}{\simeq} Q \dashv UC}{\Theta; \Xi \vDash \uparrow P \overset{u}{\simeq} \uparrow Q \dashv UC} \ (\uparrow^{\overset{u}{\simeq}})$$

$$\frac{\Theta; \Xi \vDash P \overset{u}{\simeq} Q \dashv UC_1 \quad \Theta; \Xi \vDash N \overset{u}{\simeq} M \dashv UC_2}{\Theta; \Xi \vDash P \to N \overset{u}{\simeq} Q \to M \dashv UC_1 \ \& \ UC_2} \ (\to^{\overset{u}{\simeq}})$$

$$\frac{\Theta, \overrightarrow{\alpha^+}; \Xi \vDash N \overset{u}{\simeq} M \dashv UC}{\Theta; \Xi \vDash \forall \overrightarrow{\alpha^+}. N \overset{u}{\simeq} \forall \overrightarrow{\alpha^+}. M \dashv UC} \ (\forall^{\overset{u}{\simeq}})$$

$$\frac{\Xi(\widehat{\alpha}^-) \vdash N}{\Theta; \Xi \vDash \widehat{\alpha}^- \overset{u}{\simeq} N \dashv (\widehat{\alpha}^- :\simeq N)} \ (\textsc{UVar}^{\overset{u}{\simeq}}_-)$$

$\boxed{\Theta; \Xi \vDash P \overset{u}{\simeq} Q \dashv UC}$ Positive unification

$$\frac{}{\Theta; \Xi \vDash \alpha^+ \overset{u}{\simeq} \alpha^+ \dashv \cdot} \ (\textsc{Var}^{\overset{u}{\simeq}}_+)$$

$$\frac{\Theta; \Xi \vDash N \overset{u}{\simeq} M \dashv UC}{\Theta; \Xi \vDash \downarrow N \overset{u}{\simeq} \downarrow M \dashv UC} \ (\downarrow^{\overset{u}{\simeq}})$$

$$\frac{\Theta, \overrightarrow{\alpha^-}; \Xi \vDash P \overset{u}{\simeq} Q \dashv UC}{\Theta; \Xi \vDash \exists \overrightarrow{\alpha^-}. P \overset{u}{\simeq} \exists \overrightarrow{\alpha^-}. Q \dashv UC} \ (\exists^{\overset{u}{\simeq}})$$

$$\frac{\Xi(\widehat{\alpha}^+) \vdash P}{\Theta; \Xi \vDash \widehat{\alpha}^+ \overset{u}{\simeq} P \dashv (\widehat{\alpha}^+ :\simeq P)} \ (\textsc{UVar}^{\overset{u}{\simeq}}_+)$$

Fig. 15. Unification Algorithm

The algorithm works as one might expect: if both sides are formed by constructors, it is required that the constructors are the same, and the types unify recursively. If one of the sides is a unification variable (in our case it can only be the left-hand side), we create a new unification constraint restricting it to be equal to the other side. Let us discuss the rules that implement this strategy.

Variables. The variable rules $(\textsc{Var}^{\overset{u}{\simeq}}_-)$ and $(\textsc{Var}^{\overset{u}{\simeq}}_+)$ are trivial: as the input types do not have algorithmic variables, and are already equal, the unification returns no constraints.

Shifts. The shift rules $(\downarrow^{\overset{u}{\simeq}})$ and $(\uparrow^{\overset{u}{\simeq}})$ require the two input types to be formed by the same shift constructor. They remove this constructor, unify the types recursively, and return the resulting set of constraints.

Quantifiers. Similarly, the quantifier rules $(\forall^{\overset{u}{\simeq}})$ and $(\exists^{\overset{u}{\simeq}})$ require the quantifier variables on the left-hand side and the right-hand side to be the same. This requirement is complete because we assume the input types of the unification to be normalized, and thus, the equivalence implies alpha-equivalence. In the implementation of this rule, an alpha-renaming might be needed to ensure that the quantified variables are the same, however, we omit it for brevity.

Functions. Rule $(\to^{\overset{u}{\simeq}})$ unifies two functional types. First, it unifies their argument types and their result types recursively. Then it merges the resulting constraints using the procedure described in Section 4.5.

Notice that the resulting constraints can only have *unification* entries. It means that they can be merged in a simpler way than general constraints. In particular, the merging procedure does not call any of the subtyping subroutines but rather simply checks the matching constraint entries for equality.

Algorithmic variable. Finally, if the left-hand side of the unification is an algorithmic variable, ($\textsc{Var}^{\overset{u}{\simeq}}_{-}$) or ($\textsc{Var}^{\overset{u}{\simeq}}_{+}$) is applied. It simply checks that the right-hand side type is well-formed in the required instantiation context, and returns a newly created constraint restricting the variable to be equal to the right-hand side type.

As one can see, the unification procedure is standard; the only peculiarity (although it is common for type inference) is that it makes sure that the resulting instantiations agree with the input instantiation context Ξ. As a subroutine, the unification algorithm only uses the (unification) constraint merge procedure and the well-formedness checking.

4.5 Constraint Merge

In this section, we discuss the constraint merging procedure. It allows one to combine two constraint sets into one. A simple union of two constraint sets is not sufficient, since the resulting set must not contain two entries restricting the same algorithmic variable—we call such entries *matching*.

The matching entries must be combined into *one* constraint entry, that would represent their conjunction. This way, to merge two constraint sets, we unite the entries of two sets and then merge the matching pairs.

Merging matching constraint entries. Two *matching* entries formed in the same context Θ can be merged as shown in Fig. 16. Suppose that e_1 and e_2 are input entries. The result of the merge $e_1 \mathbin{\&} e_2$ is *the weakest entry which implies both* e_1 *and* e_2.

$\boxed{\Theta \vdash e_1 \mathbin{\&} e_2 = e_3}$ Subtyping Constraint Entry Merge

$$\frac{\Theta \vDash P_1 \vee P_2 = Q}{\Theta \vdash (\widehat{\alpha}^+ :\geqslant P_1) \mathbin{\&} (\widehat{\alpha}^+ :\geqslant P_2) = (\widehat{\alpha}^+ :\geqslant Q)} \quad (\geqslant \&^+ \geqslant)$$

$$\frac{\Theta; \cdot \vDash P \geqslant Q \dashv \cdot}{\Theta \vdash (\widehat{\alpha}^+ :\simeq P) \mathbin{\&} (\widehat{\alpha}^+ :\geqslant Q) = (\widehat{\alpha}^+ :\simeq P)} \quad (\simeq \&^+ \geqslant)$$

$$\frac{\Theta; \cdot \vDash Q \geqslant P \dashv \cdot}{\Theta \vdash (\widehat{\alpha}^+ :\geqslant P) \mathbin{\&} (\widehat{\alpha}^+ :\simeq Q) = (\widehat{\alpha}^+ :\simeq Q)} \quad (\geqslant \&^+ \simeq)$$

$$\frac{\mathsf{nf}(P) = \mathsf{nf}(P')}{\Theta \vdash (\widehat{\alpha}^+ :\simeq P) \mathbin{\&} (\widehat{\alpha}^+ :\simeq P') = (\widehat{\alpha}^+ :\simeq P)} \quad (\simeq \&^+ \simeq)$$

$$\frac{\mathsf{nf}(N) = \mathsf{nf}(N')}{\Theta \vdash (\widehat{\alpha}^- :\simeq N) \mathbin{\&} (\widehat{\alpha}^- :\simeq N') = (\widehat{\alpha}^- :\simeq N)} \quad (\simeq \&^- \simeq)$$

Fig. 16. Merge of Matching Constraint Entries

Suppose that one of the input entries, say e_1, is a *unification* constraint entry. Then the resulting entry e_1 must coincide with it (up-to-equivalence), and thus, it is only required to check that e_2 is implied by e_1. We consider two options:

(i) if e_2 is also a *unification* entry, then the types on the right-hand side of e_1 and e_2 must be equivalent, as given by rules $(\simeq \&^+ \simeq)$ and $(\simeq \&^- \simeq)$;
(ii) if e_2 is a *supertype* constraint entry $\widehat{\alpha}^+ :\geqslant P$, the algorithm must check that the type assigned by e_1 is a supertype of P. The corresponding symmetric rules are $(\geqslant \&^+ \simeq)$ and $(\simeq \&^+ \geqslant)$.

In the premises of these rules, P is the same as P below, and Q is the same as Q below but relaxed to an algorithmic type.

If both input entries are supertype constraints: $\widehat{\alpha}^{+} :\geqslant P$ and $\widehat{\alpha}^{+} :\geqslant Q$, then their conjunction is $\widehat{\alpha}^{+} :\geqslant P \vee Q$, as given by $(\geqslant \&^{+} \geqslant)$. The least upper bound—$P \vee Q$—is the least supertype of both P and Q, and this way, $\widehat{\alpha}^{+} :\geqslant P \vee Q$ is the weakest constraint entry that implies $\widehat{\alpha}^{+} :\geqslant P$ and $\widehat{\alpha}^{+} :\geqslant Q$. The algorithm finding the least upper bound is discussed in Section 4.6.

Merging constraint sets. The algorithm for merging constraint sets is shown in Fig. 17. As discussed, the result of merge C_1 and C_2 consists of three parts: (i) the entries of C_1 that do not match any entry of C_2; (ii) the entries of C_2 that do not match any entry of C_1; and (iii) the merge (Fig. 16) of matching entries.

Suppose that $\Xi \vdash C_1$ and $\Xi \vdash C_2$.
Then $\Xi \vdash C_1 \,\&\, C_2 = C$ defines a set of constraints C such that $e \in C$ iff either:

- $e \in C_1$ and there is no matching $e' \in C_2$; or
- $e \in C_2$ and there is no matching $e' \in C_1$; or
- $\Xi(\widehat{\alpha}^{\pm}) \vdash e_1 \,\&\, e_2 = e$ for some $e_1 \in C_1$ and $e_2 \in C_2$ such that e_1 and e_2 both restrict variable $\widehat{\alpha}^{\pm}$.

Fig. 17. Constraint Merge

As shown in Fig. 16, the merging procedure relies substantially on the least upper bound algorithm. In the next section, we discuss this algorithm in detail, together with the upgrade procedure, selecting the least supertype well-formed *in a given context.*

4.6 Type Upgrade and the Least Upper Bounds

Both type upgrade and the least upper bound algorithms are used to find a minimal supertype under certain conditions. For a given type P well-formed in Θ, the *upgrade* operation finds the least among those supertypes of P that are well-formed in a smaller context $\Theta_0 \subseteq \Theta$. For given two types P_1 and P_2 well-formed in Θ, the *least upper bound* operation finds the least among common supertypes of P_1 and P_2 well-formed in Θ. These algorithms are shown in Fig. 18.

$\boxed{\mathsf{upgrade}\ \Theta \vdash P\ \mathsf{to}\ \Theta_0 = Q}$ Type Upgrade

$$\frac{\Theta = \Theta_0, \overrightarrow{\alpha^{\pm}} \quad \overrightarrow{\beta^{\pm}}\ \mathsf{are\ fresh} \quad \overrightarrow{\gamma^{\pm}}\ \mathsf{are\ fresh} \quad \Theta_0, \overrightarrow{\beta^{\pm}}, \overrightarrow{\gamma^{\pm}} \vDash [\overrightarrow{\beta^{\pm}}/\overrightarrow{\alpha^{\pm}}]P \vee [\overrightarrow{\gamma^{\pm}}/\overrightarrow{\alpha^{\pm}}]P = Q}{\mathsf{upgrade}\ \Theta \vdash P\ \mathsf{to}\ \Theta_0 = Q} \ (\textsc{Upg})$$

$\boxed{\Theta \vDash P_1 \vee P_2 = Q}$ Least Upper Bound

$$\frac{\Theta, \overrightarrow{\alpha^{-}}, \overrightarrow{\beta^{-}} \vDash P_1 \vee P_2 = Q}{\Theta \vDash \exists\overrightarrow{\alpha^{-}}.\, P_1 \vee \exists\overrightarrow{\beta^{-}}.\, P_2 = Q} \ (\exists^{\vee})$$

$$\frac{}{\Theta \vDash \alpha^{+} \vee \alpha^{+} = \alpha^{+}} \ (\textsc{Var}^{\vee})$$

$$\frac{\Theta \vDash \mathsf{nf}\,(\downarrow N) \stackrel{a}{\cong} \mathsf{nf}\,(\downarrow M) \dashv (\widehat{\Theta}, P, \widehat{\tau}_1, \widehat{\tau}_2)}{\Theta \vDash \downarrow N \vee \downarrow M = \exists\overrightarrow{\alpha^{-}}.\, [\overrightarrow{\alpha^{-}}/\widehat{\Theta}]P} \ (\downarrow^{\vee})$$

Fig. 18. Type Upgrade and Leas Upper Bound Algorithms

The Type Upgarde. The type upgrade algorithm uses the least upper bound algorithm as a subroutine. It exploits the idea that the free variables of a positive type Q cannot disappear in its subtypes (see Property 1). It means that if a type P has free variables not occurring in P', then any common supertype of P and P' must not contain these variables either. This way, any supertype of P not containing certain variables $\overrightarrow{\alpha^{\pm}}$ must also be a supertype of $P' = [\overrightarrow{\beta^{\pm}}/\overrightarrow{\alpha^{\pm}}]P$, where $\overrightarrow{\beta^{\pm}}$ are fresh; and vice versa: any common supertype of P and P' does not contain $\overrightarrow{\alpha^{\pm}}$ nor $\overrightarrow{\beta^{\pm}}$.

This way, to find the least supertype of P well-formed in $\Theta_0 = \Theta \setminus \overrightarrow{\alpha^{\pm}}$ (i.e., not containing $\overrightarrow{\alpha^{\pm}}$), we can do the following. First, construct a new type P' by renaming $\overrightarrow{\alpha^{\pm}}$ in P to fresh $\overrightarrow{\beta^{\pm}}$, and second, find *the least upper bound* of P and P' in the appropriate context. However, for reasons of symmetry, in rule (Upg) we employ a different but equivalent approach: we create *two* types P_1 and P_2 constructed by renaming $\overrightarrow{\alpha^{\pm}}$ in P to fresh disjoint variables $\overrightarrow{\beta^{\pm}}$ and $\overrightarrow{\gamma^{\pm}}$ respectively, and then find the least upper bound of P_1 and P_2.

