Implicit Polarized F: local type inference for impredicativity

HENRY MERCER, United Kingdom
CAMERON RAMSAY, United Kingdom
NEEL KRISHNASWAMI, University of Cambridge, United Kingdom

System F, the polymorphic lambda calculus, features the principle of *impredicativity*: polymorphic types may be (explicitly) instantiated at other types, enabling many powerful idioms such as Church encoding and data abstraction. Unfortunately, type applications need to be implicit for a language to be human-usable, and the problem of inferring all type applications in System F is undecidable. As a result, language designers have historically avoided impredicative type inference.

We reformulate System F in terms of call-by-push-value, and study type inference for it. Surprisingly, this new perspective yields a novel type inference algorithm which is extremely simple to implement (not even requiring unification), infers many types, and has a simple declarative specification. Furthermore, our approach offers type theoretic explanations of how many of the heuristics used in existing algorithms for impredicative polymorphism arise.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → Polymorphism**; *Functional languages.*

Additional Key Words and Phrases: type systems, impredicative polymorphism, local type inference

1 INTRODUCTION

System F, the polymorphic lambda calculus, was invented in the early 1970s by John Reynolds [16] and Jean-Yves Girard [6], and has remained one of the fundamental objects of study in the theory of the lambda calculus ever since.

Syntactically, it is a tiny extension of the simply-typed lambda calculus with type variables and quantification over them, with only five typing rules for the whole calculus. Despite its small size, it is capable of modeling inductive data types via Church encodings[1], and supports reasoning about data abstraction and modularity via the theory of parametricity [17, 25].

Offering a combination of parsimony and expressive power, System F has been important in the study of semantics, and has also served as inspiration for much work on language design [21]. However, practical languages have historically shied away from adopting the distinctive feature of System F — full impredicative polymorphism. This is because the natural specification for type inference for full System F is undecidable.

To understand this specification, consider the following two variables:

$$\mathsf{fst} : \forall \alpha, \beta.\, (\alpha \times \beta) \to \alpha$$
$$\mathsf{pair} : \mathsf{Int} \times \mathsf{String}$$

Here, fst has the polymorphic type of the first projection for pairs, and pair is a pair with a concrete type Int × String. To project out the first component in System F, we would write:

$$\mathsf{fst}\ [\mathsf{Int}]\ [\mathsf{String}]\ \mathsf{pair}$$

Note that each type parameter must be given explicitly. This is a syntactically heavy discipline: even in this tiny example, the number of tokens needed for type arguments equals the number of

[1]Indeed, any first-order function provably total in second-order arithmetic can be expressed in System F.

Authors' addresses: Henry Mercer, United Kingdom, henry@henrymercer.name; Cameron Ramsay, United Kingdom, cfr26@cantab.ac.uk; Neel Krishnaswami, Department of Computer Science and Technology, University of Cambridge, United Kingdom, nk480@cl.cam.ac.uk.

tokens which compute anything! Instead, we would prefer to write:

$$\mathsf{fst}\ \mathsf{pair}$$

Leaving the redundant type arguments *implicit* makes the program more readable. But how can we specify this? In the application fst pair, the function fst has the type $\forall \alpha, \beta.\ (\alpha \times \beta) \to \alpha$, but we wish to use it at a function type $(\mathsf{Int} \times \mathsf{String}) \to \mathsf{Int}$.

The standard technical device for using one type in the place of another is *subtyping*. With subtyping, the application rule is:

$$\frac{\Theta, \Gamma \vdash \mathsf{f} : A \qquad \Theta \vdash A \leq B \to C \qquad \Theta, \Gamma \vdash \mathsf{v} : B}{\Theta, \Gamma \vdash \mathsf{f}\ \mathsf{v} : C}$$

As long our subtype relation shows that $\forall \alpha, \beta.\ (\alpha \times \beta) \to \alpha \leq (\mathsf{Int} \times \mathsf{String}) \to \mathsf{Int}$, we are in the clear! Since we want subtyping to instantiate quantifiers, we can introduce subtyping rules for the universal type which reflect the *specialization order*:

$$\frac{\Theta \vdash A\ \mathsf{type} \qquad \Theta \vdash [A/\alpha]B \leq C}{\Theta \vdash \forall \alpha.\ B \leq C}\ \forall \mathrm{L} \qquad \frac{\Theta, \alpha \vdash B \leq C}{\Theta \vdash B \leq \forall \alpha.\ C}\ \forall \mathrm{R}$$

$$\frac{\Theta \vdash X \leq A \qquad \Theta \vdash B \leq Y}{\Theta \vdash A \to B \leq X \to Y} \qquad \frac{\alpha \in \Theta}{\Theta \vdash \alpha \leq \alpha}$$

Since a more general type like $\forall \alpha, \beta.\ (\alpha \times \beta) \to \alpha$ can be used in the place of a more specific type like $(\mathsf{Int} \times \mathsf{String}) \to \mathsf{Int}$, we take the standard subtyping rules for functions and base types, and add rules reflecting the principle that a universal type is a subtype of all of its instantiations.

This subtype relation is small, natural, and expressive. Thus, it has been the focus of a great deal of research in subtyping. Unfortunately, that research has revealed that this relation is *undecidable* [4, 22]. This has prompted many efforts to understand the causes of undecidability and to develop strategies which work around it. To contextualize our contributions, we briefly describe some of these efforts, but delay extended discussion of the related work until the end of the paper.

Impredicativity. One line of research locates the source of the undecidability in the fact that the ∀L rule is *impredicative* — it permits a quantifier to be instantiated at any type, including types with quantifiers in them.

For example, the identity function id can be typed in System F at $\forall \alpha.\ \alpha \to \alpha$, and impredicativity means that self-application is typeable with a polymorphic type:

$$\mathsf{id}\ [\forall \alpha.\ \alpha \to \alpha]\ \mathsf{id} : \forall \alpha.\ \alpha \to \alpha$$

A perhaps more useful example is the low-precedence application function in Haskell:

```
($) : ∀α. ∀β. (α → β) → α → β
($) f x = f x
```

This function simply applies a function to an argument, and is used in languages like Haskell and OCaml to avoid parentheses.

In Haskell, impredicative instantiation in the apply operator, for example in expressions like `runST $ (return True)` [2], is important enough that GHC, the primary Haskell compiler, has a special case typing rule for it [19]. However, this special case typing rule is non-modular and can lead to

[2]Here, runST has type $\forall \alpha.\ (\forall \beta.\ \mathsf{ST}\ \beta\ \alpha) \to \alpha$, which ensures that the internal state β is confined within the ST monad and remains inaccessible to the rest of the program. This polymorphic parameter of runST necessitates impredicative instantiation of $.

surprising results. For example, if we define an alias app for $, then runST $ (return True) typechecks, but runST `app` (return True) does not (unless the user is using the *ImpredicativeTypes* extension).

The reason for this special case rule is that GHC's type inference has built on a line of work [5, 11, 13] which restricts the ∀L rule to be *predicative*: universally quantified types can only be instantiated at monotypes (i.e. types which do not contain any quantifiers). This recovers decidability of subsumption, but at the price of giving up any inference for uses of impredicative types.

Type Syntax. Another line of work sites the difficulty of type inference in the inexpressivity of the type language of System F. The work on ML^F [1, 15] extends System F with a form of bounded quantification. These bounds are rich enough that the grammar of types can precisely record exactly which types a polymorphic quantifier might be instantiated at, which is enough to make the type inference problem tractable once more.

Of course, not just any language of bounds will work — the ML^F system was very carefully crafted to make inference both decidable and expressive, and is startlingly successful at this goal, only requiring type annotations on variables which are used at two different polymorphic types. Unfortunately, the ML^F algorithm is somewhat notorious for its implementation complexity, with multiple attempts to integrate it into GHC [7, 24] failing due to the difficulty of software maintenance.

Heuristic Approaches. A third line of work observes that type inference is actually easy in most cases, and that an algorithm which only does the easy things and fails otherwise may well suffice to be an algorithm which works in practice.

One of the oldest such approaches is that of Cardelli [2], which was invented as part of the implementation of F<:, a language with impredicative polymorphism and recursive types. The heuristic algorithm Cardelli proposed was to permit impredicative instantiation without tracking bounds — his algorithm simply took any bound it saw as an equation telling it what to instantiate the quantifier to. Pierce and Turner [14], seeing how well this algorithm worked in practice, formalized it and proved it sound (naming it *local type inference*), but were not able to give a clear non-algorithmic specification.

1.1 Contributions

Historically, approaches to impredicative type inference have been willing to pay a high cost in complexity in order to achieve high levels of expressiveness and power of type inference. In this paper, we rethink that historical approach by focusing upon the opposite tradeoff. We prioritize simplicity — of implementation, specification, and connections to existing type theories — over the expressiveness of type inference.

By being willing to make this sacrifice, we discover a easy-to-implement type inference algorithm for impredicative polymorphism, which has a clear declarative specification and is even easier to implement than traditional Damas-Milner type inference. Our algorithm sheds light on a classical implementation technique of Cardelli, and works over a foundational calculus, call-by-push-value.

Call-by-push-value was invented to study the interaction of evaluation order and side-effects, and has been used extensively in semantics. In this paper, we show that it is also well-adapted to being a kernel calculus for type inference by introducing Implicit Polarized F, a polarized variant of System F.

Specifically, our contributions are:

- To model type inference declaratively, we first introduce a subtyping relation for Implicit Polarized F that models the specialization order. Our subtyping relation is extremely simple, and is *almost* the "off-the-shelf" relation that one would first write down. The only unusual

restriction is that subtyping for shifts (the modalities connecting value and computation types) is invariant rather than covariant.
We then give a type system for our variant of call-by-push-value, using subtyping as a component. Surprisingly, features of call-by-push-value invented for semantic or type-theoretic reasons turn out to be perfectly suited for specifying type inference.
For example, argument lists (sometimes called spines in the literature [3, 20]) are commonly used in practical type inference systems [7, 18, 19]. They also arise in call-by-push-value due to the shift structure mediating between functions and values. This lets us naturally scope inference to individual function applications. So we infer types for fully unambiguous function calls, and otherwise require type annotations.
This illustrates our commitment to simplicity over powerful inference, and also results in very regular and predictable annotation behavior — our language looks like ordinary (polarized) System F, except that many of the "obvious" type instantiations are implicit.

- We give algorithmic subtyping rules corresponding to the specialization order, and algorithmic typing rules corresponding to declarative typing.
  Both of these algorithms are startlingly simple, as well — neither algorithm even needs unification, making them *simpler* than traditional Hindley-Damas-Milner [10] type inference! We prove that our algorithm is decidable, and that it is sound and complete with respect to the declarative specification.
- In fact, the combination of spine inference plus specialization yields an algorithm that is very similar to Cardelli's inference algorithm for bounded System F (as well as Pierce and Turner's local type inference). It has been known for decades that Cardelli's algorithm works extremely well in practice. However, its type-theoretic origins have not been fully understood, nor has it been understood how to characterize the kinds of problems that Cardelli's algorithm succeeds on using a declarative specification.
  The restrictions we impose to recover decidability of specialization turn out to be natural both from the perspective of type theory and implementation, and thereby offer a theoretically motivated reconstruction of Cardelli's algorithm. Polarity also ends up shedding light on many of the design choices underpinning a number of other type inference algorithms for System F [7, 18, 19].

An *explicit non-contribution* of this work is to propose a practical type inference algorithm for any existing language. For example, we work with a call-by-push-value calculus, which is a type structure no practical language currently has. Furthermore, we make no attempt to infer the types of arguments to functions; instead we focus solely on inferring type applications in a (polarized) variant of System F. Restricting the scope so much lets us isolate the key design issues underpinning a foundational problem in type inference.

2 IMPLICIT POLARIZED F

We first present Implicit Polarized F, a language with type inference that combines call-by-push-value style polarization with System F style polymorphism[3].

Like call-by-push-value, Implicit Polarized F partitions terms into *values* and *computations*, ascribing *positive types* to values and *negative types* to computations. Levy [9] described the difference between values and computations in terms of the operational semantics: "a value *is*, whereas a computation *does*". Another rule of thumb for differentiating between values and computations is to look at how we eliminate them: we tend to eliminate values by using pattern matching and

[3] Adding polarization does not result in us losing any expressiveness compared to typed System F: we demonstrate in the appendix that typeable System F can be embedded within Implicit Polarized F.

