# 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), 25 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), the polymorphic type $\forall a.\, A$ has a big lambda $\Lambda a.\, e$ 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 sophisticated enough to model things like parametricity and representation independence. However, System F by itself is unwieldy as a programming language. The fact that the universal type $\forall a.\, A$ 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 $\mathsf{len} : \forall a.\, \mathsf{List}\, a \to \mathsf{int}$ at any concrete list type without explicitly instantiating the quantifier in len's type. That is, we want to write $\mathsf{len}\,[1, 2, 3]$ instead of writing $\mathsf{len}\,[\mathsf{int}]\,[1, 2, 3]$.

The most famous of the algorithms for solving these constraints is the Damas-Hindley-Milner algorithm. The idea is that type instantiation induces a subtype ordering: the type $\forall a.\, A$ is a subtype of all of its instantiations. So we wish to be free to use the same function len at many different types


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

such as List int → int, List bool → int, List (int × bool) → int, and so on. However, the subtype relation is nondeterministic: it tells us that whenever we see a polymorphic type $\forall a.\, A$, we know it is a subtype of *any* of its instantiations. To turn this into an algorithm, we have to actually make some choices, and DHM works by using *unification*. Whenever we would have had to introduce 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 quantifer $\forall$ 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 Laufer. 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.

There has been a limited amount of work on support for existential types as a first-class concept. In an unpublished draft, **leijen06** studied an extension of Damas-Milner inference in which type schemes contain alternating universal and existential quantifiers. Quantifiers still range over monotypes, and higher-rank polymorphism is not permitted. More recently, **dk19** studed 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). **existential-crisis** 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 (i.e., types without any occurences of quantifiers). 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 undecidabilty barriers by doing *partial* type inference. That is, we no longer try to do full type inference, 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 what the partial algorithm does – for example, **existential-crisis** observe that their algorithm lacks a declarative specification, and explain why existential types make this particularly difficult to do.

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 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 combination of features 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.

*Contributions.* Our contributions are as follows:

- We give a declarative type system (again, based on call-by-push-value) to serve as a specification of our algorithm, and we we prove our algorithm is sound and complete with respect to the declarative type system. The specification makes it easy to see that all type applications (for ∀-elimination) and all packs (for ∃-introdution) are inferred.
- We give a local type inference algorithm which supports both first-class existential and universal quantifiers, both of which can be instantiated impredicatively. To evade the undecidability results surrounding type inference for System F, we work in a variant of call-by-push-value, which lets us formulate a subtyping relation which is still decidable.
- Our algorithm breaks some of the fundamental invariants of HM-style type inference. As a result, it needs to mix unification and anti-unification, uses the call-by-push-value structure to control function arities and how quantifiers can be instantiated.
- 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

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

Positive declarative types

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

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** $x : P = v(\overrightarrow{v})\ ;\ c$ and **let** $x = v(\overrightarrow{v})\ ;\ c$ 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 $\forall$) 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. Ilya: refer to the extension chapter

Computation Terms

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $T$ 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 $T_2 \vdash \sigma : T_1$ specifies its domain and range. For instance, $T, \overrightarrow{\beta^+} \vdash \sigma : \overrightarrow{\alpha^+}$ means that $\sigma$ maps the variables from $\overrightarrow{\alpha^+}$ to (positive) types well-formed in $T, \overrightarrow{\beta^+}$. 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 is important for the subtyping algorithm, in particular for the type *upgrade* procedure (**??**), 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 $T$. Then $T \vdash N_1 \leqslant N_2$ implies $\mathbf{fv}(N_1) \subseteq \mathbf{fv}(N_2)$, and $T \vdash P_1 \geqslant P_2$ implies $\mathbf{fv}(P_1) \subseteq \mathbf{fv}(P_2)$.*

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

**Property 2** (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 $T$ such that $T \vdash N_1 \leqslant N_2$; symmetrically, two types $P_1$ and $P_2$ are in the subtyping relation if there exists $T$ such that $T \vdash P_2 \geqslant P_1$. Then the subtyping relation defined this way is reflexive and transitive.*

**Property 3** (Subtyping agrees with substitution)**.** *Suppose that $\sigma$ is a substitution such that $T_2 \vdash \sigma : T_1$. Then*

