# Local Type Inference for Polarised System F with Existentials

ANONYMOUS AUTHOR(S)

CHANGE!! This paper addresses the challenging problem of type inference for Impredicative System F with existential types, a critical aspect of many programming languages. While System F serves as the basis for type systems in numerous languages, existing type inference techniques for Impredicative System F are undecidable due to the presence of existential (∃) and polymorphic (∀) types. Consequently, current algorithms are often ad-hoc and sub-optimal. This paper presents novel contributions in the form of a local type inference algorithm for Impredicative System F with existential types. The algorithm introduces innovative techniques, such as a unique combination of unification and anti-unification, a full correctness proof, and the use of control structures inspired by Call-By-Push-Value . Additionally, the paper discusses a type inference framework that allows the algorithm to be applied to different type systems, offering insights into the under-researched area of impredicative existential type inference.

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), 16 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

- There has been very little work on type inference for existentials.
- The state of the art in languages like Ocaml and Haskell is the Odersky-Läufer algorithm, which makes existentials a second-class construct tied to datatypes.
- More recently, there's the existential crisis paper, which showed how to extend ML-style inference with some support for existentials. However, it is (a) predicative-only, (b) restricted for ∀∃ types, and (c) does not have a declarative specification. (In fact, the authors comment on the need for such.)
- In this paper, we lift all of these restrictions. We give a local type inference algorithm which supports first-class existentials, for impredicative types, with no restrictions on nested quantifers, and we have declarative specification we prove our algorithm sound and complete with respect to.
- Developing the algorithm required us to break some of the fundamental invariants of HM-style typin, and as a result, our algorithm needs to mix unification and anti-unification, and works over a call-by-push-value metalanguage to control how quantifiers are instantiated and things like function arities.


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

- Unfortunately, local type inference does not infer types anywhere nearly as comprehensively as full Damas-Milner type inference. We do guarantee that all type applications (for ∀-elimination) and all packs (for ∃-introdution) are inferred, but functions and some let-bindings still need annotations on their arguments.
- The original local type inference paper combined local type inference with bidirectional typechecking to minimize the number of needed annotations, but we show how existential types complicate the integration of bidirectionality with local type inference, and we explore the design space to show how the same scheme could be applied to work with different type systems.

Neel: We need to explain what local type inference is, and how it is and isn't related to bidirectional typechecking.

## 2 OVERVIEW

### 2.1 What types do we infer?

### 2.2 The Language of Types

The types of $F^{\pm}\exists$ are given in fig. 1. They are stratified into two syntactic categories (polarities): positive and negative, similarly to the Call-By-Push-Value system [Levy 2006]. The negative types represent computations, and the positive types represent values:

- $\alpha^-$ is a negative type variable, which can be taken from a context or introduced by ∃.
- a function $P \to N$ takes a value as input and returns a computation;
- a polymorphic abstraction $\forall\overrightarrow{\alpha^+}.\ N$ quantifies a computation over a list of positive type variables $\overrightarrow{\alpha^+}$. The polarities are chosen to follow the definition of functions.
- a shift $\uparrow P$ allows a value to be used as a computation, which at the term level corresponds to a pure computation **return** $v$.
+ $\alpha^+$ is a positive type variable, taken from a context or introduced by ∀.
+ $\exists\overrightarrow{\alpha^-}.\ P$, symmetrically to ∀, binds negative variables in a positive type $P$.
+ a shift $\downarrow N$, symmetrically to the up-shift, thunks a computation, which at the term level corresponds to $\{c\}$.

Negative declarative types

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

Positive declarative types

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

Fig. 1. Declarative Types of $F^{\pm}\exists$

*Definitional Equalities.* For simplicity, we assume that alpha-equivalent terms are equal. This way, we assume that substitutions do not capture bound variables. Besides, we equate $\forall\overrightarrow{\alpha^+}.\ \forall\overrightarrow{\beta^+}.\ N$ with $\forall\overrightarrow{\alpha^+}, \overrightarrow{\beta^+}.\ N$, as well as $\exists\overrightarrow{\alpha^-}.\ \exists\overrightarrow{\beta^-}.\ P$ with $\exists\overrightarrow{\alpha^-}, \overrightarrow{\beta^-}.\ P$, and lift these equations transitively and congruently to the whole system.

### 2.3 The Language of Terms

In fig. 2, we define the language of terms of $F^{\pm}\exists$. The language combines System F with the Call-By-Push-Value approach.

- \+ $x$ denotes a (positive) term variable; Ilya: why no negatives? Following CBPV
- \+ $\{c\}$ is a value corresponding to a thunked or suspended computation;
- ± $(c : N)$ and $(v : P)$ allow one to annotate positive and negative terms;
- – **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 $\lambda$;
- – **let** $x = v;\ c$ is a standard let, binding a value $v$ to a variable $x$ in a computation $c$;
- – Applicative let forms **let** $x : P = v(\vec{v});\ c$ and **let** $x = v(\vec{v});\ c$ operate similarly to the bind of a monad: they take a suspended computation $v$, apply it to a list of arguments, bind the result (which is expected to be pure) to a variable $x$, and continue 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;
- – **let**$^{\exists}(\overrightarrow{\alpha^{-}}, x) = v;\ c$ is the standard unpack of an existential type: expecting $v$ to be 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 ∀) and dually, it does not have pack (i.e. the constructor of ∃). This is because the instantiation of polymorphic and existential types is inferred by the algorithm. Ilya: refer to the extension chapter