The Least Upper Bound. The Least Upper Bound algorithm we use operates on *positive* types. This way, the inference rules of the algorithm analyze the three possible shapes of the input types: a variable type, an existential type, and a shifted computation.

Rule ($\exists^{\vee}$) covers the case when at least one of the input types is an existential type. In this case, we can simply move the existential quantifiers from both sides to the context, and make a tail-recursive call. However, it is important to make sure that the quantified variables $\overrightarrow{\alpha^-}$ and $\overrightarrow{\beta^-}$ are disjoint (i.e., alpha-renaming might be required in the implementation).

Rule (Var$^{\vee}$) applies when both sides are variables. In this case, the common supertype only exists if these variables are the same. And if they are, the common supertypes must be equivalent to this variable.

Rule ($\downarrow^{\vee}$) is the most interesting. If both sides are not quantified, and one of the sides is a shift, so must be the other side. However, the set of common upper bounds is not trivial in this case. For example, $\downarrow(\beta^+ \to \gamma_1^-)$ and $\downarrow(\beta^+ \to \gamma_2^-)$ have two non-equivalent common supertypes: $\exists\alpha^-.\downarrow\alpha^-$ (by instantiating α^- with $\beta^+ \to \gamma_1^-$ and $\beta^+ \to \gamma_2^-$ respectively) and $\exists\alpha^-.\downarrow(\beta^+ \to \alpha^-)$ (by instantiating α^- with γ_1^- and γ_2^- respectively). As one can see, the second supertype $\exists\alpha^-.\downarrow(\beta^+ \to \alpha^-)$ is the least among them because it abstracts over a 'deeper' negative subexpression.

In general, we must (i) find the most detailed pattern (a type with 'holes' at negative positions) that matches both sides, and (ii) abstract over the 'holes' by existential quantifiers. The algorithm that finds the most detailed common pattern is called *anti-unification*. As output, it returns $(\widehat{\Theta}, P, \widehat{\tau}_1, \widehat{\tau}_2)$, where important for us is P—the pattern—and $\widehat{\Theta}$–the set of 'holes' represented by negative algorithmic variables. We discuss the anti-unification algorithm in detail in the following section.

4.7 Anti-Unification

The anti-unification algorithm [Plotkin 1970; Reynolds 1970], is a procedure dual to unification. For two given (potentially different) expressions, it finds the most specific generalizer—the most detailed pattern that matches both of the input expressions. As evidence, it can also return two substitutions that instantiate the 'holes' of the pattern to the input expressions.

In our case, we have to be more demanding on the anti-unification algorithm. Since we use it to construct an existential type, whose (negative) quantified variables can only be instantiated with negative types, we must make sure that the pattern has 'holes' only at negative positions. Moreover, we must make sure that the resulting substitutions for the 'holes' are well-formed in the context from the past—at the moment when the corresponding polymorphic variables were introduced—and do not contain variables bound later. For example, the anti-unification of $N_1 = \forall\beta^+.\,\alpha_1^+ \to \uparrow\beta^+$

and $N_2 = \forall\beta^+.\, \alpha_2^+ \to \uparrow\beta^+$ is a singleton 'hole', which we model as an algorithmic type variable $\widehat{\gamma}^-$, with a pair of substitutions $\widehat{\gamma}^- \mapsto N_1$ and $\widehat{\gamma}^- \mapsto N_2$. But it *cannot* be more specific such as $\forall\beta^+.\, \widehat{\gamma}^+ \to \uparrow\beta^+$ (since the hole cannot be positive) or $\forall\beta^+.\, \widehat{\gamma}^-$ (since *the instantiation of* $\widehat{\gamma}^-$ *cannot capture the bound variable* β^+).

The algorithm that finds the most specific generalizer of two types under required conditions is given in Fig. 19. It consists of two mutually recursive procedures: the positive and the negative anti-unification. As the positive and the negative anti-unification procedures are symmetric in their interface, let us discuss how to read the positive judgment.

The positive anti-unification judgment has form $\Theta \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\widehat{\Theta}, Q, \widehat{\tau}_1, \widehat{\tau}_2)$. As an input, it takes a context Θ, in which the 'holes' instantiations must be well-formed, and two positive types: P_1 and P_2; it returns a tuple of four components: $\widehat{\Theta}$—a set of 'holes' represented by negative algorithmic variables, Q—a pattern represented as a positive algorithmic type, whose algorithmic variables are in $\widehat{\Theta}$, and two substitutions $\widehat{\tau}_1$ and $\widehat{\tau}_2$ instantiating the variables from $\widehat{\Theta}$ such that $[\widehat{\tau}_1]\, Q = P_1$ and $[\widehat{\tau}_2]\, Q = P_2$.

$$\boxed{\Theta \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\widehat{\Theta}, Q, \widehat{\tau}_1, \widehat{\tau}_2)}$$

$$\frac{}{\Theta \vDash \alpha^+ \overset{a}{\simeq} \alpha^+ \dashv (\cdot, \alpha^+, \cdot, \cdot)}\ (\textsc{Var}^{\overset{a}{\simeq}}_+)$$

$$\frac{\Theta \vDash N_1 \overset{a}{\simeq} N_2 \dashv (\widehat{\Theta}, M, \widehat{\tau}_1, \widehat{\tau}_2)}{\Theta \vDash \downarrow N_1 \overset{a}{\simeq} \downarrow N_2 \dashv (\widehat{\Theta}, \downarrow M, \widehat{\tau}_1, \widehat{\tau}_2)}\ (\downarrow^{\overset{a}{\simeq}})$$

$$\frac{\overrightarrow{\alpha^-} \cap \Theta = \emptyset \quad \Theta \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\widehat{\Theta}, Q, \widehat{\tau}_1, \widehat{\tau}_2)}{\Theta \vDash \exists\overrightarrow{\alpha^-}.\, P_1 \overset{a}{\simeq} \exists\overrightarrow{\alpha^-}.\, P_2 \dashv (\widehat{\Theta}, \exists\overrightarrow{\alpha^-}.\, Q, \widehat{\tau}_1, \widehat{\tau}_2)}\ (\exists^{\overset{a}{\simeq}})$$

$$\boxed{\Theta \vDash N_1 \overset{a}{\simeq} N_2 \dashv (\widehat{\Theta}, M, \widehat{\tau}_1, \widehat{\tau}_2)}$$

$$\frac{}{\Theta \vDash \alpha^- \overset{a}{\simeq} \alpha^- \dashv (\cdot, \alpha^-, \cdot, \cdot)}\ (\textsc{Var}^{\overset{a}{\simeq}}_-)$$

$$\frac{\Theta \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\widehat{\Theta}, Q, \widehat{\tau}_1, \widehat{\tau}_2)}{\Theta \vDash \uparrow P_1 \overset{a}{\simeq} \uparrow P_2 \dashv (\widehat{\Theta}, \uparrow Q, \widehat{\tau}_1, \widehat{\tau}_2)}\ (\uparrow^{\overset{a}{\simeq}})$$

$$\frac{\overrightarrow{\alpha^+} \cap \Theta = \emptyset \quad \Theta \vDash N_1 \overset{a}{\simeq} N_2 \dashv (\widehat{\Theta}, M, \widehat{\tau}_1, \widehat{\tau}_2)}{\Theta \vDash \forall\overrightarrow{\alpha^+}.\, N_1 \overset{a}{\simeq} \forall\overrightarrow{\alpha^+}.\, N_2 \dashv (\widehat{\Theta}, \forall\overrightarrow{\alpha^+}.\, M, \widehat{\tau}_1, \widehat{\tau}_2)}\ (\forall^{\overset{a}{\simeq}})$$

$$\frac{\Theta \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\widehat{\Theta}_1, Q, \widehat{\tau}_1, \widehat{\tau}_2) \quad \Theta \vDash N_1 \overset{a}{\simeq} N_2 \dashv (\widehat{\Theta}_2, M, \widehat{\tau}'_1, \widehat{\tau}'_2)}{\Theta \vDash P_1 \to N_1 \overset{a}{\simeq} P_2 \to N_2 \dashv (\widehat{\Theta}_1 \cup \widehat{\Theta}_2, Q \to M, \widehat{\tau}_1 \cup \widehat{\tau}'_1, \widehat{\tau}_2 \cup \widehat{\tau}'_2)}\ (\to^{\overset{a}{\simeq}})$$

$$\frac{\text{if other rules are not applicable} \quad \Theta \vDash N \quad \Theta \vDash M}{\Theta \vDash N \overset{a}{\simeq} M \dashv (\widehat{\alpha}^-_{\{N,M\}}, \widehat{\alpha}^-_{\{N,M\}}, (\widehat{\alpha}^-_{\{N,M\}} \mapsto N), (\widehat{\alpha}^-_{\{N,M\}} \mapsto M))}\ (\text{AU})$$

Fig. 19. Anti-Unification Algorithm

At the high level, the algorithm scheme follows the standard approach [Plotkin 1970] represented as a recursive procedure. Specifically, it follows two principles:

(i) if the input terms start with the same constructor, we anti-unify the corresponding parts recursively and unite the results. This principle is followed by all the rules except (AU), which works as follows:
(ii) if the first principle does not apply to the input terms N and M (for instance, if they have different outer constructors), the anti-unification algorithm returns a 'hole' such that one substitution maps it to N and the other maps it to M. This 'hole' should have a name uniquely defined by the pair (N, M), so that it automatically merges with other 'holes' mapped to the same pair of types, and thus, the initiality of the generalizer is ensured.

Let us discuss the specific rules of the algorithm in detail.

Variables. Rules ($\textsc{Var}^{\overset{a}{\simeq}}_{+}$) and ($\textsc{Var}^{\overset{a}{\simeq}}_{-}$) generalize two equal variables. In this case, the resulting pattern is the variable itself, and no 'holes' are needed.

Shifts. Rules ($\downarrow^{\overset{a}{\simeq}}$) and ($\uparrow^{\overset{a}{\simeq}}$) operate by congruence: they anti-unify the bodies of the shifts recursively and add the shift constructor back to the resulting pattern.

Quantifiers. Rules ($\forall^{\overset{a}{\simeq}}$) and ($\exists^{\overset{a}{\simeq}}$) are symmetric. They generalize two quantified types congruently, similarly to the shift rules. However, we also require that the quantified variables are fresh, and that the left-hand side variables are equal to the corresponding variables on the right-hand side. To ensure it, alpha-renaming might be required in the implementation.

Notice that the context Θ is *not* extended with the quantified variables. In this algorithm, Θ does not play the role of a current typing context, but rather a snapshot of a context at the moment of calling the anti-unification, i.e., the context in which the instantiations of the 'holes' must be well-formed.

Functions. Rule ($\rightarrow^{\overset{a}{\simeq}}$) congruently generalizes two function types. An arrow type is the only binary constructor, and thus, it is the only rule where the union of the anti-unification results is substantial. The interesting is the case when the resulting generalization of the input types and the resulting generalization of the output types have 'holes' mapped to the same pair of types. In this case, the algorithm must merge the 'holes' into one. For example, the anti-unification of $\downarrow\alpha^- \to \alpha^-$ and $\downarrow\beta^- \to \beta^-$ must result in $\downarrow\widehat{\gamma}^- \to \widehat{\gamma}^-$, rather than $\downarrow\widehat{\gamma_1}^- \to \widehat{\gamma_2}^-$.

In our representation of the anti-unification algorithm, this 'merge' happens automatically: following the rule (AU), the name of the 'hole' is uniquely defined by the pair of types it is mapped to. Specifically, when anti-unifying $\downarrow\alpha^- \to \alpha^-$ and $\downarrow\beta^- \to \beta^-$ our algorithm returns $\downarrow\widehat{\alpha}^-_{\{\alpha^-,\beta^-\}} \to \widehat{\alpha}^-_{\{\alpha^-,\beta^-\}}$, that is a renaming of $\downarrow\widehat{\gamma}^- \to \widehat{\gamma}^-$.

This way, as the output the rule returns the following tuple:

- $\widehat{\Theta}_1 \cup \widehat{\Theta}_2$—a simple union of the sets of 'holes' returned from by the recursive calls,
- $Q \to M$—the resulting pattern constructed from the patterns returned recursively.
- $\widehat{\tau}_1 \cup \widehat{\tau}'_1$ and $\widehat{\tau}_2 \cup \widehat{\tau}'_2$ — a union (in a relational sense) of the substitutions returned by the recursive calls. It is worth noting that the union is well-defined because the result of the substation on a 'hole' is determined by the name of the 'hole'.

The Anti-Unification Rule. Rule (AU) is the base case of the anti-unification algorithm. If the congruent rules are not applicable, it means that the input types have a substantially different structure, and thus, the only option is to create a 'hole'. There are three important aspects of this rule that we would like to discuss.

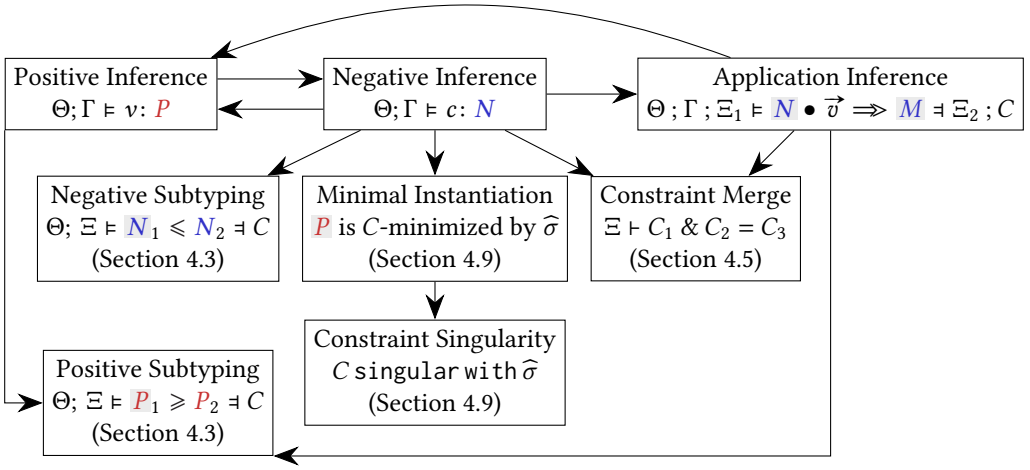

Fig. 20. Dependency graph of the typing algorithm

First, as mentioned earlier, the freshly created 'hole' has a name that is uniquely defined by the pair of input types. It is ensured by the following invariant: all the 'holes' in the algorithm have name $\widehat{\alpha}^-$ indexed by the pair of negative types it is mapped to. This way, the returning set of 'holes' is a singleton set $\{\widehat{\alpha}^-_{\{N,M\}}\}$; the resulting pattern is the 'hole' $\widehat{\alpha}^-_{\{N,M\}}$, and the mappings simply send it to the corresponding types: $\widehat{\alpha}^-_{\{N,M\}} \mapsto N$ and $\widehat{\alpha}^-_{\{N,M\}} \mapsto M$.

