Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism

Jana Dunfield  Neelakantan R. Krishnaswami
Max Planck Institute for Software Systems
Kaiserslautern and Saarbrücken, Germany
jd169@queensu.ca nk480@cl.cam.ac.uk

Abstract
Bidirectional typechecking, in which terms either synthesize a type or are checked against a known type, has become popular for its scalability (unlike Damas-Milner type inference, bidirectional typing remains decidable even for very expressive type systems), its error reporting, and its relative ease of implementation. Following design principles from proof theory, bidirectional typing can be applied to many type constructs. The principles underlying a bidirectional approach to polymorphism, however, are less obvious. We give a declarative, bidirectional account of higher-rank polymorphism, grounded in proof theory; this calculus enjoys many properties such as \( \gamma \)-reduction and predictability of annotations. We give an algorithm for implementing the declarative system; our algorithm is remarkably simple and well-behaved, despite being both sound and complete.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—polymorphism

Keywords bidirectional typechecking, higher-rank polymorphism

1. Introduction
Bidirectional typechecking [Pierce and Turner 2000] has become one of the most popular techniques for implementing typecheckers in new languages. This technique has been used for dependent types [Coquand 1996; Abel et al. 2008; Lambe et al. 2012; subtyping (Pierce and Turner 2000)], intersection, union, indexed and refinement types [Kleivan 1998; Davies and Pfenning 2000; Dunfield and Pfenning 2004; termination checking (Abel 2004); higher-rank polymorphism (Peyton Jones et al. 2007; Dunfield 2009); refinement types for LF (Lovas 2010); contextual modal types (Pientka 2008); compiler intermediate representations (Chlipala et al. 2005); and object-oriented languages including C# (Bierman et al. 2007) and Scala (Odersky et al. 2001). As can be seen, it scales well to advanced type systems; moreover, it is easy to implement, and yields relatively high-quality error messages (Peyton Jones et al. 2007).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICFP ’13, September 25–27, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2326-0/13/09...$15.00.
This is the 2020 version, which corrects the name of the first author and updates URLs for the appendix and Dunfield (2009).

However, the theoretical foundation of bidirectional typechecking has lagged behind its application. As shown by Watkins et al. (2004), bidirectional typechecking can be understood in terms of the normalization of intuitionistic type theory, in which normal forms correspond to the checking mode of bidirectional typechecking, and neutral (or atomic) terms correspond to the synthesis mode. This enables a proof of the elegant property that type annotations are only necessary at reducible expressions, and that normal forms need no annotations at all. The benefit of the proof-theoretic view is that it gives a simple and easy-to-understand declarative account of where type annotations are necessary, without reference to the details of the typechecking algorithm.

While the proof-theoretic account of bidirectional typechecking has been scaled up as far as type refinements and intersection and union types (Pfenning 2008), as yet there has been no completely satisfactory account of how to extend the proof-theoretic approach to handle polymorphism. This is especially vexing, since the ability of bidirectional algorithms to gracefully accommodate polymorphism (even higher-rank polymorphism) has been one of their chief attractions.

In this paper, we extend the proof-theoretic account of bidirectional typechecking to full higher-rank polymorphism (i.e., predicative System F), and consequently show that bidirectional typechecking is not merely sound with respect to the declarative semantics, but also that it is complete. Better still, the algorithm we give for doing so is extraordinarily simple.

First, as a specification of type checking, we give a declarative bidirectional type system which guesses all quantifier instantiations. This calculus is a small but significant contribution of this paper, since it possesses desirable properties, such as the preservation of typability under \( \gamma \)-reduction, that are missing from the type assignment version of System F. Furthermore, we can use the bidirectional character of our declarative calculus to show a number of refactoring theorems, enabling us to precisely characterize what sorts of substitutions (and reverse substitutions) preserve typability, where type annotations are needed, and when programmers may safely delete type annotations.

Then, we give a bidirectional algorithm that always finds corresponding instantiations. As a consequence of completeness, we can show that our algorithm never needs explicit type applications, and that type annotations are only required for polymorphic, reducible expressions—which, in practice, means that only let-bindings of functions at polymorphic type need type annotations; no other expressions need annotations.

Our algorithm is both simple to understand and simple to implement. The key data structure is an ordered context containing all bindings, including type variables, term variables, and existential variables denoting partial type information. By maintaining order, we are able to easily manage scope information, which is particu-
Proofs of the main results, as well as state-
completeness proofs of the algorithmic system with respect to the
deductive system. Surprisingly, it turns out that finding the correct
complete (up to \(\beta\eta\)-reduction) continues to
in modeling type instantiation using

dualizing sequent calculi, it is natural to give terms in
the proof-theoretic foundation of bidirectional typechecking. In fo-
more, ordered contexts admit a notion of \textit{extension or informa-
tion increase}, which organizes and simplifies the soundness and
completeness proofs of the algorithmic system with respect to the
deductive system. As a result of completeness, programmers may safely “pay no
attention to the implementor behind the curtain”, and ignore
all the algorithmic details of unification and type inference: the
algorithm does exactly what the declarative specification says,
more and no less.

\textbf{Contributions.} We make the following contributions:

\begin{itemize}
\item We give a declarative, bidirectional account of higher-rank
polymorphism, grounded strongly in proof theory. This cal-
culus has important properties (such as \(\eta\)-reduction) that the
type assignment variant of System F lacks, yet is sound and
complete (up to \(\beta\eta\)-equivalence) with respect to System F.

\item We give a very simple algorithm for implementing the declar-
system. Our algorithm does not need any data structure
more sophisticated than a list, but can still solve all of the prob-
lems which arise in typechecking higher-rank polymorphism
without any need for search or backtracking.

\item We prove that our algorithm is both sound and complete
with respect to our declarative specification of typing. This proof is
cleanly structured around \textit{context extension}, a relational notion
of information increase, corresponding to the intuition that our
algorithm progressively resolves type constraints.

\item As a result of completeness, programmers may safely “pay no
attention to the implementor behind the curtain”, and ignore
all the algorithmic details of unification and type inference: the
algorithm does exactly what the declarative specification says,
no more and no less.
\end{itemize}

\textbf{Lemmas and proofs.} Proofs of the main results, as well as state-
ments of all lemmas (and their proofs), can be found in the ap-
endix, available at \url{www.cs.queensu.ca/~jana/papers/bidir/}

\section{Declarative Type System}

In order to show that our algorithm is sound and complete, we need
to give a declarative type system to serve as the specification for
our algorithm. Surprisingly, it turns out that finding the correct
declarative system to use as a specification is itself an interesting
problem!

Much work on type inference for higher-rank polymorphism
takes the type assignment variant of System F as a specification of type inference. Unfortunately, under these rules typing is not stable under \(\eta\)-reductions. For example, suppose \(f\) is a variable of type \(1 \rightarrow \forall \alpha \alpha\). Then the term \(\lambda x. (x \alpha f)\) can be ascribed the type \(1 \rightarrow 1\), since the polymorphic quantifier can be instantiated to \(1\) between the \(f\) and the \(x\). But the \(\eta\)-reduct of \(\alpha\) cannot be ascribed the type \(1 \rightarrow 1\), because the quantifier cannot be instantiated until after \(f\) has been applied. This is especially unfortunate in pure languages like Haskell, where the \(\eta\) law is a valid program equality.

Therefore, we do not use the type assignment version of System F as our declarative specification of type checking and inference.

\begin{figure}
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Terms} & \(\mathbf{e} \::=\ x \mid () \mid \lambda x. e \mid e \; e \mid (e : A)\) \\
\hline
\end{tabular}
\caption{Source expressions}
\end{figure}

\begin{figure}
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Types} & \(\mathbf{A, B, C} \::=\ 1 \mid \alpha \mid \forall \alpha \alpha \mid A \rightarrow B\) \\
\hline
\textbf{Monotypes} & \(\mathbf{\tau, \sigma} \::=\ 1 \mid \alpha \mid \tau \rightarrow \sigma\) \\
\hline
\textbf{Contexts} & \(\mathbf{\Psi} \::=\ \|\Psi, \alpha \mid \Psi, x : A\) \\
\hline
\end{tabular}
\caption{Syntax of declarative types and contexts}
\end{figure}

Instead, we give a declarative, bidirectional system as the specifi-
cation. Traditionally, bidirectional systems are given in terms of a
\textit{checking} judgment \(\Psi \vdash e \in A\), which takes a type \(A\) as input and
ensures that the term \(e\) \textit{checks against} that type, and a \textit{synthesis}
judgment \(\Psi \vdash e \Rightarrow A\), which takes a term \(e\) and \textit{produces} a type
\(A\). This two-judgment formulation is satisfactory for simple types,
but breaks down in the presence of polymorphism.