Values	$v ::= x \mid \{t\}$
Computations	$t ::= \lambda x : P.\, t \mid \Lambda\alpha.\, t \mid \mathsf{return}\ v \mid$
	$\quad \mathsf{let}\ x = v(s); t \mid \mathsf{let}\ x : P = v(s); t$
Argument lists	$s ::= \epsilon \mid v, s$
Positive types	$P ::= \alpha \mid \downarrow N$
Negative types	$N ::= P \to N \mid \forall\alpha.\, N \mid \uparrow P$
Typing contexts	$\Theta ::= \cdot \mid \Theta, \alpha$
Typing environments	$\Gamma ::= \cdot \mid \Gamma, x : P$

Fig. 1. Implicit Polarized F

computations by supplying an argument. For example, datatypes are values because they are eliminated by pattern matching, whereas functions are computations because they are eliminated by supplying arguments.

We present the term language for Implicit Polarized F in Figure 1. Terms are split between values and computations as follows:

- Variables x are values, since this is one of the invariants of call-by-push-value.
- Lambda abstractions $\lambda x : P.\, t$ are computations, not values, since they are eliminated by passing an argument. We can think of a lambda abstraction as a computation t of type N with holes in it representing the (positive) bound variable, so lambda abstractions have the negative type $P \to N$.
- Thunks $\{t\}$ are values that suspend arbitrary computations. Surrounding a computation t of type N with braces suspends that computation, giving us a thunk $\{t\}$ of type $\downarrow N$.
  To create a traditional function value, we make a lambda abstraction $\lambda x : P.\, t$ into a thunk $\{\lambda x : P.\, t\}$. Traditional thunks therefore have the type $\downarrow(P \to N)$.
- System-F style type abstractions $\Lambda\alpha.\, t$ are computations of type $\forall\alpha.\, N$. Our choice of polarities for the types of polymorphic terms is motivated by the correspondence between function abstraction $\lambda x : P.\, t$ and type abstraction $\Lambda\alpha.\, t$.
- The let forms $\mathsf{let}\ x = v(s); t$ and $\mathsf{let}\ x : P = v(s); t$ are sequencing computations that operate much like the bind operation in a monad. Each form takes the thunked computation v, passes it the arguments s it needs, binds the result to the variable x and continues the computation t.
- $\mathsf{return}\ v$ is a trivial computation that just returns a value, completing the sequencing monad. We can also use return to return values from a function — in fact since all the terminal symbols for terms live in the grammar of values, the syntax mandates that functions eventually return a value. The type structure of returned values is symmetric to that of thunks: returned values have type $\uparrow P$.

Besides the addition of type annotations to guide inference and some differences to the syntax, our language is nothing more than call-by-push-value plus polymorphism. Our use of argument lists serves only to reverse the order of argument lists from Levy's original work so that our syntax matches traditional languages with n-ary functions like JavaScript and C. Just like those languages, all of a function's arguments must be passed at once in Implicit Polarized F.

$\boxed{\Theta; \Gamma \vdash e : A}$ The term e synthesizes the type A

$\boxed{\Theta; \Gamma \vdash s : N \gg M}$ When passed to a head of type N, the argument list s synthesizes the type M

$$\frac{x : P \in \Gamma}{\Theta; \Gamma \vdash x : P}\ \text{Dvar} \qquad \frac{\Theta; \Gamma \vdash t : N}{\Theta; \Gamma \vdash \{t\} : \downarrow N}\ \text{Dthunk}$$

$$\frac{\Theta; \Gamma, x : P \vdash t : N}{\Theta; \Gamma \vdash \lambda x : P.\, t : P \to N}\ \text{D}\lambda\text{abs} \qquad \frac{\Theta, \alpha; \Gamma \vdash t : N}{\Theta; \Gamma \vdash \Lambda\alpha.\, t : \forall \alpha.\, N}\ \text{Dgen}$$

$$\frac{\Theta; \Gamma \vdash v : P}{\Theta; \Gamma \vdash \mathsf{return}\ v : \uparrow P}\ \text{Dreturn}$$

$$\frac{\Theta; \Gamma \vdash v : \downarrow M \qquad \Theta; \Gamma \vdash s : M \gg \uparrow Q \qquad \Theta \vdash \uparrow Q \leq^{-} \uparrow P \qquad \Theta; \Gamma, x : P \vdash t : N}{\Theta; \Gamma \vdash \mathsf{let}\ x : P = v(s); t : N}\ \text{Dambiguouslet}$$

$$\frac{\begin{array}{c}\Theta; \Gamma \vdash v : \downarrow M \qquad \Theta; \Gamma \vdash s : M \gg \uparrow Q \\ \Theta; \Gamma, x : Q \vdash t : N \qquad \forall P.\ \text{if}\ \Theta; \Gamma \vdash s : M \gg \uparrow P\ \text{then}\ \Theta \vdash Q \cong^{+} P\end{array}}{\Theta; \Gamma \vdash \mathsf{let}\ x = v(s); t : N}\ \text{Dunambiguouslet}$$

$$\frac{}{\Theta; \Gamma \vdash \epsilon : N \gg N}\ \text{Dspinenil} \qquad \frac{\Theta; \Gamma \vdash v : P \qquad \Theta \vdash P \leq^{+} Q \qquad \Theta; \Gamma \vdash s : N \gg M}{\Theta; \Gamma \vdash v, s : (Q \to N) \gg M}\ \text{Dspinecons}$$

$$\frac{\Theta \vdash P\ \text{type}^{+} \qquad \Theta; \Gamma \vdash s : [P/\alpha]N \gg M}{\Theta; \Gamma \vdash s : (\forall \alpha.\, N) \gg M}\ \text{Dspinetypeabs}$$

Fig. 2. Declarative type system

2.1 Declarative typing

We present in Figure 2 a declarative type system for Implicit Polarized F. Our system has simple, mostly syntax directed rules, with its main complexity lying in the unusual premise to Dunambiguouslet and the rules for typing argument lists.

The system has five main judgments:

- $\Theta; \Gamma \vdash v : P$: In the context Θ and typing environment Γ, the value v synthesizes the positive type P.
- $\Theta; \Gamma \vdash t : N$: In the context Θ and typing environment Γ, the computation t synthesizes the negative type N.
- $\Theta; \Gamma \vdash s : N \gg M$: In the context Θ and typing environment Γ, and when passed to a head of type N, the argument list s synthesizes the type M.
- $\Theta \vdash P \leq^{+} Q$: In the context Θ, P is a positive subtype of Q.
- $\Theta \vdash N \leq^{-} M$: In the context Θ, N is a negative subtype of M. We reverse the alphabetical order of M and N in the negative judgment to better indicate the symmetry between positive and negative subtyping — we will see what this symmetry is in Section 4.1.

When we want to be ambiguous between values and computations, we write $\Theta; \Gamma \vdash e : A$ and $\Theta \vdash A \leq^{\pm} B$.

Our inference rules are the following:

- Dvar lets us reference variables within the type environment Γ.
- Dthunk states that if we know that a computation has type N, then thunking it produces a value of type $\downarrow N$.
- Dλabs is the standard typing rule for lambda abstraction. The value hypothesis $x : P$ in this rule maintains the invariant that the type environment only contains bindings to values.
- Dgen is to Dλabs as type abstraction is to function abstraction. We add a type variable to the typing context, as opposed to adding a variable binding to the typing environment.
- Dreturn complements Dthunk: if a value has type P, then returning it produces a computation of type $\uparrow P$.
- Dambiguouslet allows us to let-bind the results of function applications.
  - In the first premise, $\Theta; \Gamma \vdash v : \downarrow M$, we require v to be a thunked computation.
  - We then take the type M of this computation and use the spine judgment $\Theta; \Gamma \vdash s : M \gg \uparrow Q$ to type the function application $v(s)$. The requirement that the argument list produces an upshifted type $\uparrow Q$ means that the type has to be maximally applied. This encodes the fact that partial application is forbidden.
    Typically v will be a downshifted function and s will be the argument list to pass to that function. For example, v could be a function stored within the type environment, e.g. f when $f : \downarrow(P \to N) \in \Gamma$, or one written inline $\{\lambda x : P.\, t\}$. However we can also use an embedded value $\{\mathsf{return}\ v\}$ as a head along with an empty argument list to let-bind values.
  - Now that we have the type $\uparrow Q$ that the argument list takes the head to, we use the subtyping judgment $\Theta \vdash \uparrow Q \leq^{-} \uparrow P$ to check that the type synthesized by the function application is compatible with the annotation P. For our algorithm to work, we need to check the stronger compatibility constraint that $\uparrow Q$ is a subtype of $\uparrow P$, rather than just checking that Q is a subtype of P.
  - Finally, $\Theta; \Gamma, x : P \vdash t : N$ tells us that the type of the let term is given by the type of t in the type environment Γ extended with x bound to P.
- Dunambiguouslet is a variant of Dambiguouslet that lets us infer the return types of function applications when they are unambiguous. The first and second premises are identical to those of Dambiguouslet, and the third premise is the same as the last premise of Dambiguouslet except that it binds x directly to the type synthesized by the function application.
  However the final premise is unusual, and one might wonder whether it is in fact well founded. This premise encodes the condition that in order to omit the annotation, the return type of the function application must be unambiguous. By quantifying over all possible inferred types, we check that there is only one type (modulo isomorphism) that can be inferred for the return type of the function application. If every inferred type is equivalent to Q, then we can arbitrarily choose Q to bind x to, since any other choice would synthesize the same type for the let-binding.
  The well-foundedness of our rule stems from the use of a syntactically smaller subterm in the premise of Dunambiguouslet compared to the conclusion. In $\Theta; \Gamma \vdash s : M \gg \uparrow P$, s is a smaller subterm than $\mathsf{let}\ x = v(s); t$, and every subderivation of that premise acts on a smaller subterm too.
  A consequence of this unusual premise is that the soundness and completeness theorems end up being mutually recursive.

$\boxed{\Theta \vdash A\ \text{type}^{\pm}}$ In the context Θ, A is a well-formed positive/negative type

$$\frac{\alpha \in \Theta}{\Theta \vdash \alpha\ \text{type}^{+}}\ \text{Twfuvar} \qquad \frac{\Theta \vdash N\ \text{type}^{-}}{\Theta \vdash \downarrow N\ \text{type}^{+}}\ \text{Twfshift}\downarrow \qquad \frac{\Theta, \alpha \vdash N\ \text{type}^{-}}{\Theta \vdash \forall \alpha.\, N\ \text{type}^{-}}\ \text{Twfforall}$$

$$\frac{\Theta \vdash P\ \text{type}^{+} \qquad \Theta \vdash N\ \text{type}^{-}}{\Theta \vdash P \to N\ \text{type}^{-}}\ \text{Twfarrow} \qquad \frac{\Theta \vdash P\ \text{type}^{+}}{\Theta \vdash \uparrow P\ \text{type}^{-}}\ \text{Twfshift}\uparrow$$

Fig. 3. Well-formedness of declarative types

$\boxed{\Theta \vdash A \leq^{\pm} B}$ In the context Θ, A is a positive/negative declarative subtype of B

$$\frac{\Theta \vdash \alpha\ \text{type}^{+}}{\Theta \vdash \alpha \leq^{+} \alpha}\ \leq^{\pm}\text{Drefl} \qquad \frac{\Theta \vdash M \leq^{-} N \qquad \Theta \vdash N \leq^{-} M}{\Theta \vdash \downarrow N \leq^{+} \downarrow M}\ \leq^{\pm}\text{Dshift}\downarrow$$

$$\frac{\Theta, \alpha \vdash N \leq^{-} M}{\Theta \vdash N \leq^{-} \forall \alpha.\, M}\ \leq^{\pm}\text{Dforallr} \qquad \frac{\Theta \vdash P\ \text{type}^{+} \qquad \Theta \vdash [P/\alpha]N \leq^{-} M}{\Theta \vdash \forall \alpha.\, N \leq^{-} M}\ \leq^{\pm}\text{Dforalll}$$

$$\frac{\Theta \vdash Q \leq^{+} P \qquad \Theta \vdash N \leq^{-} M}{\Theta \vdash P \to N \leq^{-} Q \to M}\ \leq^{\pm}\text{Darrow} \qquad \frac{\Theta \vdash Q \leq^{+} P \qquad \Theta \vdash P \leq^{+} Q}{\Theta \vdash \uparrow P \leq^{-} \uparrow Q}\ \leq^{\pm}\text{Dshift}\uparrow$$

Fig. 4. Declarative subtyping

The last three rules teach us how to type function applications:

- Dspinenil states that an empty argument list does not change the type of the head of a function application.
- Dspinecons tells us how to type a non-empty argument list. The argument list v, s takes $Q \to N$ to M if the value v synthesizes a type P, this type P is either Q or a subtype of it, and the remaining argument list s takes N to M.
- Dspinetypeabs lets us instantiate the head that an argument list can be passed to by replacing a type variable α with a well-formed (see Figure 3) type P. Note that the instantiation is implicit — there is no annotation for P, so we are doing inference here.