Computation Terms

$$
\begin{array}{lll}
c,\ d & ::= & \\
 & \mid & (c : N) \\
 & \mid & \lambda x : P.\ c \\
 & \mid & \Lambda\alpha^{+}.\ c \\
 & \mid & \textbf{return}\ v \\
 & \mid & \textbf{let}\ x = v;\ c \\
 & \mid & \textbf{let}\ x : P = v(\vec{v});\ c \\
 & \mid & \textbf{let}\ x = v(\vec{v});\ c \\
 & \mid & \textbf{let}^{\exists}(\overrightarrow{\alpha^{-}}, x) = v;\ c
\end{array}
$$

Value Terms

$$
\begin{array}{lll}
v,\ w & ::= & \\
 & \mid & x \\
 & \mid & \{c\} \\
 & \mid & (v : P)
\end{array}
$$

Fig. 2. Declarative Terms of $F^{\pm}\exists$

### 2.4 The key ideas of the algorithm

## 3 DECLARATIVE SYSTEM

The declarative system serves as a specification of the type inference algorithm. It consists of two main parts: the subtyping and the type inference.

### 3.1 Subtyping

It is represented by a set of inference rules shown in fig. 3.

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

$$\frac{}{\Gamma \vdash \alpha^- \leqslant \alpha^-}\ (\textsc{Var}^{\leqslant}_-)$$

$$\frac{\Gamma \vdash P \simeq^{\leqslant} Q}{\Gamma \vdash \uparrow P \leqslant \uparrow Q}\ (\uparrow^{\leqslant})$$

$$\frac{\Gamma \vdash P \geqslant Q \quad \Gamma \vdash N \leqslant M}{\Gamma \vdash P \to N \leqslant Q \to M}\ (\to^{\leqslant})$$

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

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

$$\frac{}{\Gamma \vdash \alpha^+ \geqslant \alpha^+}\ (\textsc{Var}^{\geqslant}_+)$$

$$\frac{\Gamma \vdash N \simeq^{\leqslant} M}{\Gamma \vdash \downarrow N \geqslant \downarrow M}\ (\downarrow^{\geqslant})$$

$$\frac{\Gamma, \overrightarrow{\beta^-} \vdash \sigma : \overrightarrow{\alpha^-} \quad \Gamma, \overrightarrow{\beta^-} \vdash [\sigma]P \geqslant Q}{\Gamma \vdash \exists \overrightarrow{\alpha^-}.\ P \geqslant \exists \overrightarrow{\beta^-}.\ Q}\ (\exists^{\geqslant})$$

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

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

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

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

Fig. 3. Declarative Subtyping

*Quantifiers.* Symmetric rules ($\forall^{\leqslant}$) and ($\exists^{\geqslant}$) specify the 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 the subtyping. For conciseness of representation, we compose these rules into one. First, our rule extends context $\Gamma$ 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.

The instantiation of quantifiers is modeled by substitution $\sigma$. The notation $\Gamma_2 \vdash \sigma : \Gamma_1$ specifies its domain and range. For instance, $\Gamma, \overrightarrow{\beta^+} \vdash \sigma : \overrightarrow{\alpha^+}$ means that $\sigma$ maps the variables from $\overrightarrow{\alpha^+}$ to (positive) types well-formed in $\Gamma, \overrightarrow{\beta^+}$. This way, application $[\sigma]N$ instantiates (replaces) every $\alpha_i^-$ in $N$ with $\sigma(\alpha_i^-)$.

*Invariant Shifts.* An important restriction that we put on the subtyping system is that the subtyping on shifted types requires their equivalence, as shown in ($\downarrow^{\geqslant}$) and ($\uparrow^{\leqslant}$). Relaxing both of these invariants make the system equivalent to System F , and thus, undecidable. However, after certain changes ($\uparrow^{\leqslant}$) can be relaxed to the covariant form, as we will discuss in **??**.

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

*Variables.* The subtyping of variables is defined reflexively, which is enough to ensure the reflexivity of subtyping in general. The algorithm will use the fact that the subtypes of a variable coincide with its supertypes, which however is not true for an arbitrary type.

*3.1.1 Properties of the Declarative Subtyping.* A property that we extensively use is that the subtyping is reflexive and transitive, and agrees with substitution.

Property 1 (Subtyping forms a preorder). *Let us say that two types $N_1$ and $N_2$ are in the subtyping relation if there exists a context $\Gamma$ such that $\Gamma \vdash N_1 \leqslant N_2$; symmetrically, two types $P_1$ and*

$P_2$ *are in the subtyping relation if there exists* $\Gamma$ *such that* $\Gamma \vdash P_2 \geqslant P_1$. *Then the subtyping relation defined this way is reflexive and transitive.*

Property 2 (Subtyping agrees with substitution). *Suppose that* $\sigma$ *is a substitution such that* $\Gamma_2 \vdash \sigma : \Gamma_1$. *Then*

- $\Gamma_1 \vdash N \leqslant M$ *implies* $\Gamma_2 \vdash [\sigma]N \leqslant [\sigma]M$, *and*
- $\Gamma_1 \vdash P \geqslant Q$ *implies* $\Gamma_2 \vdash [\sigma]P \geqslant [\sigma]Q$.