The essential problem is as follows: the standard bidirectional
rule for checking applications \(e_1 e_2\) in non-polymorphic systems
is to synthesize type \(A \rightarrow B\) for \(e_1\), and then check \(e_2\) against \(A\),
returning \(B\) as the type. With polymorphism, however, we may have an application \(e_1 e_2\) in which \(e_1\) synthesizes a type of polymorphic
\(\{\forall \alpha \alpha \rightarrow \alpha\}\). Furthermore, we do not know \textit{a priori} how
many quantifiers we need to instantiate.

To solve this problem, we turn to \textit{focalization} (Andreoli [1992]),
the proof-theoretic foundation of bidirectional typechecking. In fo-
\begin{figure}
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{\(\Psi \vdash A \leq B\)} & Under context \(\Psi\), type \(A\) is a subtype of \(B\) \\
\hline
\begin{align*}
\alpha &\in \Psi \\
\Psi &\vdash \alpha \leq \alpha \textbf{-Var} \\
\Psi &\vdash 1 \leq 1 \textbf{-Unit} \\
\Psi &\vdash B_1 \leq A_1 \\
\Psi &\vdash A_2 \leq B_2 \\
\Psi &\vdash A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2 \\
\Psi &\vdash \forall \alpha \alpha \leq \forall \beta B \\
\Psi &\beta \vdash A \leq B \\
\end{align*}
\end{tabular}
\caption{Well-formedness of types and subtyping in the declarative
system}
\end{figure}

\begin{tabular}{|c|c|}
\hline
\textbf{\(\Psi \vdash A\)} & Under context \(\Psi\), type \(A\) is well-formed \\
\hline
\begin{align*}
\alpha &\in \Psi \\
\Psi &\vdash \alpha \textbf{-DeclUvarWF} \\
\Psi &\vdash 1 \textbf{-DeclUnitWF} \\
\Psi &\vdash A \textbf{-DeclArrowWF} \\
\Psi &\vdash \forall \alpha \alpha \textbf{-DeclForallWF} \\
\end{align*}
\end{tabular}

\begin{verbatim}
2
\end{verbatim}
the fundamental algorithmic problem in extending bidirectional typechecking to polymorphism is precisely the problem of figuring out what the missing type applications are.

Preserving the η-rule for functions comes at a cost. The subtyping relation induced by instantiation is undesirable for impredicative polymorphism [Tiuryn and Urzyczyn 1996; Chrzȩszcz 1998]. Since we want a complete typechecking algorithm, we consequently restrict our system to predicative polymorphism, where polymorphic quantifiers can be instantiated only with monomorphic types. We discuss alternatives in Section 9.

2.1 Typing in Detail

**Language overview.** In Figure 1, we give the grammar for our language. We have a unit term (1), variables x, lambda-abstraction λx.e, application e1 e2, and type annotation (e : A). We write A, B, C for types (Figure 2): types are the unit type 1, type variables α, universal quantification ∀α.A, and functions A → B. Monotypes τ and σ are the same, less the universal quantifier. Contexts Ψ are lists of type variable declarations, and term variables x : A, with the expected well-formedness condition. (We give a single-context formulation mixing type and term hypotheses to simplify the presentation.)

**Checking, synthesis, and application.** Our type system has three main judgments, given in Figure 3. The checking judgment Ψ ⊢ e : A asserts that e checks against the type A in the context Ψ.

The synthesis judgment Ψ ⊢ e ⇒ A says that we can synthesize the type A for e in the context Ψ. Finally, an application judgment Ψ ⊢ A • e ⇒ C says that if a (possibly polymorphic) function of type A is applied to argument e, then the whole application synthesizes C for the whole application.

As is standard in the proof-theoretic presentations of bidirectional typechecking, each of the introduction forms in our calculus has a corresponding checking rule. The DeclI rule says that Ψ ⊢ e :: A asserts that e checks against the unit type 1. The Decl→I rule says that λα.e checks against the function type A → B if e checks against B with the additional hypothesis that x has type A. The DeclI rule says that e has type ∀α.A if e has type A in a context extended with a fresh α. Sums, products and recursive types can be added similarly (we leave them out for simplicity). Rule DeclSub mediates between synthesis and checking: it says that e can be checked against B, if e synthesizes A and A is a subtype of B (that is, A is at least as polymorphic as B).

As expected, we can infer a type for a variable (the DeclVar rule) just by looking it up in the context. Likewise, the DeclAnno rule says that we can synthesize a type A for a term with a type annotation (e : A) just by returning that type (after checking that the term does actually check against A).

Application is a little more complex: we have to eliminate universals until we reach an arrow type. To do this, we use an application judgment Ψ ⊢ A • e ⇒ C, which says that if we apply a term of type A to an argument e, we get something of type C. This judgment works by guessing instantiations of polymorphic quantifiers in rule DeclvApp. Once we have instantiated enough quantifiers to expose an arrow A → C, we check e against A and return C in rule Decl→App.

In the following example, where we are applying some function polymorphic in α, DeclvApp instantiates the outer quantifier (to the unit type 1); we elide the premise Ψ ⊢ 1, but leaves the inner quantifier over β alone.

\[
Ψ \vdash x \triangleleft (\forall \beta. \beta \rightarrow \beta) \\
Ψ \vdash (\forall \beta. \beta \rightarrow \beta) \rightarrow 1 \rightarrow 1 \bullet x \triangleright 1 \rightarrow 1 \\
Ψ \vdash (\forall x. (\forall \beta. \beta \rightarrow \beta) \rightarrow \alpha \rightarrow \alpha) \bullet x \triangleright 1 \rightarrow 1
\]

In the minimal proof-theoretic formulation of bidirectionality [Davies and Pfenning 2000; Dunfield and Pfenning 2003], introduction forms are checked and elimination forms synthesize, full stop. Even () cannot synthesize its type! Actual bidirectional typecheckers tend to take a more liberal view, adding synthesis rules for at least some introduction forms. To show that our system can accommodate these kinds of extensions, we add the DeclI and Decl→I rules, which synthesize a unit type for () and a monomorphic function type for lambda expressions λx.e. We examine other variations, including a purist bidirectional no-inference alternative, and a liberal Damas-Milner alternative, in Section 8.

**Instantiating types.** We express the fact that one type is a polymorphic generalization of another by means of the subtyping judgment given in Figure 4. One important aspect of the judgment is that types are compared relative to a context of free variables. This simplifies our rules, by letting us eliminate the awkward side conditions on sets of free variables that plague many presentations. Most of the subtyping judgment is typical: it proceeds structurally on types, with a contravariant twist for the arrow; all the real ac-

![Figure 4. Declarative typing]
tion is contained within the two subtyping rules for the universal quantifier.

The left rule, \( \subseteq \forall \), says that a type \( \forall \alpha. A \) is a subtype of \( B \), if some instance \( [\tau/\alpha]A \) is a subtype of \( B \). This is what makes these rules only a declarative specification: \( \subseteq \forall \) guesses the instantiation \( \tau \) “out of thin air”, and so the rules do not directly yield an algorithm.

The right rule \( \subseteq \forall R \) is a little more subtle. It says that \( A \) is a subtype of \( \forall \beta. B \) if we can show that \( A \) is a subtype of a \( B \) in a context extended with \( \beta \). There are two intuitions for this rule, one semantic, the other proof-theoretic. The semantic intuition is that since \( \forall \beta. B \) is a subtype of \( [\tau/\beta]B \) for any \( \tau \), we need \( A \) to be a subtype of \( [\tau/\beta]B \) for any \( \tau \). Then, if we can show that \( A \) is a subtype of \( B \), with a free variable \( \beta \), we can appeal to a substitution principle for subtyping to conclude that for all \( \tau \), type \( A \) is a subtype of \( [\tau/\beta]B \).

The proof-theoretic intuition is simpler. The rules \( \subseteq \forall L \) and \( \subseteq \forall R \) are just the left and right rules for universal quantification in the sequent calculus. Type inference is a form of theorem proving, and our subtype relation gives some of the inference rules a sensitivity, in turn, simplifies a number of proofs. In fact, the rules are practically syntax-directed: the only exception is when both types are quantifiers, and either \( \subseteq \forall L \) or \( \subseteq \forall R \) could be tried. Since \( \subseteq \forall R \) is invertible, however, in practice one can apply it eagerly.