2.2 Subtyping

To enable implicit type application, our type system uses a subtyping relation $\Theta \vdash A \leq^{\pm} B$ which reifies the specialization order, expressing when a term of type A can be safely used where a term of type B is expected. This relation is defined in Figure 4 and consists of the following rules:

- $\leq^{\pm}$Drefl is the standard rule for type variables: if we are expecting a term of type α, then the only terms we can safely use in its place are those that also have type α.
- $\leq^{\pm}$Darrow is the classic rule for function subtyping: we can replace a term of type $Q \to M$ with a function that takes inputs that are at least as general as Q and produces outputs that are at least as specific as M.

$\boxed{\Theta \vdash A \cong^{\pm} B}$ In the context Θ, the types A and B are isomorphic

$$\Theta \vdash A \cong^{\pm} B \text{ iff } \Theta \vdash A \leq^{\pm} B \text{ and } \Theta \vdash B \leq^{\pm} A.$$

Fig. 5. Isomorphic types

- $\leq^{\pm}$Dforalll and $\leq^{\pm}$Dforallr correspond to the left and right rules governing $\forall$ quantifiers in the LK sequent calculus system. Surprisingly, the $\leq^{\pm}$Dforalll rule does not have any restrictions beyond well-formedness: P is an arbitrary well-formed type, so full impredicative instantiation is possible.
- $\leq^{\pm}$Dshift$\downarrow$ and $\leq^{\pm}$Dshift$\uparrow$ are unusual since the standard denotational and operational semantics of shifts both give rise to covariant rules. However using these covariant rules would lead to an undecidable system [4, 22]. To restore decidability, we use more restrictive invariant shift rules.
  This idea is motivated by a similar restriction that Cardelli [2] introduced as a heuristic for type inference in an implementation of System $F_{<:}$. Instead of using subtyping to infer the types assigned to type parameters, Cardelli used first-order unification to "synthesize" these types.
  Since the goal of unification is to make expressions equal, an equivalent notion of Cardelli's restriction in our system is requiring the types to be *isomorphic*. We say that types A and B are isomorphic, denoted as $\Theta \vdash A \cong^{\pm} B$, if both $\Theta \vdash A \leq^{\pm} B$ and $\Theta \vdash B \leq^{\pm} A$ hold (see Figure 5). Indeed, this requirement corresponds exactly to the premises of the shift rules. To our knowledge, this is the first time that Cardelli's restriction has been presented as part of a declarative specification of typing.

3 EXAMPLES

In this section we characterize the behavior of our system by giving some examples of terms and subtyping relationships that are permitted and some that are not.

3.1 Subtyping

- Our subtyping relation allows us to swap quantifiers at identical depths. As indicated by the subtyping in both directions below, we can substitute a term of either of the types below for a term of the other type:

$$\forall \alpha, \beta. \downarrow(\alpha \to \uparrow\beta) \to [\alpha] \to \uparrow[\beta] \quad \underset{-}{\overset{\leq^{-}}{>}} \quad \forall \beta, \alpha. \downarrow(\alpha \to \uparrow\beta) \to [\alpha] \to \uparrow[\beta]$$

- Subtypes can push in quantifiers as long as those quantifiers do not cross a shift boundary:

$$\forall \alpha. \alpha \to (\forall \beta. \beta \to \uparrow(\alpha \times \beta)) \quad \leq^{-} \quad \forall \alpha, \beta. \alpha \to \beta \to \uparrow(\alpha \times \beta)$$

- Unusually, our subtyping relation allows the instantiations of type variables to be impredicative. For example, here we can instantiate the α in $[\alpha]$ to be an identity function $\forall \beta. \beta \to \uparrow\beta$, giving us a list of *polymorphic* identity functions. Predicative systems would allow us to produce a list of identity functions that are all parameterized over the same type, i.e. $\forall \beta. [\beta \to \uparrow\beta]$, but the list produced by our system $[\forall \beta. \beta \to \uparrow\beta]$ is truly polymorphic with each element of the list having its own type variable.

head	$: \downarrow(\forall\alpha.\, [\alpha] \to \uparrow\alpha)$	inc	$: \downarrow(\mathsf{Int} \to \uparrow\mathsf{Int})$
tail	$: \downarrow(\forall\alpha.\, [\alpha] \to \uparrow[\alpha])$	choose	$: \downarrow(\forall\alpha.\, \alpha \to \alpha \to \uparrow\alpha)$
[]	$: \downarrow(\forall\alpha.\, \uparrow[\alpha])$	poly	$: \downarrow(\downarrow(\forall\alpha.\, \alpha \to \uparrow\alpha) \to \uparrow(\mathsf{Int} \times \mathsf{Bool}))$
(::)	$: \downarrow(\forall\alpha.\, \alpha \to [\alpha] \to \uparrow[\alpha])$	auto	$: \downarrow(\downarrow(\forall\alpha.\, \alpha \to \uparrow\alpha) \to (\forall\alpha.\, \alpha \to \uparrow\alpha))$
single	$: \downarrow(\forall\alpha.\, \alpha \to \uparrow[\alpha])$	auto'	$: \downarrow(\forall\alpha.\downarrow(\forall\beta.\, \beta \to \uparrow\beta) \to \alpha \to \uparrow\alpha)$
append	$: \downarrow(\forall\alpha.\, [\alpha] \to [\alpha] \to \uparrow[\alpha])$	app	$: \downarrow(\forall\alpha, \beta.\downarrow(\alpha \to \uparrow\beta) \to \alpha \to \uparrow\beta)$
length	$: \downarrow(\forall\alpha.\, [\alpha] \to \uparrow\mathsf{Int})$	revapp	$: \downarrow(\forall\alpha, \beta.\, \alpha \to \downarrow(\alpha \to \uparrow\beta) \to \uparrow\beta)$
map	$: \downarrow(\forall\alpha, \beta.\downarrow(\alpha \to \beta) \to [\alpha] \to \uparrow[\beta])$	flip	$: \downarrow(\forall\alpha, \beta, \gamma.\downarrow(\alpha \to \beta \to \uparrow\gamma) \to \beta \to \alpha \to \uparrow\gamma)$
id	$: \downarrow(\forall\alpha.\, \alpha \to \uparrow\alpha)$	runST	$: \downarrow(\forall\alpha.\downarrow(\forall\beta.\, \mathsf{ST}\ \beta\ \alpha) \to \uparrow\alpha)$
ids	$: [\downarrow(\forall\alpha.\, \alpha \to \uparrow\alpha)]$	argST	$: \downarrow(\forall\alpha.\, \mathsf{ST}\ \alpha\ \mathsf{Int})$
a9	$: \downarrow(\forall\alpha.\downarrow(\alpha \to \uparrow\alpha) \to [\alpha] \to \uparrow\alpha)$	c8	$: \downarrow(\forall\alpha.\, [\alpha] \to [\alpha] \to \uparrow\alpha)$
h	$: \downarrow(\mathsf{Int} \to (\forall\alpha.\, \alpha \to \uparrow\alpha))$	k	$: \downarrow(\forall\alpha.\, \alpha \to [\alpha] \to \uparrow\alpha)$
lst	$: [\downarrow(\forall\alpha.\, \mathsf{Int} \to \alpha \to \uparrow\alpha)]$	r	$: \downarrow(\downarrow(\forall\alpha.\, \alpha \to \forall\beta.\, \beta \to \uparrow\beta) \to \uparrow\mathsf{Int})$

Fig. 6. Environment for the examples from Serrano et al. [19], translated into Implicit Polarized F

$$\forall\alpha.\, \uparrow[\alpha] \quad \leq^- \quad \uparrow[\downarrow(\forall\beta.\, \beta \to \uparrow\beta)]$$

- An important restriction of our system that is essential to decidability is that types underneath shifts must be isomorphic. For example, we must establish both $M \leq^- N$ and $N \leq^- M$ before we can infer that $\downarrow M \leq^+ \downarrow N$.
  This means that terms of either of the types below may not be substituted for terms of the other type. While $\forall\alpha, \beta.\, \alpha \to \beta \to \uparrow(\alpha \times \beta)$ is indeed a subtype of $\mathsf{Int} \to \mathsf{String} \to \uparrow(\mathsf{Int} \times \mathsf{String})$, the subtyping does not hold the other way around. Therefore in this example where we zip a list of integers and a list of strings together, the type of the pairing function $\downarrow(\mathsf{Int} \to \mathsf{String} \to \uparrow(\mathsf{Int} \times \mathsf{String}))$ is neither a subtype nor a supertype of the type of the generic pairing function $\downarrow(\forall\alpha, \beta.\, \alpha \to \beta \to \uparrow(\alpha \times \beta))$.

$$\downarrow(\mathsf{Int} \to \mathsf{String} \to \uparrow(\mathsf{Int} \times \mathsf{String})) \to [\mathsf{Int}] \to [\mathsf{String}] \to \uparrow[\mathsf{Int} \times \mathsf{String}]$$
$$\not\uparrow \quad \not\downarrow$$
$$\downarrow(\forall\alpha, \beta.\, \alpha \to \beta \to \uparrow(\alpha \times \beta)) \to [\mathsf{Int}] \to [\mathsf{String}] \to \uparrow[\mathsf{Int} \times \mathsf{String}]$$

3.2 Typing

To illustrate Implicit Polarized F's type system, we translate the examples used in the comparison of type systems in Serrano et al. [19] (GI) to Implicit Polarized F and investigate which of these translations (a) typecheck without needing additional annotations beyond the GI example, (b) typecheck but require additional annotations beyond the GI example, and (c) do not typecheck. These examples make use of a few common functions whose types we give in Figure 6.

Despite the extreme bias towards simplicity that is reflected in our straightforward local heuristic, surprisingly many of these examples typecheck. We discuss a few characteristic ones below; the full set are in Figure 7.