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

Property 3 (Positive Least Upper Bound exists). *Suppose that* $P_1$ *and* $P_2$ *are positive types well-formed in* $\Gamma$. *Then there exists the least common supertype— a type* $P$ *such that*

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

*Negative Greatest Lower Bound does not exist.* However, the symmetric construction—the greatest lower bound of two negative types does not always exist. Let us consider the following counterexample. Let us 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 $P \to \downarrow\uparrow P_1 \to \downarrow\uparrow P_2 \to N$? One of the common subtypes is $\forall\alpha^+, \beta^+, \gamma^+.\ \alpha^+ \to \downarrow\uparrow\beta^+ \to \downarrow\uparrow\gamma^+ \to N$, which, however is not the greatest one.

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 $\alpha^+$ and $\beta^+$ with $Q$ ensures that both of these types are subtypes of $Q \to \downarrow\uparrow Q \to \downarrow\uparrow Q \to N$; instantiating $\alpha^+$ with $P_1$ and $\beta^+$ 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. Since $M_1$ and $M_2$ are not equivalent, it 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 a.k.a. 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 reacher than equality. Specifically,

(ii) one can introduce redundant quantifiers. For example, $\forall\alpha^+, \beta^+.\ \uparrow\alpha^+$ is equivalent but not equal to $\forall\alpha^+.\ \uparrow\alpha^+$;

(i) one can reorder 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) the transformations (i) and (ii) can happen 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. 4. The steps (i-ii) are implemented by the ordering function, which takes a set of variables *vars* and a type and returns a list of variables from *vars* that occur in the type in the order of their first occurrence. Its formal definition can be found in **??**.

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

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

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

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

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

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

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

Fig. 4. Type Normalization Procedure

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

Property 4 (Correctness of normalization).

– *For* $N$ *and* $M$ *well-formed in* $\Gamma$, $\Gamma \vdash N \simeq^{\leqslant} M$ *is equivalent to* $\mathbf{nf}\,(N) = \mathbf{nf}\,(M)$;
\+ *analogously, for* $P$ *and* $Q$ *well-formed in* $\Gamma$, $\Gamma \vdash P \simeq^{\leqslant} Q$ *is equivalent to* $\mathbf{nf}\,(P) = \mathbf{nf}\,(Q)$.

### 3.3 Type Inference

The declarative specification of the type inference is shown in fig. 5. The positive typing judgment $\Gamma; \Phi \vdash v\colon P$ is read as "under the type context $\Gamma$ and variable context $\Phi$, the term $v$ is allowed to have the type $P$", where $\Phi$—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 of $\Gamma;\ \Phi \vdash N \bullet \vec{v} \Longrightarrow M$, which reads "under the type context $\Gamma$ and variable context $\Phi$, the application of a function of type $N$ to the list of arguments $\vec{v}$ is allowed to have the type $M$".

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

*Variables.* Rule ($\textsc{Var}^{\textsc{inf}}$) allows to infer the type of a variable from the context. In literature can be found 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 in general case by ($\simeq^{\textsc{inf}}_+$).

$\boxed{\Gamma;\Phi \vdash c\colon N}$ Negative typing

$$\frac{\Gamma \vdash P \quad \Gamma;\Phi, x : P \vdash c\colon N}{\Gamma;\Phi \vdash \lambda x : P.\ c\colon P \to N}\ (\lambda^{\text{INF}})$$

$$\frac{\Gamma, \alpha^+;\Phi \vdash c\colon N}{\Gamma;\Phi \vdash \Lambda\alpha^+.\ c\colon \forall\alpha^+.\ N}\ (\Lambda^{\text{INF}})$$

$$\frac{\Gamma;\Phi \vdash v\colon P}{\Gamma;\Phi \vdash \mathbf{return}\ v\colon \uparrow P}\ (\text{RET}^{\text{INF}})$$

$$\frac{\Gamma;\Phi \vdash v\colon P \quad \Gamma;\Phi, x : P \vdash c\colon N}{\Gamma;\Phi \vdash \mathbf{let}\ x = v;\ c\colon N}\ (\text{LET}^{\text{INF}})$$

$$\frac{\begin{array}{l}\Gamma;\Phi \vdash v\colon \downarrow M\\ \Gamma;\ \Phi \vdash M \bullet \vec{v} \Longrightarrow \uparrow Q \text{ unique}\\ \Gamma;\Phi, x : Q \vdash c\colon N\end{array}}{\Gamma;\Phi \vdash \mathbf{let}\ x = v(\vec{v});\ c\colon N}\ (\text{LET}^{\text{INF}}_{@})$$

$$\frac{\begin{array}{l}\Gamma \vdash P \quad \Gamma;\Phi \vdash v\colon \downarrow M\\ \Gamma;\ \Phi \vdash M \bullet \vec{v} \Longrightarrow \uparrow Q\\ \Gamma \vdash \uparrow Q \leqslant \uparrow P \quad \Gamma;\Phi, x : P \vdash c\colon N\end{array}}{\Gamma;\Phi \vdash \mathbf{let}\ x : P = v(\vec{v});\ c\colon N}\ (\text{LET}^{\text{INF}}_{:@})$$

$$\frac{\begin{array}{l}\Gamma;\Phi \vdash v\colon \exists\overrightarrow{\alpha}.\ P\\ \mathbf{nf}\ (\exists\overrightarrow{\alpha}.\ P) = \exists\overrightarrow{\alpha}.\ P\\ \Gamma, \overrightarrow{\alpha};\Phi, x : P \vdash c\colon N \quad \Gamma \vdash N\end{array}}{\Gamma;\Phi \vdash \mathbf{let}^{\exists}(\overrightarrow{\alpha}, x) = v;\ c\colon N}\ (\text{LET}^{\text{INF}}_{\exists})$$

$$\frac{\Gamma \vdash M \quad \Gamma;\Phi \vdash c\colon N \quad \Gamma \vdash N \leqslant M}{\Gamma;\Phi \vdash (c : M)\colon M}\ (\text{ANN}^{\text{INF}}_{-})$$

$$\frac{\Gamma;\Phi \vdash c\colon N \quad \Gamma \vdash N \simeq^{\leqslant} N'}{\Gamma;\Phi \vdash c\colon N'}\ (\simeq^{\text{INF}}_{-})$$