- *$T_1 \vdash N \leqslant M$ implies $T_2 \vdash [\sigma]N \leqslant [\sigma]M$, and*
- *$T_1 \vdash P \geqslant Q$ implies $T_2 \vdash [\sigma]P \geqslant [\sigma]Q$.*

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

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

*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 5** (Correctness of normalization)**.**

- *For $N$ and $M$ well-formed in $T$, $T \vdash N \simeq^{\leqslant} M$ is equivalent to* $\mathbf{nf}\,(N) = \mathbf{nf}\,(M)$*;*
+ *analogously, for $P$ and $Q$ well-formed in $T$, $T \vdash P \simeq^{\leqslant} Q$ 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 $T; \Gamma \vdash v \colon P$ is read as "under the type context $T$ and variable context $\Gamma$, the term $v$ is allowed to have the type $P$", where $\Gamma$—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 $T\,; \Gamma \vdash N \bullet \vec{v} \Rrightarrow M$, which reads "under the type context $T$ and variable context $\Gamma$, 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 ($\simeq^{\textsc{inf}}_{+}$).

*Annotations.* Subtyping is also used by the annotation rules ($\textsc{ann}^{\textsc{inf}}_{-}$) and ($\textsc{ann}^{\textsc{inf}}_{+}$). 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 ($\lambda^{\textsc{inf}}$) 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 $\alpha^{+}$ 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 ($\textsc{ret}^{\textsc{inf}}$) and ($\{\}^{\textsc{inf}}$) 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 $T \vdash \uparrow Q \leqslant \uparrow P$. This judgment is more restrictive than checking bare $T \vdash P \geqslant Q$, 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

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

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

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

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

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

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

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

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

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

$$\frac{T;\Gamma \vdash c\colon N \quad T \vdash N \simeq^{\leqslant} N'}{T;\Gamma \vdash c\colon N'}\ (\simeq^{\text{INF}}_{-})$$

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

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

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

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

$$\frac{T;\Gamma \vdash v\colon P \quad T \vdash P \simeq^{\leqslant} P'}{T;\Gamma \vdash v\colon P'}\ (\simeq^{\text{INF}}_{+})$$

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

$$\frac{T \vdash N \simeq^{\leqslant} N'}{T\ ;\Gamma \vdash N \bullet \cdot \Longrightarrow N'}\ (\emptyset^{\text{INF}}_{\bullet\Rightarrow})$$

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

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

Fig. 5. Declarative Inference

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 $(\emptyset^{\text{INF}}_{\bullet\Rightarrow})$. 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 $(\simeq^{\text{INF}}_{+})$, 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 ($\forall^{\text{INF}}_{\bullet\Rightarrow}$): 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 ($\emptyset^{\text{INF}}_{\bullet\Rightarrow}$), we also check that the list of arguments is not empty.

*Application of an Arrow Type.* Another important application rule is ($\rightarrow^{\text{INF}}_{\bullet\Rightarrow}$). 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 6** (Declarative Typing is Defined up to Equivalence)**.** *Let us assume that* $T \vdash \Gamma_1 \simeq^{\leqslant} \Gamma_2$, *i.e., the corresponding types assigned by* $\Gamma_1$ *and* $\Gamma_2$ *are equivalent in* $T$. *Also, let us assume that* $T \vdash N_1 \simeq^{\leqslant} N_2$, $T \vdash P_1 \simeq^{\leqslant} P_2$, *and* $T \vdash M_1 \simeq^{\leqslant} M_2$. *Then*

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

Ilya: Other properties?

## 4 THE ALGORITHM

In this section, we present the algorithmization of the 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, must be mode-correct ([Dunfield et al. 2020]), i.e., the 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* (a.k.a. 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 $\alpha^{+}, \beta^{-}$. 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 $\overrightarrow{\widehat{\alpha}^{+}}/\overrightarrow{\alpha^{+}}$.