Second, this rule is only applicable to negative types; moreover, the input types are checked to be well-formed in the outer context Θ. This is required by the usage of anti-unification: we call it to build an existential type that would be an upper bound of two input types via abstracting some of their subexpressions under existential quantifiers. The existentials quantify over *negative* variables, and they must be instantiated in the context available at that moment.

Third, the rule is only applicable if all other rules fail. Notice that it could happen even when the input types have matching constructors. For example, the generalizer of $\uparrow\alpha^+$ and $\uparrow\beta^+$ is $\widehat{\gamma}^-$ (with mappings $\widehat{\gamma}^- \mapsto \uparrow\alpha^+$ and $\widehat{\gamma}^- \mapsto \uparrow\beta^+$), rather than $\uparrow\widehat{\gamma}^+$. This way, the algorithm must try to apply the congruent rules first, and only if they fail, apply (AU). As it is not known a priori whether the congruent rules will be applicable, the implementation must use backtracking.

4.8 Type Inference

Finally, we present the type inference algorithm. Similarly to the subtyping algorithm, it structurally corresponds to the declarative inference specification, meaning that most of the algorithmic rules have declarative counterparts, with respect to which they are sound and complete.

This way, the inference algorithm also consists of three mutually recursive procedures: the positive type inference, the negative type inference, and the application type inference. As subroutines, the inference algorithm uses subtyping, constraint merge, and minimal instantiation. The corresponding dependency is shown in Fig. 20.

The positive and the negative type inference judgments have symmetric forms: $\Theta; \Gamma \vDash v\colon P$ and $\Theta; \Gamma \vDash c\colon N$. Both of these algorithms take as an input typing context Θ, a variable context Γ, and a term (a value or a computation) taking its type variables from Θ, and term variables from Γ. As an output, they return a type of the given term, which we guarantee to be normalized.

The application type inference judgment has form $\Theta\,;\Gamma\,;\Xi_1 \vDash N \bullet \vec{v} \Longrightarrow M \dashv \Xi_2\,;C$. As an input, it takes three contexts: typing context Θ, a variable context Γ, and an instantiation context Ξ_1. It also takes a head type N and a list of arguments (terms) $\vec{v}$ the head is applied to. The head may contain algorithmic variables specified by Ξ_1, in other words, $\Theta\,;\,\mathtt{dom}\,(\Xi_1) \vdash N$. As a result, the application inference judgment returns M—a normalized type of the result of the application. Type M may contain new algorithmic variables, and thus, the judgment also returns Ξ_2—an updated instantiation context and C—a set of subtyping constraints. Together Ξ_2 and C specify how the algorithmic variables must be instantiated.

The inference rules are shown in Fig. 21. Next, we discuss them in detail.

Variables. Rule ($\textsc{Var}^{\textsc{Inf}}$) infers the type of a positive variable by looking it up in the term variable context and normalizing the result.

Annotations. Rules ($\textsc{Ann}^{\textsc{Inf}}_{+}$) and ($\textsc{Ann}^{\textsc{Inf}}_{-}$) are symmetric. First, they check that the annotated type is well-formed in the given context Θ. Then they make a recursive call to infer the type of annotated expression, check that the inferred type is a subtype of the annotation, and return the normalized annotation.

Abstractions. Rule ($\lambda^{\textsc{Inf}}$) infers the type of a lambda abstraction. It checks the well-formedness of the annotation P, makes a recursive call to infer the type of the body in the extended context, and returns the corresponding arrow type. Since the annotation P is allowed to be non-normalized, the rule also normalizes the resulting type.

Rule ($\Lambda^{\textsc{Inf}}$) infers the type of a big lambda. Similarly to the previous case, it makes a recursive call to infer the type of the body in the extended *type* context. After that, it returns the corresponding universal type. It is also required to normalize the result. For instance, if α^{+} does not occur in the body of the lambda, the corresponding $\forall$ will be removed.

Return and Thunk. Rules ($\{\}^{\textsc{Inf}}$) and ($\textsc{Ret}^{\textsc{Inf}}$) are similar to the declarative rules: they make a recursive call to type the body of the thunk or the return expression and put the shift on top of the result.

Unpack. Rule ($\textsc{Let}^{\textsc{Inf}}_{\exists}$) allows one to unpack an existential type. First, it infers the existential type $\exists\overrightarrow{\alpha}.\,P$ of the value being unpacked, and since the type is guaranteed to be normalized, binds the quantified variables with $\overrightarrow{\alpha}$. Then it infers the type of the body in the appropriately extended context and checks that the inferred type does not depend on $\overrightarrow{\alpha}$ by checking well-formedness $\Theta \vdash N$.

Let Binders. Rule ($\textsc{Let}^{\textsc{Inf}}$) represents the type inference of a standard let binder. It infers the type of the bound value v, and makes a recursive call to infer the type of the body in the extended context.

Rule ($\textsc{Let}^{\textsc{Inf}}_{\mathrm{C}}$) infers a type of computational let binder. It follows the corresponding declarative rule ($\textsc{Let}^{\textsc{Inf}}_{\mathrm{C}}$) but uses algorithmic judgments instead of declarative ones. It is worth noting that when calling the subtyping $\Theta;\,\cdot \vDash M \leqslant \uparrow P \dashv \cdot$, both M and P are free of algorithmic variables: M is a type inferred for c, and P is given as an annotation.

Rule ($\textsc{Let}^{\textsc{Inf}}_{:@}$) infers a type of *annotated* applicative let binder. First, it infers the type of the head of the application, ensuring that it is a *thunked computation* $\downarrow M$. After that, it makes a recursive call to the application inference procedure, returning an algorithmic type M', that must be a subtype of the annotation $\uparrow P$.

$\boxed{\Theta; \Gamma \vDash v \colon P}$ Positive typing

$$\frac{x : P \in \Gamma}{\Theta; \Gamma \vDash x \colon \mathsf{nf}\,(P)}\ (\textsc{Var}^{\textsc{inf}}) \qquad \frac{\Theta \vdash Q \quad \Theta; \Gamma \vDash v \colon P \quad \Theta; \cdot \vDash Q \geqslant P \dashv \cdot}{\Theta; \Gamma \vDash (v : Q) \colon \mathsf{nf}\,(Q)}\ (\textsc{ann}_{+}^{\textsc{inf}}) \qquad \frac{\Theta; \Gamma \vDash c \colon N}{\Theta; \Gamma \vDash \{c\} \colon \downarrow N}\ (\{\}^{\textsc{inf}})$$

$\boxed{\Theta; \Gamma \vDash c \colon N}$ Negative typing

$$\frac{\Theta \vdash M \quad \Theta; \Gamma \vDash c \colon N \quad \Theta; \cdot \vDash N \leqslant M \dashv \cdot}{\Theta; \Gamma \vDash (c : M) \colon \mathsf{nf}\,(M)}\ (\textsc{ann}_{-}^{\textsc{inf}}) \qquad \frac{\Theta; \Gamma \vDash v \colon P \quad \Theta; \Gamma, x : P \vDash c \colon N}{\Theta; \Gamma \vDash \mathtt{let}\, x = v;\ c \colon N}\ (\textsc{let}^{\textsc{inf}})$$

$$\frac{\Theta \vdash P \quad \Theta; \Gamma, x : P \vDash c \colon N}{\Theta; \Gamma \vDash \lambda x : P.\, c \colon \mathsf{nf}\,(P \to N)}\ (\lambda^{\textsc{inf}}) \qquad \frac{\Theta \vdash P \quad \Theta; \Gamma \vDash c \colon M \quad \Theta; \cdot \vDash M \leqslant \uparrow P \dashv \cdot \quad \Theta; \Gamma, x : P \vDash c' \colon N}{\Theta; \Gamma \vDash \mathtt{let}\, x : P = c;\ c' \colon N}\ (\textsc{let}_{\textsc{c}}^{\textsc{inf}})$$

$$\frac{\Theta, \alpha^{+}; \Gamma \vDash c \colon N}{\Theta; \Gamma \vDash \Lambda \alpha^{+}.\, c \colon \mathsf{nf}\,(\forall \alpha^{+}.\, N)}\ (\Lambda^{\textsc{inf}}) \qquad \frac{\Theta; \Gamma \vDash v \colon \exists \overrightarrow{\alpha^{-}}.\, P \quad \Theta, \overrightarrow{\alpha^{-}}; \Gamma, x : P \vDash c \colon N \quad \Theta \vdash N}{\Theta; \Gamma \vDash \mathtt{let}^{\exists}(\overrightarrow{\alpha^{-}}, x) = v;\ c \colon N}\ (\textsc{let}_{\exists}^{\textsc{inf}})$$

$$\frac{\Theta; \Gamma \vDash v \colon P}{\Theta; \Gamma \vDash \mathtt{return}\, v \colon \uparrow P}\ (\textsc{ret}^{\textsc{inf}})$$

$$\frac{\Theta \vdash P \quad \Theta; \Gamma \vDash v \colon \downarrow M \quad \Theta\,; \Gamma\,; \cdot \vDash M \bullet \vec{v} \Longrightarrow M' \dashv \Xi\,; C_1 \quad \Theta; \Xi \vDash M' \leqslant \uparrow P \dashv C_2 \quad \Xi \vdash C_1 \,\&\, C_2 = C \quad \Theta; \Gamma, x : P \vDash c \colon N}{\Theta; \Gamma \vDash \mathtt{let}\, x : P = v(\vec{v});\ c \colon N}\ (\textsc{let}_{:@}^{\textsc{inf}})$$

$$\frac{\Theta; \Gamma \vDash v \colon \downarrow M \quad \Theta\,; \Gamma\,; \cdot \vDash M \bullet \vec{v} \Longrightarrow \uparrow Q \dashv \Xi\,; C \quad Q \text{ is } C\text{-minimized by } \widehat{\sigma} \quad \Theta; \Gamma, x : [\widehat{\sigma}]\, Q \vDash c \colon N}{\Theta; \Gamma \vDash \mathtt{let}\, x = v(\vec{v});\ c \colon N}\ (\textsc{let}_{@}^{\textsc{inf}})$$

$\boxed{\Theta\,; \Gamma\,; \Xi_1 \vDash N \bullet \vec{v} \Longrightarrow M \dashv \Xi_2\,; C}$ Application typing

$$\frac{}{\Theta\,; \Gamma\,; \Xi \vDash N \bullet \Longrightarrow \mathsf{nf}\,(N) \dashv \Xi\,; \cdot}\ (\emptyset_{\bullet\Rightarrow}^{\textsc{inf}}) \qquad \frac{\Theta; \Gamma \vDash v \colon P \quad \Theta; \Xi \vDash Q \geqslant P \dashv C_1 \quad \Theta\,; \Gamma\,; \Xi \vDash N \bullet \vec{v} \Longrightarrow M \dashv \Xi'\,; C_2 \quad \Xi \vdash C_1 \,\&\, C_2 = C}{\Theta\,; \Gamma\,; \Xi \vDash Q \to N \bullet v, \vec{v} \Longrightarrow M \dashv \Xi'\,; C}\ (\to_{\bullet\Rightarrow}^{\textsc{inf}})$$

$$\frac{\Theta\,; \Gamma\,; \Xi, \overrightarrow{\widehat{\alpha^{+}}}\{\Theta\} \vDash [\overrightarrow{\widehat{\alpha^{+}}}/\overrightarrow{\alpha^{+}}]\, N \bullet \vec{v} \Longrightarrow M \dashv \Xi'\,; C \quad \overrightarrow{\widehat{\alpha^{+}}} \text{ are fresh} \quad \vec{v} \neq \quad \overrightarrow{\alpha^{+}} \neq \cdot}{\Theta\,; \Gamma\,; \Xi \vDash \forall \overrightarrow{\alpha^{+}}.\, N \bullet \vec{v} \Longrightarrow M \dashv \Xi'\,; C|_{\mathsf{fav}(N) \cup \mathsf{fav}(M)}}\ (\forall_{\bullet\Rightarrow}^{\textsc{inf}})$$

Fig. 21. Algorithmic Type Inference

Then premise $\Theta; \Xi \vDash M' \leqslant \uparrow P \dashv C_2$ together with $\Xi \vdash C_1 \,\&\, C_2 = C$ check that M' can be instantiated to the annotated type $\uparrow P$, and if it is, the algorithm infers the type of the body in the extended context, and returns it as the result.

Rule ($\textsc{let}^{\textsc{inf}}_{@}$) works similarly to ($\textsc{let}^{\textsc{inf}}_{:@}$). However, since no annotation is given, the algorithm must ensure that the inferred Q has the 'canonical' minimal instantiation. To find it, it makes a call to the minimal instantiation algorithm (Section 4.9) that finds the substitution that satisfies the inferred constraints C and instantiates Q to the minimal (among other such instantiations) type $[\widehat{\sigma}]\,Q$.

Application to an Empty List of Arguments. Rule ($\emptyset^{\textsc{inf}}_{\bullet\Rightarrow}$) is the base case of application inference. If the list of applied arguments is empty, the inferred type is the type of the head, and the algorithm returns it after normalizing.

Application of a Polymorphic Type $\forall$. Rule ($\forall^{\textsc{inf}}_{\bullet\Rightarrow}$), analogously to the declarative case, is the rule ensuring the implicit elimination of the universal quantifiers. This is the place where the algorithmic variables are introduced. The algorithm simply replaces the quantified variables $\overrightarrow{\alpha^+}$ with fresh algorithmic variables $\overrightarrow{\widehat{\alpha}^+}$, and makes a recursive call in the extended context.

To ensure the rule precedence, we also require the head type to have at least one $\forall$-quantifier, and the list of arguments to be non-empty.

Application of an Arrow Type. Rule ($\rightarrow^{\textsc{inf}}_{\bullet\Rightarrow}$) is the main rule of algorithmic application inference. It is applied when the head has an arrow type $Q \rightarrow N$. First, it infers the type of the first argument v, and then, calling the algorithmic subtyping, finds C_1—the *minimal* constraint ensuring that Q is a supertype of the type of v. Then it makes a recursive call applying N to the rest of the arguments and merges the resulting constraint with C_1.