$\boxed{\Gamma;\Phi \vdash v\colon P}$ Positive typing

$$\frac{x : P \in \Phi}{\Gamma;\Phi \vdash x\colon P}\ (\text{VAR}^{\text{INF}})$$

$$\frac{\Gamma;\Phi \vdash c\colon N}{\Gamma;\Phi \vdash \{c\}\colon \downarrow N}\ (\{\}^{\text{INF}})$$

$$\frac{\Gamma \vdash Q \quad \Gamma;\Phi \vdash v\colon P \quad \Gamma \vdash Q \geqslant P}{\Gamma;\Phi \vdash (v : Q)\colon Q}\ (\text{ANN}^{\text{INF}}_{+})$$

$$\frac{\Gamma;\Phi \vdash v\colon P \quad \Gamma \vdash P \simeq^{\leqslant} P'}{\Gamma;\Phi \vdash v\colon P'}\ (\simeq^{\text{INF}}_{+})$$

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

$$\frac{\Gamma \vdash N \simeq^{\leqslant} N'}{\Gamma;\ \Phi \vdash N \bullet \cdot \Longrightarrow N'}\ (\emptyset^{\text{INF}}_{\bullet\Rightarrow})$$

$$\frac{\begin{array}{l}\Gamma;\Phi \vdash v\colon P \quad \Gamma \vdash Q \geqslant P\\ \Gamma;\ \Phi \vdash N \bullet \vec{v} \Longrightarrow M\end{array}}{\Gamma;\ \Phi \vdash Q \to N \bullet v, \vec{v} \Longrightarrow M}\ (\to^{\text{INF}}_{\bullet\Rightarrow})$$

$$\frac{\begin{array}{l}\vec{v} \neq \cdot \quad \overrightarrow{\alpha^+} \neq \cdot \quad \Gamma \vdash \sigma : \overrightarrow{\alpha^+}\\ \Gamma;\ \Phi \vdash [\sigma]N \bullet \vec{v} \Longrightarrow M\end{array}}{\Gamma;\ \Phi \vdash \forall\overrightarrow{\alpha^+}.\ N \bullet \vec{v} \Longrightarrow M}\ (\forall^{\text{INF}}_{\bullet\Rightarrow})$$

Fig. 5. Declarative Subtyping

*Annotations.* Subtyping is also used by the annotation rules ($\text{ANN}^{\text{INF}}_{-}$) and ($\text{ANN}^{\text{INF}}_{+}$). 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 ($\lambda^{\text{INF}}$) 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 ($\Lambda^{\text{INF}}$) infers polymorphic $\forall$-type. It extends the type context with the quantifying variable $\alpha^+$ and infers the type of the body. As a result, it returns a polymorphic type quantifying the abstracted variable $\alpha^+$ over the type of the body.

*Return and Thunk.* Rules ($\text{RET}^{\text{INF}}$) and ($\{\}^{\text{INF}}$) add the corresponding shifts 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. 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.

*Applicative Let Binders.* Rules ($\textsc{let}^{\textsc{inf}}_{@}$) and ($\textsc{let}^{\textsc{inf}}_{:@}$) 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 difference is that the former rule is for the *unannotated* let binder, and thus it requires the resulting type of application to be unique (up to equivalence), so that the type of the bound variable $x$ is known before it is put into the context. 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 $\Gamma \vdash \uparrow Q \leqslant \uparrow P$. This judgment is more restrictive than checking bare $\Gamma \vdash P \geqslant Q$, however, it is necessary to make the algorithm complete as it allows us to preserve certain invariants (see **??**). In **??** we discuss how this restriction can be relaxed together with invariant shift subtyping.

*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 ($\simeq^{\textsc{inf}}_{+}$) and ($\simeq^{\textsc{inf}}_{-}$). 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 ($\emptyset^{\textsc{inf}}_{\bullet\Rightarrow}$). 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 ($\simeq^{\textsc{inf}}_{+}$), however, the local relaxation is sufficient to prove the global property.

*Application of a Polymorphic Type* $\forall$. The complexity of the system is hidden in the rules, whose output type is not immediately defined by their input and the output of their premises (a.k.a. not mode-correct [Dunfield et al. 2020]). In our typing system, such rule is ($\forall^{\textsc{inf}}_{\bullet\Rightarrow}$): the instantiation of the quantifying variables is not known a priori. The algorithm we present in **??** delays this instantiation until more information about it (in particular, typing constraints) is collected.

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

*Application of an Arrow Type.* Another important application rule is ($\rightarrow^{\textsc{inf}}_{\bullet\Rightarrow}$). 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 (**??**), this subtyping check will provide the constraints we need to resolve the delayed instantiations of the quantifying variables.