*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

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

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

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

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

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

Constraint Type Context
$\Sigma ::= \{\widehat{\alpha}_1^{\pm}\{T_1\}, \ldots, \widehat{\alpha}_n^{\pm}\{T_n\}\}$ where $\widehat{\alpha}_1^{\pm}, \ldots, \widehat{\alpha}_n^{\pm}$ are pairwise distinct

Fig. 6. Algorithmic Syntax

constructors of the system can be applied to *algorithmic* types, however, algorithmic variables cannot be abstracted by the quantifiers $\forall$ and $\exists$.

*Algorithmic Contexts and Well-formedness.* To specify when algorithmic types are well-formed, we define algorithmic contexts $\Upsilon$ as sets of algorithmic variables. Then $T ; \Upsilon \vdash P$ and $T ; \Upsilon \vdash N$ represent the well-formedness judgment of algorithmic terms defined as expected. Informally, they check that all free declarative variables are in $T$, and all free algorithmic variables are in $\Upsilon$. In addition to the rules repeating the declarative definition **??**, we have two base cases for the algorithmic variables: ($\text{UVAR}_+^{\text{WF}}$) and ($\text{UVAR}_-^{\text{WF}}$) (see section 4.1)

$$\frac{\vdots}{\dfrac{\widehat{\alpha}^+ \in \Upsilon}{T ; \Upsilon \vdash \widehat{\alpha}^+}\ (\text{UVAR}_+^{\text{WF}})} \qquad \frac{\vdots}{\dfrac{\widehat{\alpha}^- \in \Upsilon}{T ; \Upsilon \vdash \widehat{\alpha}^-}\ (\text{UVAR}_-^{\text{WF}})}$$

Fig. 7. Well-formedness of Algorithmic Types

*Algorithmic Normalization.* Similarly to well-formedness, the normalization of algorithmic types is defined by extending the declarative definition with the algorithmic variables. To the rules repeating the declarative normalization, we add rules saying that normalization is trivial on algorithmic variables (see section 4.1).

$$\frac{\vdots}{\dfrac{}{\mathbf{nf}\,(\widehat{\alpha}^+) = \widehat{\alpha}^+}\ (\text{UVAR}_+^{\text{NF}})} \qquad \frac{\vdots}{\dfrac{}{\mathbf{nf}\,(\widehat{\alpha}^-) = \widehat{\alpha}^-}\ (\text{UVAR}_-^{\text{NF}})}$$

Fig. 8. Normalization of Algorithmic Types

## 4.2 Type Constraints

As the algorithm proceeds, it accumulates the information about the algorithmic type variables in the form of *constraints*. In our system, the constraints can be of two kinds: *subtyping constraints* and *unification constraints*. The subtyping constraint can only have a positive shape $\widehat{\alpha}^{+} :\geqslant P$, i.e., it restricts a positive algorithmic variable to be a supertype of a certain declarative type—this is one of the invariants that we preserve in the algorithm. The unification constraint can have either a positive form $\widehat{\alpha}^{+} :\simeq P$ or a negative form $\widehat{\alpha}^{-} :\simeq N$, however, the right-hand side of the constraint cannot contain algorithmic type variables. The set of constraints is denoted as $C$. We assume that each algorithmic variable can be restricted by at most one constraint.

We separately define $UC$ as a set consisting of unification constraints only. This is done to simplify the representation of the algorithm. The unification algorithm, which we use as a subroutine of the subtyping algorithm, can only produce unification constraints. A set of unification constraints can be resolved in a simpler way than a general constraint set. This way, the separation of the unification constraint resolution into a separate procedure allows us to better decompose the structure of the algorithm, and thus, simplify the inductive proofs.

Constraint Entry

$$\begin{array}{lll} e & ::= & \\ & \mid & \widehat{\alpha}^{+} :\simeq P \\ & \mid & \widehat{\alpha}^{-} :\simeq N \\ & \mid & \widehat{\alpha}^{+} :\geqslant P \end{array}$$

Unification Constraint Entry

$$\begin{array}{lll} ue & ::= & \\ & \mid & \widehat{\alpha}^{+} :\simeq P \\ & \mid & \widehat{\alpha}^{-} :\simeq N \end{array}$$

Constraint Set

$$C \quad ::= \quad \{e_1, \ldots, e_n\}$$

Unification Constraint Set

$$UC \quad ::= \quad \{ue_1, \ldots, ue_n\}$$

Fig. 9. Constraint Entries and Sets

*Constraint Contexts.* When one instantiates an algorithmic variable, they 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 a *constraint context* $\Sigma$—a set of pairs associating algorithmic variables and declarative contexts.