	GI example	**Implicit Polarized F translation**	
A1	const2 = λx y. y	let const2 = {return {$\Lambda\alpha.\,\Lambda\beta.\,\lambda$x : $\alpha.\,\lambda$y : β. return y}}; ...	Ann
A2	choose id	λx : $\downarrow(\forall\alpha.\,\alpha \to \uparrow\alpha)$. (let t = choose id x; return t)	Ann
A3	choose [] ids	let n : $[\downarrow(\forall\alpha.\,\alpha \to \uparrow\alpha)]$ = []; let t = choose n ids; return t	Ann
A4	λx : $(\forall\alpha.\,\alpha \to \alpha)$. x x	λx : $(\forall\alpha.\,\alpha \to \uparrow\alpha)$. (let y = x x; return y)	✓
A5	id auto	let t = id auto; return t	✓
A6	id auto′	let t = id auto′; return t	✓
A7	choose id auto	let t = choose id auto; return t	×
A8	choose id auto′	let t = choose id auto′; return t	×
A9	a9 (choose id) ids	let f = {return {λx : $\downarrow(\forall\alpha.\,\alpha \to \uparrow\alpha)$. let y = choose id x; return y}}; let t = a9 f ids; return t	Ann
A10	poly id	let t = poly id; return t	✓
A11	poly (λx. x)	let t = poly {$\Lambda\alpha.\,\lambda$x : α. return x}; return t	Ann
A12	id poly (λx. x)	let x = id poly; let t = x {$\Lambda\alpha.\,\lambda$y : α. return y}; return t	Ann
B1	λf. (f 1, f True)	λf : $\downarrow(\forall\alpha.\,\alpha \to \uparrow\alpha)$. (let l = f 1; let r = f true; return (l, r))	Ann
B2	λxs. poly (head xs)	λxs : $[\downarrow(\forall\alpha.\,\alpha \to \uparrow\alpha)]$. (let x = head xs; let y = poly x; return y)	Ann
C1	length ids	let t = length ids; return t	✓
C2	tail ids	let t = tail ids; return t	✓
C3	head ids	let t = head ids; return t	✓
C4	single id	let t = single id; return t	✓
C5	cons id ids	let t = cons id ids; return t	✓
C6	cons (λx. x) ids	let t = cons {$\Lambda\alpha.\,\lambda$x : α. return x} ids; return t	Ann
C7	append (single inc) (single ids)	let x = single inc; let y = single id; let t = append x y; return t	×
C8	c8 (single id) ids	let x = single id; let t = c8 x ids; return t	✓
C9	map poly (single id)	let x = single id; let t = map poly x; return t	✓
C10	map head (single ids)	let x = single ids; let t = map head x; return t	×
D1	app poly id	let t = app poly id; return t	✓
D2	revapp id poly	let t = revapp id poly; return t	✓
D3	runST argST	let t = runST argST; return t	✓
D4	app runST argST	let t = app runST argST; return t	×
D5	revapp argST runST	let t = revapp argST runST; return t	×
E1	k h lst	let t = k h lst; return t	×
E2	k ($\lambda x.\,h\,x$) lst	let f = {return {$\Lambda\alpha.\,\lambda$y : Int. return {λx : α. let z = h y z; return z}}}; let t = k f lst; return t	×
E3	r ($\lambda x\,y.\,y$)	let t = r {$\Lambda\alpha.\,\Lambda\beta.\,\lambda$x : $\alpha.\,\lambda$y : β. return y}; return t	×

Fig. 7. Examples used in the comparison of type systems in Serrano et al. [19] along with their translations to Implicit Polarized F. In the right hand column:

- ✓ indicates that an example typechecks without needing additional annotations beyond the GI example.
- Ann indicates that the example typechecks, but additional annotations are required beyond the GI example.
- × indicates that the example does not typecheck in Implicit Polarized F.

Shifts make types much more fine-grained. The shift structure in Implicit Polarized F types makes them much more fine-grained compared to systems like GI. For example, in Implicit Polarized F the number of arguments we can pass to a function is encoded in its type. If we instantiate id at $\downarrow(\forall\alpha.\,\alpha \to \uparrow\alpha)$, it still takes only a single argument. In contrast, in systems like GI, instantiating id at $\forall\alpha.\,\alpha \to \alpha$ gives us a function that can take two arguments: first id and then an arbitrary value.

As a result, while id instantiated at $\forall\alpha.\,\alpha \to \alpha$ and auto represent the same thing in systems like GI, id instantiated at $\downarrow(\forall\alpha.\,\alpha \to \uparrow\alpha)$ and auto represent fundamentally different things in Implicit Polarized F. Therefore in A7 we can not synthesize a consistent return type for choose id auto and so it does not typecheck.

Examples A8, C7, and E2 do not work for similar reasons. For instance, in C7 our use of invariant shifts prevents us from using single ids at a less polymorphic type.

Another example of the shift structure in Implicit Polarized F making types much more fine-grained is A3: a direct translation let t = choose [] ids; return t will not typecheck since [] has a shift at its top-level while ids does not. As a result, we need to unwrap the top-level shift in the type of [] using a let-binding like so: let n : $[\downarrow(\forall\alpha.\,\alpha \to \uparrow\alpha)]$ = [].

Hyperlocal inference. Inference in Implicit Polarized F refuses to incorporate far away information. As encoded in the last premise of the Dunambiguouslet rule, we must annotate a let-bound variable unless its type is completely determined by the application $v(s)$.

For example, in A3 there are many types we can infer for the empty list constructor [], for instance [Int], $[\forall\alpha.\,\alpha]$, and $\forall\alpha.\,[\alpha]$. We can see from the later application choose n ids that $[\downarrow(\forall\alpha.\,\alpha \to \uparrow\alpha)]$ is the right choice for this type. However, Implicit Polarized F only considers the immediate context of the function application. In this context the type for n is ambiguous, so we need to annotate the let-binding.

While A3 needs an annotation, examples like C4 do not. While the type $[\downarrow(\forall\alpha.\,\alpha \to \uparrow\alpha)]$ that is synthesized for the let-bound variable in C4 contains universal quantification, this quantification occurs underneath a shift. Since shifts are invariant, we can not perform subtyping underneath them, so this type is the only one modulo isomorphism that we can infer for the let-bound variable.

We sometimes need to eta-expand terms to get them to type check. An example of this is C10 map head (single ids). Here map expects an argument of type $\downarrow(\alpha \to \uparrow\beta)$, but head has type $\downarrow(\forall\alpha.\,[\alpha] \to \uparrow\alpha)$. Since shifts are invariant in Implicit Polarized F, these types are incompatible. We can remedy this by eta-expanding the term, for instance for C10 we can create a new term $\lambda x : \downarrow(\forall\alpha.\,\alpha \to \uparrow\alpha).$ let y = head x; return y that specializes head to the type $\downarrow([\downarrow\forall\alpha.\,\alpha \to \uparrow\alpha] \to \uparrow\downarrow(\forall\alpha.\,\alpha \to \uparrow\alpha))$. Examples D4, D5, E1, and E3 fail to typecheck for similar reasons.

Another situation where eta-expansion is necessary is when replicating partial application. Since Implicit Polarized F does not allow partial application, partial applications like choose id in GI must be represented as $\lambda x : \downarrow(\forall\alpha.\,\alpha \to \uparrow\alpha).$ let y = choose id x; return y in Implicit Polarized F. Since we syntactically require users to annotate lambda-bound variables, these examples all require annotations in Implicit Polarized F. This affects examples A2, A9, and E2.

Implicit Polarized F tends to prefer inferring impredicative types to inferring type schemes. Adding impredicativity to Hindley-Milner style systems means that, without changes to the type language like the introduction of bounded quantification, terms are no longer guaranteed to have a most general type. For instance, when typing single id we find that $[\forall\alpha.\,\alpha \to \alpha]$ and $\forall\alpha.\,[\alpha \to \alpha]$ are equally general. Due to its simple local heuristic, Implicit Polarized F will instantiate the type variable in single impredicatively, thereby typing single as $[\forall\alpha.\,\alpha \to \alpha]$. In GI, the choice we make depends on whether the type variable α in the definition of single appears under a type constructor

Positive types	$P ::= \ldots \mid \hat{\alpha}$
Contexts	$\Theta ::= \ldots \mid \Theta, \hat{\alpha} \mid \Theta, \hat{\alpha} = P$

Fig. 8. Additions to declarative types and contexts to form their algorithmic counterparts

in any of the arguments to it. Since it does not, GI assigns α a "top-level monomorphic" type, thereby typing single as $\forall\alpha.\, [\alpha \to \alpha]$.

Implicit Polarized F's preference for inferring impredicative types is not an unquestionable pro or con: while it allows us to infer examples like C8 and C9 that systems that prefer type schemes can not, systems that prefer type schemes can deal better with eta-reduced terms.

4 ALGORITHMIC TYPING

In this section, we discuss how to implement our declarative system algorithmically.

There are two rules that prevent us from directly implementing the declarative system. These are Dspinetypeabs and $\leq^{\pm}$Dforalll, both of which have a type P that appears in the premises but not in the conclusion of each rule. As a result, a direct implementation of type inference would need to somehow invent a type P out of thin air.

To get an algorithm, we follow Dunfield and Krishnaswami [5] in replacing these types P with *existential type variables* $\hat{\alpha}$. These represent unsolved types that the algorithm will solve at a later stage. Since type variables and their solutions are positive, existential type variables must also be positive.

The introduction of existential variables has some knock-ons on our system: each judgment needs to be adapted to manage existential variables and their instantiation. For instance, as shown in Figure 8, the contexts of algorithmic judgments will contain not only type variables, but also solved and unsolved existential type variables.

4.1 Algorithmic subtyping

We first describe the algorithmic judgments for subtyping. These will be used to solve all relevant existential type variables.

Our algorithm for subtyping consists of two mutually recursive judgments $\Theta \vdash P \leq^{+} Q \dashv \Theta'$ and $\Theta \vdash N \leq^{-} M \dashv \Theta'$. Each judgment takes an input context Θ, checks that the left-hand side type is a subtype of the right-hand side type, and produces an output context Θ'. The shape of this output context is identical to that of the corresponding input context: the only difference between them is that the output context contains the solutions to any newly solved existential variables. Since existential type variables can only have a single solution, output contexts can only solve existential variables that are unsolved in the input context.

As a result of polarization, existential type variables only appear on the left-hand side of negative subtyping judgments and the right-hand side of positive subtyping judgments. This *groundness invariant* is critical to the workings of our algorithm. Since we will frequently make use of this invariant, we briefly introduce some terminology related to it. We call types that do not contain any existential variables *ground*. The side of a judgment that can not contain any existential type variables is the *ground side*, and the opposite side is the *non-ground side*. We will observe symmetries between properties of the positive and negative judgments based on the groundness of the types (see Lemma 6.1 for instance). This is why we swap the alphabetical order of M and N when writing negative judgments.

$\boxed{\Theta \vdash A \leq^{\pm} B \dashv \Theta'}$ In the context Θ, A checks algorithmically as a subtype of B, producing the context Θ'

$$\frac{}{\Theta_L, \alpha, \Theta_R \vdash \alpha \leq^{+} \alpha \dashv \Theta_L, \alpha, \Theta_R} \leq^{\pm}\mathsf{Arefl} \qquad \frac{\Theta_L \vdash P\ \mathrm{type}^{+} \qquad P\ \mathrm{ground}}{\Theta_L, \hat{\alpha}, \Theta_R \vdash P \leq^{+} \hat{\alpha} \dashv \Theta_L, \hat{\alpha} = P, \Theta_R} \leq^{\pm}\mathsf{Ainst}$$

$$\frac{\Theta \vdash M \leq^{-} N \dashv \Theta' \qquad \Theta' \vdash N \leq^{-} [\Theta']M \dashv \Theta''}{\Theta \vdash \downarrow N \leq^{+} \downarrow M \dashv \Theta''} \leq^{\pm}\mathsf{Ashift}{\downarrow}$$

$$\frac{\Theta, \hat{\alpha} \vdash [\hat{\alpha}/\alpha]N \leq^{-} M \dashv \Theta', \hat{\alpha}[= P] \qquad M \neq \forall \beta. M'}{\Theta \vdash \forall \alpha. N \leq^{-} M \dashv \Theta'} \leq^{\pm}\mathsf{Aforalll}$$

$$\frac{\Theta, \alpha \vdash N \leq^{-} M \dashv \Theta', \alpha}{\Theta \vdash N \leq^{-} \forall \alpha. M \dashv \Theta'} \leq^{\pm}\mathsf{Aforallr} \qquad \frac{\Theta \vdash Q \leq^{+} P \dashv \Theta' \qquad \Theta' \vdash [\Theta']N \leq^{-} M \dashv \Theta''}{\Theta \vdash P \to N \leq^{-} Q \to M \dashv \Theta''} \leq^{\pm}\mathsf{Aarrow}$$

$$\frac{\Theta \vdash Q \leq^{+} P \dashv \Theta' \qquad \Theta' \vdash [\Theta']P \leq^{+} Q \dashv \Theta''}{\Theta \vdash \uparrow P \leq^{-} \uparrow Q \dashv \Theta''} \leq^{\pm}\mathsf{Ashift}{\uparrow}$$

Fig. 9. Algorithmic subtyping

We present our algorithmic subtyping rules in Figure 9.