**Let-generalization.** In many accounts of type inference, let-bindings are treated specially. For example, traditional Damas-Milner type inference only does polymorphic generalization at let-bindings. Instead, we have sought to avoid a special treatment of let-bindings. In logical terms, let-bindings internalize the cut rule, and so special treatment puts the cut-elimination property of the calculus at risk—that is, typability may not be preserved when a let-binding is substituted away. To make let-generalization safe, additional properties like the principal types property are needed, a property endangered by rich type system features like higher-rank polymorphism, refinement types (Dunfield 2007) and GADTs (Vytiniotis et al. 2010).

To emphasize this point, we have omitted let-binding from our formal development. But since cut is admissible—i.e., the substitution theorem holds—restoring let-bindings is easy, as long as they get no special treatment incompatible with substitution. For example, the standard bidirectional rule for let-bindings is suitable:

\[
\begin{align*}
\Psi \vdash e \Rightarrow A \quad \Psi, x : A \vdash e' \iff C
\end{align*}
\]

Note the absence of generalization in this rule.

### 2.2 Bidirectional Typing and Type Assignment System F

Since our declarative specification is (quite consciously) not the usual type-assignment presentation of System F, a natural question is to ask what the relationship is. Luckily, the two systems are quite closely related: we can show that if a term is well-typed in our type assignment system, it is always possible to add type annotations to make the term well-typed in the bidirectional system; conversely, if the bidirectional system types a term, then some \( \beta \eta \)-equal term is well-typed under the type assignment system.

We formalize these properties with the following theorems, taking \( [e] \) to be the erasure of all type annotations from a term. We give the rules for our type assignment System F in Figure 5.

<table>
<thead>
<tr>
<th>( \Psi \vdash e : A )</th>
<th>Under context ( \Psi ), ( e ) has type ( A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( {x : A} \in \Psi )</td>
<td>( \Psi \vdash x : A ) ( A \forall )</td>
</tr>
<tr>
<td>( \Psi \vdash \lambda x. e : A \rightarrow B )</td>
<td>( \Psi \vdash e_1 : A \rightarrow B ) ( \Psi \vdash e_2 : A \rightarrow B ) ( A \rightarrow \text{E} )</td>
</tr>
<tr>
<td>( \Psi, x : A \vdash e : \forall \alpha. A )</td>
<td>( \Psi \vdash e : \forall \alpha. A ) ( \Psi \vdash \tau ) ( \forall \alpha )</td>
</tr>
</tbody>
</table>

**Figure 5. Type assignment rules for predicative System F**

**Theorem 1** (Completeness of Bidirectional Typing). If \( \Psi \vdash e : A \) then there exists \( e' \) such that \( \Psi \vdash e' \Rightarrow A \) and \( \beta \eta \vdash e \).

**Theorem 2** (Soundness of Bidirectional Typing). If \( \Psi \vdash e \Rightarrow A \) then there exists \( e' \) such that \( \Psi \vdash e' \Rightarrow A \) and \( \beta \eta \vdash e \).

Note that in the soundness theorem, the equality is up to \( \beta \) and \( \eta \). We may need to \( \eta \)-expand bidirectionally-typed terms to make them typecheck under the type assignment system, and within the proof of soundness, we \( \beta \)-reduce identity coercions.

### 2.3 Robustness of Typing

Type annotations are an essential part of the bidirectional approach: they mediate between type checking and type synthesis. However, we want to relieve programmers from having to write redundant type annotations, and even more importantly, enable programmers to easily predict where type annotations are needed.

Since our declarative system is bidirectional, the basic property is that type annotations are required only at redexes. Additionally, these typing rules can infer (actually, guess) all monomorphic types, so the answer to the question of where annotations are needed is: only on bindings of polymorphic type. Where bidirectional typing really stands out is in its robustness under substitution. We can freely substitute and “unsubstitute” terms:

**Theorem 3** (Substitution). Assume \( \Psi \vdash e \Rightarrow A \).

- If \( \Psi, x : A \vdash e' \iff C \) then \( \Psi \vdash e/x/e' \iff C \).
- If \( \Psi, x : A \vdash e' \iff C \) then \( \Psi \vdash e/x/e' \iff C \).

**Theorem 4** (Inverse Substitution). Assume \( \Psi \vdash e \Rightarrow A \).

- If \( \Psi \vdash [e : A]/x/e' \iff C \) then \( \Psi, x : A \vdash e' \iff C \).
- If \( \Psi \vdash [e : A]/x/e' \iff C \) then \( \Psi, x : A \vdash e' \iff C \).

Substitution is stated in terms of synthesizing expressions, since any checking term can be turned into a synthesizing term by adding an annotation. Dually, inverse substitution allows extracting any checking term into a let-binding with a type annotation. However, doing so indiscriminately can lead to a term with many redundant annotations, and so we also characterize when annotations can safely be removed:

**Theorem 5** (Annotation Removal). We have that:

- If \( \Psi \vdash \left( \lambda x. e \right) \Rightarrow A \) then \( \Psi \vdash \lambda x. e \Rightarrow C \).
- If \( \Psi \vdash \left( \lambda x. e \right) \Rightarrow C \) then \( \Psi \vdash \lambda x. e \Rightarrow C \).
- If \( \Psi \vdash e_1 (e_2 : A) \Rightarrow C \) then \( \Psi \vdash e_1 (e_2 : A) \Rightarrow C \).
- If \( \Psi \vdash \left( \lambda x. e \right) \Rightarrow C \) then \( \Psi \vdash \lambda x. e \Rightarrow C \).

2 The number of annotations can be reduced still further; see Section 4 for how to infer the types of all terms typable under Damas-Milner.

3 The generalization of Theorem 3 to any synthesizing term—not just \( (e : A) \)—does not hold. For example, given \( e = \lambda y. y \) and \( e' = x \) and \( \Psi \vdash \lambda y. y \Rightarrow 1 \rightarrow 1 \) and \( \Psi \vdash \lambda y. y \leftarrow C_1 \rightarrow C_2 \), we cannot derive \( \Psi, x : 1 \rightarrow 1 \vdash x \leftarrow C_1 ightarrow C_2 \) unless \( C_1 \) and \( C_2 \) happen to be 1.

---

4
3. Algorithmic Type System

Our declarative bidirectional system is a fine specification of how typing should behave, but it enjoys guessing entirely too much: the typing rules `DecllApp` and `Decl→l⇒` could only be implemented with the help of an oracle. The declarative subtyping rule \( \subseteq \forall \) has the same problem.

The first step in building our algorithmic bidirectional system will be to modify the three oracle rules so that, instead of guessing a type, they defer the choice by creating an existential type variable, to be solved later. However, our existential variables are not exactly unification variables; they are organized into ordered algorithmic contexts (Section 3.3), and typing rules (Figure 11) discussed in Section 3.4. All of the rules manipulate the contexts in a way consistent with context extension, a metatheoretic notion described in Section 4; context extension is key in stating and proving decidability, soundness and completeness.

3.1 Algorithmic Contexts

A notion of (ordered) algorithmic context is central to our approach. Like declarative contexts \( \Psi \), algorithmic contexts \( \Gamma \) (see Figure 6) we also use the letters \( \Delta \) and \( \Theta \) contain declarations of universal type variables \( \alpha \) and term variable typings \( x : A \). Unlike declarative contexts, algorithmic contexts also contain declarations of existential type variables \( \tilde{\alpha} \), which are either unsolved (and we simply write \( \tilde{\alpha} \)) or solved to some monotype \( (\tilde{\alpha} = \tau) \). Finally, for scoping reasons that will become clear when we examine the rules, algorithmic contexts also include a marker \( \triangledown \).

Complete contexts \( \Omega \) are the same as contexts, except that they cannot have unsolved variables.

The well-formedness rules for contexts (Figure 7) do not only prohibit duplicate declarations, but also enforce order: if \( \Gamma = (\Gamma_1, x : A, \Gamma_R) \), the type \( A \) must be well-formed under \( \Gamma_1 \); it cannot refer to variables \( \alpha \) or \( \tilde{\alpha} \) in \( \Gamma_R \). Similarly, if \( \Gamma = (\Gamma_1, \tilde{\alpha} = \tau, \Gamma_R) \), the solution type \( \tau \) must be well-formed under \( \Gamma_1 \). Consequently, circularity is ruled out: \( (\tilde{\alpha} = \beta, \beta = \tilde{\alpha}) \) is not well-formed.

### Figure 6. Syntax of types, monotypes, and contexts in the algorithmic system

- If \( \Psi \vdash ((e_1, e_2) : A) \Rightarrow A \), then \( \Psi \vdash e_1 e_2 \Rightarrow B \) and \( \Psi \vdash B \Rightarrow A \).
- If \( \Psi \vdash ((e : B) : A) \Rightarrow A \), then \( \Psi \vdash \{e : B\} \Rightarrow B \) and \( \Psi \vdash B \Rightarrow A \).
- If \( \Psi \vdash (\lambda x . e) : \sigma \Rightarrow \tau \Rightarrow \sigma \Rightarrow \tau \) then \( \Psi \vdash \lambda x . e \Rightarrow \sigma \Rightarrow \tau \).