*Auxiliary Functions.* We define **dom** $(C)$—a domain of a constraint set $C$ as a set of algorithmic variables that it restricts. Similarly, we define **dom** $(\Sigma)$—a domain of constraint context as a set of algorithmic variables that $\Sigma$ associates with their contexts. We write $\Sigma(\widehat{\alpha}^{\pm})$ to denote the context associated with $\widehat{\alpha}^{\pm}$ in $\Sigma$.

## 4.3 Subtyping Algorithm

For convenience and scalability, we decompose the subtyping algorithm into several procedures. Figure 10 shows these procedures and the dependencies between them: arrows denote the invocation of one procedure from another. The label «**nf**» annotating arrows means that the calling procedure normalizes the input before passing it to the callee.

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 $T$, a constraint context $\Sigma$, 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

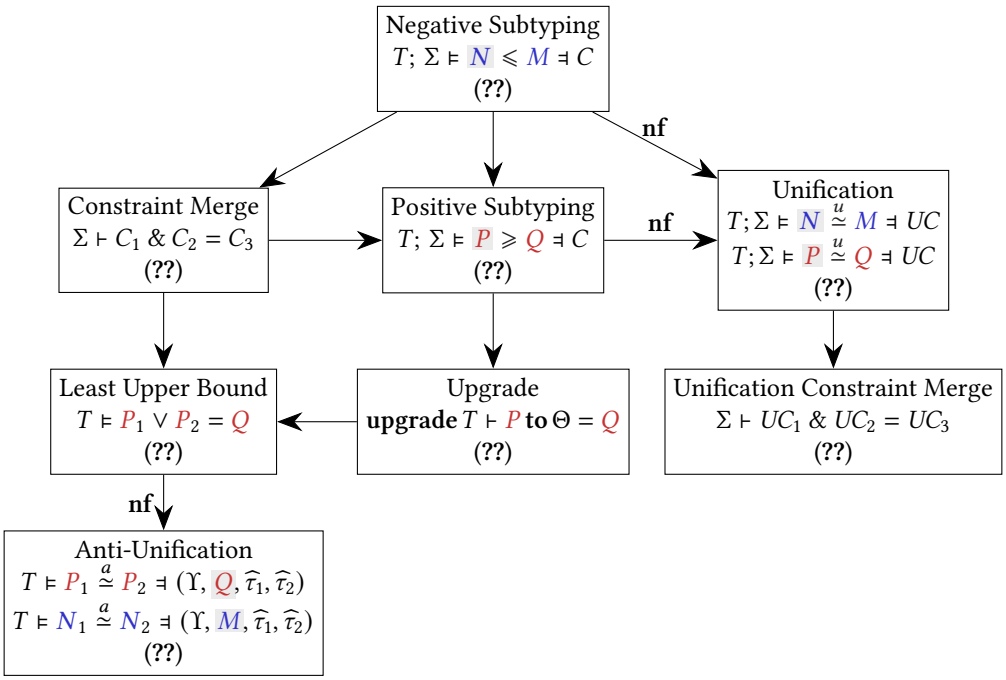


Fig. 10. Dependency graph of the subtyping algorithm

variables) and well-formed in $T$, but the first type ($N$ and $P$) may contain algorithmic variables, whose instantiation contexts are specified by $\Sigma$.

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).

*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, which make it equivalent to the right-hand side. For this purpose, the algorithm invokes the unification procedure **??** preceded by normalization of the input types. It returns the resulting constraints given by the unification algorithm.

*Quantifiers.* Rules ($\forall^{\leqslant}$) and **??** 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. The rule introduces fresh algorithmic variables $\overrightarrow{\widehat{\alpha}^{+}}$ or $\overrightarrow{\widehat{\alpha}^{-}}$, puts them into the constraint context $\Sigma$ (specifying that they must be instantiated in the extended context $T, \overrightarrow{\beta^{+}}$ or $T, \overrightarrow{\beta^{-}}$) and substitute the quantified variables for them in the input type.