### 3.3.1 *Declarative Typing Properties.*

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

Property 5 (Declarative Typing is Defined up to Equivalence). *Let us assume that* $\Gamma \vdash \Phi_1 \simeq^{\leqslant} \Phi_2$, *i.e., the corresponding types assigned by* $\Phi_1$ *and* $\Phi_2$ *are equivalent in* $\Gamma$. *Also, let us assume that* $\Gamma \vdash N_1 \simeq^{\leqslant} N_2$, $\Gamma \vdash P_1 \simeq^{\leqslant} P_2$, *and* $\Gamma \vdash M_1 \simeq^{\leqslant} M_2$. *Then*

- $\Gamma; \Phi_1 \vdash c : N_1$ *holds if and only if* $\Gamma; \Phi_2 \vdash c : N_2$,
- $\Gamma; \Phi_1 \vdash v : P_1$ *holds if and only if* $\Gamma; \Phi_2 \vdash v : P_2$, *and*
- $\Gamma;\ \Phi_1 \vdash N_1 \bullet \vec{v} \Longrightarrow M_1$ *holds if and only if* $\Gamma;\ \Phi_2 \vdash N_2 \bullet \vec{v} \Longrightarrow M_2$.

Ilya: Other properties?

## 4 THE ALGORITHM

In this section, we present the algorithmization of the system described before Shadowing the declarative system, the algorithm has two main parts: the subtyping and the type inference, which we discuss in this section one after another. First, let us discuss the syntax of the algorithmic system

**Positive Algorithmic Variables** $\widehat{\alpha}^+, \widehat{\beta}^+, \widehat{\gamma}^+, \ldots$
**Negative Algorithmic Variables** $\widehat{\alpha}^-, \widehat{\beta}^-, \widehat{\gamma}^-, \ldots$
**Positive Algorithmic Types** $P = \cdots \mid \widehat{\alpha}^+$
**Negative Algorithmic Types** $N = \cdots \mid \widehat{\alpha}^-$
**Algorithmic Type Context** $\Xi$ = set of Algorithmic Variables

*Algorithmic Variables.* Both subtyping and the inference algorithms are represented by sets of inference rules, to simplify the soundness and completeness proofs w.r.t. the declarative specification. The terms these rules manipulate we call *algorithmic*. They extend the previously defined declarative terms and types by adding *algorithmic type variables* (a.k.a. existential 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 $\alpha, \beta, \ldots$ and also to smooth out the transition from the declarative to the algorithmic system, when we replace the quantified variables $\overrightarrow{\alpha^+}$ that we are not able to instantiate 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 $\overrightarrow{\widehat{\alpha}^+}/\overrightarrow{\alpha^+}$.

*Algorithmic Types.* The syntax of algorithmic types is the syntax of declarative types extended with algorithmic type variables: we add positive algorithmic variables $\widehat{\alpha}^+$ to the positive types, and negative algorithmic variables $\widehat{\alpha}^-$ to the negative types. Notice that these variables cannot be abstracted by the quantifiers $\forall$ and $\exists$. We denote the algorithmic types

*Algorithmic Contexts and Well-formedness.* To specify well-formedness, we define algorithmic contexts $\Xi$ as sets of algorithmic variables. Then $\Gamma;\ \Xi \vdash P$ and $\Gamma;\ \Xi \vdash N$ represent the well-formedness judgment of algorithmic terms defined as expected: in addition to the declarative definition **??**, we also check that each algorithmic variable is in the context $\Xi$.

]

### 4.1 Subtyping Algorithm

Hello Subtyping

### 4.2 Unification

Hello Unification

### 4.3 Constraint Merge

Definition 1 (Matching Entries). *We call two unification constraint entries or two subtyping constraint entries matching if they are restricting the same unification variable.*

Two matching entries formed in the same context $\Gamma$ can be merged in the following way:

- $(\simeq \&^+ \simeq)$ and $(\simeq \&^- \simeq)$ are symmetric cases. To merge two matching entries restricting a variable to be equivalent to certain types, we check that these types are equivalent to each

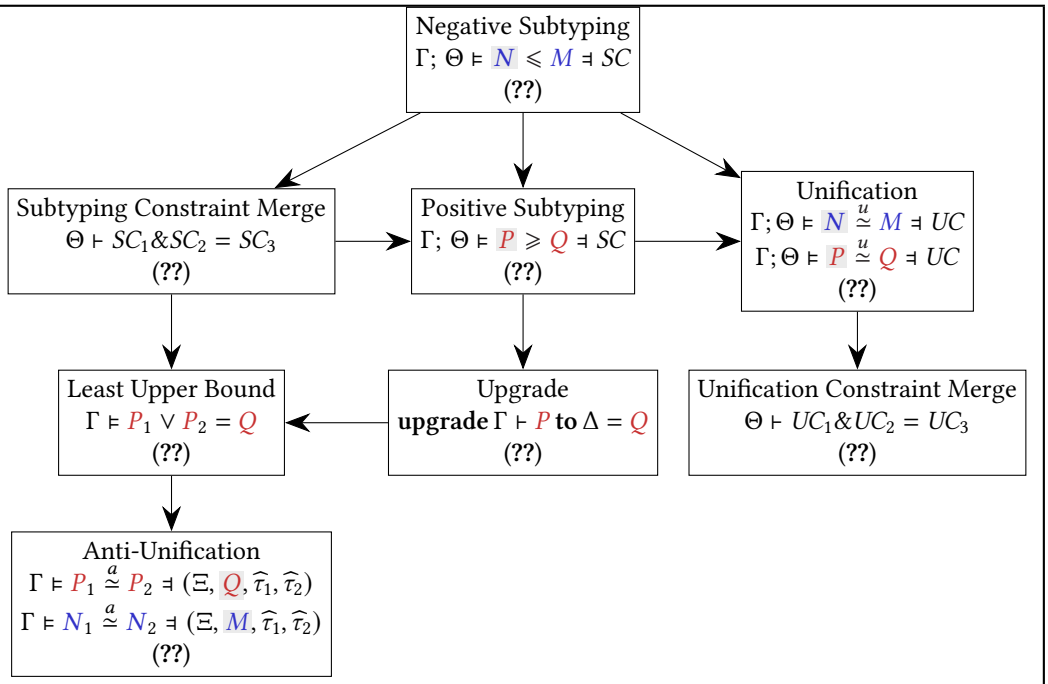


Fig. 6. Dependency graph of the subtyping algorithm

other. To do so, it suffices to check for *equality* of their normal forms, as discussed in **??**. After that, we return the left-hand entry.