We can also show that the expected \( \eta \)-laws hold:

**Theorem 6** (Soundness of Eta). If \( \Psi \vdash \lambda x . e \iff A \) and \( x \not\in \operatorname{FV}(e) \), then \( \Psi \vdash e \iff A \).

### Figure 7. Well-formedness of types and contexts in the algorithmic system

- \( \Gamma \vdash A \) under context \( \Gamma \), type \( A \) is well-formed
- \( \Gamma[\alpha] \vdash \alpha \) \text{ UvarWF} \quad \Gamma \vdash \alpha \) \text{ UnitWF}
- \( \Gamma \vdash A \) \quad \Gamma \vdash B \) \text{ ArrowWF} \quad \Gamma, \alpha \vdash A \) \text{ ForallWF}
- \( \Gamma[\tilde{\alpha}] \vdash \tilde{\alpha} \) \text{ EvarWF} \quad \Gamma[\tilde{\alpha} = \tau] \vdash \tilde{\alpha} \) \text{ SolvedEvarWF}

### Figure 8. Applying a context, as a substitution, to a type

**Contexts as substitutions on types.** An algorithmic context can be viewed as a substitution for its solved existential variables. For example, \( \tilde{\alpha} = 1, \beta = \tilde{\alpha} \mapsto 1 \) can be applied as if it were the substitution \( 1/\tilde{\alpha}, (\tilde{\alpha} \mapsto 1)/\beta \) (applied right to left), or the simultaneous substitution \( 1/\tilde{\alpha}, (1\mapsto 1)/\beta \). We write \( [\Gamma]A \) for \( \Gamma \) applied as a substitution to type \( A \); this operation is defined in Figure 8.

**Complete contexts.** Complete contexts \( \Omega \) (Figure 6) have no unsolved variables. Therefore, applying such a context to a type \( A \) (provided it is well-formed: \( \Omega \vdash A \)) yields a type \( [\Omega]A \) with no existentials. Complete contexts are essential for stating and proving soundness and completeness, but are not explicitly distinguished in any of our rules.

**Hole notation.** Since we will manipulate contexts not only by appending declarations (as in the declarative system) but by inserting and replacing declarations in the middle, a notation for contexts with a hole is useful:

\[ \Gamma = \Gamma_0[\Theta] \] means \( \Gamma \) has the form \( (\Gamma_1, \Theta, \Gamma_R) \)

For example, if \( \Gamma = \Gamma_0[\beta] \) (\( \{\tilde{\alpha}, \beta, x : \beta\} \)), then \( \Gamma_0[\beta = \tilde{\alpha}] = (\{\tilde{\alpha}, \beta = \tilde{\alpha}, x : \beta\} \). Since this notation is concise, we use it even
in rules that do not replace declarations, such as the rules for type well-formedness in Figure 1.

Occasionally, we also need contexts with two ordered holes:

\[\Gamma = \Delta \subseteq \Theta \cap \Omega\] means \(\Gamma\) has the form \((\Gamma_1, \Theta_1, \Gamma_1, \Theta_2, \Gamma_2)\)

### Input and output contexts

Our declarative system used a subtyping judgment and three typing judgments: checking, synthesis, and application. Our algorithmic system includes similar judgment typing judgments and three typing judgments: checking, synthesis, and application. Our declarative system used a subtyping judgment and three typing judgments: checking, synthesis, and application.

Our declarative system used a subtyping judgment and three typing judgments: checking, synthesis, and application. Our algorithmic system includes similar judgment typing judgment and three typing judgments: checking, synthesis, and application.

Rule 4 and application. Our algorithmic system includes similar judgment typing judgment and three typing judgments: checking, synthesis, and application.

In our setting, it is safe to drop trailing existentials that are unsolved: \(\Theta\) is an input to the second premise. But this is an arbitrary choice; the Rule 4 and application. Our algorithmic system includes similar judgment typing judgment and three typing judgments: checking, synthesis, and application. Rule 4 and application. Our algorithmic system includes similar judgment typing judgment and three typing judgments: checking, synthesis, and application.

The last two rules are essential: they derive subtypings with an unsolved existential on one side, and an arbitrary type on the other. Rule 4 and application. Our algorithmic system includes similar judgment typing judgment and three typing judgments: checking, synthesis, and application. Rule 4 and application. Our algorithmic system includes similar judgment typing judgment and three typing judgments: checking, synthesis, and application.

The differences between the declarative and algorithmic systems, particularly manipulations of existential variables, are most prominent in the subtyping rules, so we discuss those first.

#### 3.2 Algorithmic Subtyping

The first four subtyping rules in Figure 1 do not directly manipulate the context, but do illustrate how contexts are propagated.

Rules \(<:\text{Var}\) and \(<:\text{Unit}\) are reflexive rules; neither involves existential variables, so the output context in the conclusion is the same as the input context \(\Gamma\). Rule \(<:\text{Exvar}\) concludes that any unsolved existential variable is a subtype of itself, but this gives no clue as to how to solve that existential, so the output context is similarly unchanged.

Rule \(<:\rightarrow\) is a bit more interesting: it has two premises, where the first premise has an output context \(\Theta\), which is used as the input context to the second (subtyping) premise; the second premise has output context \(\Delta\), which is the output of the conclusion. Note that in \(<:\rightarrow\)'s second premise, we do not simply check that \(A_2 <: B_2\), but apply the first premise's output \(\Theta\) to those types:

\[\Theta \vdash \Theta A_2 : \Theta B_2 \rightarrow \Delta\]

This maintains a general invariant: whenever we try to derive \(\Gamma \vdash A : B \rightarrow \Delta\), the types \(A\) and \(B\) are already fully applied under \(\Gamma\). That is, they contain no existential variables already solved in \(\Gamma\). In this case, the type \(\forall \alpha \cdot A\) has the form \(A_1 \rightarrow A_2\). It follows that \(\alpha\)’s solution must have the form \(\cdots \rightarrow \cdots\), so we “articulate” \(\alpha\), giving it the solution \(\alpha_1 \rightarrow \alpha_2\) where the \(\alpha_i\) are fresh existentials. We insert their declarations just before \(\alpha\)—they must be to the left of \(\alpha\) so they can be mentioned in its subtyping. However, they must be close enough to \(\alpha\) that they appear to the right of the marker \(\alpha\) introduced by \(\forall\). Note that the first premise \(\forall \alpha \cdot \alpha_1\) switches to the other instantiation judgment. Also, the second premise \(\Theta \vdash \alpha \cdot A_2 : \Theta A_2 \rightarrow \Delta\) applies \(\Theta\) to \(A_2\), to apply any solutions found in the first premise.

The other rules are somewhat subtle. Rule \(\text{InstLAr}\) and \(\alpha\) are direct analogues of the first three

\[\Gamma[\alpha]\beta \vdash \alpha : \alpha \beta \rightarrow \Gamma[\alpha]\beta = \alpha\]

where, as explained in Section 3.1, \(\Gamma[\alpha]\beta\) denotes a context where some unsolved existential variable \(\hat{\alpha}\) is declared to the left of \(\beta\). In this situation, we cannot use \(\text{InstLS}\) to set \(\alpha\) to \(\beta\) because \(\beta\) is not well-formed under the part of the context to the left of \(\alpha\). Instead, we set \(\beta\) to \(\hat{\alpha}\).

Rule \(\text{InstLAir}\) is the instantiation version of \(<:\forall\). Since our polymorphism is predicative, we can’t assign \(\forall \beta\) to \(\hat{\alpha}\), but we can decompose the quantifier in the same way that subtyping does.

The rules for the second judgment \(\Lambda \vdash \alpha\) are similar: \(\text{InstR}\) and \(\text{InstRAr}\) are direct analogues of the first three \(\vdash \Lambda \alpha\) rules, and \(\text{InstRAiR}\) is the instantiation version of \(<:\forall\).