$\boxed{T;\,\Sigma \vDash N \leqslant M \dashv C}$ Negative subtyping

$$\frac{}{T;\,\Sigma \vDash \alpha^- \leqslant \alpha^- \dashv \cdot}\ (\textsc{Var}^{\leqslant}_-)$$

$$\frac{T;\Sigma \vDash \mathbf{nf}\,(P) \stackrel{u}{\cong} \mathbf{nf}\,(Q) \dashv UC}{T;\,\Sigma \vDash \uparrow P \leqslant \uparrow Q \dashv UC}\ (\uparrow^{\leqslant})$$

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

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

$\boxed{T;\,\Sigma \vDash P \geqslant Q \dashv C}$ Positive supertyping

$$\frac{}{T;\,\Sigma \vDash \alpha^+ \geqslant \alpha^+ \dashv \cdot}\ (\textsc{Var}^{\geqslant}_+)$$

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

$$\frac{T;\Sigma \vDash \mathbf{nf}\,(N) \stackrel{u}{\cong} \mathbf{nf}\,(M) \dashv UC}{T;\,\Sigma \vDash \downarrow N \geqslant \downarrow M \dashv UC}\ (\downarrow^{\geqslant})$$

$$\frac{\mathbf{upgrade}\,T \vdash P\ \mathbf{to}\ \Sigma(\widehat{\alpha}^+) = Q}{T;\,\Sigma \vDash \widehat{\alpha}^+ \geqslant P \dashv (\widehat{\alpha}^+ :\geqslant Q)}\ (\textsc{UVar}^{\geqslant})$$

Fig. 11. Subtyping Algorithm

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 constraint 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 later in **??**) and returned as the output.

*Algorithmic Variable.* If one of the sides of the subtyping is a unification variable, the algorithm must create 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 context required for $\widehat{\alpha}^+$ by $\Sigma$, because this context might be smaller than the current context $T$. As we wish the resulting constraint set to be sound w.r.t. $\Sigma$, 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 $\Sigma(\widehat{\alpha}^+)$. It is done by the *upgrade* procedure, which we discuss in detail in **??**.

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 ($\to^{\leqslant}$) *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 $T$, a constraint context $\Sigma$, 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 $T$ and $\Sigma$, whereas the right-hand side is well-formed declaratively in $T$.

Since only the left-hand side may contain algorithmic variables, that the unification instantiates, we could have called this procedure *matching*. However, in **??** we will discuss several modifications of the type system, where this invariant is not preserved, and thus, this procedure becomes a genuine 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.

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

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

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

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

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

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

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

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

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

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

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

Fig. 12. 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 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 the argument types and their result types recursively Then it merges the resulting constraints using the constraint merge procedure (**??**).

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 subroutines discussed here, 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 constraint 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, except that it makes sure that the resulting instantiations agree with the input constraint context $\Sigma$. 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 Entreies.* Two *matching* entries formed in the same context $T$ can be merged as shown in fig. 13. Suppose that $e_1$ and $e_2$ are input entries. The result of the merge $e_1 \,\&\, e_2$ must be the weakest entry which implies both $e_1$ and $e_2$.

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 $e_1$.

- If $e_2$ is also a restricting 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 $(\simeq \&^{-} \simeq)$.
- If $e_2$ is a supertype constraint $\widehat{\alpha}^{+} :\geqslant P$, the algorithm must check that the type assigned by $e_1$ is a supertype of $P$. The corresponding symmetric rules are $(\geqslant \&^{+} \simeq)$ and $(\simeq \&^{+} \geqslant)$.

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 for finding the least upper bound is discussed in **??**.

*Merging Constraint Sets.* The algorithm for merging constraint sets is shown in fig. 14. 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. 13) of matching entries.