4.9 Minimal Instantiation and Constraint Singularity

Multiple types M can be inferred for a type application: $\Theta\,;\Gamma \vdash N \bullet \overrightarrow{v} \Longrightarrow M$ but only one *principal* type should be chosen for a variable in an unannotated let binder. Declaratively, we require the principal type P to be minimal among other all types P' that can be inferred for the application $\Theta\,;\Gamma \vdash N \bullet \overrightarrow{v} \Longrightarrow \uparrow P'$. Algorithmically, the inference returns an *algorithmic* type P together with a set of *necessary and sufficient* constraints C that any instantiation of P must satisfy. This way, the principal type is the minimal instantiation of P, obtained by a substitution satisfying the given constraints C.

To find this substitution, we use the minimal instantiation algorithm (Fig. 22). The judgment 'P is C-minimized by $\widehat{\sigma}$' $\widehat{\sigma}$ instantiates P to a subtype of any other instantiation of P that satisfies C. First, it removes the existential quantifiers by ($\exists^{\textsc{min}}$). Then it considers two cases.

(1) If the type P is an algorithmic variable $\widehat{\alpha}^+$ restricted by a *subtyping* constraint $(\widehat{\alpha}^+ :\geqslant Q) \in C$, its minimal instantiation is the type $\mathsf{nf}\,(Q)$, and ($\textsc{uVar}^{\textsc{min}}$) is applied.

(2) Otherwise, the type P is either a shifted computation, a declarative variable, an unrestricted algorithmic variable, or an algorithmic variable restricted by an *equivalence* constraint. In all of these cases, the minimal instantiation exists if and only if there is only one possible instantiation of P that satisfies C.
The algorithm finds this instantiation by two premises: (i) checking that all the algorithmic variables of the type are restricted by the constraint set; and (ii) building the substitution that satisfies the constraint set on these variables (and simultaneously checking that such substitution is unique up to equivalence) using the *constraint singularity algorithm*.

$$\frac{(\widehat{\alpha}^{+} :\geqslant P) \in C}{\widehat{\alpha}^{+} \text{ is } C\text{-minimized by } (\mathtt{nf}\,(P)/\widehat{\alpha}^{+})}\ (\text{UVAR}^{\text{MIN}}) \qquad \frac{\mathtt{fav}(P) \subseteq \mathtt{dom}\,(C) \quad C|_{\mathtt{fav}(P)}\ \mathtt{singular\ with}\ \widehat{\sigma}}{P \text{ is } C\text{-minimized by } \widehat{\sigma}}\ (\text{SING}^{\text{MIN}})$$

$$\frac{P \text{ is } C\text{-minimized by } \widehat{\sigma}}{\exists\overrightarrow{\alpha^{-}}.\ P \text{ is } C\text{-minimized by } \widehat{\sigma}}\ (\exists^{\text{MIN}})$$

Fig. 22. Minimal Instantiation

'$C\ \mathtt{singular\ with}\ \widehat{\sigma}$' means:

+ for any *positive* constraint entry $e \in C$ restricting a variable $\widehat{\beta}^{+}$, there exists P such that $e\ \mathtt{singular\ with}\ P$ (as defined in Fig. 24), and $[\widehat{\sigma}]\widehat{\beta}^{+} = P$; and
− the symmetric property holds for all *negative* $e \in C$; and
∉ for any $\widehat{\alpha}^{\pm} \notin \mathtt{dom}\,(C)$, $[\widehat{\sigma}]\widehat{\alpha}^{\pm} = \widehat{\alpha}^{\pm}$.

Fig. 23. Singular Constraint

$$\frac{}{\widehat{\alpha}^{+} :\simeq P\ \mathtt{singular\ with\ nf}\,(P)}\ (\simeq_{+}^{\text{SING}}) \qquad \frac{}{\widehat{\alpha}^{-} :\simeq N\ \mathtt{singular\ with\ nf}\,(N)}\ (\simeq_{-}^{\text{SING}})$$

$$\frac{}{\widehat{\alpha}^{+} :\geqslant \exists\overrightarrow{\alpha^{-}}.\ \alpha^{+}\ \mathtt{singular\ with}\ \alpha^{+}}\ (:\geqslant\alpha^{\text{SING}}) \qquad \frac{\mathtt{nf}\,(N) = \alpha_i^{-} \in \overrightarrow{\alpha^{-}}}{\widehat{\alpha}^{+} :\geqslant \exists\overrightarrow{\alpha^{-}}.\ {\downarrow}N\ \mathtt{singular\ with}\ \exists\beta^{-}.\ {\downarrow}\beta^{-}}\ (:\geqslant{\downarrow}^{\text{SING}})$$

Fig. 24. Singular Constraint Entry

The singularity algorithm performs two tasks: it checks that a *constraint set* has a single substitution satisfying it, and if it does, it builds this substitution.

To implement the singularity algorithm, we define a partial function '$C\ \mathtt{singular\ with}\ \widehat{\sigma}$', taking a subtyping constraint C as an argument and returning a substitution $\widehat{\sigma}$—the only solution of C.

The constraint C is composed of constraint entries. Therefore, we define singularity by combining the singularity of each constraint entry (Fig. 23). In order for C to be singular, each entry must have a unique instantiation, and the resulting substitution $\widehat{\sigma}$ must be a union of these instantiations. In addition, $\widehat{\sigma}$ must act as an identity on the variables not restricted by C.

The singularity of constraint *entries* is defined in Fig. 24.

- The *equivalence* entries are always singular, as the only possible type satisfying them is the one given in the constraint itself, which is reflected in rules $(\simeq_{-}^{\text{SING}})$ and $(\simeq_{+}^{\text{SING}})$.
- The *subtyping* constraints are trickier. As will be discussed in Section 5.3, variables (and equivalent them $\exists\overrightarrow{\alpha^{-}}.\ \alpha^{+}$) do not have proper supertypes, and thus, the constraints of a form $\widehat{\alpha}^{+} :\geqslant \exists\overrightarrow{\alpha^{-}}.\ \alpha^{+}$ are singular with the only possible normalized solution α^{+}—see rule $(:\geqslant\alpha^{\text{SING}})$.
- However, if the body of the existential type is guarded by a shift $\exists\overrightarrow{\alpha^{-}}.\ {\downarrow}N$, it is singular if and only if N is equivalent to some $\alpha_i^{-} \in \overrightarrow{\alpha^{-}}$ bound by the quantifier—see rule $(:\geqslant{\downarrow}^{\text{SING}})$. The completeness of this criterion is justified by the fact that if N is *not* equivalent to any α_i^{-}, then there are two non-equivalent solutions of constraint $(\widehat{\alpha}^{+} :\geqslant \exists\overrightarrow{\alpha^{-}}.\ {\downarrow}N)$: the trivial $\exists\overrightarrow{\alpha^{-}}.\ {\downarrow}N$ and $\exists\overrightarrow{\alpha^{-}}.\ {\downarrow}\alpha_1^{-}$ (which is a supertype of $\exists\overrightarrow{\alpha^{-}}.\ {\downarrow}N$ since α_1^{-} can be instantiated to N).

5 ALGORITHM CORRECTNESS

The central results ensuring the correctness of the inference algorithm are its soundness and completeness with respect to the declarative specification. The soundness means the algorithm

will always produce a typing *allowed* by the declarative system; dually, the completeness says that once a term has some type declaratively, the inference algorithm succeeds.

The formal statements of soundness and completeness are given in the theorems below. Notice that the theorems also include the soundness and completeness of *application inference* (labeled as •), which is more complex. As such, let us discuss it in more detail.

Both soundness and completeness of application inference assume that the input head type N is free from *negative* algorithmic variables—it is achieved by polarization invariants preserved by the inference rules. The soundness states that the output of the algorithm—M and C—is viable. Specifically, that the constraint set C provides a sufficient set of restrictions that any substitution $\widehat{\sigma}$ must satisfy to ensure the *declarative* inference of the output type M, that is $\Theta\,;\Gamma \vdash [\widehat{\sigma}]N \bullet \vec{v} \Longrightarrow [\widehat{\sigma}]M$.

The application inference completeness means that if there exists a substitution $\widehat{\sigma}$ and the resulting type M ensuring the declarative inference $\Theta\,;\Gamma \vdash [\widehat{\sigma}]N \bullet \vec{v} \Longrightarrow M$ then the algorithm succeeds and gives the most general result M_0 and C_0. The property of 'being the most general' is specified in pt. (2). Intuitively, it means that for any other solution—substitution $\widehat{\sigma}$ and the resulting type M, if it ensures the declarative inference, then $\widehat{\sigma}$ can be extended in a C_0-complying way to equate M_0 with M.

Theorem (Soundness of Typing). *Suppose that* $\Theta \vdash \Gamma$. *Then*[1]

- $+$ $\Theta;\Gamma \vDash v\colon P$ *implies* $\Theta;\Gamma \vdash v\colon P$,
- $-$ $\Theta;\Gamma \vDash c\colon N$ *implies* $\Theta;\Gamma \vdash c\colon N$,
- • $\Theta\,;\Gamma\,;\Xi \vDash N \bullet \vec{v} \Longrightarrow M \dashv \Xi'\,;C$ *implies* $\Theta\,;\Gamma \vdash [\widehat{\sigma}]N \bullet \vec{v} \Longrightarrow [\widehat{\sigma}]M$, *for any instantiation of* $\widehat{\sigma}$ *satisfying constraints* C. *All of it under assumptions that* $\Theta \vdash^{\supseteq} \Xi$ *and* $\Theta\,;\mathsf{dom}\,(\Xi) \vdash N$ *and that* N *is free from* negative *algorithmic variables.*

Theorem (Completeness of Typing). *Suppose that* $\Theta \vdash \Gamma$. *Then*[1]

- $+$ $\Theta;\Gamma \vdash v\colon P$ *implies* $\Theta;\Gamma \vDash v\colon \mathsf{nf}\,(P)$,
- $-$ $\Theta;\Gamma \vdash c\colon N$ *implies* $\Theta;\Gamma \vDash c\colon \mathsf{nf}\,(N)$,
- • *If* $\Theta\,;\Gamma \vdash [\widehat{\sigma}]N \bullet \vec{v} \Longrightarrow M$ *where (1)* $\Theta \vdash^{\supseteq} \Xi$, *(2)* $\Theta \vdash M$, *(3)* $\Theta\,;\mathsf{dom}\,(\Xi) \vdash N$ *(free from* negative *algorithmic variables), and (4)* $\Xi \vdash \widehat{\sigma} : \mathsf{fav}(N)$, *then there exist* M_0, Ξ_0, *and* C_0 *such that*
  - *(1)* $\Theta\,;\Gamma\,;\Xi \vDash N \bullet \vec{v} \Longrightarrow M_0 \dashv \Xi_0\,;C_0$ *and*
  - *(2) for any other* $\widehat{\sigma}$ *and* M *(where* $\Xi \vdash \widehat{\sigma} : \mathsf{fav}(N)$ *and* $\Theta \vdash M$*) such that* $\Theta\,;\Gamma \vdash [\widehat{\sigma}]N \bullet \vec{v} \Longrightarrow M$, *there exists* $\widehat{\sigma}'$ *such that (a)* $\Xi_0 \vdash \widehat{\sigma}' : \mathsf{fav}N \cup \mathsf{fav}M_0$ *and* $\Xi_0 \vdash \widehat{\sigma}' : C_0$, *(b)* $\Xi \vdash \widehat{\sigma}' \simeq^{\leqslant} \widehat{\sigma} : \mathsf{fav}N$, *and (c)* $\Theta \vdash [\widehat{\sigma}']M_0 \simeq^{\leqslant} M$.

The proof of soundness and completeness result is done gradually for all the subroutines, following the structure of the algorithm (Figs. 13 and 20) bottom-up. Next, we discuss the main of these results.

5.1 Normalization

The point of type normalization is factoring out non-trivial equivalence by selecting a representative from each equivalence class. This way, *the correctness of normalization* means that checking for equivalence of two types is the same as checking for equality of their normal forms.

Lemma (Normalization correctness). *Assuming all types are well-formed in* Θ, *we have* $\Theta \vdash N \simeq^{\leqslant} M \iff \mathsf{nf}\,(N) = \mathsf{nf}\,(M)$ *and* $\Theta \vdash P \simeq^{\leqslant} Q \iff \mathsf{nf}\,(P) = \mathsf{nf}\,(Q)$.

[1]The presented properties hold, but the actual inductive proof requires strengthening of the statement and the corresponding theorem is more involved. See the appendix (??) for details.

To prove the correctness of normalization, we introduce an *intermediate* relation on types—*declarative equivalence* (the notation is $N \simeq^D M$ and $P \simeq^D Q$). In contrast to $\Theta \vdash N \simeq^{\leqslant} M$ (which means mutual subtyping), $N \simeq^D M$ does not depend on subtyping judgments, but explicitly allows quantifier reordering and redundant quantifier removal. Then the statement $\Theta \vdash N \simeq^{\leqslant} M \iff \mathsf{nf}(N) = \mathsf{nf}(M)$ (as well as its positive counterpart) is split into two steps: $\Theta \vdash N \simeq^{\leqslant} M \iff N \simeq^D M \iff \mathsf{nf}(N) = \mathsf{nf}(M)$.

5.2 Anti-Unification

The anti-unifier of the two types is the most specific pattern that matches against both of them. In our setting, the anti-unifiers are restricted further: first, the pattern might only contain 'holes' at *negative* positions (because eventually, the 'holes' become variables abstracted by the existential quantifier); second, the anti-unification is parametrized with a context Θ, in which the pattern instantiations must be well-formed.

This way, the correctness properties of the anti-unification algorithm are refined accordingly. The soundness of anti-unification not only ensures that the resulting pattern matches with the input types, but also that the pattern instantiations are well-formed in the corresponding context, and that all the 'hole' variables are negative. In turn, completeness states that if there exists a solution satisfying the soundness criteria, then the algorithm succeeds.

The correctness properties are formulated by the following lemmas. For brevity, we only provide the statements for the positive case, since the negative case is symmetric.