### Example

The interplay between instantiation and quantifiers is delicate. For example, consider the problem of instantiating \(\hat{\beta}\) to a supertype of \(\forall \alpha \cdot \alpha\). In this case, the type \(\forall \alpha \cdot \alpha\) is so polymorphic that it places no constraints at all on \(\hat{\beta}\). Therefore, it seems we are at risk of being forced to make a necessarily incomplete choice—but the instantiation judgment’s ability to “change its mind” about which variable to instantiate saves the day:

instantiated to any well-formed type, such as 1. In a dependently typed setting, we would need to check that at least one solution exists.
**Figure 9. Algorithmic subtyping**

\[ \Gamma \vdash \alpha : \Delta \]

Under input context \( \Gamma \), instantiate \( \alpha \) such that \( \alpha : A \), with output context \( \Delta \)

\[ \Gamma \vdash \tau \]

\[ \Gamma, \alpha, \Gamma' \vdash \alpha : \tau \vdash \Gamma, \alpha = \tau, \Gamma' \]

\[ \Gamma[\hat{\alpha}, \hat{\alpha}, \hat{\alpha} = \hat{\alpha} \rightarrow \hat{\alpha}_2] \vdash A_1 \vdash \Theta \vdash \hat{\alpha}_2 : \Theta A_2 \vdash \Delta \]

\[ \Gamma[\hat{\alpha}] \vdash \hat{\alpha} : A_1 \rightarrow \hat{\alpha}_2 \vdash \Delta \]

**Figure 10. Instantiation**

\[ \Gamma \vdash e \Leftrightarrow A \vdash \Delta \]

Under input context \( \Gamma \), e checks against input type \( A \), with output context \( \Delta \)

\[ \Gamma \vdash e \Rightarrow A \vdash \Delta \]

Under input context \( \Gamma \), e synthesizes output type \( A \), with output context \( \Delta \)

\[ \Gamma \vdash A \bullet e \Rightarrow C \vdash \Delta \]

Under input context \( \Gamma \), applying a function of type \( A \) to \( e \) synthesizes type \( C \), with output context \( \Delta \)
Here, we introduce a new variable \( \hat{\alpha} \) to go under the universal quotient; then, instantiation applies InstRReach to set \( \hat{\alpha} \), not \( \hat{\beta} \). Hence, \( \hat{\beta} \) is, correctly, *not constrained* by this subtyping problem.

Thus, instantiation does not necessarily solve any existential variable. However, instantiation to any monotype \( \tau \) will solve an existential variable—that is, for input context \( \Gamma \) and output \( \Delta \), we have unsolved\((\Delta) < \) unsolved\((\Gamma)\). This will be critical for decidability of subtyping (Section 5.2).

**Another example.** In Figure 12 we show a derivation that uses quantifier instantiation (InstRAll), articulation (InstRArr) and “reaching” (InstLReach), as well as InstRSolve. In the output context \( \Delta = \Gamma[\beta_2, \beta_1 = \hat{\beta}, \hat{\alpha} = \beta_1 \rightarrow \beta_2] \) note that \( \hat{\alpha} \) is solved to \( \beta_1 \rightarrow \hat{\beta} \), and \( \beta_2 \) is solved to \( \hat{\beta}_1 \). Thus, \( \Delta[\hat{\alpha} = \beta_1 \rightarrow \beta_1] \) which is a monomorphic approximation of \( \forall \hat{\beta}. \hat{\beta} \rightarrow \beta \).

3.4 Algorithmic Typing

We now turn to the typing rules in Figure 11. Many of these rules follow the declarative rules, with extra context machinery. Rule Var uses an assumption \( x : \Lambda \) without generating any new information, so the output context in its conclusion \( \Gamma \vdash x \rightarrow \Lambda \rightarrow \Gamma \) is just the input context. Rule Sub’s first premise has an output context \( \Theta \) used as the input context to the second (subtyping) premise, which has output context \( \Delta \), the output of the conclusion. Rule Anne does not directly change the context, but the derivation of its premise might include the use of some rule that does, so we propagate the premise’s output context \( \Delta \) to the conclusion.

**Unit and \( \forall \).** In the second row of typing rules, 11 and 11\( \Rightarrow \) generate no new information and simply propagate the input context.

\( \forall \) is more interesting: Like the declarative rule DeclIV, it adds a universal type variable \( \alpha \) to the (input) context. The output context of the premise \( \Gamma \vdash \alpha \rightarrow \Lambda \rightarrow \Delta \rightarrow \Theta \) allows for some additional (existential) variables to appear after \( \alpha \), in a trailing context \( \Theta \). These existential variables could depend on \( \alpha \); since \( \alpha \) goes out of scope in the conclusion, we must drop them from the concluding output context, which is just \( \Delta \): the part of the premise’s output context that cannot depend on \( \alpha \).

The application-judgment rule \( \forall \)App serves a similar purpose to the subtyping rule \( \ll \), but does not place a marker before \( \hat{\alpha} \): the variable \( \alpha \) may appear in the output type \( C \), so \( \hat{\alpha} \) must survive in the output context \( \Delta \).

**Functions.** In the third row of typing rules, rule \( \rightarrow I \) follows the same scheme: the declarations \( \Theta \) following \( x : \Lambda \) are dropped in the conclusion’s output context.

Rule \( \rightarrow I \Rightarrow \) corresponds to Decl\( \rightarrow I \Rightarrow \), one of the guessing rules, so we create new existential variables \( \hat{\alpha} \) (for the function domain) and \( \hat{\beta} \) (for the codomain) and check the function body against \( \hat{\beta} \). As in \( \forall \)App, we do not place a marker before \( \hat{\alpha} \), because \( \hat{\alpha} \) and \( \hat{\beta} \) appear in the output type \( \left\langle \lambda \alpha . e \Rightarrow \hat{\alpha} \rightarrow \hat{\beta} \right\rangle \).

Rule \( \rightarrow E \) is the expected analogue of Decl\( \rightarrow E \); like other rules with two premises, it applies the intermediate context \( \Theta \).

On the last row of typing rules, \( \hat{\alpha} \)App derives \( \hat{\alpha} \) \( \cdot e \Rightarrow \hat{\alpha}_2 \) where \( \hat{\alpha} \) is unsolved in the input context. Here we have an application judgment, which is supposed to synthesize a type for an application \( e_1 e \) where \( e_1 \) has type \( \hat{\alpha} \). We know that \( e_1 \) should have function type; similarly to InstLArr/InstRArr, we introduce \( \hat{\alpha}_1 \) and \( \hat{\alpha}_2 \) and add \( \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2 \) to the context. (Rule \( \hat{\alpha} \)App is the only algorithmic typing rule that does not correspond to a declarative rule.)

Finally, rule \( \rightarrow \)App is analogous to Decl\( \rightarrow \)App.

4. Context Extension

We motivated the algorithmic rules by saying that they evolved input contexts to output contexts that were “more solved”. To state and prove the metatheoretic results of decidability, soundness and completeness (Sections 4 and 5), we introduce a context extension judgment \( \Gamma \rightarrow \Delta \). This judgment captures a notion of information increase from an input context \( \Gamma \) to an output context \( \Delta \), and relates algorithmic contexts \( \Gamma \) and \( \Delta \) to completely solved extensions \( \Omega \), which correspond—via the context application described in Section 4.1—to declarative contexts \( \Psi \).

The judgment \( \Gamma \rightarrow \Delta \) is read “\( \Gamma \) is extended by \( \Delta \)” (or \( \Delta \) extends \( \Gamma \)). Another reading is that \( \Delta \) carries at least as much information as \( \Gamma \). A third reading is that \( \Gamma \rightarrow \Delta \) means that \( \Gamma \) is entailed by \( \Delta \): all positive information derivable from \( \Gamma \) (say, that existential variable \( \hat{\alpha} \) is in scope) can also be derived from \( \Delta \) (which may have more information, say, that \( \hat{\alpha} \) is equal to a particular type). This reading is realized by several key lemmas; for instance, extension preserves well-formedness: if \( \Gamma \vdash A \) and \( \Gamma \rightarrow \Delta \), then \( \Delta \vdash A \).

The rules deriving the context extension judgment (Figure 13) say that the empty context extends the empty context (\( \Gamma \rightarrow \Delta \)); a term variable typing \( x : \Lambda \rightarrow \Delta \) if applying the extending context \( \Delta \) to \( \Lambda \) and \( \Lambda \rightarrow \Delta \) yields the same type (\( \rightarrow \)Unsolved); universal type variables must match (\( \rightarrow \)Add); scope markers must match (\( \rightarrow \)AddSolved); and, existential variables may:

- appear unsolved in both contexts (\( \rightarrow \)Unsolved),
- appear solved in both contexts, if applying the extending context \( \Delta \) makes the solutions \( \tau \rightarrow \) equal (\( \rightarrow \)Solved),
- get solved by the extending context (\( \rightarrow \)Solve),
- be solved by the extending context, either without a solution (\( \rightarrow \)Add) or with a solution (\( \rightarrow \)AddSolved).