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

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

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

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

$$\frac{\mathbf{nf}\,(P) = \mathbf{nf}\,(P')}{T \vdash (\widehat{\alpha}^+ :\simeq P) \,\&\, (\widehat{\alpha}^+ :\simeq P') = (\widehat{\alpha}^+ :\simeq P)} \quad (\simeq \&^+ \simeq)$$

$$\frac{\mathbf{nf}\,(N) = \mathbf{nf}\,(N')}{T \vdash (\widehat{\alpha}^- :\simeq N) \,\&\, (\widehat{\alpha}^- :\simeq N') = (\widehat{\alpha}^- :\simeq N)} \quad (\simeq \&^- \simeq)$$

Fig. 13. Merge of Matching Constraint Entries

Suppose that $\Sigma \vdash C_1$ and $\Sigma \vdash C_2$.
Then $\Sigma \vdash C_1 \,\&\, C_2 = C$ 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
- $\Sigma(\widehat{\alpha}^\pm) \vdash e_1 \,\&\, e_2 = e$ 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. 14. Constraint Merge

As shown in fig. 13, 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 ell-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 $T$, the *upgrade* operation finds the least among those supertypes of $P$ that are well-formed in a smaller context $\Theta \subseteq T$. For given two types $P_1$ and $P_2$ well-formed in $T$, the *least upper bound* operation finds the least among common supertypes of $P_1$ and $P_2$ well-formed in $T$. These algorithms are shown in fig. 15.

*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 $\Theta = T \setminus \overrightarrow{\alpha^\pm}$ (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

$\boxed{\textbf{upgrade}\ T \vdash P\ \textbf{to}\ \Theta = Q}$ Type Upgrade

$$\frac{\begin{array}{l} T = \Theta, \overrightarrow{\alpha^{\pm}} \\ \overrightarrow{\beta^{\pm}}\ \textbf{are fresh} \quad \overrightarrow{\gamma^{\pm}}\ \textbf{are fresh} \\ \Theta, \overrightarrow{\beta^{\pm}}, \overrightarrow{\gamma^{\pm}} \vDash [\overrightarrow{\beta^{\pm}}/\overrightarrow{\alpha^{\pm}}]P \vee [\overrightarrow{\gamma^{\pm}}/\overrightarrow{\alpha^{\pm}}]P = Q \end{array}}{\textbf{upgrade}\ T \vdash P\ \textbf{to}\ \Theta = Q} \quad (\textsc{Upg})$$

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

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

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

$$\frac{T \vDash \textbf{nf}\ (\downarrow N) \overset{a}{\simeq} \textbf{nf}\ (\downarrow M) \dashv (\Upsilon, P, \widehat{\tau}_1, \widehat{\tau}_2)}{T \vDash \downarrow N \vee \downarrow M = \exists\overrightarrow{\alpha^{-}}.\ [\overrightarrow{\alpha^{-}}/\Upsilon]P} \quad (\downarrow^{\vee})$$

Fig. 15. Type Upgrade and Leas Upper Bound Algorithms

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 ($\textsc{Var}^{\vee}$) 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 $\alpha^{-}$ with $\beta^{+} \to \gamma_1^{-}$ and $\beta^{+} \to \gamma_2^{-}$ respectively) and $\exists\alpha^{-}.\downarrow(\beta^{+} \to \alpha^{-})$ (by instantiating $\alpha^{-}$ with $\gamma_1^{-}$ and $\gamma_2^{-}$ 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 $(\Upsilon, P, \widehat{\tau}_1, \widehat{\tau}_2)$, where important for us is $P$—the pattern and $\Upsilon$—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 [**todo**], 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 initial contest, and do not contain variables bound later. For example, the anti-unification of $N_1 = \forall\beta^+.\ \alpha_1^+ \to \uparrow\beta^+$ and $N_2 = \forall\beta^+.\ \alpha_2^+ \to \uparrow\beta^+$ must result in a '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 cannot capture the bound variable $\beta^+$).

The algorithm that finds the most specific generalizer of two types under required conditions is given in fig. 16. 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 $T \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\Upsilon, Q, \widehat{\tau}_1, \widehat{\tau}_2)$. As an input, it takes a context $T$, 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: $\Upsilon$—a set of 'holes' represented by negative algorithmic variables, $Q$—a pattern represented as a positive algorithmic type, whose algorithmic variables are in $\Upsilon$, and two substitutions $\widehat{\tau}_1$ and $\widehat{\tau}_2$ instantiating the variables from $\Upsilon$ such that $[\widehat{\tau}_1]\,Q = P_1$ and $[\widehat{\tau}_2]\,Q = P_2$.