- $(\simeq \&^+ \geqslant)$ and $(\geqslant \&^+ \simeq)$ are also symmetric. In this case, since one of the entries requires the variable to be equal to a type, the resulting entry must also imply that. However, for the soundness, it is needed to ensure that the equating restriction is stronger than the subtyping restriction. For this purpose, the premise invokes the positive subtyping.
- $(\geqslant \&^+ \geqslant)$ In this case, we find the least upper bound of the types from the input restrictions, and as the output, restrict the variable to be a supertype of the result. The least upper bound procedure will be discussed in **??**.

### 4.4 Type Upgrade and the Least Upper Bounds

### 4.5 Anti-Unification

### 4.6 Type Inference

## 5 CORRECTNESS

## 6 EXTENSIONS

## 7 RELATED WORK

## 8 CONCLUSION

[Botlan et al. 2003] [**dunfieldBidirectionalTyping2020**]

## REFERENCES

Didier Le Botlan and Didier Rémy (Aug. 2003). "MLF Raising ML to the Power of System F." In: *ICFP '03*. Uppsala, Sweden: ACM Press, pp. 52–63.

$\boxed{\Gamma;\Theta \vDash N \leqslant M \dashv SC}$ Negative subtyping

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

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

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

$$\frac{\Gamma;\Theta \vDash P \geqslant Q \dashv SC_1 \quad \Gamma;\Theta \vDash N \leqslant M \dashv SC_2 \quad \Theta \vdash SC_1 \& SC_2 = SC}{\Gamma;\Theta \vDash P \to N \leqslant Q \to M \dashv SC}\ (\to^{\leqslant})$$

$\boxed{\Gamma;\Theta \vDash P \geqslant Q \dashv SC}$ Positive supertyping

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

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

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

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

Fig. 7. Subtyping Algorithm

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

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.

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

$\boxed{\Gamma;\Theta \vDash N \overset{u}{\simeq} M \dashv UC}$ Negative unification $\qquad$ $\boxed{\Gamma;\Theta \vDash P \overset{u}{\simeq} Q \dashv UC}$ Positive unification

$$\frac{}{\Gamma;\Theta \vDash \alpha^- \overset{u}{\simeq} \alpha^- \dashv \cdot}\ (\text{VAR}^{\overset{u}{\simeq}}_-) \qquad \frac{}{\Gamma;\Theta \vDash \alpha^+ \overset{u}{\simeq} \alpha^+ \dashv \cdot}\ (\text{VAR}^{\overset{u}{\simeq}}_+)$$

$$\frac{\Gamma;\Theta \vDash P \overset{u}{\simeq} Q \dashv UC}{\Gamma;\Theta \vDash \uparrow P \overset{u}{\simeq} \uparrow Q \dashv UC}\ (\uparrow^{\overset{u}{\simeq}}) \qquad \frac{\Gamma;\Theta \vDash N \overset{u}{\simeq} M \dashv UC}{\Gamma;\Theta \vDash \downarrow N \overset{u}{\simeq} \downarrow M \dashv UC}\ (\downarrow^{\overset{u}{\simeq}})$$

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

$$\frac{\Gamma, \overrightarrow{\alpha^+};\Theta \vDash N \overset{u}{\simeq} M \dashv UC}{\Gamma;\Theta \vDash \forall\overrightarrow{\alpha^+}.\, N \overset{u}{\simeq} \forall\overrightarrow{\alpha^+}.\, M \dashv UC}\ (\forall^{\overset{u}{\simeq}}) \qquad \frac{\Gamma, \overrightarrow{\alpha^-};\Theta \vDash P \overset{u}{\simeq} Q \dashv UC}{\Gamma;\Theta \vDash \exists\overrightarrow{\alpha^-}.\, P \overset{u}{\simeq} \exists\overrightarrow{\alpha^-}.\, Q \dashv UC}\ (\exists^{\overset{u}{\simeq}})$$

$$\frac{\Theta(\widehat{\alpha}^-) \vdash N}{\Gamma;\Theta \vDash \widehat{\alpha}^- \overset{u}{\simeq} N \dashv (\widehat{\alpha}^- :\simeq N)}\ (\text{UVAR}^{\overset{u}{\simeq}}_-) \qquad \frac{\Theta(\widehat{\alpha}^+) \vdash P}{\Gamma;\Theta \vDash \widehat{\alpha}^+ \overset{u}{\simeq} P \dashv (\widehat{\alpha}^+ :\simeq P)}\ (\text{UVAR}^{\overset{u}{\simeq}}_+)$$