Extension does not allow solutions to disappear: information must increase. It does allow solutions to change, but only if the change preserves or increases information. The extension

\( \left( \hat{\alpha}, \hat{\beta} = \hat{\alpha} \rightarrow \hat{\beta} = \hat{\alpha} \right) \rightarrow \left( \hat{\alpha} = 1, \hat{\beta} = \hat{\alpha} \right) \)

directly increases information about \( \hat{\alpha} \), and indirectly increases information about \( \hat{\beta} \). Perhaps more interestingly, the extension

\( \left( \hat{\alpha} = 1, \hat{\beta} = \hat{\alpha} \right) \rightarrow \left( \hat{\alpha} = 1, \hat{\beta} = 1 \right) \)

also holds: while the solution of \( \hat{\beta} \) in \( \Omega \) is different, in the sense that \( \Omega \) contains \( \hat{\beta} = 1 \) while \( \Delta \) contains \( \hat{\beta} = \hat{\alpha} \), applying \( \Omega \) to the two solutions gives the same thing: applying \( \Omega \) to \( \Delta \)’s solution of \( \hat{\beta} \) gives \( \left( \Omega \right)\hat{\beta} = \left( \Omega \right)1 = 1 \), while applying \( \Omega \) to \( \Delta \)’s own solution for \( \hat{\beta} \) also gives 1, because \( \left| \Omega \right| = 1 \).

Extension is quite rigid, however, in two senses. First, if a declaration appears in \( \Gamma \), it appears in all extensions of \( \Gamma \). Second, *extension preserves order*. For example, if \( \hat{\beta} \) is declared after \( \hat{\alpha} \) in \( \Gamma \), then \( \hat{\beta} \) will also be declared after \( \hat{\alpha} \) in every extension of \( \Gamma \). This holds for every variety of declaration. This rigidity aids in enforcing type variable scoping and dependencies, which are nontrivial in a setting with higher-rank polymorphism.

This combination of rigidity (in demanding that the order of declarations be preserved) with flexibility (in how existential type variables are expanded) manages to satisfy scoping and dependency relations and give enough room to maneuver in the algorithm and metatheory.

4.1 Context Application

A complete context \( \Omega \) (Figure 6) has no unsolved variables, so applying it to a (well-formed) type yields a type \( \left( \Omega \right)A \) with no existen-
As discussed in Section 3.3, deriving $\Gamma \vdash \alpha \triangleq \Delta$ does not necessarily instantiate any existential variable (unless $\Delta$ is a monotype). However, the instantiation rules do preserve the size of (substituted) types:

**Lemma (Instantiation Size Preservation).**
If $\Gamma = (\Gamma_0, \delta, \Gamma_1)$ and $\Gamma \vdash \alpha \triangleq \Delta$ or $\Delta \vdash A \triangleq \alpha$, and $\Gamma \vdash B$ and $\alpha \notin \text{FV}(\Gamma[B])$, then $|\Gamma[B]| = |\Delta|$, where $|C|$ is the plain size of $C$.

Using this lemma, we can show that the type $A$ in the instantiation judgment always gets smaller, even in rule InstLArr: the second premise applies the intermediate context $\Theta$ to $A_2$, but the lemma tells us that this application cannot make $A_2$ larger, and $A_2$ is smaller than the conclusion’s type $(A_1 \rightarrow A_2)$.

Now we can prove decidability of instantiation, assuming that $\alpha$ is unsolved in the input context $\Gamma$, that $A$ is well-formed under $\Gamma$, that $A$ is fully applied $(\lceil A = A \rceil)$, and that $\alpha$ does not occur in $A$. These conditions are guaranteed when instantiation is invoked, because the typing rule Sub applies the input substitution, and the subtyping rules apply the substitution where needed—in exactly one place: the second premise of $\ll$: $\rightarrow$. The proof is based on the (substituted) types in the premises being smaller than the (substituted) type in the conclusion.

**Theorem 7 (Decidability of Instantiation).**
If $\Gamma = \Gamma_0(\delta)$ and $\Gamma \vdash A$ such that $\lceil \Gamma[A] = A \rceil$ and $\alpha \notin \text{FV}(A)$, then:

1. Either there exists $\Delta$ such that $\Gamma_0(\delta) \vdash \alpha \triangleq \Delta$, and $\Gamma \vdash \Delta$, or not.
2. Either there exists $\Delta$ such that $\Gamma_0(\delta) \vdash A \triangleq \Delta$, and $\Gamma \vdash \Delta$, or not.

### 5.2 Decidability of Algorithmic Subtyping
To prove decidability of the subtyping system in Figure 9, measure judgments $\Gamma \vdash A \triangleq B$ in $\Delta$ lexicographically by

(S1) the number of $\forall$ quantifiers in $\Gamma$ and $B$;
(S2) the number of unsolved existentials in $\Gamma$;
(S3) the number of $\exists$ quantifiers in $\Gamma$.

Part (S3) uses contextual size, which penalizes solved variables (*):

**Definition (Contextual Size).**

\[
\begin{align*}
|\Gamma| & = 1, \\
|\Gamma(\delta)\vdash \delta| & = 1, \\
|\Gamma(\delta)\vdash \alpha| & = 1 + |\Gamma(\delta)\vdash \delta|, \\
|\Gamma\vdash \forall w.A| & = 1 + |\Gamma, \delta|, \\
|\Gamma\vdash A \rightarrow B| & = 1 + |\Gamma\vdash A| + |\Gamma\vdash B|.
\end{align*}
\]

For example, if $\Gamma = (\beta, \delta) = \beta$ then $|\Gamma\vdash \delta| = 1 + |\Gamma\vdash \beta| = 1 + 1 = 2$, whereas the plain size of $\delta$ is simply 1.
The connection between (S1) and (S2) may be clarified by examining rule $\vartriangleleft \rightarrow$, whose conclusion says that $A_1 \rightarrow A_2$ is a subtype of $B_1 \rightarrow B_2$. If $A_2$ or $B_2$ is polymorphic, then the first premise on $A_1 \rightarrow A_2$ is smaller by (S1). Otherwise, the first premise has the same input context as the conclusion, so it has the same (S2), but is smaller by (S3). If $B_1$ or $A_1$ is polymorphic, then the second premise is smaller by (S1). Otherwise, we use the property that instantiating a monotype always solves an existential:

**Lemma (Monotypes Solve Variables).** If $\Gamma \vdash \alpha \leq \tau \rightarrow \Delta$ or $\Gamma \vdash \tau \leq \Delta$, then if $|\Gamma|=\tau$ and $\Delta \notin \text{FV}(|\Gamma|\tau)$, we have $|\text{unsolved}(\Gamma)| = |\text{unsolved}(\Delta)| + 1$.

A couple of other lemmas are worth mentioning: subtyping on two monotypes cannot increase the number of unsolved existentials, and applying a substitution $\Gamma$ to a type does not increase the type's size with respect to $\Gamma$.

**Lemma (Monotype Monotonicity).** If $\Gamma \vdash \alpha; \beta \leq \Delta$ then $|\text{unsolved}(\Delta)| \leq |\text{unsolved}(\Gamma)|$.

**Lemma (Substitution Decreases Size).** If $\Gamma \vdash \alpha$ then $|\Gamma| + |\Gamma|\alpha| \leq |\Gamma| - |\alpha|$.

## Decidability of Algorithmic Typing

**Theorem 8 (Decidability of Subtyping).** Given a context $\Gamma$ and types $A$, $B$ such that $\Gamma \vdash A$ and $\Gamma \vdash B$ and $|\Gamma|A = A$ and $|\Gamma|B = B$, it is decidable whether there exists $\Delta$ such that $\Gamma \vdash A < B \rightarrow \Delta$.

### 3. Decidability of Algorithmic Typing

**Theorem 9 (Decidability of Typing).**

(i) Synthesis: Given a context $\Gamma$ and a term $e$, it is decidable whether there exists a type $A$ and a context $\Delta$ such that $\Gamma \vdash e : A \rightarrow \Delta$.

(ii) Checking: Given a context $\Gamma$, a term $e$, and a type $A$ such that $\Gamma \vdash B$, it is decidable whether there is a context $\Delta$ such that $\Gamma \vdash e : B \rightarrow \Delta$.

(iii) Application: Given a context $\Gamma$, a term $e$, and a type $A$ such that $\Gamma \vdash A$, it is decidable whether there exists a type $C$ and a context $\Delta$ such that $\Gamma \vdash A \cdot e : C \rightarrow \Delta$.