- $\leq^{\pm}$Arefl is identical to our declarative rule for type variables, however we have moved the $\Theta \vdash \alpha\ \mathrm{type}^{+}$ judgment into an equivalent requirement that the input and output contexts have the form $\Theta_L, \alpha, \Theta_R$.
- $\leq^{\pm}$Ainst is a new rule that describes how to solve existential type variables. The idea is that if we reach a subtyping judgment like $P \leq^{+} \hat{\alpha}$, then there is no further work to be done, and we can just set $\hat{\alpha}$ to be equal to P. Since this rule is the only way we solve existential variables, all solutions to existential variables are ground.
  The premises of $\leq^{\pm}$Ainst ensure that the rule preserves well-formedness: as shown in Figure 14, our contexts are ordered, and a solution P to an existential variable $\hat{\alpha}$ must be well-formed with respect to the context items prior to $\hat{\alpha}$, i.e. Θ_L. This means that the solution for $\hat{\alpha}$ can only contain type variables that were in scope when $\hat{\alpha}$ was introduced: an important property for the soundness of the system.
- $\leq^{\pm}$Aforalll implements the strategy we described to make the $\leq^{\pm}$Dforalll rule algorithmic: rather than replacing the type variable α with a ground type P, we procrastinate deciding what P should be and instead replace α with a fresh existential type variable $\hat{\alpha}$. The bracketed $\hat{\alpha}[= P]$ in the output context indicates that the algorithm doesn't have to solve $\hat{\alpha}$: we will later see in Section 6.1.1 that it solves $\hat{\alpha}$ if and only if the type variable α appears in the term N.
  The second premise of $\leq^{\pm}$Aforalll states that the right-hand side must not be a prenex polymorphic form. We introduce this premise to ensure that the system is strictly syntax-oriented: we have to eliminate all the quantifiers on the right with $\leq^{\pm}$Aforallr before eliminating any quantifiers on the left with $\leq^{\pm}$Aforalll. In general, introducing a premise like this could be problematic for completeness, as in completeness (see Theorem 7.2) we want to be able to take apart the declarative derivation in the same order that the algorithmic

$\boxed{[\Theta]A}$ Applying a context Θ, as a substitution, to a type A

$$[\cdot]A = A \qquad [\Theta, \hat{\alpha}]A = [\Theta]A$$
$$[\Theta, \alpha]A = [\Theta]A \qquad [\Theta, \hat{\alpha} = P]A = [\Theta]([P/\hat{\alpha}]A)$$

Fig. 10. Applying a context to a type

system would derive the term. Thankfully, the invertibility of $\leq^{\pm}$Dforallr ensures that we can transform declarative derivations concluding with $\leq^{\pm}$Dforalll into equivalent ones that, like the algorithmic systems, use $\leq^{\pm}$Dforallr as much as possible to eliminate all the prenex quantifiers on the right-hand side before using $\leq^{\pm}$Dforalll.

- $\leq^{\pm}$Aforallr is almost an exact copy of $\leq^{\pm}$Dforalll. The forall rules must preserve the groundness invariant, so while $\leq^{\pm}$Aforalll can introduce existential type variables when eliminating a quantifier on the left-hand side, $\leq^{\pm}$Aforallr can not introduce any existential variables while eliminating a quantifier on the right-hand side. This ensures that M remains ground from the conclusion of the rule to its premise.
- $\leq^{\pm}$Aarrow describes how to check function types. This rule is interesting since it has multiple premises. After checking the first premise $\Theta \vdash Q \leq^{+} P \dashv \Theta'$, we have an output context Θ' that might solve some existential variables that appear in P. Since some of these existential variables might also appear in N, we need to ensure that we propagate their solutions when we check $N \leq^{-} M$. We do this by substituting all the existential variables that appear in the possibly non-ground N by their solutions in Θ'. We denote this operation as $[\Theta']N$ and define it formally in Figure 10.
- $\leq^{\pm}$Ashift$\downarrow$ and the symmetric $\leq^{\pm}$Ashift$\uparrow$ are our shift rules. In these rules the groundness invariant defines the order in which the premises need to be be checked. For instance, in $\leq^{\pm}$Ashift$\downarrow$ only N is ground in the conclusion, therefore we need to check $M \leq^{-} N$ first. As we will see in Section 6.1.1, checking this gives us an algorithmic context Θ' that solves all the existential variables that appear in M. We can use this context to *complete* M by substituting all the existential variables in M by their solutions in Θ', giving us the ground type $[\Theta']M$. Now that we have a ground type $[\Theta']M$, we can check $N \leq^{-} M$ by verifying $\Theta' \vdash N \leq^{-} [\Theta']M \dashv \Theta''$.

4.2 Algorithmic type system

With a mechanism to solve existential variables in hand, we can now construct an algorithmic system to implement our declarative typing rules. We present our algorithmic type system in Figure 11.

- Most of the rules (Avar, Aλabs, Agen, Athunk, Areturn, Aspinenil, and Aspinecons) are identical to their declarative counterparts, modulo adding output contexts and applying output contexts to subsequent premises.
- We split the declarative Dspinetypeabs rule into two rules depending on whether the new universal variable α appears in the type we are quantifying over N. If it does not, as in Aspinetypeabsnotin, then we do not need to introduce a new existential variable. If it does, as in Aspinetypeabsin, then as with $\leq^{\pm}$Aforalll we introduce a fresh existential variable $\hat{\alpha}$ to replace α. Note that unlike most rules, Aspinetypeabsin adds a new existential variable to the output context of the conclusion *that does not appear in its input context*. This will prove to be important in the Aunambiguouslet and Aambiguouslet rules.

$\boxed{\Theta; \Gamma \vdash e : A \dashv \Theta'}$ The term e synthesizes the type A, producing the context Θ'

$\boxed{\Theta; \Gamma \vdash s : N \gg M \dashv \Theta'}$ When passed to a head of type N, the argument list s synthesizes the type M, producing the context Θ'

$$\frac{x : P \in \Gamma}{\Theta; \Gamma \vdash x : P \dashv \Theta} \text{ Avar}$$

$$\frac{\Theta; \Gamma, x : P \vdash t : N \dashv \Theta'}{\Theta; \Gamma \vdash \lambda x : P.\, t : P \to N \dashv \Theta'} \text{ A}\lambda\text{abs} \qquad \frac{\Theta, \alpha; \Gamma \vdash t : N \dashv \Theta', \alpha}{\Theta; \Gamma \vdash \Lambda\alpha.\, t : \forall \alpha.\, N \dashv \Theta'} \text{ Agen}$$

$$\frac{\Theta; \Gamma \vdash t : N \dashv \Theta'}{\Theta; \Gamma \vdash \{t\} : \downarrow N \dashv \Theta'} \text{ Athunk} \qquad \frac{\Theta; \Gamma \vdash v : P \dashv \Theta'}{\Theta; \Gamma \vdash \text{return } v : \uparrow P \dashv \Theta'} \text{ Areturn}$$

$$\frac{\begin{array}{c}\Theta; \Gamma \vdash v : \downarrow M \dashv \Theta' \qquad \Theta'; \Gamma \vdash s : M \gg \uparrow Q \dashv \Theta'' \qquad \Theta'' \vdash P \leq^{+} Q \dashv \Theta''' \\ \Theta''' \vdash [\Theta''']Q \leq^{+} P \dashv \Theta^{(4)} \qquad \Theta^{(5)} = \Theta^{(4)} \upharpoonright \Theta \qquad \Theta^{(5)}; \Gamma, x : P \vdash t : N \dashv \Theta^{(6)}\end{array}}{\Theta; \Gamma \vdash \text{let } x : P = v(s); t : N \dashv \Theta^{(6)}} \text{ Aambiguouslet}$$

$$\frac{\begin{array}{c}\Theta; \Gamma \vdash v : \downarrow M \dashv \Theta' \qquad \Theta'; \Gamma \vdash s : M \gg \uparrow Q \dashv \Theta'' \\ \text{FEV}(Q) = \emptyset \qquad \Theta''' = \Theta'' \upharpoonright \Theta \qquad \Theta'''; \Gamma, x : Q \vdash t : N \dashv \Theta^{(4)}\end{array}}{\Theta; \Gamma \vdash \text{let } x = v(s); t : N \dashv \Theta^{(4)}} \text{ Aunambiguouslet}$$

$$\frac{}{\Theta; \Gamma \vdash \epsilon : N \gg N \dashv \Theta} \text{ Aspinenil}$$

$$\frac{\Theta; \Gamma \vdash v : P \dashv \Theta' \qquad \Theta' \vdash P \leq^{+} [\Theta']Q \dashv \Theta'' \qquad \Theta''; \Gamma \vdash s : [\Theta'']N \gg M \dashv \Theta'''}{\Theta; \Gamma \vdash v, s : Q \to N \gg M \dashv \Theta'''} \text{ Aspinecons}$$

$$\frac{\Theta; \Gamma \vdash s : N \gg M \dashv \Theta' \qquad \alpha \notin \text{FUV}(N)}{\Theta; \Gamma \vdash s : (\forall \alpha.\, N) \gg M \dashv \Theta'} \text{ Aspinetypeabsnotin}$$

$$\frac{\Theta, \hat{\alpha}; \Gamma \vdash s : [\hat{\alpha}/\alpha]N \gg M \dashv \Theta', \hat{\alpha}\,[= P] \qquad \alpha \in \text{FUV}(N)}{\Theta; \Gamma \vdash s : (\forall \alpha.\, N) \gg M \dashv \Theta', \hat{\alpha}\,[= P]} \text{ Aspinetypeabsin}$$

Fig. 11. Algorithmic type system

- The first, second, and last premises of Aambiguouslet are the same as their declarative counterparts. The third and fourth just inline the algorithmic $\leq^{\pm}$Ashift$\uparrow$ rule corresponding to the declarative premise $\Theta \vdash \uparrow Q \leq^{-} \uparrow P$ in Dambiguouslet.
  We saw in the Aspinetypeabsin rule that performing type inference on spines can introduce new existential variables. To simplify our proofs, we do not want these to leak. Therefore we introduce a notion of *context restriction* in Figure 12. The fifth premise $\Theta^{(5)} = \Theta^{(4)} \upharpoonright \Theta$ creates a new context that restricts the context $\Theta^{(4)}$ to contain only the existential variables in Θ. So if $\Theta^{(4)}$ contains new existential variables compared to Θ, then these will not be

$\boxed{\Theta' \upharpoonright \Theta}$ Θ' restricted to only contain existential variables which appear in Θ

$$\frac{}{\cdot \upharpoonright \cdot = \cdot}\ \upharpoonright\text{empty} \qquad \frac{\Theta' \upharpoonright \Theta = \Theta''}{\Theta', \alpha \upharpoonright \Theta, \alpha = \Theta'', \alpha}\ \upharpoonright\text{uvar}$$

$$\frac{\Theta' \upharpoonright \Theta = \Theta''}{\Theta', \hat{\alpha}\ [= P] \upharpoonright \Theta, \hat{\alpha}\ [= Q] = \Theta'', \hat{\alpha}\ [= P]}\ \upharpoonright\text{guessin} \qquad \frac{\Theta' \upharpoonright \Theta = \Theta'' \qquad \hat{\alpha}\ [= Q] \notin \Theta}{\Theta', \hat{\alpha}\ [= P] \upharpoonright \Theta = \Theta''}\ \upharpoonright\text{guessnotin}$$

Fig. 12. Definition of context restriction

present in $\Theta^{(5)}$. However any new solutions in $\Theta^{(4)}$ to existential variables already in Θ will be present in $\Theta^{(5)}$, so the algorithm will not solve any existential variable as two incompatible solutions.

- The first, second, and last premises of Aunambiguouslet are lifted from the declarative rule Dunambiguouslet. We replace the quantification in the last premise of Dunambiguouslet with a statement that there are no existential variables left in the type Q, i.e. $\text{FEV}(Q) = \emptyset$. Since no instantiations are possible, there are not any other types left that Q could have. Like Aambiguouslet, we restrict the output context of the spine judgment to remove any existential variables newly introduced by the spine.