Fig. 8. Unification Algorithm

$\boxed{\Gamma \vdash e_1 \,\&\, e_2 = e_3}$ Subtyping Constraint Entry Merge

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

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

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

$$\frac{\mathbf{nf}\,(P) = \mathbf{nf}\,(P')}{\Gamma \vdash (\widehat{\alpha}^+ :\simeq P) \,\&\, (\widehat{\alpha}^+ :\simeq P') = (\widehat{\alpha}^+ :\simeq P)}\ (\simeq \&^+ \simeq)$$

$$\frac{\mathbf{nf}\,(N) = \mathbf{nf}\,(N')}{\Gamma \vdash (\widehat{\alpha}^- :\simeq N) \,\&\, (\widehat{\alpha}^- :\simeq N') = (\widehat{\alpha}^- :\simeq N)}\ (\simeq \&^- \simeq)$$

Fig. 9. Merge Matching Constraint Entries

Suppose that $\Theta \vdash SC_1$ and $\Theta \vdash SC_2$. Then $\Theta \vdash SC_1 \& SC_2 = SC$ defines a set such that $e \in SC$ iff either

- $e \in SC_1$ and there is no matching $e' \in SC_2$; or
- $e \in SC_2$ and there is no matching $e' \in SC_1$; or
- $\Theta(\widehat{\alpha}^{\pm}) \vdash e_1 \& e_2 = e$ for some $e_1 \in SC_1$ and $e_2 \in SC_2$ such that $e_1$ matches with $e_2$ restricting variable $\widehat{\alpha}^{\pm}$.

Fig. 10. Constraint Merge

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

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

$$\frac{\Gamma \vDash \mathbf{nf}\,(\downarrow N) \overset{a}{\simeq} \mathbf{nf}\,(\downarrow M) \dashv (\Xi, P, \widehat{\tau_1}, \widehat{\tau_2})}{\Gamma \vDash \downarrow N \vee \downarrow M = \exists \overrightarrow{\alpha^-}.\, [\overrightarrow{\alpha^-}/\Xi]\, P} \quad (\downarrow^\vee)$$

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

Fig. 11. Least Upper Bound Algorithm

$$\boxed{\mathbf{upgrade}\,\Gamma \vdash P\,\mathbf{to}\,\Delta = Q}$$

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

Fig. 12. Type Upgrade Algorithm

$$\boxed{\Gamma \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\Xi, Q, \widehat{\tau_1}, \widehat{\tau_2})}$$

$$\frac{}{\Gamma \vDash \alpha^+ \overset{a}{\simeq} \alpha^+ \dashv (\cdot, \alpha^+, \cdot, \cdot)} \quad (\text{VAR}^{\overset{a}{\simeq}}_{+})$$

$$\frac{\Gamma \vDash N_1 \overset{a}{\simeq} N_2 \dashv (\Xi, M, \widehat{\tau_1}, \widehat{\tau_2})}{\Gamma \vDash \downarrow N_1 \overset{a}{\simeq} \downarrow N_2 \dashv (\Xi, \downarrow M, \widehat{\tau_1}, \widehat{\tau_2})} \quad (\downarrow^{\overset{a}{\simeq}})$$

$$\frac{\overrightarrow{\alpha^-} \cap \Gamma = \emptyset \quad \Gamma \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\Xi, Q, \widehat{\tau_1}, \widehat{\tau_2})}{\Gamma \vDash \exists \overrightarrow{\alpha^-}.\, P_1 \overset{a}{\simeq} \exists \overrightarrow{\alpha^-}.\, P_2 \dashv (\Xi, \exists \overrightarrow{\alpha^-}.\, Q, \widehat{\tau_1}, \widehat{\tau_2})} \quad (\exists^{\overset{a}{\simeq}})$$

$$\boxed{\Gamma \vDash N_1 \overset{a}{\simeq} N_2 \dashv (\Xi, M, \widehat{\tau_1}, \widehat{\tau_2})}$$

$$\frac{}{\Gamma \vDash \alpha^- \overset{a}{\simeq} \alpha^- \dashv (\cdot, \alpha^-, \cdot, \cdot)} \quad (\text{VAR}^{\overset{a}{\simeq}}_{-})$$

$$\frac{\Gamma \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\Xi, Q, \widehat{\tau_1}, \widehat{\tau_2})}{\Gamma \vDash \uparrow P_1 \overset{a}{\simeq} \uparrow P_2 \dashv (\Xi, \uparrow Q, \widehat{\tau_1}, \widehat{\tau_2})} \quad (\uparrow^{\overset{a}{\simeq}})$$

$$\frac{\overrightarrow{\alpha^+} \cap \Gamma = \emptyset \quad \Gamma \vDash N_1 \overset{a}{\simeq} N_2 \dashv (\Xi, M, \widehat{\tau_1}, \widehat{\tau_2})}{\Gamma \vDash \forall \overrightarrow{\alpha^+}.\, N_1 \overset{a}{\simeq} \forall \overrightarrow{\alpha^+}.\, N_2 \dashv (\Xi, \forall \overrightarrow{\alpha^+}.\, M, \widehat{\tau_1}, \widehat{\tau_2})} \quad (\forall^{\overset{a}{\simeq}})$$

$$\frac{\Gamma \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\Xi_1, Q, \widehat{\tau_1}, \widehat{\tau_2}) \quad \Gamma \vDash N_1 \overset{a}{\simeq} N_2 \dashv (\Xi_2, M, \widehat{\tau_1'}, \widehat{\tau_2'})}{\Gamma \vDash P_1 \to N_1 \overset{a}{\simeq} P_2 \to N_2 \dashv (\Xi_1 \cup \Xi_2, Q \to M, \widehat{\tau_1} \cup \widehat{\tau_1'}, \widehat{\tau_2} \cup \widehat{\tau_2'})} \quad (\to^{\overset{a}{\simeq}})$$