At the high level, the algorithm scheme follows the standard approach [**todo**] consisting of 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 $M$. The name of 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 $T$ is *not* extended with the quantified variables. In this algorithm, $T$ 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 ($\to^{\overset{a}{\simeq}}$) 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

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

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

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

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

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

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

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

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

$$\frac{T \vDash P_1 \overset{a}{\simeq} P_2 \dashv (\Upsilon_1, Q, \widehat{\tau}_1, \widehat{\tau}_2) \quad T \vDash N_1 \overset{a}{\simeq} N_2 \dashv (\Upsilon_2, M, \widehat{\tau}'_1, \widehat{\tau}'_2)}{T \vDash P_1 \to N_1 \overset{a}{\simeq} P_2 \to N_2 \dashv (\Upsilon_1 \cup \Upsilon_2, Q \to M, \widehat{\tau}_1 \cup \widehat{\tau}'_1, \widehat{\tau}_2 \cup \widehat{\tau}'_2)} \ (\to^{\overset{a}{\simeq}})$$

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

Fig. 16. Anti-Unification Algorithm

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:

- $\Upsilon_1 \cup \Upsilon_2$—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.

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 $T$. 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). This principle makes the inference system not syntax-directed: it is not known a priori (before the recursive call) whether the corresponding congruent rule or rule (AU) will be applied.

### 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.

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

The application type inference judgment has form $T\,;\Gamma\,;\Sigma_1 \vDash N \bullet \vec{v} \Longrightarrow M \dashv \Sigma_2\,;C$. As an input, it takes three contexts: typing context $T$, a variable context $\Gamma$, and a constraint context $\Sigma_1$. 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 $\Sigma_1$, in other words, $T\,;\mathbf{dom}\,(\Sigma_1) \vdash N$. 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 $\Sigma_2$—an updated constraint context and $C$—a set of subtyping constraints. Together $\Sigma_2$ and $C$ specify how the algorithmic variables must be instantiated.

The inference rules are shown in fig. 17. 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 $T$. 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.

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

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

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

$$\frac{T \vdash M \quad T;\Gamma \vDash c\colon N \quad T;\cdot \vDash N \leqslant M \dashv \cdot}{T;\Gamma \vDash (c : M)\colon \mathbf{nf}\,(M)}\ (\textsc{ann}_{-}^{\textsc{inf}}) \qquad \frac{T;\Gamma \vDash v\colon P}{T;\Gamma \vDash \mathbf{return}\ v\colon \uparrow P}\ (\textsc{ret}^{\textsc{inf}})$$

$$\frac{T \vdash P \quad T;\Gamma, x : P \vDash c\colon N}{T;\Gamma \vDash \lambda x : P.\ c\colon \mathbf{nf}\,(P \to N)}\ (\lambda^{\textsc{inf}}) \qquad \frac{T;\Gamma \vDash v\colon P \quad T;\Gamma, x : P \vDash c\colon N}{T;\Gamma \vDash \mathbf{let}\ x = v\,;\, c\colon N}\ (\textsc{let}^{\textsc{inf}})$$

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

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

$$\frac{T;\Gamma \vDash v\colon \downarrow M \quad T\,;\Gamma\,;\cdot \vDash M \bullet \vec{v} \Longrightarrow \uparrow Q \dashv \Sigma\,;C \quad \mathbf{uv}\ Q = \mathbf{dom}\,(C) \quad C\ \mathbf{singular\ with}\ \widehat{\sigma} \quad T;\Gamma, x : [\widehat{\sigma}]\,Q \vDash c\colon N}{T;\Gamma \vDash \mathbf{let}\ x = v(\vec{v})\,;\, c\colon N}\ (\textsc{let}_{@}^{\textsc{inf}})$$