Lemma (Soundness of (positive) anti-unification)**.** *Assuming P_1 and P_2 are normalized, if $\Theta \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\widehat{\Theta}, Q, \widehat{\tau}_1, \widehat{\tau}_2)$ then (1) $\Theta ; \widehat{\Theta} \vdash Q$, (2) $\Theta ; \cdot \vdash \widehat{\tau}_i : \widehat{\Theta}$ for $i \in \{1, 2\}$ are anti-unification substitutions (in particular, $\widehat{\Theta}$ contains only negative algorithmic variables), and (3) $[\widehat{\tau}_i] Q = P_i$ for $i \in \{1, 2\}$.*

Lemma (Completeness of (positive) anti-unification)**.** *Assuming that P_1 and P_2 are normalized terms well-formed in Θ and there exist $(\widehat{\Theta}', Q', \widehat{\tau}_1', \widehat{\tau}_2')$ such that (1) $\Theta ; \widehat{\Theta}' \vdash Q'$, (2) $\Theta ; \cdot \vdash \widehat{\tau}_i' : \widehat{\Theta}'$ for $i \in \{1, 2\}$ are anti-unification substitutions, and (3) $[\widehat{\tau}_i'] Q' = P_i$ for $i \in \{1, 2\}$.*

Then the anti-unification algorithm terminates, that is there exists $(\widehat{\Theta}, Q, \widehat{\tau}_1, \widehat{\tau}_2)$ such that $\Theta \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\widehat{\Theta}, Q, \widehat{\tau}_1, \widehat{\tau}_2)$.

Notice that for the anti-unification substitution $\widehat{\tau}$ we use a separate signature specifying the domain and the range. $\Theta ; \widehat{\Theta}_2 \vdash \widehat{\tau} : \widehat{\Theta}_1$ means that $\widehat{\tau}$ maps the 'holes' (i.e., algorithmic variables) from $\widehat{\Theta}_1$ to *algorithmic* types well-formed in Θ and $\widehat{\Theta}_2$. To put it formally, $\Theta ; \widehat{\Theta}_2 \vdash [\widehat{\tau}]\widehat{\alpha}^-$ for any $\widehat{\alpha}^- \in \widehat{\Theta}_1$. Although in the formulation of soundness and completeness, the resulting types are declarative (i.e., $\widehat{\Theta}_2$ is always empty), we need the possibility of mapping the 'holes' to types with 'holes' to formulate the *initiality* of anti-unification.

The initiality shows that the anti-unifier that the algorithm provides is the most specific (or the most 'detailed'). Specifically, it means that any other sound anti-unification solution can be 'refined' to the result of the algorithm. The 'refinement' is represented as an instantiation of the anti-unifier—a substitution $\Theta ; \widehat{\Theta}_2 \vdash \widehat{\rho} : \widehat{\Theta}_1$ replacing the 'holes' from $\widehat{\Theta}_1$ with types that themselves can contain 'holes' from $\widehat{\Theta}_2$.

Lemma (Initiality of Anti-Unification)**.** *Assume that P_1 and P_2 are normalized types well-formed in Θ, and the anti-unification algorithm succeeds: $\Theta \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\widehat{\Theta}, Q, \widehat{\tau}_1, \widehat{\tau}_2)$. Then $(\widehat{\Theta}, Q, \widehat{\tau}_1, \widehat{\tau}_2)$ is more specific than any other sound anti-unifier $(\widehat{\Theta}', Q', \widehat{\tau}_1', \widehat{\tau}_2')$, i.e., if (1) $\Theta ; \widehat{\Theta}' \vdash Q'$, (2) $\Theta ; \cdot \vdash \widehat{\tau}_i' : \widehat{\Theta}'$ for $i \in \{1, 2\}$, and (3) $[\widehat{\tau}_i'] Q' = P_i$ for $i \in \{1, 2\}$ then there exists a 'refining' substitution $\widehat{\rho}$ such that $\Theta ; \widehat{\Theta} \vdash \widehat{\rho} : (\widehat{\Theta}'|_{\mathsf{fav} Q'})$ and $[\widehat{\rho}] Q' = Q$.*

To prove the correctness properties of the anti-unification algorithm, one extra observation is essential. The algorithm relies on the fact that in the resulting tuple $(\widehat{\Theta}, Q, \widehat{\tau}_1, \widehat{\tau}_2)$, there are no two different 'holes' $\widehat{\beta}^-$ mapped to the same pair of types by $\widehat{\tau}_1$ and $\widehat{\tau}_2$. This is used to ensure that, for example, the anti-unifier of $\downarrow\uparrow\mathtt{Int} \to \uparrow\mathtt{Int}$ and $\downarrow\uparrow\mathtt{Bool} \to \uparrow\mathtt{Bool}$ is $\downarrow\widehat{\alpha}^- \to \widehat{\alpha}^-$, but not (less specific) $\downarrow\widehat{\alpha}^- \to \widehat{\beta}^-$. To preserve this property, we arrange the algorithm in such a way that the name of the 'hole' is determined by the types it is mapped to. The following lemma specifies this observation.

Lemma (Uniqueness of Anti-unification Variable Names). *Names of the anti-unification variables are uniquely defined by the types they are mapped to by the resulting substitutions. Assuming P_1 and P_2 are normalized, if $\Theta \vDash P_1 \stackrel{a}{\cong} P_2 \dashv (\widehat{\Theta}, Q, \widehat{\tau}_1, \widehat{\tau}_2)$ then for any $\widehat{\beta}^- \in \widehat{\Theta}$, $\widehat{\beta}^- = \widehat{\alpha}^-_{\{[\widehat{\tau}_1]\widehat{\beta}^-, [\widehat{\tau}_2]\widehat{\beta}^-\}}$.*

5.3 Least Upper Bound and Upgrade

The Least Upper Bound algorithm finds the least type that is a supertype of two given types. Its *soundness* means that the returned type is indeed a supertype of the given ones; the *completeness* means that the algorithm succeeds if the least upper bound exists; and the *initiality* means that the returned type is the least among common supertypes.

Lemma (Least Upper Bound soundness). *For types $\Theta \vdash P_1$, and $\Theta \vdash P_2$, if $\Theta \vDash P_1 \vee P_2 = Q$ then*

(i) $\Theta \vdash Q$
(ii) $\Theta \vdash Q \geqslant P_1$ *and* $\Theta \vdash Q \geqslant P_2$

Lemma (Least Upper Bound completeness and initiality). *For types $\Theta \vdash P_1$, $\Theta \vdash P_2$, and $\Theta \vdash Q$ such that $\Theta \vdash Q \geqslant P_1$ and $\Theta \vdash Q \geqslant P_2$, there exists Q' s.t. $\Theta \vDash P_1 \vee P_2 = Q'$ and $\Theta \vdash Q \geqslant Q'$.*

The key observation that allows us to prove these properties is the characterization of positive supertypes. The following lemma justifies the case analysis used in the Least Upper Bound algorithm (in Section 4.6). In particular, it establishes the correspondence between the upper bounds of shifted types $\downarrow M$ and *patterns* fitting M (represented by existential types), which explains the usage of anti-unification as a way to find a common pattern.

Lemma (Characterization of normalized supertypes). *For a normalized positive type P well-formed in Θ, the set of normalized Θ-formed supertypes of P is the following:*

- *if P is a variable β^+, its only normalized supertype is β^+ itself;*
- *if P is an existential type $\exists\overrightarrow{\beta^-}.\, P'$ then its Θ-formed supertypes are the $(\Theta, \overrightarrow{\beta^-})$-formed supertypes of P' not using $\overrightarrow{\beta^-}$;*
- *if P is a downshift type $\downarrow M$, its supertypes have form $\exists\overrightarrow{\alpha^-}.\, \downarrow M'$ such that there exists a Θ-formed instantiation of $\overrightarrow{\alpha^-}$ in $\downarrow M'$ that makes $\downarrow M'$ equal to $\downarrow M$, i.e., $[\vec{N}/\overrightarrow{\alpha^-}]\downarrow M' = \downarrow M$.*

Similarly to the Least Upper Bound algorithm, the Upgrade finds the least type among upper bounds (this time the set of considered upper bounds consists of supertypes well-formed in a *smaller* context). This way, we also use the supertype characterization to prove the following properties of the Upgrade algorithm.

Lemma (Upgrade soundness). *Assuming P is well-formed in $\Theta = \Theta_0, \overrightarrow{\alpha^{\pm}}$, if $\mathtt{upgrade}\ \Theta \vdash P\ \mathtt{to}\ \Theta_0 = Q$ then (1) $\Theta_0 \vdash Q$ and (2) $\Theta \vdash Q \geqslant P$.*

Lemma (Upgrade Completeness). *Assuming P is well-formed in $\Theta = \Theta_0, \overrightarrow{\alpha^{\pm}}$, for any Q' such that Q' is a Θ_0-formed upper bound of P, i.e., (1) $\Theta_0 \vdash Q'$ and (2) $\Theta \vdash Q' \geqslant P$, there exists Q such that ($\mathtt{upgrade}\ \Theta \vdash P\ \mathtt{to}\ \Theta_0 = Q$) and $\Theta_0 \vdash Q' \geqslant Q$.*

5.4 Subtyping

As for other properties, the correctness of subtyping means that the algorithm produces a valid result (soundness) whenever it exists (completeness). In the rules defining the subtyping algorithm (Fig. 14), one can see that the positive and negative subtyping relations are not *mututally* recursive: negative subtyping algorithm uses the positive subtyping, but not vice versa. Because of that, the inductive proofs of the *positive* subtyping correctness are done independently.

The soundness of positive subtyping states that the output constraint C provides a sufficient set of restrictions. In other words, any substitution satisfying C ensures the desired declarative subtyping: $\Theta \vdash [\widehat{\sigma}]P \geqslant Q$. Notice that the soundness formulation assumes that only the left-hand side input type (P) can contain algorithmic variables. This is one of the invariants of the algorithm that significantly simplifies the unification and constraint resolution.

Lemma (Soundness of Positive Subtyping). *If* $\Theta \vdash^{\supseteq} \Xi$ *and* $\Theta \vdash Q$ *and* $\Theta\,;\mathsf{dom}(\Xi) \vdash P$ *and* $\Theta; \Xi \vDash P \geqslant Q \dashv C$, *then* $\Xi \vdash C : \mathsf{fav} P$ *and for any* $\widehat{\sigma}$ *such that* $\Xi \vdash \widehat{\sigma} : C$, *we have* $\Theta \vdash [\widehat{\sigma}]P \geqslant Q$.

The completeness of subtyping says that if the substitution ensuring the declarative subtyping exists, then the subtyping algorithm succeeds (terminates).

Lemma (Completeness of Positive Subtyping). *Suppose that* $\Theta \vdash^{\supseteq} \Xi$, $\Theta \vdash Q$ *and* $\Theta\,;\mathsf{dom}(\Xi) \vdash P$. *Then if there exists* $\widehat{\sigma}$ *such that* $\Xi \vdash \widehat{\sigma} : \mathsf{fav}(P)$ *and* $\Theta \vdash [\widehat{\sigma}]P \geqslant Q$, *then the subtyping algorithm succeeds:* $\Theta; \Xi \vDash P \geqslant Q \dashv C$.

After the correctness properties of positive subtyping are established, they are used to prove the correctness of the negative subtyping. The soundness is formulated symmetrical to the positive case, however, the completeness requires an additional invariant to be preserved. The algorithmic input type N must be free from *negative* algorithmic variables. In particular, it ensures that the constraint restricting a *negative* algorithmic variable will never be generated, and thus, we do not need the Greatest Common Subtype procedure to resolve the constraints.

Lemma (Completeness of Negative Subtyping). *Suppose that* $\Theta \vdash^{\supseteq} \Xi$ *and* $\Theta \vdash M$ *and* $\Theta;\mathsf{dom}(\Xi) \vdash N$ *and* N *does not contain negative unification variables* ($\widehat{\alpha}^- \notin \mathsf{fav} N$). *Then for any* $\Xi \vdash \widehat{\sigma} : \mathsf{fav}(N)$ *such that* $\Theta \vdash [\widehat{\sigma}]N \leqslant M$, *the subtyping algorithm succeeds:* $\Theta; \Xi \vDash N \leqslant M \dashv C$.

5.5 Minimal Instantiation and Singularity

Algorithmic typing relies on the *minimal instantiation* procedure. For a given positive type P and a set of constraints C it returns a substitution $\Xi \vdash \widehat{\sigma} : C$ such that $[\widehat{\sigma}]P$ is a subtype of all the other instantiations of P satisfying C. This way, the correctness of minimal instantiation is formulated as the following two lemmas.

Lemma (Soundness of Minimal Instantiation). *Suppose that* $\Theta \vdash^{\supseteq} \Xi$, $\Xi \vdash C$, *and* $\Theta\,;\mathsf{dom}(\Xi) \vdash P$. *Then '*$P$ *is* C*-minimized by* $\widehat{\sigma}$*' implies that* $\Xi \vdash \widehat{\sigma} : \mathsf{fav} P$ *is a normalized substitution satisfying* C *(i.e.,* $\Xi \vdash \widehat{\sigma} : C$*) and for any other substitution* $\Xi \vdash \widehat{\sigma}' : \mathsf{fav} P$ *satisfying* C, *we have* $\Theta \vdash [\widehat{\sigma}']P \geqslant [\widehat{\sigma}]P$.

Lemma (Completeness of Minimal Instantiation). *Suppose that* $\Theta \vdash^{\supseteq} \Xi, \Xi \vdash C, \Theta\,;\mathsf{dom}(\Xi) \vdash P$, *and there exists* $\Xi \vdash \widehat{\sigma} : \mathsf{fav} P$ *satisfying* C ($\Xi \vdash \widehat{\sigma} : C$) *such that for any other* $\Xi \vdash \widehat{\sigma}' : \mathsf{fav} P$ *satisfying* C, *we have* $\Theta \vdash [\widehat{\sigma}']P \geqslant [\widehat{\sigma}]P$. *Then the minimal instantiation procedure succeeds; specifically,* P *is* C*-minimized by* $\mathsf{nf}(\widehat{\sigma})$ *holds.*

The soundness is rather straightforwardly proved by induction on the judgment inference tree, relying on the soundness of the singularity procedure. The proof of completeness is done by induction on the structure of the type P. It uses the fact that $\Theta \vdash \exists\overrightarrow{\alpha}.\,[\widehat{\sigma}']P \geqslant \exists\overrightarrow{\alpha}.\,[\widehat{\sigma}]P$ implies $\Theta, \overrightarrow{\alpha} \vdash [\widehat{\sigma}']P \geqslant [\widehat{\sigma}]P$.