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

Fig. 13. Anti-unification Algorithm

$\boxed{\Gamma; \Phi \vDash c \colon N}$

Negative typing

$$\frac{\Gamma \vdash M \quad \Gamma; \Phi \vDash c \colon N \quad \Gamma; \cdot \vDash N \leqslant M \dashv \cdot}{\Gamma; \Phi \vDash (c : M) \colon \mathbf{nf}\ (M)}\ (\mathrm{ANN}_{-}^{\mathrm{INF}})$$

$$\frac{\Gamma \vdash P \quad \Gamma; \Phi, x : P \vDash c \colon N}{\Gamma; \Phi \vDash \lambda x : P.\ c \colon \mathbf{nf}\ (P \to N)}\ (\lambda^{\mathrm{INF}})$$

$$\frac{\Gamma, \alpha^{+}; \Phi \vDash c \colon N}{\Gamma; \Phi \vDash \Lambda \alpha^{+}.\ c \colon \mathbf{nf}\ (\forall \alpha^{+}.\ N)}\ (\Lambda^{\mathrm{INF}})$$

$$\frac{\Gamma; \Phi \vDash v \colon P}{\Gamma; \Phi \vDash \mathbf{return}\ v \colon \uparrow P}\ (\mathrm{RET}^{\mathrm{INF}})$$

$$\frac{\Gamma; \Phi \vDash v \colon P \quad \Gamma; \Phi, x : P \vDash c \colon N}{\Gamma; \Phi \vDash \mathbf{let}\ x = v;\ c \colon N}\ (\mathrm{LET}^{\mathrm{INF}})$$

$$\frac{\begin{array}{l}\Gamma \vdash P \quad \Gamma; \Phi \vDash v \colon \downarrow M \\ \Gamma;\ \Phi;\ \cdot \vDash M \bullet \vec{v} \Longrightarrow \uparrow Q \dashv \Theta;\ SC_1 \\ \Gamma;\ \Theta \vDash \uparrow Q \leqslant \uparrow P \dashv SC_2 \\ \Theta \vdash SC_1 \& SC_2 = SC \quad \Gamma; \Phi, x : P \vDash c \colon N\end{array}}{\Gamma; \Phi \vDash \mathbf{let}\ x : P = v(\vec{v});\ c \colon N}\ (\mathrm{LET}_{:@}^{\mathrm{INF}})$$

$$\frac{\begin{array}{l}\Gamma; \Phi \vDash v \colon \downarrow M \quad \Gamma;\ \Phi;\ \cdot \vDash M \bullet \vec{v} \Longrightarrow \uparrow Q \dashv \Theta;\ SC \\ \mathbf{uv}\ Q = \mathbf{dom}\ (SC) \quad SC\ \mathbf{singular\ with}\ \widehat{\sigma} \\ \Gamma; \Phi, x : [\widehat{\sigma}]\, Q \vDash c \colon N\end{array}}{\Gamma; \Phi \vDash \mathbf{let}\ x = v(\vec{v});\ c \colon N}\ (\mathrm{LET}_{@}^{\mathrm{INF}})$$

$$\frac{\Gamma; \Phi \vDash v \colon \exists \overrightarrow{\alpha^{-}}.\ P \quad \Gamma, \overrightarrow{\alpha^{-}}; \Phi, x : P \vDash c \colon N \quad \Gamma \vdash N}{\Gamma; \Phi \vDash \mathbf{let}^{\exists}(\overrightarrow{\alpha^{-}}, x) = v;\ c \colon N}\ (\mathrm{LET}_{\exists}^{\mathrm{INF}})$$

$\boxed{\Gamma; \Phi \vDash v \colon P}$

Positive typing

$$\frac{x : P \in \Phi}{\Gamma; \Phi \vDash x \colon \mathbf{nf}\ (P)}\ (\mathrm{VAR}^{\mathrm{INF}})$$

$$\frac{\Gamma; \Phi \vDash c \colon N}{\Gamma; \Phi \vDash \{c\} \colon \downarrow N}\ (\{\}^{\mathrm{INF}})$$

$$\frac{\begin{array}{l}\Gamma \vdash Q \quad \Gamma; \Phi \vDash v \colon P \\ \Gamma; \cdot \vDash Q \geqslant P \dashv \cdot\end{array}}{\Gamma; \Phi \vDash (v : Q) \colon \mathbf{nf}\ (Q)}\ (\mathrm{ANN}_{+}^{\mathrm{INF}})$$

$\boxed{\Gamma;\ \Phi;\ \Theta_1 \vDash N \bullet \vec{v} \Longrightarrow M \dashv \Theta_2;\ SC}$

Application typing

$$\frac{}{\Gamma;\ \Phi;\ \Theta \vDash N \bullet \cdot \Longrightarrow \mathbf{nf}\ (N) \dashv \Theta;\ \cdot}\ (\emptyset_{\bullet\Rightarrow}^{\mathrm{INF}})$$

$$\frac{\begin{array}{l}\Gamma; \Phi \vDash v : P \to Q \geqslant P \dashv \Theta; \\ \Gamma;\ \Phi;\ \Theta \vDash N \bullet \vec{v} \Longrightarrow M \dashv \Theta';\ SC_2 \\ \Theta \vdash SC_1 \& SC_2 = SC\end{array}}{\Gamma;\ \Phi;\ \Theta \vDash Q \to N \bullet v, \vec{v} \Longrightarrow M \dashv \Theta';\ SC}\ (\to_{\bullet\Rightarrow}^{\mathrm{INF}})$$

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