Our algorithm is really easy to implement — we wrote a bare-bones implementation in 250 lines of OCaml.

5 PROPERTIES OF DECLARATIVE TYPING

Implicit Polarized F should support the full complement of metatheoretic properties, but in this paper we focus on the properties needed to establish our theorems about type inference. For example, we will only prove substitution at the type level, and totally ignore term-level substitution.

5.1 Subtyping

Because we use subtyping to model type instantiation, we need to know quite a few properties about how subtyping works. For example, we are able to show that the subtyping relation admits both reflexivity and transitivity:

Lemma 5.1 (Declarative subtyping is reflexive). *If* $\Theta \vdash A\,\text{type}^{\pm}$ *then* $\Theta \vdash A \leq^{\pm} A$.

Lemma 5.2 (Declarative subtyping is transitive). *If* $\Theta \vdash A\,\text{type}^{\pm}$, $\Theta \vdash B\,\text{type}^{\pm}$, $\Theta \vdash C\,\text{type}^{\pm}$, $\Theta \vdash A \leq^{\pm} B$, *and* $\Theta \vdash B \leq^{\pm} C$, *then* $\Theta \vdash A \leq^{\pm} C$.

Because of impredicativity, transitivity is surprisingly subtle to get right. We discuss the needed metric in Section 6.2 as this metric is also used to show the decidability of algorithmic subtyping.

We also show that subtyping is stable under substitution, which is useful for proving properties of type instantiation:

Lemma 5.3 (Declarative subtyping is stable under substitution). *If* $\Theta_L, \Theta_R \vdash P\,\text{type}^{+}$, *then:*

- *If* $\Theta_L, \alpha, \Theta_R \vdash Q\,\text{type}^{+}$, $\Theta_L, \alpha, \Theta_R \vdash R\,\text{type}^{+}$, *and* $\Theta_L, \alpha, \Theta_R \vdash Q \leq^{+} R$, *then* $\Theta_L, \Theta_R \vdash [P/\alpha]Q \leq^{+} [P/\alpha]R$.
- *If* $\Theta_L, \alpha, \Theta_R \vdash N\,\text{type}^{-}$, $\Theta_L, \alpha, \Theta_R \vdash M\,\text{type}^{-}$, *and* $\Theta_L, \alpha, \Theta_R \vdash N \leq^{-} M$, *then* $\Theta_L, \Theta_R \vdash [P/\alpha]N \leq^{-} [P/\alpha]M$.

$\boxed{\Theta \vdash \Gamma_1 \cong \Gamma_2}$ In the context Θ, the environments Γ_1 and Γ_2 are isomorphic

$$\frac{}{\Theta \vdash \cdot \cong \cdot}\ \text{Eisoempty} \qquad \frac{\Theta \vdash \Gamma_1 \cong \Gamma_2 \qquad \Theta \vdash P \cong^{+} Q}{\Theta \vdash \Gamma_1, x : P \cong \Gamma_2, x : Q}\ \text{Eisovar}$$

Fig. 13. Isomorphic environments

$\boxed{\Theta\ \mathrm{ctx}}$ Θ is a well-formed context

$$\frac{}{\cdot\ \mathrm{ctx}}\ \text{Cwfempty} \qquad \frac{\Theta\ \mathrm{ctx}}{\Theta, \alpha\ \mathrm{ctx}}\ \text{Cwfuvar} \qquad \frac{\Theta\ \mathrm{ctx}}{\Theta, \hat{\alpha}\ \mathrm{ctx}}\ \text{Cwfunsolvedguess}$$

$$\frac{\Theta\ \mathrm{ctx} \qquad \Theta \vdash P\ \mathrm{type}^{+} \qquad P\ \mathrm{ground}}{\Theta, \hat{\alpha} = P\ \mathrm{ctx}}\ \text{Cwfsolvedguess}$$

Fig. 14. Well-formedness of contexts

5.2 Typing

We also show an important property about our typing judgment. Suppose we have two types A and B that are isomorphic, i.e. $\Theta \vdash A \cong^{\pm} B$. Then, by the intuitive definition of subtyping, it should be possible to safely use any term of type A where a term of type B is expected, and vice versa. Concretely, the map function can be given both the type $P = \downarrow(\forall\alpha, \beta. \downarrow(\alpha \to \beta) \to [\alpha] \to \uparrow[\beta])$ and also the type $Q = \downarrow(\forall\beta, \alpha. \downarrow(\alpha \to \beta) \to [\alpha] \to \uparrow[\beta])$, and there is no reason to prefer one to the other. Therefore if we have an environment Γ where map : P and an environment Γ' where map : Q, the inference we get within each environment should be the same.

We formalize this idea with the following lemma, which makes use of the isomorphic environment judgment we define in Figure 13:

Lemma 5.4 (Isomorphic environments type the same terms). *If* $\Theta \vdash \Gamma \cong \Gamma'$, *then:*

- *If* $\Theta; \Gamma \vdash v : P$ *then* $\exists P'$ *such that* $\Theta \vdash P \cong^{-} P'$ *and* $\Theta; \Gamma' \vdash v : P'$.
- *If* $\Theta; \Gamma \vdash t : N$ *then* $\exists N'$ *such that* $\Theta \vdash N \cong^{-} N'$ *and* $\Theta; \Gamma' \vdash t : N'$.
- *If* $\Theta; \Gamma \vdash s : N \gg M$ *and* $\Theta \vdash N \cong^{-} N'$, *then* $\exists M'$ *such that* $\Theta \vdash M \cong^{-} M'$ *and* $\Theta; \Gamma \vdash s : N' \gg M'$.

This lemma tells us that regardless of which type we give the variable map, any term which uses it will have the same type (up to isomorphism).

6 PROPERTIES OF ALGORITHMIC TYPING

6.1 Well-formedness

We first establish some of the invariants our type system maintains.

6.1.1 *Subtyping.*

Context well-formedness. A simple property that our algorithm maintains is that any solutions the algorithm chooses are well-formed: they are ground and only contain universal variables in

$\boxed{\Theta \vdash A\ \mathrm{type}^{\pm}}$ In the context Θ, A is a well-formed positive/negative type

$$\frac{\hat{\alpha} \in \mathrm{EV}(\Theta)}{\Theta \vdash \hat{\alpha}\ \mathrm{type}^{+}}\ \mathrm{Twfguess}$$

Fig. 15. Additional well-formedness rules for algorithmic types. $\mathrm{EV}(\Theta)$ contains all the existential type variables in Θ, independently of whether they are solved or unsolved.

$\boxed{\Theta \longrightarrow \Theta'}$ Θ extends to Θ'

$$\frac{}{\cdot \longrightarrow \cdot}\ \mathrm{Cempty} \qquad \frac{\Theta \longrightarrow \Theta'}{\Theta, \alpha \longrightarrow \Theta', \alpha}\ \mathrm{Cuvar} \qquad \frac{\Theta \longrightarrow \Theta'}{\Theta, \hat{\alpha} \longrightarrow \Theta', \hat{\alpha}}\ \mathrm{Cunsolvedguess}$$

$$\frac{\Theta \longrightarrow \Theta'}{\Theta, \hat{\alpha} \longrightarrow \Theta', \hat{\alpha} = P}\ \mathrm{Csolveguess} \qquad \frac{\Theta \longrightarrow \Theta' \qquad \|\Theta\| \vdash P \cong^{+} Q}{\Theta, \hat{\alpha} = P \longrightarrow \Theta', \hat{\alpha} = Q}\ \mathrm{Csolvedguess}$$

Fig. 16. Context extension

$\boxed{\|\Theta\|}$ The declarative context corresponding to the algorithmic context Θ

$$\|\cdot\| = \cdot \qquad \|\Theta, \alpha\| = \|\Theta\|, \alpha \qquad \|\Theta, \hat{\alpha}\| = \|\Theta\| \qquad \|\Theta, \hat{\alpha} = P\| = \|\Theta\|$$

Fig. 17. Producing a declarative context from an algorithmic context

scope at the time of the corresponding existential variable's creation. We formalize this notion of the well-formedness of contexts in Figures 14 and 15.

Output context solves all existential variables. Another property of our algorithm is that given well-formed inputs, the algorithm will solve all the necessary solutions: in other words, the output context contains solutions to all existential variables appearing in the non-ground type.

Context extension. Finally, we want to show that the algorithm produces an output context that only adds solutions to the input context. For instance, our algorithm should not change an existing solution to an incompatible one. We formalize this idea with a context extension judgment based on the earlier work of Dunfield and Krishnaswami [5]. This judgment $\Theta \longrightarrow \Theta'$, as defined in Figure 16, indicates information gain: Θ' must contain at least as much information as Θ.

Most of the rules defining context extension are homomorphic. The exceptions are Csolveguess, which allows the algorithm to solve an existential variable, and Csolvedguess, which allows the algorithm to change the solution P for an existential variable $\hat{\alpha}$ to an isomorphic type Q. This ability is not crucial to the algorithm; in fact the algorithm will maintain syntactically identical solutions. However since the declarative system can invent any solutions it likes when instantiating quantified types, we need this little bit of flex to prove completeness.

Solutions to existential types are ground, so we do not need to introduce a new notion of isomorphism over algorithmic types. However since the context Θ is algorithmic rather than declarative, we do need to introduce a simple operation $\|\Theta\|$ that converts an algorithmic context

$\boxed{\Theta \Longrightarrow \Theta'}$ The context Θ' weakly extends the context Θ'

$$\frac{}{\cdot \Longrightarrow \cdot}\ \text{Wcempty} \qquad \frac{\Theta \Longrightarrow \Theta'}{\Theta, \alpha \Longrightarrow \Theta', \alpha}\ \text{Wcuvar} \qquad \frac{\Theta \Longrightarrow \Theta'}{\Theta, \hat{\alpha} \Longrightarrow \Theta', \hat{\alpha}}\ \text{Wcunsolvedguess}$$

$$\frac{\Theta \Longrightarrow \Theta'}{\Theta, \hat{\alpha} \Longrightarrow \Theta', \hat{\alpha} = P}\ \text{Wcsolveguess} \qquad \frac{\Theta \Longrightarrow \Theta' \qquad \|\Theta\| \vdash P \cong^{+} Q}{\Theta, \hat{\alpha} = P \Longrightarrow \Theta', \hat{\alpha} = Q}\ \text{Wcsolvedguess}$$

$$\frac{\Theta \Longrightarrow \Theta'}{\Theta \Longrightarrow \Theta', \hat{\alpha}}\ \text{Wcnewunsolvedguess} \qquad \frac{\Theta \Longrightarrow \Theta'}{\Theta \Longrightarrow \Theta', \hat{\alpha} = P}\ \text{Wcnewsolvedguess}$$

Fig. 18. Weak context extension. We highlight the rules that are "new" compared with context extension.

$\boxed{\Theta \vdash \Gamma\ \text{env}}$ The environment Γ is well-formed with respect to the context Θ

$$\frac{}{\Theta \vdash \cdot\ \text{env}}\ \text{Ewfempty} \qquad \frac{\Theta \vdash \Gamma\ \text{env} \qquad \Theta \vdash P\ \text{type}^{+} \qquad P\ \text{ground}}{\Theta \vdash \Gamma, x : P\ \text{env}}\ \text{Ewfvar}$$

Fig. 19. Well-formedness of typing environments

into a declarative one by dropping all the context items involving existential variables. This operation is defined in Figure 17.

We formalize all these properties in the well-formedness statement about the algorithmic subtyping relation below.

Lemma 6.1 (Algorithmic subtyping is w.f.).

- *If* $\Theta \vdash P \leq^{+} Q \dashv \Theta'$, $\Theta\,\text{ctx}$, P *ground, and* $[\Theta]Q = Q$, *then* $\Theta'\,\text{ctx}$, $\Theta \longrightarrow \Theta'$, *and* $[\Theta']Q$ *ground.*
- *If* $\Theta \vdash N \leq^{-} M \dashv \Theta'$, $\Theta\,\text{ctx}$, M *ground, and* $[\Theta]N = N$, *then* $\Theta'\,\text{ctx}$, $\Theta \longrightarrow \Theta'$, *and* $[\Theta']N$ *ground.*

The premises of each case encode our expectations about the inputs to the subtyping algorithm: the input context should be well-formed, the type on the ground side should be ground, and the type on the non-ground side should not contain any existential variables that have already been solved. We prove that when given these inputs, the subtyping algorithm produces an output context that is well-formed, is compatible with the input context, and that solves all the existential variables that appear in the input types.