The soundness of *singularity* states that C `singular with` $\widehat{\sigma}$ implies that any substitution satisfying C is equivalent to $\widehat{\sigma}$ on the domain. The completeness states that if all the C-compliant substitutions are equivalent, then the singularity procedure succeeds.

Lemma (Soundness of Singularity). *Suppose* $\Xi \vdash C : \widehat{\Theta}$, *and* C `singular with` $\widehat{\sigma}$. *Then* $\Xi \vdash \widehat{\sigma} : \widehat{\Theta}$, $\Xi \vdash \widehat{\sigma} : C$ *and for any* $\widehat{\sigma}'$ *such that* $\Xi \vdash \widehat{\sigma} : C$, $\Xi \vdash \widehat{\sigma}' \simeq^{\leqslant} \widehat{\sigma} : \widehat{\Theta}$.

Lemma (Completeness of Singularity). *For a given set of constraints* $\Xi \vdash C$, *suppose that all the substitutions satisfying* C *are equivalent on* $\mathsf{dom}(C)$. *In other words, suppose that there exists* $\Xi \vdash \widehat{\sigma}_1 : \mathsf{dom}(C)$ *such that for any* $\Xi \vdash \widehat{\sigma} : \mathsf{dom}(C)$, $\Xi \vdash \widehat{\sigma} : C$ *implies* $\Xi \vdash \widehat{\sigma} \simeq^{\leqslant} \widehat{\sigma}_1 : \mathsf{dom}(C)$. *Then* C `singular with` $\widehat{\sigma}_0$ *for some* $\widehat{\sigma}_0$.

5.6 Typing

Finally, we discuss the proofs of the soundness and completeness of type inference algorithm that we stated at the beginning of this section. There are three subtleties that we will cover that are important for the proof to go through: the determinacy of the algorithm, the mutual dependence of the soundness and completeness proofs, and the non-trivial inductive metric that we use.

Determinacy. One of the properties that our proof relies on is the determinacy of the typing algorithm: the output (the inferred type) is uniquely determined by the input (the term and the contexts). Determinacy is not hard to demonstrate by structural induction: in every algorithmic inference system, only one inference rule can be applied for a given input. However, we need to prove it for *every* subroutine of the algorithm. Ultimately, it requires the determinacy of such procedures as *generation of fresh variables*, which is easy to ensure, but must be taken into account in the implementation.

Lemma (Determinacy of the Typing Algorithm). *Suppose that* $\Theta \vdash \Gamma$ *and* $\Theta \vdash^{\supseteq} \Xi$. *Then*

- $+$ *If* $\Theta; \Gamma \vDash v : P$ *and* $\Theta; \Gamma \vDash v : P'$ *then* $P = P'$.
- $-$ *If* $\Theta; \Gamma \vDash c : N$ *and* $\Theta; \Gamma \vDash c : N'$ *then* $N = N'$.
- $\bullet$ *If* $\Theta \,;\, \Gamma \,;\, \Xi \vDash N \bullet \vec{v} \Longrightarrow M \dashv \Xi' \,;\, C$ *and* $\Theta \,;\, \Gamma \,;\, \Xi \vDash N \bullet \vec{v} \Longrightarrow M' \dashv \Xi' \,;\, C'$ *then* $M = M'$, $\Xi = \Xi'$, *and* $C = C'$.

Mutuality of the Soundness and Completeness Proofs. Typically in our inductive proofs, the soundness is proven before completeness, as the completeness requires the premise subtrees to satisfy certain properties (which can be given by the soundness). However, in the case of the typing algorithm, the soundness and completeness proofs cannot be separated: the inductive proof of one requires another, and vice versa.

The proof of soundness can be viewed as a mapping from an algorithmic tree to a declarative one. We show that each algorithmic inference rule can be transformed into the corresponding declarative rule, as long as the premises are transformed accordingly, and apply the induction principle.

The soundness requires completeness in the case of ($\textsc{let}^{\textsc{inf}}_{@}$). To prove the soundness, in other words, to transform this rule into its declarative counterpart ($\textsc{let}^{\textsc{inf}}_{@}$), one needs to prove the *principality* of the inferred application type $\uparrow[\widehat{\sigma}]\, Q$. In other words, we need to conclude that any other *declarative* tree infers for the application a supertype of $[\widehat{\sigma}]\, Q$.

The only way to do so is by applying the soundness of minimal instantiation (see the lemma above). However, first, the declarative tree must be converted to the corresponding algorithmic one (by completeness!). This way, the soundness and completeness proofs are mutually dependent.

Inductive Metric. The soundness and completeness lemmas are proven by *mutual* induction. Since the declarative and the algorithmic systems do not depend on each other, we must introduce

a uniform *metric* on which the induction is conducted. We define the metric gradually, starting from the auxiliary function—the size of a judgment $\mathsf{size}(J)$.

The size of *declarative* and *algorithmic* judgments is defined as a pair: the first component is the syntactic size of the terms used in the judgment. The second component depends on the kind of judgment. For regular type inference judgments (such as $\Theta;\Gamma \vdash v\colon P$ or $\Theta;\Gamma \vDash c\colon N$), it is always zero. For application inference judgments ($\Theta;\Gamma \vdash N \bullet \vec{v} \Longrightarrow M$ or $\Theta;\Gamma;\Xi \vDash N \bullet \vec{v} \Longrightarrow M \dashv \Xi';C$), it is equal to the number of prenex quantifiers of the head type N. We need this adjustment to ensure the monotonicity of the metric, since the rules ($\forall^{\mathrm{INF}}_{\bullet\Rightarrow}$) and ($\forall^{\mathrm{INF}}_{\bullet\Rightarrow}$) only reduce the quantifiers in the head type but they do not change the list of arguments.

Definition (Judgement Size). *For a declarative or an algorithmic typing judgment J, we define a metric* $\mathsf{size}(J)$ *as a pair of integers in the following way:*

- $+\ \mathsf{size}(\Theta;\Gamma \vdash v\colon P) = (\mathsf{size}(v), 0)$;
- $-\ \mathsf{size}(\Theta;\Gamma \vdash c\colon N) = (\mathsf{size}(c), 0)$;
- $\bullet\ \mathsf{size}(\Theta;\Gamma \vdash N \bullet \vec{v} \Longrightarrow M) = (\mathsf{size}(\vec{v}), \mathsf{npq}(N))$;
- $+\ \mathsf{size}(\Theta;\Gamma \vDash v\colon P) = (\mathsf{size}(v), 0)$;
- $-\ \mathsf{size}(\Theta;\Gamma \vDash c\colon N) = (\mathsf{size}(c), 0)$;
- $\bullet\ \mathsf{size}(\Theta;\Gamma;\Xi \vDash N \bullet \vec{v} \Longrightarrow M \dashv \Xi';C) = (\mathsf{size}(\vec{v}), \mathsf{npq}(N))$.

Here $\mathsf{size}(v)$ *and* $\mathsf{size}(c)$ *is the size of the syntax tree of the term, and* $\mathsf{size}(\vec{v})$ *is the sum of sizes of the terms in* $\vec{v}$*; and* $\mathsf{npq}(N)$ *and* $\mathsf{npq}(P)$ *represent the number of prenex quantifiers, i.e.,*

$$+\ \mathsf{npq}(\exists\overrightarrow{\alpha^-}.P) = |\overrightarrow{\alpha^-}|, \text{ if } P \neq \exists\overrightarrow{\beta^-}.P',$$
$$-\ \mathsf{npq}(\forall\overrightarrow{\alpha^+}.N) = |\overrightarrow{\alpha^+}|, \text{ if } N \neq \forall\overrightarrow{\beta^+}.N'.$$

Notice that for *algorithmic* inference system, $\mathsf{size}(J)$ decreases in all the inductive steps, i.e., for each inference rule, the size of the premise judgments is strictly less than the size of the conclusion. However, the *declarative* inference system has rules ($\simeq^{\mathrm{INF}}_{-}$) and ($\simeq^{\mathrm{INF}}_{+}$), that 'step to' an equivalent type, and thus, technically, might keep the judgment unchanged altogether.

To deal with this issue, we introduce the metric on the entire *inference trees* rather than on judgments, and plug into this metric the parameter that certainly decreases in rules ($\simeq^{\mathrm{INF}}_{-}$) and ($\simeq^{\mathrm{INF}}_{+}$)—the number of such nodes in the inference tree. We denote this number as $\mathsf{eq\_nodes}(T)$. Then the final metric is defined as a pair in the following way.

Definition (Inference Tree Metric). *For a tree T, inferring a declarative or an algorithmic judgement J, we define* $\mathsf{metric}(T)$ *as follows:*

$$\mathsf{metric}(T) = \begin{cases} (\mathsf{size}(J), \mathsf{eq\_nodes}(T)) & \text{if } J \text{ represents a declarative judgement,} \\ (\mathsf{size}(J), 0) & \text{if } J \text{ represents an algorithmic judgement.} \end{cases}$$

Here $\mathsf{eq\_nodes}(T)$ *is the number of nodes in T labeled with* ($\simeq^{INF}_{+}$) *or* ($\simeq^{INF}_{-}$).

This metric is suitable for mutual induction on the soundness and completeness of the typing algorithm. First, it is monotonous w.r.t. the inference rules, and this way, we can always apply the induction hypothesis to premises of each rule. Second, the induction hypothesis is powerful enough, so we can use the completeness of the algorithm in the soundness proof, where required. For instance, to prove the soundness of typing in case of $\Theta;\Gamma \vDash \mathtt{let}\ x = v(\vec{v});\ c'\colon N$, we can assume, that *completeness* holds for a term of shape $\Theta;\Gamma \vdash M \bullet \vec{v} \Longrightarrow K$, since $\mathsf{size}(args) < \mathsf{size}(letx = v(args);c')$. This is exactly what allows us to deal with the case of ($\textsc{let}^{\mathrm{INF}}_{@}$), because then we can conclude that the inferred type (of a declarative judgment $\Theta;\Gamma \vdash M \bullet \vec{v} \Longrightarrow \uparrow[\widehat{\sigma}]\,Q$ constructed by the induction hypothesis) is unique.

6 EXTENSIONS AND FUTURE WORK

In this section, we discuss several extensions to the system and the algorithm. Some of them can be incorporated into the system immediately, while others are beyond the scope of this work, and thus are left for future research.

6.1 Explicit Type Application

In our system, all type applications are inferred implicitly: the algorithm automatically instantiates the variables abstracted by $\forall$. The *explicit* type application can be added to the declarative system by the following rule:

$$\frac{\Theta; \Gamma \vdash c : \forall \overrightarrow{\alpha^+}. N}{\Theta; \Gamma \vdash c\{\overrightarrow{P}\} : [\overrightarrow{P}/\overrightarrow{\alpha^+}]N} \quad (TApp^{\text{INF}})$$

However, this rule alone would cause ambiguity. The declarative system does not fix the order of the quantifiers, which means that $\forall \alpha^+. \forall \beta^+. N$ and $\forall \beta^+. \forall \alpha^+. N$ can be inferred as a type of c interchangeably. But that would make explicit instantiation $c\{P\}$ ambiguous, as it is unclear whether α^+ or β^+ should be instantiated with P.

Solution 1: Declarative Normalization. One way to resolve this ambiguity is to fix the order of quantifiers. The algorithm already performs the ordering of the quantifiers in the normalization procedure. This way, we could require the inferred type to be normalized to specify the order of the quantifiers:

$$\frac{\Theta; \Gamma \vdash c : \forall \overrightarrow{\alpha^+}. N \quad \mathsf{nf}\,(\forall \overrightarrow{\alpha^+}. N) = \forall \overrightarrow{\alpha^+}. N}{\Theta; \Gamma \vdash c\{\overrightarrow{P}\} : [\overrightarrow{P}/\overrightarrow{\alpha^+}]N} \quad (TApp^{\text{INF}})$$

The drawback of this approach is that it adds another point where the internal algorithmic concept—normalization—is exposed to the 'surface' declarative system.

Solution 2: Elementary Type Inference. An alternative approach to provide explicit type application was proposed by [Zhao et al. 2022]. In this work, the subtyping relation is restricted in such a way that $\forall \alpha^+. \forall \beta^+. N$ and $\forall \beta^+. \forall \alpha^+. N$ are *not* mutual subtypes (as long as $\alpha^+ \in \mathsf{fv}\,(N)$ and $\beta^+ \in \mathsf{fv}\,(N)$). It implies that the order of the quantifiers of the inferred type is unique, and thus, the explicit type application is unambiguous.

These restrictions can be incorporated into our system by replacing the polymorphic subtyping rules ($\forall^{\leqslant}$) and ($\exists^{\geqslant}$) with the following stronger versions:

$$\frac{\Theta, \overrightarrow{\beta^+} \vdash \sigma : \overrightarrow{\alpha^+} \quad \Theta, \overrightarrow{\beta^+} \vdash [\sigma]N \leqslant M}{\Theta \vdash \forall \overrightarrow{\beta^+}, \overrightarrow{\alpha^+}. N \leqslant \forall \overrightarrow{\beta^+}. M} \quad (\text{E}\forall^{\leqslant}) \qquad \frac{\Theta, \overrightarrow{\beta^-} \vdash \sigma : \overrightarrow{\alpha^-} \quad \Theta, \overrightarrow{\beta^-} \vdash [\sigma]P \geqslant Q}{\Theta \vdash \exists \overrightarrow{\beta^-}, \overrightarrow{\alpha^-}. P \geqslant \exists \overrightarrow{\beta^-}. Q} \quad (\text{E}\exists^{\geqslant})$$

According to these rules, if two quantified ($\forall$ or $\exists$) types are subtypes of each other, they must have the same top-level quantifiers. Moreover, the equivalence on types (mutual subtyping) degenerates to *equality* up to alpha-conversion.