The following induction measure suffices to prove decidability:

$$\Rightarrow_{e}, \quad \left\langle e, \Gamma \vdash B \right\rangle, \quad \left\langle e, \Gamma \vdash A \right\rangle$$

where $\left\langle \ldots \right\rangle$ denotes lexicographic order, and where (when comparing two judgments typing the same term $e$) the synthesis judgment (top line) is considered smaller than the checking judgment (second line), which in turn is considered smaller than the application judgment (bottom line). That is, $\Rightarrow_{e} < \left\langle e, \Gamma \vdash B \right\rangle$ and $\left\langle e, \Gamma \vdash A \right\rangle$. In Sub, this makes the synthesis premise smaller than the checking conclusion; in $\rightarrow\text{App}$ and $\&\text{App}$, this makes the checking premise smaller than the application conclusion.

Since we have no explicit introduction form for polymorphism, the rule $\forall i$ has the same term $e$ in its premise and conclusion, and both the premise and conclusion are the same kind of judgment (checking). The rule $\forall \text{App}$ is similar (with application judgments in premise and conclusion). Therefore, given two judgments on the same term, and that are both checking judgments or both application judgments, we use the size of the input type expression—which does get smaller in $\forall i$ and $\forall \text{App}$.

## Soundness

We want the algorithmic specifications of subtyping and typing to be sound with respect to the declarative specifications. Roughly, given a derivation of an algorithmic judgment with input context $\Gamma$ and output context $\Delta$, and some complete context $\Omega$ that extends $\Delta$ (which therefore extends $\Gamma$), applying $\Omega$ throughout the given algorithmic judgment should yield a derivable declarative judgment.

Let's make that rough outline concrete for instantiation, showing that the action of the instantiation rules is consistent with declarative subtyping:

**Theorem 10 (Instantiation Soundness).**

- Given $\Delta \rightarrow \Omega$ and $|\Gamma|B = B$ and $\alpha \notin \text{FV}(\Omega)$:
  1. If $\Gamma \vdash \alpha \leq B \rightarrow \Delta$ then $|\Omega|A \vdash \alpha \leq |\Omega|B$.
  2. If $\Gamma \vdash B \leq \alpha \rightarrow \Delta$ then $|\Omega|A \vdash |\Omega|B \leq |\Omega|\alpha$.

Note that the declarative derivation is under $|\Omega|\Delta$, which is $\Omega$ applied to the algorithmic output context $\Delta$.

With instantiation soundness, we can prove the expected soundness property for subtyping:

**Theorem 11 (Soundness of Algorithmic Subtyping).**

If $\Gamma \vdash \alpha : B \rightarrow \Delta$ where $|\Gamma|A = A$ and $|\Gamma|B = B$ and $\Delta \rightarrow \Omega$ then $|\Omega|A \vdash |\Omega|A \leq |\Omega|B$.

Finally, knowing that subtyping is sound, we can prove that typing is sound:

**Theorem 12 (Soundness of Algorithmic Typing).** Given $\Delta \rightarrow \Omega$:

(i) If $\Gamma \vdash e : \alpha \rightarrow \Delta$ then $|\Omega|\Delta \vdash e \leq |\Omega|\alpha$.

(ii) If $\Gamma \vdash e : \alpha \rightarrow \Delta$ then $|\Omega|\Delta \vdash e \leq |\Omega|\alpha$.

(iii) If $\Gamma \vdash A \cdot e \rightarrow \Delta$ then $|\Omega|\Delta \vdash |\Omega|A \cdot e \leq |\Omega|C$.

The proofs need several lemmas, including this one:

**Lemma (Typing Extension).**

If $\Gamma \vdash e : \alpha \rightarrow \Delta$ or $\Gamma \vdash e : A \rightarrow \Delta$ or $\Gamma \vdash A \cdot e \rightarrow \Delta$ then $|\Omega|\Delta \rightarrow \Omega \cdot \Delta$.

## Completeness

Completeness of the algorithmic system is something like soundness in reverse: given a declarative derivation of $|\Omega|\Gamma \vdash |\Omega|\cdot \Delta \cdot \ldots$, we want to get an algorithmic derivation of $\Gamma \rightarrow \cdot \Delta \cdot \ldots$.

For soundness, the output context $\Delta$ such that $\Delta \rightarrow \Omega$ was given; $\Delta \rightarrow \Omega$ followed from Typing Extension (the above lemma) and transitivity of extension. For completeness, only $\Gamma$ is given, so we have $\Gamma \rightarrow \Omega$ in the antecedent. Then we might expect to show, along with $\Gamma \rightarrow \cdot \Delta \cdot \ldots$, that $\Delta \rightarrow \Omega$. But this is not general enough: the algorithmic rules generate fresh existential variables, so $\Delta$ may have existentials that are not found in $\Gamma$, nor in $\Omega$. In completeness, we are given a declarative derivation, which contains no existentials; the completeness proof must build up the completing context $\Omega$ along with the algorithmic derivation. Thus, completeness will produce an $\Omega'$ which extends both the given $\Omega$ and the output context of the algorithmic derivation: $\Omega \rightarrow \Omega'$ and $\Delta \rightarrow \Omega'$ (By transitivity, we also get $\Gamma \rightarrow \Omega'$).

As with soundness, we have three main completeness results, for instantiation, subtyping and typing:

**Theorem 13 (Instantiation Completeness).** Given $\Gamma \rightarrow \Omega$ and $A = [\Gamma]A$ and $\& \notin \text{unsolved}(\Gamma)$ and $\& \notin \text{FV}(\Omega)$:

(i) If $|\Omega|\Gamma \vdash [\Omega]A \leq [\Omega]B$ then there are $\Delta$, $\Omega'$ such that $\Omega \rightarrow \Omega'$ and $\Delta \rightarrow \Omega'$.

(ii) If $|\Omega|\Gamma \vdash [\Omega]A \leq [\Omega]B$ then there exist $\Delta$ and $\Omega'$ such that $\Delta \rightarrow \Omega'$ and $\Omega \rightarrow \Omega'$ and $\Gamma \vdash [\Gamma]A \leq [\Gamma]B$.

**Theorem 14 (Generalized Completeness of Subtyping).**

If $\Gamma \vdash \Omega$ and $\Gamma \vdash A$ and $\Gamma \vdash B$ and $|\Gamma|A \leq |\Gamma|B$ then there exist $\Delta$ and $\Omega'$ such that $\Delta \rightarrow \Omega'$.
Theorem 15 (Completeness of Algorithmic Typing). Given $\Gamma \vdash \Omega$ and $\Gamma \vdash A$,

(i) If $[\Omega]\Gamma \vdash e \leftrightarrow [\Omega]A$
then there exist $\Delta$ and $\Omega'$
such that $\Delta \rightarrow \Omega'$ and $\Gamma \vdash e \leftrightarrow [\Gamma]A \rightarrow \Delta$.

(ii) If $[\Omega]\Gamma \vdash e \Rightarrow A$
then there exist $\Delta$, $\Omega'$, and $A'$
such that $\Delta \rightarrow \Omega'$ and $\Gamma \vdash e \Rightarrow A' \rightarrow \Delta$
and $A = (\Omega')A'$.

(iii) If $[\Omega]\Gamma \vdash [\Omega]A \bullet e \Rightarrow C$
then there exist $\Delta$, $\Omega'$, and $C'$
such that $\Delta \rightarrow \Omega'$ and $\Gamma \vdash [\Gamma]A \bullet e \Rightarrow C' \rightarrow \Delta$
and $C = ([\Omega']C')$.

8. Design Variations

The rules we give infer monomorphic types, but require annotations
for all polymorphic bindings. In this section, we consider alternatives
to this choice.

Eliminating type inference. To eliminate type inference from the
declarative system, it suffices to drop the Decl→I⇒ and Decl1⇒ rules.
The corresponding alterations to the algorithmic system are a little more delicate: simply deleting the →I⇒ and 1I⇒ rules breaks completeness.
To see why, suppose that we have a variable $\alpha$ of type $\forall x. \alpha \rightarrow \alpha$, and consider the application $f \langle \alpha \rangle$. Our algorithm will introduce a new existential variable $\hat{\alpha}$ for $\alpha$, and then check $\hat{\alpha}$ against $\alpha$. Without the 1I⇒ rule, typechecking will fail. To restore completeness, we need to modify these two rules. Instead of being synthesis rules, we will change them to checking rules that check values against an unknown existential variable.

$\Gamma[I]\vdash \hat{\alpha} \Leftrightarrow \hat{\alpha} \vdash I[\alpha] = I\hat{\alpha}$

$\Gamma[\hat{\alpha}_{2}, \hat{\alpha}_{1}, \hat{\alpha} = \hat{\alpha}_{1} \rightarrow \hat{\alpha}_{2}, x : \hat{\alpha}_{1} \vdash e \leftrightarrow \hat{\alpha}_{2} \rightarrow \Delta, x : \hat{\alpha}_{1}, \Delta' \vdash \Gamma[I] \vdash \lambda x.e \leftrightarrow \forall \hat{\alpha}, [\hat{\alpha}/\hat{\alpha}]e \rightarrow \Delta$.