$\boxed{T\,;\Gamma\,;\Sigma_1 \vDash N \bullet \vec{v} \Longrightarrow M \dashv \Sigma_2\,;C}$ Application typing

$$\frac{}{T\,;\Gamma\,;\Sigma \vDash N \bullet \cdot \Longrightarrow \mathbf{nf}\,(N) \dashv \Sigma\,;\cdot}\ (\emptyset_{\bullet\Rightarrow}^{\textsc{inf}}) \qquad \frac{T;\Gamma \vDash v\colon P \quad T;\Sigma \vDash Q \geqslant P \dashv C_1 \quad T\,;\Gamma\,;\Sigma \vDash N \bullet \vec{v} \Longrightarrow M \dashv \Sigma'\,;C_2 \quad \Sigma \vdash C_1 \,\&\, C_2 = C}{T\,;\Gamma\,;\Sigma \vDash Q \to N \bullet v, \vec{v} \Longrightarrow M \dashv \Sigma'\,;C}\ (\to_{\bullet\Rightarrow}^{\textsc{inf}})$$

$$\frac{T\,;\Gamma\,;\Sigma, \overrightarrow{\widehat{\alpha}^{+}}\{T\} \vDash [\overrightarrow{\widehat{\alpha}^{+}}/\overrightarrow{\alpha^{+}}]\,N \bullet \vec{v} \Longrightarrow M \dashv \Sigma'\,;C \quad \overrightarrow{\widehat{\alpha}^{+}}\ \mathbf{are\ fresh} \quad \vec{v} \neq \cdot \quad \overrightarrow{\alpha^{+}} \neq \cdot}{T\,;\Gamma\,;\Sigma \vDash \forall\overrightarrow{\alpha^{+}}.\ N \bullet \vec{v} \Longrightarrow M \dashv \Sigma'\,;C|_{\mathbf{uv}\,(N)\cup\mathbf{uv}\,(M)}}\ (\forall_{\bullet\Rightarrow}^{\textsc{inf}})$$

Fig. 17. Algorithmic Type Inference

*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 ($\Lambda^{\text{INF}}$) 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 $\alpha^+$ does not occur in the body of the lambda, the corresponding $\forall$ will be removed.

*Return and Thunk.* Rules ($\{\}^{\text{INF}}$) and ($\text{RET}^{\text{INF}}$) 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 ($\text{LET}^{\text{INF}}_{\exists}$) 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 $T \vdash N$.

*Let Binders.* Rule ($\text{LET}^{\text{INF}}$) 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 ($\text{LET}^{\text{INF}}_{:@}$) 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 $\uparrow Q$, that must be instantiated to a subtype of the annotation $\uparrow P$. Then premise $T;\ \Sigma \vDash \uparrow Q \leqslant \uparrow P \dashv C_2$ together with $\Sigma \vdash C_1 \,\&\, C_2 = C$ check whether the instantiation to the annotated type $\uparrow P$ is possible, and if it is, the algorithm infers the type of the body in the extended context, and returns it as the result.

Rule ($\text{LET}^{\text{INF}}_{:@}$) works similarly to ($\text{LET}^{\text{INF}}_{:@}$), However, since there is no annotation, instead of checking that the inferred type $\uparrow Q$ can be a subtype of the annotation, the algorithm checks that $\uparrow Q$ is unique. As we prove, uniqueness means that all the algorithmic variables of $\uparrow Q$ are sufficiently restricted by $C$. This way, uniqueness is checked by the combination of $\mathbf{uv}\ Q = \mathbf{dom}\ (C)$ and $C$ **singular with** $\widehat{\sigma}$. Together, these two premises guarantee that the only possible instantiation of $Q$ is $[\widehat{\sigma}]\ Q$.

*Application to an Empty List of Arguments.* Rule ($\emptyset^{\text{INF}}_{\bullet\Rightarrow}$) 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 ($\forall^{\text{INF}}_{\bullet\Rightarrow}$), 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 that this step does not cause infinite recursion, we also check that the head type has at least one $\forall$-quantifier. Also, to force the algorithm to apply rule ($\emptyset^{\text{INF}}_{\bullet\Rightarrow}$) when there are no arguments, we require $\vec{v} \neq \cdot$.

*Application of an Arrow Type.* Rule ($\rightarrow^{\text{INF}}_{\bullet\Rightarrow}$) 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 $C_1$.

## 5 ALGORITHM 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.

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.

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.

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.

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.

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.

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