6.1.2 Typing. The well-formedness statement for typing is very similar to the one for subtyping. However stating well-formedness for the spine judgment is more complex because this judgment (specifically the Aspinetypeabsin rule) can introduce new existential variables that do not appear within its input context.

To tackle this, we introduce a new notion of context extension, *weak context extension*, in Figure 18. This is identical to normal context extension, except it has two additional rules to permit adding new existential variables: Wcnewunsolvedguess lets it add an unsolved variable,

$\boxed{|A|_{\text{NQ}}}$ The size of a type A, ignoring quantification

$$|\alpha|_{\text{NQ}} = 1 \qquad |{\downarrow}N|_{\text{NQ}} = |N|_{\text{NQ}} + 1 \qquad |P \to N|_{\text{NQ}} = |P|_{\text{NQ}} + |N|_{\text{NQ}} + 1$$
$$|\hat{\alpha}|_{\text{NQ}} = 1 \qquad |\forall \alpha.\, N|_{\text{NQ}} = |N|_{\text{NQ}} \qquad |{\uparrow}P|_{\text{NQ}} = |P|_{\text{NQ}} + 1$$

$\boxed{\text{NPQ}(A)}$ The number of prenex quantifiers in a type A

$$\text{NPQ}(\alpha) = 0 \qquad \text{NPQ}({\downarrow}N) = 0 \qquad \text{NPQ}(P \to N) = 0$$
$$\text{NPQ}(\hat{\alpha}) = 0 \qquad \text{NPQ}(\forall \alpha.\, N) = 1 + \text{NPQ}(N) \qquad \text{NPQ}({\uparrow}P) = 0$$

Fig. 20. Definitions used in the decidability metric for Implicit Polarized F

and Wcnewsolvedguess lets it add a solved variable. We also extend in Figure 19 our notion of the well-formedness of algorithmic contexts to typing environments in the obvious way, with Ewfvar paralleling Cwfsolvedguess.

With this weaker notion of context extension we can state well-formedness of typing as follows:

Lemma 6.2 (Algorithmic typing is w.f.). *Given a typing context* Θ *and typing environment* Γ *such that* $\Theta\,\text{ctx}$ *and* $\Theta \vdash \Gamma\,\text{env}$*:*

- *If* $\Theta; \Gamma \vdash v : P \dashv \Theta'$*, then* $\Theta'\,\text{ctx}$*,* $\Theta \longrightarrow \Theta'$*,* $\Theta' \vdash P\,\text{type}^+$*, and* P *ground.*
- *If* $\Theta; \Gamma \vdash t : N \dashv \Theta'$*, then* $\Theta'\,\text{ctx}$*,* $\Theta \longrightarrow \Theta'$*,* $\Theta' \vdash N\,\text{type}^-$*, and* N *ground.*
- *If* $\Theta; \Gamma \vdash s : N \gg M \dashv \Theta', \Theta \vdash N\,\text{type}^-$*, and* $[\Theta]N = N$*, then* $\Theta'\,\text{ctx}, \Theta \Longrightarrow \Theta', \Theta' \vdash M\,\text{type}^-$*,* $[\Theta']M = M$*, and* $\text{FEV}(M) \subseteq \text{FEV}(N) \cup (\text{FEV}(\Theta') \setminus \text{FEV}(\Theta))$.

In addition to the standard postconditions, we also prove in the spine judgment case that the free existential variables in the output type M either come from the input type N or were added to the context by Aspinetypeabsin while instantiating universal quantifiers.

6.2 Determinism and decidability

Since our system is syntax-directed, determinism of algorithmic typing follows from straightforward rule inductions on the typing and subtyping rules. However, decidability is more intricate.

6.2.1 Subtyping. Our goal in proving the decidability of subtyping is finding a metric that decreases from the conclusion to each premise of the algorithmic rules. However the obvious metrics do not work. For instance, a metric based on the size of the type will not work because types can get bigger when we instantiate type variables. Predicative systems might use the lexicographic ordering of the number of prenex quantifiers followed by the size of the type as a metric. But while instantiations will not increase the number of quantifiers in predicative systems, they can do in impredicative systems.

In order to find a metric that works, we go back to the declarative system. Our declarative subtyping relation has the property that (using a size metric $|\_|_{\text{NQ}}$ for terms that ignores quantifiers), a well-formed[4] $\Theta \vdash P \leq^+ Q$ judgment implies that $|Q|_{\text{NQ}} \leq |P|_{\text{NQ}}$, and similarly a well-formed $\Theta \vdash N \leq^- M$ judgment implies that $|N|_{\text{NQ}} \leq |M|_{\text{NQ}}$. The intuition behind this property is that subtyping reflects the specialization order, and when a polymorphic type is instantiated, occurrences of type variables α get replaced with instantiations P. As a result, the size of a subtype has to be smaller than its supertypes. In ordinary System F, this intuition is actually false due to the contravariance of the function type. However, in Implicit Polarized F, universal quantification is

[4]One for which the well-formedness assumptions in Lemma 6.1 (Algorithmic subtyping is w.f.) hold.

a *negative type* of the form $\forall\alpha.\, N$, and ranges over *positive types*. Since positive type variables α are occurring within a negative type N, each occurrence of α must be underneath a shift, which is invariant in our system. As a result, the intuitive property is true.

Now, note that our algorithm always solves all of the existential problems in a well-formed subtyping problem. In particular, given an algorithmic derivation $\Theta \vdash P \leq^{+} Q \dashv \Theta'$, we know that $[\Theta']Q$ will be ground. So we therefore expect $[\Theta']Q$ to be the same size as it would be in a declarative derivation. This gives us a hint about what to try for the metric.

Since the size of the ground side (e.g. P) bounds the size of the completed non-ground side (e.g. $[\Theta']Q$), we will incorporate the size of the ground side into our metric. This size decreases between the conclusions and the premises of all of the rules apart from $\leq^{\pm}$Aforalll and $\leq^{\pm}$Aforallr, where it stays the same. $\leq^{\pm}$Aforalll and $\leq^{\pm}$Aforallr both however pick off a prenex quantifier, so we can take the lexicographic ordering of this size followed by the total number of prenex quantifiers in the subtyping judgment. This is $(|P|_{\text{NQ}}, \text{NPQ}(P) + \text{NPQ}(Q))$ for positive judgments $\Theta \vdash P \leq^{+} Q \dashv \Theta'$ and $(|M|_{\text{NQ}}, \text{NPQ}(M) + \text{NPQ}(N))$ for negative judgments $\Theta \vdash N \leq^{-} M \dashv \Theta'$. We give formal definitions of $|\_|_{\text{NQ}}$ and $\text{NPQ}(\_)$ in Figure 20.

Note that this is somewhat backwards from a typical metric for predicative systems where we count quantifiers first, then the size.

We prove that this metric assigns a total ordering to derivations in the algorithmic subtyping system. In each rule with multiple hypotheses, we invoke the bounding property described above, which we formalize in the following lemma:

Lemma 6.3 (Completed non-ground size bounded by ground size).

- *If* $\Theta \vdash P \leq^{+} Q \dashv \Theta'$, $\Theta\,\text{ctx}$, *P ground, and* $[\Theta]Q = Q$, *then* $|[\Theta']Q|_{NQ} \leq |P|_{NQ}$.
- *If* $\Theta \vdash N \leq^{-} M \dashv \Theta'$, $\Theta\,\text{ctx}$, *M ground, and* $[\Theta]N = N$, *then* $|[\Theta']N|_{NQ} \leq |M|_{NQ}$.

As a result of our key tactic for deriving this metric coming from the declarative system, it turns out that this metric assigns total orderings to derivations both in the algorithmic and the declarative subtyping systems. This allows us to reuse it in both the proof of completeness and the proof of transitivity, where in both cases using the height of the derivation as an induction metric is too weak.

6.2.2 Typing. The decidability of algorithmic typing ends up being relatively straightforward. Almost every rule decreases in the standard structural notions of size $|\_|$ on terms and spines. The Dspinetypeabs rule however preserves the size of the spine involved, so taking this size alone is insufficient. Therefore when comparing two spine judgments, we take the lexicographic ordering of $(|s|, \text{NPQ}(N))$, where s is the spine and N is the input type. We prove that this metric assigns a total ordering to algorithmic typing derivations, thus demonstrating that our typing algorithm is decidable. Alongside our proof, we include an exact statement of this metric in the appendix.

Note that, as with subtyping, this metric assigns a total ordering not only to algorithmic typing derivations but also to declarative typing derivations. Therefore we reuse this metric in the proofs of soundness, completeness, and the behavior of isomorphic types. In each of these cases, using the height of the derivation as an induction metric is too weak.

7 SOUNDNESS AND COMPLETENESS

We have now set out declarative subtyping and typing systems for Implicit Polarized F as well as algorithms to implement them. In this section, we will demonstrate that the algorithms are sound and complete with respect to their declarative counterparts. Proofs of all of these results are in the appendix.

7.1 Subtyping

7.1.1 Soundness. Consider the algorithmic subtyping judgment $\Theta \vdash P \leq^+ Q \dashv \Theta'$. The non-ground input Q might contain some unsolved existential variables, which are solved in Θ'. For soundness, we want to say that no matter what solutions are ever made for these existential variables, there will be a corresponding declarative derivation. To state this, we introduce a notion of a *complete context*. These are algorithmic contexts which contain no unsolved existential variables:

$$\text{Complete contexts} \quad \Omega ::= \cdot \mid \Omega, \alpha \mid \Omega, \hat{\alpha} = P$$

We can now use the context extension judgment $\Theta' \longrightarrow \Omega$ to state that Ω is a complete context that contains solutions to all the unsolved existential variables in Θ'. Applying this complete context to Q gives us our hoped-for declarative subtyping judgment $\|\Theta\| \vdash P \leq^+ [\Omega]Q$. (The $\|\Theta\|$ judgment (defined in Figure 17) drops the existential variables to create a declarative context.)

This gives us the following statement of soundness, which we prove in the appendix:

Theorem 7.1 (Soundness of algorithmic subtyping). *Given a well-formed algorithmic context Θ and a well-formed complete context Ω:*

- *If $\Theta \vdash P \leq^+ Q \dashv \Theta'$, $\Theta' \longrightarrow \Omega$, P ground, $[\Theta]Q = Q$, $\Theta \vdash P\,\text{type}^+$, and $\Theta \vdash Q\,\text{type}^+$, then $\|\Theta\| \vdash P \leq^+ [\Omega]Q$.*
- *If $\Theta \vdash N \leq^- M \dashv \Theta'$, $\Theta' \longrightarrow \Omega$, M ground, $[\Theta]N = N$, $\Theta \vdash N\,\text{type}^-$, and $\Theta \vdash M\,\text{type}^-$, then $\|\Theta\| \vdash [\Omega]N \leq^- M$.*

The side conditions for soundness are similar to those of well-formedness (Lemma 6.1), except we also require the complete context and the types in the algorithmic judgment to be well-formed.

7.1.2 Completeness. Completeness is effectively the reverse of soundness. For each positive declarative judgment $\|\Theta\| \vdash P \leq^+ [\Omega]Q$ where $\Theta \longrightarrow \Omega$ we want to find a corresponding algorithmic derivation $\Theta \vdash P \leq^+ Q \dashv \Theta'$, and likewise for negative judgments. No matter what declarative solutions we had for a solved type, the algorithmic system infers compatible solutions.

Since context extension $\Theta \longrightarrow \Omega$ now appears in the premise, we need to strengthen our induction hypothesis to prove completeness for rules with multiple premises. To do this, we prove that the algorithm's output context Θ' is also compatible with Ω, i.e. $\Theta' \longrightarrow \Omega$. This gives us the following statement of completeness, in which the side conditions are identical to soundness.