To accommodate these changes in the *algorithm*, it suffices to (i) replace the normalization procedure with identity: $\mathsf{nf}\,(N) \stackrel{\text{def}}{=} N$, $\mathsf{nf}\,(P) \stackrel{\text{def}}{=} P$, since the new equivalence classes are singletons; (ii) modify the least upper bound polymorphic rule ($\exists^{\vee}$) so that it requires the quantifiers to be equal (and performs alpha-conversion if necessary):

$$\frac{\Theta, \overrightarrow{\alpha^-} \vDash P_1 \vee P_2 = Q \quad \overrightarrow{\alpha^-} \subseteq \mathsf{fv}\,Q}{\Theta \vDash \exists \overrightarrow{\alpha^-}. P_1 \vee \exists \overrightarrow{\alpha^-}. P_2 = Q} \quad (\text{E}\exists^{\vee})$$

(iii) replace the subtyping polymorphic rule ($\forall^{\leqslant}$) by the following rule:

$$\frac{\overrightarrow{\widehat{\alpha}^{+}} \text{ are fresh} \quad \Theta, \overrightarrow{\beta^{+}}; \Xi, \overrightarrow{\widehat{\alpha}^{+}} \{\Theta, \overrightarrow{\beta^{+}}\} \vDash [\overrightarrow{\widehat{\alpha}^{+}}/\overrightarrow{\alpha^{+}}]N \leqslant M \dashv C}{\Theta; \Xi \vDash \forall \overrightarrow{\beta^{+}}, \overrightarrow{\alpha^{+}}. N \leqslant \forall \overrightarrow{\beta^{+}}. M \dashv C} \quad (\mathrm{E}\forall^{\leqslant})$$

and (iv) update the existential rule ($\exists^{\geqslant}$) symmetrically.

After these changes, the rule ($TApp^{\mathrm{INF}}$) and its algorithmic counterpart can be used to infer the the type of $c\{P\}$.

The elementary type inference restricts the expressiveness of the subtyping perhaps too much. As mentioned, it forbids the quantifier reordering; besides, as soon as the right-hand side of the subtyping relation is polymorphic, it restricts the instantiation of the left-hand side quantifiers, which *disallows*, for example, $\cdot \vdash \forall \alpha^{+}. \uparrow \alpha^{+} \leqslant \forall \alpha^{+}. \uparrow\downarrow\uparrow \alpha^{+}$. On the other hand, the elementary subtyping system can be smoothly extended with bounded quantification, which we discuss later.

Solution 3: labeled quantifiers. A compromise solution that resolves the ambiguity of explicit instantiation without fixing the order of quantifiers is by using *labeled* quantifiers. Let us discuss how to introduce them to the *negative* part of the system, since the positive subsystem is symmetric.

Each type abstraction $\Lambda l \triangleright \alpha^{+}. c$ must be annotated with a label l. This label propagates to the inferred polymorphic type. This way, each polymorphic quantifier $\forall \overrightarrow{l_i \triangleright \alpha_i^{+I}}. N$ is annotated with a label l_i whose index is taken from the *list* of labels I associated with the group of quantifiers. To instantiate a quantifying variable, one refers to it by its label: $c\{l_j \triangleright P\}$, which is expressed by the following typing rules:

$$\frac{\Theta, \alpha^{+}; \Gamma \vdash c : N}{\Theta; \Gamma \vdash \Lambda l \triangleright \alpha^{+}. c : \forall l \triangleright \alpha^{+}. N} \quad (\Lambda^{\mathrm{INF}}) \qquad \frac{\Theta; \Gamma \vdash c : \forall \overrightarrow{l_i \triangleright \alpha_i^{+I}}. N}{\Theta; \Gamma \vdash c\{l_j \triangleright P\} : \forall \overrightarrow{l_i \triangleright \alpha_i^{+I\setminus\{j\}}}. [P/\alpha_j^{+}]N} \quad (TApp^{\mathrm{INF}})$$

The subtyping rule permits an arbitrary order of the quantifiers. However, similarly to the elementary subtyping, each of the right-hand side quantifiers must have a matching left-hand side quantifier with the same label. These matching quantifiers synchronously removed (in other words, each variable is abstractly instantiated to itself), and the remaining quantifiers are instantiated as usual by a substitution σ.

$$\frac{\{I\} \subseteq \{\mathcal{J}\} \quad \Theta, \overrightarrow{\alpha_i^{+I}} \vdash \sigma : \overrightarrow{\alpha_j^{+\mathcal{J}\setminus\{I\}}} \quad \Theta, \overrightarrow{\alpha_i^{+I}} \vdash [\sigma]N \leqslant M}{\Theta \vdash \forall \overrightarrow{l_j \triangleright \alpha_j^{+\mathcal{J}}}. N \leqslant \forall \overrightarrow{l_i \triangleright \alpha_i^{+I}}. M} \quad (\forall^{\leqslant})$$

The equivalence (mutual subtyping) in this system allows the quantifiers within the same group to be reordered together with their labels. For instance, $\forall l \triangleright \alpha^{+}. \forall m \triangleright \beta^{+}. N$ and $\forall m \triangleright \beta^{+}. \forall l \triangleright \alpha^{+}. N$ are subtypes of each other. However, the right-hand side quantifiers are removed synchronously with the matching left-hand side quantifiers, which means that $\cdot \vdash \forall l \triangleright \alpha^{+}. \uparrow \alpha^{+} \leqslant \forall l \triangleright \alpha^{+}. \uparrow\downarrow\uparrow \alpha^{+}$ is still not allowed.

6.2 Weakening of the subtyping invariants

In order to make the subtyping relation decidable, we made the subtyping of shifts *invariant*, which manifests in the following rules:

$$\frac{\Theta \vdash P \simeq^{\leqslant} Q}{\Theta \vdash \uparrow P \leqslant \uparrow Q} \quad (\uparrow^{\leqslant}) \qquad \frac{\Theta \vdash N \simeq^{\leqslant} M}{\Theta \vdash \downarrow N \geqslant \downarrow M} \quad (\downarrow^{\geqslant})$$

To make the system more expressive, we can relax these rules. Although making both rules covariant would make the system undecidable, it is possible to make the up-shift subtyping covariant

while keeping the down-shift invariant:

$$\frac{\Theta \vdash Q \geqslant P}{\Theta \vdash \uparrow P \leqslant \uparrow Q} \quad (\uparrow^{\leqslant}_{\text{RLX}})$$

However, this change requires an update to the algorithm. As one can expect, the corresponding algorithmic rule ($\uparrow^{\leqslant}$) must be changed accordingly:

$$\frac{\Theta;\ \Xi \vDash Q \leqslant P \dashv C}{\Theta;\ \Xi \vDash \uparrow P \leqslant \uparrow Q \dashv C} \quad (\uparrow^{\leqslant}_{\text{RLX}})$$

Also, notice that now the algorithmic variables can occur on *both* sides of the positive subtyping relation, and thus, a more sophisticated constraint solver is needed. To see what kind of constraints will be generated, let us keep the other algorithmic rules unchanged for a moment and make several observations.

First, notice that the negative quantification variables still only occur at *invariant* positions: the negative variables are generated from the existential quantifiers $\exists \alpha^-.\, P$, and the switch from a positive to a negative type in the syntax path to α^- in P is 'guarded' by *invariant* down-shift. It means that in the algorithm, the negative constraint entries can only be equivalence entries ($\widehat{\alpha}^- :\simeq N$) but not subtyping entries.

Second, notice that the positive subtyping is still not dependent on the negative subtyping. It means, that any leaf-to-root path in the inference tree is monotonous: in the first phase, the negative subtyping rules are applied, and in the second phase, only the positive ones.

The invariant that only the left-hand side of the judgment is algorithmic is not preserved. However, the only rule violating this invariant is ($\uparrow^{\leqslant}_{\text{RLX}}$), and it is placed at the interface between negative subtyping and positive subtyping. It means that the *negative* algorithmic variables (produced by $\exists$ in the positive phase of the inference path) still only occur on the left-hand side of the judgment, and the positive algorithmic variables (produced by $\forall$ in the first phase) can occur either on the left-hand side or on the right-hand side (if the switch from the negative to the positive phase is made by ($\uparrow^{\leqslant}_{\text{RLX}}$)) but not both. This way, the produced entries will be of the following form: (i) $\widehat{\alpha}^+ :\simeq P$, (ii) $\widehat{\alpha}^- :\simeq N$, (iii) $\widehat{\alpha}^+ :\geqslant P$, and (iv) $\widehat{\alpha}^+ :\leqslant P$ where P does not contain positive unification variables.

We do not present the details of the constraint resolution procedure or how it is integrated into the subtyping algorithm, leaving it for future work. However, we believe that the system consisting of the mentioned kinds of constraints is solvable by the following reasons. (i) The equivalence entries $\widehat{\alpha}^+ :\simeq P$ and $\widehat{\alpha}^- :\simeq N$ are resolved by standard first-order pattern unification techniques. (ii) The resolution of supertyping entries ($\widehat{\alpha}^+ :\geqslant P$) is discussed in the original algorithm (Section 4.5). (iii) The new kind of entries $\widehat{\alpha}^+ :\leqslant P$ can be reduced by case analysis. If P is a variable β^+ or a shifted computation $\downarrow N$ then $\widehat{\alpha}^+ :\leqslant P$ is equivalent to $\widehat{\alpha}^+ :\simeq P$; otherwise, $\widehat{\alpha}^+ :\leqslant \exists \overrightarrow{\alpha^-}.\downarrow N$ is equivalently replaced by $\widehat{\alpha}^+ :\simeq [\overrightarrow{\widehat{\alpha}}/\overrightarrow{\alpha^-}]\downarrow N$ for freshly created $\overrightarrow{\widehat{\alpha}}$.

This way, the algorithm can be extended to support the *covariant* up-shift subtyping. This, for example, enriches the relation with such subtypes as $\cdot \vdash \forall \alpha^+.\, \alpha^+ \to \uparrow \alpha^+ \leqslant \downarrow\uparrow \texttt{Int} \to \uparrow \exists \beta^-.\, \downarrow \beta^-$, which does not hold in the original system. Moreover, this extension also increases the expressiveness of the *type inference*. Namely, when inferring a type of an annotated let binding $\texttt{let}\ x : P = v(\vec{v});\ c$, we require $\uparrow P$ to be a supertype of $\uparrow Q$—the resulting type of the application $v(\vec{v})$. In the relaxed system, it can be replaced with the requirement that P is a supertype of Q (without the up-shift), which is more permissive.

In addition, the refinement of the unification algorithm described above makes possible another improvement to the system—bounded quantification.

6.3 Bounded Quantification

After the weakening of subtyping invariants, it is possible to smoothly extend the *elementary* version of the system (the second solution discussed in Section 6.1) with bounded quantification. In particular, we can add upper bounds to polymorphic ∀-quantifiers: $\forall \overrightarrow{\alpha^+ \leqslant P}. N$.

The declarative subtyping rules for bounded quantification are as expected: the instantiation of quantified variables must satisfy the corresponding bounds ($\Theta, \overrightarrow{\alpha^+} \vdash Q_i \geqslant [\sigma]\beta_i^+$ holds for each i); and the right-hand side quantifiers must be more restrictive than the matching left-hand side quantifiers ($\Theta \vdash P_i \geqslant P'_i$ for each i):

$$\frac{\overrightarrow{\Theta \vdash P_i \geqslant P'_i} \quad \Theta, \overrightarrow{\alpha^+} \vdash \sigma : \overrightarrow{\beta^+} \quad \overrightarrow{\Theta, \overrightarrow{\alpha^+} \vdash Q_i \geqslant [\sigma]\beta_i^+} \quad \Theta, \overrightarrow{\alpha^+} \vdash [\sigma]N \leqslant M}{\Theta \vdash \forall \overrightarrow{\alpha^+ \leqslant P}. \forall \overrightarrow{\beta^+ \leqslant Q}. N \leqslant \forall \overrightarrow{\alpha^+ \leqslant P'}. M} \quad (\mathrm{E}\forall^{\leqslant})$$

The corresponding algorithmic rule, as before, replaces the polymorphic variables $\beta_1 \ldots$ with fresh algorithmic variables $\widehat{\beta_1}^+ \ldots$, and merges the generated constraints with the constraints given in the bounds $\widehat{\beta_i}^+ :\leqslant Q_i$:

$$\frac{\begin{array}{l}\Theta; \cdot \vDash \overrightarrow{P_i \geqslant P'_i} \dashv \cdot \quad \overrightarrow{\widehat{\beta}^+} \text{ are fresh} \\ \Theta, \overrightarrow{\alpha^+}; \Xi, \overrightarrow{\widehat{\beta}^+} \{\Theta, \overrightarrow{\alpha^+}\} \vDash N \leqslant M \dashv C_0 \quad \Xi \vdash C_0 \,\&\, (\overrightarrow{\widehat{\beta_i}^+ :\leqslant Q_i}) = C\end{array}}{\Theta; \Xi \vDash \forall \overrightarrow{\alpha^+ \leqslant P}. \forall \overrightarrow{\beta^+ \leqslant Q}. N \leqslant \forall \overrightarrow{\alpha^+ \leqslant P'}. M \dashv C} \quad (\mathrm{E}\forall^{\leqslant})$$

Notice that the algorithmic rule ($\mathrm{E}\forall^{\leqslant}$) requires the bounding types $\overrightarrow{P}$ and $\overrightarrow{Q}$ to be *declarative*. Because of that, the generated constraints have shape $\widehat{\beta_i}^+ :\leqslant Q_i$, i.e., the bounding types do not contain algorithmic variables. This kind of constraints is covered by the resolution procedure described in Section 6.2, and thus, ($\mathrm{E}\forall^{\leqslant}$) fits into the algorithmic system without changes to the constraint solver.

However, in order for this rule to be complete, the declarative system must be restricted as well. To guarantee that no algorithmic variable is introduced in the bounding types, we need to forbid the quantifying variables to occur in the bounding types in the declarative system. For that purpose, we must include this requirement in the type well-formedness (here $\mathtt{fbv}N$ denotes the set of variables freely occurring in the quantifiers of N at any depth):

$$\frac{\overrightarrow{\Theta \vdash P_i} \quad \Theta, \overrightarrow{\alpha^+} \vdash N \quad \overrightarrow{\alpha^+} \cap \mathtt{fbv}N = \emptyset}{\Theta \vdash \forall \overrightarrow{\alpha^+ \leqslant P}. N} \quad (\forall^{\mathrm{WF}})$$

For brevity, we do not discuss in detail the combination of bounded and unbounded quantification in one system. We believe this update is straightforward: the unbounded quantifiers are treated as the weakest constraints, which are trivially satisfied. Analogously, it is possible to extend the system with *lower* bound ∀-quantifiers: $\forall \overrightarrow{\alpha^+ \geqslant P}. N$. Then the generated constraints will change to $\widehat{\beta_i}^+ :\geqslant Q_i$, which is also covered by the resolution procedure. However, we do not consider bounded *existential* quantification, because in this case, the constraint resolution would require us to find the *greatest lower bound* of two negative types, which is not well-defined (see Section 4.5).

