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

1 INTRODUCTION

2 OVERVIEW

2.1 What types do we infer?

2.2 The Language of Types

The types of $\mathrm{F}^{\pm}\exists$ 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:

– α^- 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.
+ α^+ 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 $\{c\}$.

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

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

Definitional Equalities. For simplicity, we assume alpha-equivalent terms 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 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?`
- $+$ $\{c\}$ is a value corresponding to a thunked or suspended computation;
- $\pm$ $(c : N)$ 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 λ;
- $-$ **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 $\forall$) 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

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

Value Terms

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

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

$$\frac{}{\Gamma \vdash \alpha^- \leqslant \alpha^-}\ (\text{Var}^{\leqslant}_{-}) \qquad \frac{}{\Gamma \vdash \alpha^+ \geqslant \alpha^+}\ (\text{Var}^{\geqslant}_{+})$$

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

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

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

$\boxed{\Gamma \vdash N \simeq^{\leqslant} M}$ Negative equivalence $\qquad\qquad$ $\boxed{\Gamma \vdash P \simeq^{\leqslant} Q}$ Positive equivalence

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

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

A property that we extensively use is that the subtyping is reflexive and transitive. Moreover, any two *positive* types have the least upper bound, which makes the positive subtyping a semilattice.

Property 1 (Subtyping is well-defined). *The subtyping forms a preorder and is preserved under substitutions.*

3.2 Equivalence and Normalization

The subtyping relation forms a preorder on types, and thus, it induces an equivalence relation a.k.a. *bicoercibility*. 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 this, 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) once the subtyping (and thus, the equivalence) changes, we only need to modify the normalization function, while the rest of the algorithm remains the same.

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

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

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

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

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

Fig. 4. Type Normalization

3.3 Type Inference

The declarative type inference system is shown in fig. 5. The positive typing judgment $\Gamma; \Phi \vdash v \colon P$ is read as "under the type context Γ and variable context Φ, the term v is allowed to have the type P", where Φ—the variable context—is defined standardly as a set of pairs of the form $x : P$. The negative typing judgment is read similarly. To infer the resulting type of an application, the applicative let rules refer to the *Application typing* judgment. It has form of $\Gamma;\ \Phi \vdash N \bullet \vec{v} \Longrightarrow M$, which is read as "under the type context Γ and variable context Φ, the application of a function of type N to the list of arguments $\vec{v}$ is allowed to have the type M".

Non-bidirectionality. Notice that all the rules in our system are inferring.

Typing up to equivalence. As discussed in **??**, the subtyping, as a preorder, induces a non-trivial equivalence relation on types. The system must not distinguish between equivalent types, and thus, type inference must be defined up to equivalence. For this purpose, we use rules $(\simeq_+^{\text{INF}})$ and $(\simeq_-^{\text{INF}})$. The application typing is defined up to equivalence as well, although it has no special rules for that.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 5. Declarative Subtyping

Unpack. Rule $(\text{LET}^{\text{INF}}_{\exists})$ 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.

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 ($\forall^{\textsc{inf}}_{\bullet\Rightarrow\!\!\!\Rightarrow}$): the instantiation of the quantifying variables is taken out of thin air. The algorithm we present in **??** delays this instantiation until more information about it (in particular, typing constraints) is collected.

Application of an Arrow Type. Another important application rule is ($\rightarrow^{\textsc{inf}}_{\bullet\Rightarrow\!\!\!\Rightarrow}$). 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.

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.

3.4 Properties of the Declarative System

Now we present selected properties of the declarative system, which are important for the correctness of the algorithm.

As we discussed in **??**, the equivalence induced by the subtyping is not trivial. To deal with it without modifying standard operations such as unification, we use normalization. This way, it is important for the normalization to be sound and complete w.r.t. the equivalence:

Property 2 (Correctness of normalization). *For N and M well-formed in Γ, $\Gamma \vdash N \simeq^{\leqslant} M$ is equivalent to* $\mathbf{nf}\ (N) = \mathbf{nf}\ (M)$.

However, the greatest lower bound of two positive types is not always defined. Let us consider a positive type P, and two different supertypes P_1 and P_2 (i.e., $\cdot \vdash P_1 \geqslant P$, $\cdot \vdash P_2 \geqslant P$, and $\mathbf{nf}\ (P_1) \neq \mathbf{nf}\ (P_2)$. Let us also consider an arbitrary negative type N.

4 THE ALGORITHM

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.

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.

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