Theorem 7.2 (Completeness of algorithmic subtyping). *If $\Theta\,\text{ctx}$, $\Theta \longrightarrow \Omega$, and $\Omega\,\text{ctx}$, then:*

- *If $\|\Theta\| \vdash P \leq^+ [\Omega]Q$, $\Theta \vdash P\,\text{type}^+$, $\Theta \vdash Q\,\text{type}^+$, P ground, and $[\Theta]Q = Q$, then $\exists\Theta'$ such that $\Theta \vdash P \leq^+ Q \dashv \Theta'$ and $\Theta' \longrightarrow \Omega$.*
- *If $\|\Theta\| \vdash [\Omega]N \leq^- M$, $\Theta \vdash M\,\text{type}^-$, $\Theta \vdash N\,\text{type}^-$, M ground, and $[\Theta]N = N$, then $\exists\Theta'$ such that $\Theta \vdash N \leq^- M \dashv \Theta'$ and $\Theta' \longrightarrow \Omega$.*

7.2 Typing

Due to the unusual implication within Dunambiguouslet, soundness and completeness of typing are mutually recursive in our system. In soundness we use completeness while proving this implication, and in completeness we use soundness while unpacking the implication. We justify each of these uses by only applying either soundness or completeness to a judgment involving a subterm.

7.2.1 Soundness. Soundness of typing is formulated in the same way as soundness of subtyping. We introducing a complete context Ω which the output context extends to. For the spine judgment we allow the completed output type $[\Omega]M$ to be isomorphic to the declarative output type M'.

Theorem 7.3 (Soundness of algorithmic typing). *If* Θ ctx, $\Theta \vdash \Gamma$ env, $\Theta' \longrightarrow \Omega$, *and* Ω ctx, *then:*

- *If* $\Theta; \Gamma \vdash v : P \dashv \Theta'$, *then* $\|\Theta\| ; \Gamma \vdash v : [\Omega]P$.
- *If* $\Theta; \Gamma \vdash t : N \dashv \Theta'$, *then* $\|\Theta\| ; \Gamma \vdash t : [\Omega]N$.
- *If* $\Theta; \Gamma \vdash s : N \gg M \dashv \Theta'$, $\Theta \vdash N\ \mathrm{type}^-$, *and* $[\Theta]N = N$, *then* $\exists M'$ *such that* $\|\Theta\| \vdash [\Omega]M \cong^- M'$ *and* $\|\Theta\| ; \Gamma \vdash s : [\Omega]N \gg M'$.

7.3 Completeness

The main challenge in stating completeness of typing is taking into account the fact that spine judgments can introduce new existential variables. Practically, the introduction of new existential variables means that the output context of the algorithmic spine judgment Θ' will not necessarily extend to the complete context Ω. To deal with this, we make use of weak context extension and instead require there to exist some new complete context Ω' containing the new existential variables. The original complete context Ω should then weakly extend to Ω', and the output context of the spine judgment Θ' should strongly extend to Ω'.

This gives us the following statement of the completeness of typing:

Theorem 7.4 (Completeness of algorithmic typing). *If* Θ ctx, $\Theta \vdash \Gamma$ env, $\Theta \longrightarrow \Omega$, *and* Ω ctx, *then:*

- *If* $\|\Theta\| ; \Gamma \vdash v : P$ *then* $\exists \Theta'$ *such that* $\Theta; \Gamma \vdash v : P \dashv \Theta'$ *and* $\Theta' \longrightarrow \Omega$.
- *If* $\|\Theta\| ; \Gamma \vdash t : N$ *then* $\exists \Theta'$ *such that* $\Theta; \Gamma \vdash t : N \dashv \Theta'$ *and* $\Theta' \longrightarrow \Omega$.
- *If* $\|\Theta\| ; \Gamma \vdash s : [\Omega]N \gg M$, $\Theta \vdash N\ \mathrm{type}^-$, *and* $[\Theta]N = N$, *then* $\exists \Theta', \Omega'$ *and* M' *such that* $\Theta; \Gamma \vdash s : N \gg M' \dashv \Theta'$, $\Omega \Longrightarrow \Omega'$, $\Theta' \longrightarrow \Omega'$, $\|\Theta\| \vdash [\Omega']M' \cong^- M$, $[\Theta']M' = M'$, *and* Ω' ctx.

8 RELATED WORK

There has been considerable research into working around the undecidability of type inference for System F. Broadly, it falls into three main categories: enriching the language of types to make inference possible, restricting the subtype relation, and using heuristics to knock off the easy cases.

Enriching the Type Language. The "gold standard" for System F type inference is the ML^F system of Botlan and Rémy [1]. They observe that type inference for System F is complicated by the lack of principal types. To create principal types, they extend the type language of System F with bounded type constraints, and show that this admits a type inference algorithm with a simple and powerful specification: only arguments to functions used at multiple parametric types need annotation.

Unfortunately, ML^F is notoriously difficult to implement (see Rémy and Yakobowski [15] for the state-of-the-art), and there have been multiple attempts to find simpler subsystems. FPH (Vytiniotis et al. [24]) attempted to simplify ML^F by limiting the user-visible language of types to the standard System F types and only using ML^F-style constrained types internally within the type checker. Another attempt to simplify ML^F was HML (Leijen [8]). Unlike FPH, HML exposes part of the ML^F machinery (flexible types) to the user, which changes the language of types but also types more programs. Both FPH and HML require annotations only for polymorphic arguments, however unfortunately, both approaches proved too intricate to adapt to GHC.

Restricting the subtype relation. The most widely used approach for System F type inference simply abandons impredicativity. This line of work was originated by Odersky and Läufer [11], who proposed restricting type instantiation in the subtype relation to monotypes. This made subtyping decidable, and forms the basis for type inference in Haskell [13]. Dunfield and Krishnaswami [5] give a simple variant of this algorithm based on bidirectional typechecking.

This style of inference omits impredicativity from the subtype relation altogether. *Boxy types* [23] combine predicative subtyping with a generalization of bidirectional typechecking to support impredicativity. Vytiniotis et al. introduce a new type system feature, *boxes*, which merge the synthesis and checking judgments from bidirectional typechecking into marks on types (i.e. the boxes) which indicate whether part of a type came from inference or annotation. (This is somewhat reminiscent of "colored local type inference" [12].) Unfortunately, boxy types lack a clear non-algorithmic specification of when type annotations are required.

HMF [7] introduced another approach to restricting the specialization relation. It restricted subtyping for the function type constructor — instead of being contravariant, function types were invariant. This meant that inference could be done with only a modest modification to the Damas-Milner algorithm. Our work retains function contravariance, and only imposes invariance at shifts.

Heuristic Approaches. Many of the type inference problems which arise in practice are actually easy to solve. Cardelli [2] invented one of the oldest such techniques while constructing a type inference algorithm for a language with F-bounded impredicative polymorphism. Rather than doing anything difficult, Cardelli's algorithm simply instantiated each quantifier with the first type constraint it ran into. Pierce and Turner [14] formalized Cardelli's approach, and noticed that it did not need unification to implement. To make their algorithm work, they needed to use a seemingly ad-hoc representation of types, with a single type constructor that was simultaneously an n-ary small and big lambda. Furthermore, while they proved that inference was sound, they did not offer a declarative characterization of inference.

Serrano et al. [19] recently revisited the idea of controlling inference with heuristics with their system GI. They restrict impredicative instantiation to *guarded instantiations*, which are (roughly speaking) the cases when the type variable being instantiated is underneath a type constructor. This restriction is automatically achieved in our setting via the presence of shifts, which suggests that this syntactic restriction actually arises for deeper type-theoretic reasons. In follow up work, Serrano et al. [18] further simplify their approach, making some dramatic simplifications to the type theory (e.g., giving up function type contravariance) in order to achieve a simpler implementation.

Conclusions. Many researchers have noticed that type inference algorithms benefit from being able to look at the entire argument list to a function. From the perspective of plain lambda calculus, this looks ad-hoc and non-compositional. For example, HMF was originally described in two variants, one using argument lists and one without, and only the weaker algorithm without argument lists has a correctness proof. However, from the perspective of polarized type theory, argument lists are entirely type-theoretically natural: they mark the change of a polarity boundary!

This explains why Pierce and Turner's use of "jumbo" function types which combine multiple quantifiers and arguments makes sense: the merged connectives are all negative, with no interposed shifts. Making shifts explicit means that small connectives can have the same effect, which makes it possible to give a clear specification for the system.

We have also seen that many algorithms omit function contravariance from the specialization order to support impredicative inference. Our work clarifies that contravariance *per se* is not problematic, but rather that the benefits for inference arise from controlling the crossing of polarity boundaries. This again permits a simpler and more regular specification of subtyping.

ACKNOWLEDGMENTS

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project *TypeFoundry*, funded under the European Union's Horizon 2020 Framework Programme (grant agreement ID: 101002277).

REFERENCES

[1] Didier Le Botlan and Didier Rémy. 2003. MLF Raising ML to the Power of System F. In *ICFP '03*. ACM Press, Uppsala, Sweden, 52–63.

[2] Luca Cardelli. 1993. *An Implementation of F<:*. Technical Report. Systems Research Center, Digital Equipment Corporation.

[3] I. Cervesato. 2003. A Linear Spine Calculus. *Journal of Logic and Computation* 13, 5 (Oct. 2003), 639–688. https://doi.org/10.1093/logcom/13.5.639

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

[5] Jana Dunfield and Neel Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming - ICFP '13*. ACM Press, Boston, Massachusetts, USA, 429. https://doi.org/10.1145/2500365.2500582

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

[7] Daan Leijen. 2008. *HMF: Simple Type Inference for First-Class Polymorphism*. Technical Report MSR-TR-2008-65. Microsoft Research. 15 pages.

[8] Daan Leijen. 2008. *Robust Type Inference for First-Class Polymorphism*. Technical Report MSR-TR-2008-55. Microsoft Research. 10 pages.

[9] Paul Blain Levy. 2006. Call-by-Push-Value: Decomposing Call-by-Value and Call-by-Name. *Higher-Order and Symbolic Computation* 19, 4 (Dec. 2006), 377–414. https://doi.org/10.1007/s10990-006-0480-6

[10] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (Dec. 1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[11] Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '96*. ACM Press, St. Petersburg Beach, Florida, United States, 54–67. https://doi.org/10.1145/237721.237729

[12] Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored Local Type Inference. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '01*. ACM Press, London, United Kingdom, 41–53. https://doi.org/10.1145/360204.360207

[13] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-Rank Types. *Journal of Functional Programming* 17, 01 (Jan. 2007), 1. https://doi.org/10.1017/S0956796806006034

[14] Benjamin Pierce and David Turner. 2000. Local Type Inference. *ACM Transactions on Programming Languages and Systems* 22, 1 (Jan. 2000), 1–44. https://doi.org/10.1145/345099.345100

[15] Didier Rémy and Boris Yakobowski. 2008. From ML to MLF: Graphic Type Constraints with Efficient Type Inference. In *ICFP*. ACM, Victoria, BC, Canada, 63–74.

[16] John C Reynolds. 1974. Towards a Theory of Type Structure. In *Programming Symposium*. Springer, Springer, Paris, France, 408–425.

[17] John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*. North-Holland/IFIP, Paris, France, 513–523.

[18] Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A Quick Look at Impredicativity. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 1–29. https://doi.org/10.1145/3408971

[19] Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded Impredicative Polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2018*. ACM Press, Philadelphia, PA, USA, 783–796. https://doi.org/10.1145/3192366.3192389

[20] Arnaud Spiwack. 2014. A Dissection of L. (2014).

[21] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '07)*. Association for Computing Machinery, New York, NY, USA, 53–66. https://doi.org/10.1145/1190315.1190324

[22] 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)*. IEEE Computer Society, USA, 74.

[23] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2006. Boxy Types: Inference for Higher-Rank Types and Impredicativity. In *ICFP '06*. ACM Press, Portland, Oregon, USA, 251–262.

[24] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: First-class Polymorphism for Haskell. In *ICFP '08*. ACM Press, Victoria, BC, Canada, 295–306.

[25] Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture - FPCA '89*. ACM Press, Imperial College, London, United Kingdom, 347–359. https://doi.org/10.1145/99370.99404