6.4 Bidirectionalization

The algorithm we provide requires that all lambda functions are annotated. This restriction significantly simplifies the type inference by making the terms uniquely define their types. However, it leads to certain redundancies in typing, in particular, the type of a lambda expression cannot be inferred from the context it is used in: the annotated $((\lambda x.\, \mathtt{return}\, x) : (\mathtt{Int} \to \uparrow\mathtt{Int}))$ does not infer $\mathtt{Int} \to \uparrow\mathtt{Int}$.

The well-known way to incorporate this expressiveness into the system is to make the typing bidirectional [Dunfield et al. 2020]. The idea is to split the typing judgment into two kinds:

(i) *Synthesis* judgments are used when the type of a term can be *inferred* based exclusively on the term itself. Syntactically, we denote synthesizing judgments as $\Theta\,;\Gamma \vdash c \Rightarrow N$.
(ii) *Checking* judgments assume that the type of a term is *given* from the context, and it is required to *check* that the given type can be assigned to the term. In checking judgments $\Theta\,;\Gamma \vdash c \Leftarrow N$, the type N is considered as an *input*.

To bidirectionalize the system, each typing rule must be oriented either to synthesis or to checking (or both). In particular, the *annotated* lambda-abstractions are synthesizing, and the *unannotated* ones are checking.

$$\frac{\Theta \vdash P \quad \Theta\,;\Gamma, x : P \vdash c \Rightarrow N}{\Theta\,;\Gamma \vdash \lambda x : P.\, c \Rightarrow P \to N}\ (\lambda^{\Leftarrow}) \qquad \frac{\Theta \vdash P \quad \Theta\,;\Gamma, x : P \vdash c \Leftarrow N}{\Theta\,;\Gamma \vdash \lambda x.\, c \Leftarrow P \to N}\ (\lambda^{\Leftarrow})$$

As common for bidirectional systems with subtyping, we would also need to introduce the *subsumption* rules. They allow us to 'forget' the information about the type by switching to the synthesis mode, as long as the synthesized type is more polymorphic than the checked one:

$$\frac{\Theta\,;\Gamma \vdash c \Rightarrow N \quad \Theta \vdash N \leqslant M}{\Theta\,;\Gamma \vdash c \Leftarrow M}\ (\text{SUB}_{-}) \qquad \frac{\Theta\,;\Gamma \vdash v \Rightarrow P \quad \Theta \vdash Q \geqslant P}{\Theta\,;\Gamma \vdash v \Leftarrow Q}\ (\text{SUB}_{+})$$

Most of the original rules will have two bidirectional counterparts, one for each mode. The premises of the rules are oriented with respect to the conclusion. For instance, we need both checking and inferring versions for `return` v:

$$\frac{\Theta\,;\Gamma \vdash v \Leftrightarrow P}{\Theta\,;\Gamma \vdash \texttt{return}\, v \Leftrightarrow \uparrow P}\ (\text{RET}^{\Leftrightarrow})$$

For some rules, however, it only makes sense to have one mode. As mentioned above, the unannotated lambda-abstraction is always checking. On the other hand, an *annotated* lambda, type-level lambda abstraction, or a variable inference is always synthesizing, since the required type information is already given in the term:

$$\frac{\Theta, \alpha^{+}\,;\Gamma \vdash c \Rightarrow N}{\Theta\,;\Gamma \vdash \Lambda\alpha^{+}.\, c \Rightarrow \forall\alpha^{+}.\, N}\ (\Lambda^{\Leftarrow}) \qquad \frac{x : P \in \Gamma}{\Theta\,;\Gamma \vdash x \Rightarrow P}\ (\text{VAR}^{\Rightarrow})$$

Finally, let us discuss the application inference part of the declarative system. Rules $(\emptyset^{\text{INF}}_{\bullet\Rightarrow})$ and $(\forall^{\text{INF}}_{\bullet\Rightarrow})$ are unchanged: they simply do not have typing premises to be oriented. However, the arrow application rule $(\to^{\text{INF}}_{\bullet\Rightarrow})$ infers the result of the application of an arrow $Q \to N$ to a list of arguments $v, \vec{v}$, and it must make sure that the first argument v is typeable with the expected Q. As we will discuss further, simply checking $\Theta\,;\Gamma \vdash v \Leftarrow Q$ turns out to be too powerful and potentially leads to undecidability. Instead, this check is approximated by the combination of $\Theta\,;\Gamma \vdash v \Rightarrow P$ and $\Theta \vdash Q \geqslant P$.

The Algorithm. The algorithmic system is bidirectionalized in a similar way. Each typing premise and the conclusion of the rule is oriented to either 'checking' ($\Leftarrow$) or 'synthesizing' ($\Rightarrow$) mode, with respect to the declarative system.

For brevity, we omit the details of the bidirectional algorithm. Let us discuss one especially important rule—arrow application inference. An intuitive way to bidirectionalize this rule is the following:

$$\frac{\begin{array}{l}\Theta\,;\Gamma \vDash v \Leftarrow Q \dashv C_1 \\ \Theta\,;\Gamma\,;\Xi \vDash N \bullet \vec{v} \Rrightarrow M \dashv \Xi'\,;C_2 \quad \Xi \vdash C_1 \,\&\, C_2 = C\end{array}}{\Theta\,;\Gamma\,;\Xi \vDash Q \to N \bullet v, \vec{v} \Rrightarrow M \dashv \Xi'\,;C}\ (\to^{\bullet\Rrightarrow})$$

However, this rule is too permissive. The judgement $\Theta\,;\Gamma \vDash v \Leftarrow Q \dashv C_1$ requires us to check a term against an *algorithmic* type Q. Further, through the lambda function checking $(\lambda^{\Leftarrow})$ the algorithmic types infiltrate the type context and then through variable inference and subsumption, *both* sides of subtyping. This way, $(\rightarrow^{\bullet\Rightarrow})$ compromises the important invariant of the subtyping algorithm: now the same algorithmic variable can occur on *both* sides of the subtyping relation.

We believe that the relaxation of this invariant brings the constraint resolution too close to second-order pattern unification, which is undecidable [Goldfarb 1981]. In particular, the constraint $(\widehat{\alpha}^+ :\geqslant \downarrow\uparrow\widehat{\alpha}^+)$ is solvable with $\widehat{\alpha}^+ = \exists\beta^-.\downarrow\beta^-$, since by $(\exists^{\geqslant})$, the existential β^- can be impredicatively instantiated to $\uparrow\exists\beta^-.\downarrow\beta^-$.

Constraints such as $(\widehat{\alpha}^+ :\geqslant \downarrow\uparrow\widehat{\alpha}^+)$ are not merely hypothetical. It occurs, for example, when we infer the type of the following application:

$$\cdot\,;\cdot \vdash \forall\alpha^+.\downarrow(\alpha^+ \to \uparrow\mathtt{Int}) \to \downarrow\uparrow\alpha^+ \to \uparrow\mathtt{Int} \bullet \{\lambda x.\,\lambda y.\,\mathtt{let}\ y' = x(y);\ \mathtt{return}\ y'\} \Longrightarrow \uparrow\mathtt{Int}$$

For x to be applicable to y, the type of y—$\downarrow\uparrow\alpha^+$ must be a subtype of α^+—the type expected by x. This way, this inference is possible if and only if $(\widehat{\alpha}^+ :\geqslant \downarrow\uparrow\widehat{\alpha}^+)$ is solvable. Using a similar scheme, we can construct different examples whose resolution significantly relies on second-order pattern unification. Thus, we leave the resolution of this type of constraints beyond the scope of this work.

Instead, we strengthen $(\rightarrow^{\bullet\Rightarrow})$ in such a way that it never checks a term against an algorithmic type. Instead, type checking v against Q is replaced by a more restrictive premise—checking that v *synthesizes a subtype* of Q:

$$\frac{\Theta\,;\Gamma \vDash v \Rightarrow P \quad \Theta;\,\Xi \vDash Q \geqslant P \dashv C_1 \qquad \Theta\,;\Gamma\,;\Xi \vDash N \bullet \vec{v} \Longrightarrow M \dashv \Xi'\,;C_2 \quad \Xi \vdash C_1\,\&\,C_2 = C}{\Theta\,;\Gamma\,;\Xi \vDash Q \to N \bullet v, \vec{v} \Longrightarrow M \dashv \Xi'\,;C} \quad (\rightarrow^{\bullet\Rightarrow})$$

As one can expect, the declarative rule is also changed accordingly. In terms of practicality, this change disallows implicit checking against a *polymorphic* type: $\cdot;\cdot \vdash \lambda x.\,\mathtt{return}\,x \Leftarrow \forall\alpha^+.\,\alpha^+ \to \uparrow\alpha^+$ is not allowed, because it would require adding algorithmic $x : \widehat{\alpha}^+$ to the context. However, once the instantiation is made explicit or the lambda is explicitly polymorphic, the checking is allowed: $\cdot\,;\cdot \vdash \lambda x.\,\mathtt{return}\,x \Leftarrow \mathtt{Int} \to \uparrow\mathtt{Int}$ and $\cdot\,;\cdot \vdash \Lambda\alpha^+.\,\lambda x : \alpha^+.\,\mathtt{return}\,x \Leftarrow \forall\alpha^+.\,\alpha^+ \to \uparrow\alpha^+$ are both valid.

7 CONCLUSION

We have presented a type inference algorithm for an impredicative polymorphic lambda calculus with existential types and subtyping. The system is designed in call-by-push-value paradigm, which allowed us to restrict it to a decidable fragment of the language described declaratively. We presented the inference algorithm and proved its soundness and completeness with respect to the specification. The algorithm combines unification—a standard type inference technique—with anti-unification—which has not been used in type inference before.

REFERENCES

Jacek Chrzaszcz (1998). "Polymorphic Subtyping without Distributivity." In: *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*. MFCS '98. Berlin, Heidelberg: Springer-Verlag, pp. 346–355.

Luís Damas and Robin Milner (1982). "Principal type-schemes for functional programs." In: *ACM-SIGACT Symposium on Principles of Programming Languages*. URL: https://api.semanticscholar.org/CorpusID:11319320.

Jana Dunfield and Neel Krishnaswami (Nov. 2020). "Bidirectional Typing." In: arXiv: 1908.05839.

Jana Dunfield and Neelakantan R. Krishnaswami (2016). "Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types." In: *Proceedings of the ACM on Programming Languages* 3, pp. 1–28. URL: https://api.semanticscholar.org/CorpusID:19625612.

Richard A. Eisenberg, Guillaume Duboc, Stephanie Weirich, and Daniel K. Lee (2021). "An existential crisis resolved: type inference for first-class existential types." In: *Proceedings of the ACM on Programming Languages* 5, pp. 1–29. URL: https://api.semanticscholar.org/CorpusID:237205112.

Jean-Yves Girard (1971). "Une Extension de L'Interpretation de Gödel à L'Analyse, et Son Application à L'Elimination Des Coupures Dans L'Analyse et La Theorie Des Types." In: *Proceedings of the Second Scandinavian Logic Symposium*. Ed. by J.E. Fenstad. Vol. 63. Studies in Logic and the Foundations of Mathematics. North Holland: Elsevier, pp. 63–92. DOI: 10.1016/S0049-237X(08)70843-7.

Warren D. Goldfarb (1981). "The Undecidability of the Second-Order Unification Problem." In: *Theor. Comput. Sci.* 13, pp. 225–230. DOI: 10.1016/0304-3975(81)90040-2.

Roger Hindley (1969). "The Principal Type-Scheme of an Object in Combinatory Logic." In: *Transactions of the American Mathematical Society* 146, pp. 29–60. URL: https://api.semanticscholar.org/CorpusID:7223700.

Konstantin Läufer and Martin Odersky (1994). "Polymorphic type inference and abstract data types." In: *ACM Trans. Program. Lang. Syst.* 16, pp. 1411–1430. URL: https://api.semanticscholar.org/CorpusID:13189837.

Daan Leijen (2006). "First-class polymorphism with existential types." In: URL: https://api.semanticscholar.org/CorpusID:16392798.

Paul Blain Levy (Dec. 2006). "Call-by-Push-Value: Decomposing Call-by-Value and Call-by-Name." In: *Higher-Order and Symbolic Computation* 19.4, pp. 377–414. DOI: 10.1007/s10990-006-0480-6.

Dale Miller (1991). "A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification." In: *J. Log. Comput.* 1.4, pp. 497–536. DOI: 10.1093/logcom/1.4.497.

Robin Milner (1978). "A Theory of Type Polymorphism in Programming." In: *J. Comput. Syst. Sci.* 17, pp. 348–375. URL: https://api.semanticscholar.org/CorpusID:388583.

Benjamin Pierce and David Turner (Jan. 2000). "Local Type Inference." In: *ACM Transactions on Programming Languages and Systems* 22.1, pp. 1–44. DOI: 10.1145/345099.345100.

Gordon D. Plotkin (1970). "A Note on Inductive Generalization." In: *Machine Intelligence* 5.1, pp. 153–163.

John C Reynolds (1970). "Transformational systems and the algebraic structure of atomic formulas." In: vol. 5. 1, pp. 135–151.

John C Reynolds (1974). "Towards a Theory of Type Structure." In: *Programming Symposium*. Springer. Paris, France: Springer, pp. 408–425.

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis (Aug. 2020). "A Quick Look at Impredicativity." In: *Proceedings of the ACM on Programming Languages* 4.ICFP, pp. 1–29. DOI: 10.1145/3408971.

Jerzy Tiuryn (1995). "Equational Axiomatization of Bicoercibility for Polymorphic Types." In: *Foundations of Software Technology and Theoretical Computer Science, 15th Conference, Bangalore, India, December 18-20, 1995, Proceedings*. Ed. by P. S. Thiagarajan. Vol. 1026. Lecture Notes in Computer Science. Springer, pp. 166–179. DOI: 10.1007/3-540-60692-0_47.

Jerzy Tiuryn and Pawel Urzyczyn (1996). "The Subtyping Problem for Second-Order Types Is Undecidable." In: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*. LICS '96. USA: IEEE Computer Society, p. 74.

Jinxu Zhao and Bruno C. d. S. Oliveira (2022). "Elementary Type Inference." In: *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*. Ed. by Karim Ali and Jan Vitek. Vol. 222. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:28. DOI: 10.4230/LIPICS.ECOOP.2022.2.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009