With these two rules replacing 1I⇒ and →I⇒, we have a complete algorithm for the no-inference bidirectional system.

Full Damas-Milner type inference. Another alternative is to increase the amount of type inference done. For instance, a natural question is whether we can extend the bidirectional approach to subsume the inference done by the algorithm of Damas and Milner (1982). This appears feasible: we can alter the →I⇒ rule to support ML-style type inference.

In this rule, we introduce a marker $\bullet$ into the context, and then check the function body against the type $\hat{\beta}$. Then, our output type substitutes away all the solved existential variables to the right of the marker $\bullet$, and generalizes over all of the unsolved variables to the right of the marker. Using an ordered context gives precise control over the scope of the existential variables, making it easy to express polymorphic generalization.

The above is only a sketch; we have not defined the corresponding declarative system, nor proved completeness.

9. Related Work and Discussion

9.1 Type Inference for System F

Because type inference for System F is undecidable (Wells 1999),
designing type inference algorithms for first-class polymorphism inherently involves navigating a variety of design tradeoffs. As a result, there have been a wide variety of proposals for extending type systems beyond the Damas-Milner "sweet spot". The main tradeoff appears to be a "two-out-of-three" choice: language designers can keep any two of: (1) the $\eta$-law for functions, (2) impredicative instantiation, and (3) the standard type language of System F.

As discussed in Section 2 for typability under $\eta$-reductions, it is necessary for subtyping to instantiate deeply: that is, we must allow instantiation of quantifiers to the right of an arrow. However, Tiuryn and Urzyczyn (1996) and Chrzaszcz (1998) showed that the subtyping relation for impredicative System F is undecidable.

As a result, if we want $\eta$ and a complete algorithm, then either the polymorphic instantiations must be predicative, or a different language of types must be used.

Figure 15 summarizes the different choices made by the designers of this and related systems.

Impredicativity and the $\eta$-law. The designers of $\mathcal{ML}^F$ (Le Botlan and Réméy 2003) (Rémy and Yakobowska 2008) (Le Botlan and Réméy 2009) chose to use a different language of types, one with a form of bounded quantification. This increases the expressivity of types enough to ensure principal types, which means that (1) required annotations are few and predictable, and (2) their system is very robust in the face of program transformations, including $\eta$. However, the richness of the $\mathcal{ML}^F$ type structure requires a sophisticated metatheory and correspondingly intricate implementation techniques.

Impredicativity and System F types. Much of the other work on higher-rank polymorphism avoids changing the language of types.

The HML system of Leijen (2009) and the FPH system of Vyntinis et al. (2008) both retain the type language of (impredicative) System F. Each of these systems gives as a specification a slightly different extension to the declarative Damas-Milner type system, and handle the issue of inference in slightly different ways. HML is essentially a restriction of $\mathcal{ML}^F$, in which the external language of types is limited to System F, but which uses the technology of $\mathcal{ML}^F$ internally, as part of type inference. FPH, on the other hand, extends and generalizes work on boxy types (Vyntinis et al. 2006) to control type inference. The differences in expressive power between these two systems are subtle—roughly speaking, FPH requires slightly more annotations, but has a less complicated specification. However, in both systems, the same heuristic guidance to the programmer applies: place explicit annotations on binders with fancy types.

The $\eta$-law and System F types. Peyton Jones et al. (2007) developed an approach for typechecking higher-rank predicative polymorphism that is closely related to ours. They define a bidirectional declarative system similar to our own, but which lacks an application judgment. This complicates the presentation of their system, forcing them to introduce an additional grammatical category of types beyond monotypes and polytypes, and requires many rules to carry an additional subtyping premise. Next, they enrich the subtyping rules of Odersky and Läufer (1996) with the distributivity axiom of Mitchell (1988), which we rejected on ideological grounds: it is a valid coercion, but is not orthogonal (it is a single rule mixing two different type connectives) and does not correspond to a rule in the sequent calculus. They do not prove the soundness and completeness of their Haskell reference implementation, but it appears to implement behavior close to our application judgment.

History of our approach. Several of the ideas used in the present paper descend from Dunfield (2009), an approach to first-class polymorphism (including impredicativity) also based on ordered contexts with existential variables instantiated via subtyping. In
fact, the present work began as an attempt to extend [Dunfield (2009)] with type-level computation. During that attempt, we found several shortcomings and problems. The most serious is that the decidability and completeness arguments were not valid. These problems may be fixable, but instead we started over, reusing several of the high-level ideas in different technical forms.

9.2 Other Type Systems

Pierce and Turner (2000) developed bidirectional typechecking for rich subtyping, with specific techniques for instantiating polymorphism within function application (hence, local type inference). Their declarative specification is more complex than ours, and their algorithm depends on computing approximations of upper and lower bounds on types. Colored local type inference (Odersky et al. (2001)) allows different parts of type expressions to be propagated in different directions. Our approach gets a similar effect by manipulating type expressions with existential variables.

9.3 Our Algorithm

One of our main contributions is our new algorithm for type inference, which is remarkable in its simplicity. Three key ideas underpin our algorithm.

Ordered contexts. We move away from the traditional “bag of constraints” model of type inference, and instead embed existential variables and their values directly into an ordered context. Thus, straightforward scoping rules control the free variables of the types each existential variable may be instantiated with, without any need for model-theoretic techniques like skolemization, which fit awkwardly into a type-theoretic discipline. Using an ordered context permits handling quantifiers in a manner resembling the level-based generalization mechanism of Rémy (1992), used also in ML $^\ell$ (Le Botlan and Rémy 2009).

The instantiation judgment. The original inspiration for instantiation comes from the “greedy” algorithm of Cardelli (1993), which eagerly uses type information to solve existential constraints. In that setting—a language with rather ambitious subtyping—the greedy algorithm was incomplete: consider a function of type $\forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha$ applied to a $\text{Cat}$ and an $\text{Animal}$; the cat will be checked against an existential $\forall \alpha$, which instantiates $\forall \alpha$ to $\text{Cat}$, but checking the second argument, $\text{Animal} \llimap \text{Cat}$, fails. (Reversing the order of arguments makes typing succeed!)

In our setting, where subtyping represents the specialization order induced by quantifier instantiation, it is possible to get a complete algorithm, by slightly relaxing the pure greedy strategy. Rather than eagerly setting constraints, we first look under quantifiers (in the InstLAll, InstRAll, and InstRRall rules) to see if there is a feasible monotype instantiation, and we also use the the InstLReach and InstRRReach to set the “wrong” existential variable in case we need to equate an existential variable with one to its right in the context. Looking under quantifiers seems forced by our restriction to predicative polymorphism, and “reaching” seems forced by our use of an ordered context, but the combination of these mechanisms fortuitously enables our algorithm to find good upper and lower monomorphic approximations of polymorphic types.

This is surprising, since it is quite contrary to the implementation strategy of ML $^\ell$ (also used by HML and FPH). There, the language of type constraints supports bounds on fully quantified types, and the algorithm incrementally refines these constraints. In contrast, we only ever create equational constraints on existentials (bounds are not needed), and once we have a solution for an existential, our algorithm never needs to revisit its decision.

Distinguishing instantiation as a separate judgment is new in this paper, and beneficial; Dunfield (2009) baked instantiation into the subtyping rules, resulting in a system whose implementation required substantial backtracking—over a set of rules including arbitrary application of substitutions. We, instead, maintain an invariant in subtyping and instantiation that the types are always fully applied with respect to an input context, obviating the need for explicit rules to apply substitutions.

Context extension. Finally, we introduce a context-extension judgment as the central invariant in our correctness proofs. This permits us to state many properties important to our algorithm abstractly, without reference to the details of our algorithm.

We are not the only ones to study context-based approaches to type inference. Recently, Gundry et al. (2010) recast the classic Damas-Milner algorithm, which manipulates unstructured sets of equality constraints, as structured constraint solving under ordered contexts. A (semantic) notion of information increase is central to their development, as (syntactic) context extension is to ours. While their formulation supports only ML-style prenex polymorphism, the ultimate goal is a foundation for type inference for dependent types. To some extent, both our algorithm and theirs can be understood in terms of the proof system of Miller (1992) for mixed-prefix unification. We each restrict the unification problem, and then give a proof search algorithm to solve the type inference problem.

Acknowledgments

Thanks to the anonymous ICFP reviewers for their comments, which have (we hope) led to a more correct paper.

References


Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. Logical Methods in Computer Science, 8(1), 2012.


