Greedy Bidirectional Polymorphism

Jana Dunfield
McGill University, Montréal, Canada

Abstract

Bidirectional typechecking has become popular in advanced type systems because it works in many situations where inference is undecidable. In this paper, I show how to cleanly handle parametric polymorphism in a bidirectional setting. The key contribution is a bidirectional type system for a subset of ML that supports first-class (higher-rank and even impredicative) polymorphism, and is complete for predicative polymorphism (including ML-style polymorphism and higher-rank polymorphism). The system’s power comes from bidirectionality combined with a “greedy” method of finding polymorphic instances inspired by Cardelli’s early work on System F\(_\kappa\). This work demonstrates that bidirectionality is a good foundation for traditionally vexing features like first-class polymorphism.

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

General Terms languages

1. Introduction

To check programs in advanced type systems, it is often useful to split the traditional typing judgment \( e : \tau \) into two forms, \( e \uparrow \tau \) and \( e \Downarrow \tau \). A read “\( e \) synthesizes type \( \tau \)” and \( e \Downarrow \tau \) a read “\( e \) checks against type \( \tau \)”. This technique has been used for dependent types (Coquand 1996; Norell 2007; Abel et al. 2008; Löh et al. 2008); subtyping (Pierce and Turner 2000; Odersky et al. 2001); intersection, union, indexed and refinement types (Xi 1998; Davies and Pfenning 2000; Dunfield and Pfenning 2004); termination checking (Abel 2004); higher-rank polymorphism (Peyton Jones et al. 2007); refinement types for LF (Lovas and Pfenning 2007); contextual modal types (Pientka 2008; Pientka and Dunfield 2008); and compiler intermediate representations (Chlipala et al. 2005).

Bidirectional typechecking is necessary because annotation-free type inference, which works well for the lambda calculus with prenex polymorphism, becomes difficult (if not undecidable) when we add first-class polymorphism, subtyping, intersection types, and so forth. Bidirectional typechecking is nice because reports of type errors are better localized, which is useful even when type inference is feasible.

In earlier work, we gave a concise recipe for bidirectional typechecking (Dunfield and Pfenning 2004), in which annotations are needed exactly where redexes appear. But we left out a vital feature: parametric polymorphism. So what are the proper bidirectional introduction and elimination rules for parametric polymorphism? It turns out that the introduction rule is easy, but the elimination rule is hard. For example, if we have a polymorphic function \( \text{choose} : \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha \), to find the right instantiation of \( \alpha \) in the application \( \text{choose} \ x \ y \) we must look at \( x \)’s type (and, for certain mixtures of type system features, \( y \)’s type as well). Clearly, we do not know how to instantiate \( \alpha \) from the term \( \text{choose} \) alone.

How can we find polymorphic instances in a bidirectional type system that is simple to formulate and use—without a heavy type annotation burden? I adapt an idea of Cardelli (1993), greedy: the first constraint on a type variable determines the instantiation. For \( \text{choose} \ x \ y \), this means \( \alpha \) is determined by the type of \( x \).

In this paper, I show how to use greedy to find polymorphic instances in System F (Girard 1986; Reynolds 1974), where polymorphism is first-class (higher-rank and impredicative). This yields a remarkably simple algorithm that is complete for predicative polymorphism (including ML-style prenex polymorphism). That is, if a typing derivation exists that instantiates type variables at monomorphic types, the user gives no more information than the annotations already present (on redexes) if there were no polymorphism. The algorithm handles some uses of impredicative polymorphism, where type variables are instantiated with polymorphic types, without extra help: for the rest, I provide a “hint” mechanism. Using intersection and union types, the approach can even handle subtyping, as described elsewhere (Dunfield 2009).

This paper shows that first-class polymorphism, while often tricky with type inference, is manageable in bidirectional typechecking. Rather than starting with Damas-Milner inference, perhaps eventually trying to glue on some bidirectionality for the season’s latest type features, we get simplicity and power by making things bidirectional from the ground up.

I will begin by giving a point of reference: a bidirectional type system that assumes polymorphic instances are found magically (Section 2). Section 3 develops a decidable version of that system and shows that it is complete, with respect to the Section 2 system, for typing derivations that use only predicative polymorphism. Section 4 adds datatypes, Section 5 briefly sketches subtyping, and Section 6 explains the implementation.

2. System Bi

System Bi is a very simple bidirectional type system with first-class polymorphism. It does not touch the problem of finding polymorphic instances; that is left to System Bi\(^3\) (“bi ex”), described in the next section. But it is a good reference point for proving things about System Bi\(^3\).

Figure 1 gives the syntax of terms, types, etc. For simplicity, I omit some constructs such as fixed point recursion \( \text{fix} \) \( u.e \), which is easy to handle as in previous work (Dunfield and Pfenning 2004). We’ll also gloss over datatypes \( \hat{\tau} \) where \( \delta \) is the name of an \( n \)-argument inductive datatype and \( \hat{\tau} \) is a sequence of \( n \) types. For example, given a base type \( \text{int} \) and the one-argument datatype list, we can write \( \text{int list} \). Term-level data constructors have constructor...
Abstract syntax tree representation of declarations of recursive functions, but not on function calls (except where a synthesis rule—where the type being eliminated appears—can be used where a type is well-formed). This yields the smallest sensible set of rules, and of an elimination rule—where the type being eliminated appears—since I’m interested primarily in call-by-value languages with side effects. Of course this is not practical—indeed, it begs the question this paper is supposed to answer—and we will address this in System Bi.

Figure 1: Grammar and operational semantics for System Bi
in the outer annotation. The type variable $\alpha$ is bound by $\Gamma'$, and its scope is $x: \alpha \vdash \ldots$, but the program variable $x$ in $x: \alpha$ is in the scope of $\alpha$.

$$(\lambda x. \alpha. n. \ldots) ((\lambda y. \text{Cons}(y, \text{Nil})) : (\alpha, x: \alpha \vdash \alpha \rightarrow \alpha \text{ list})) \ldots$$

$\forall \alpha. \alpha \rightarrow \text{int} \rightarrow \alpha$ list

This avoids having a term-level binder for type variables. Allowing something like

$$(\lambda x. n. \ldots) ((\lambda y. \text{Cons}(y, \text{Nil})) : (\alpha, x: \alpha \vdash \alpha \rightarrow \alpha \text{ list})) \ldots$$

$\forall \alpha. \alpha \rightarrow \text{int} \rightarrow \alpha$ list

does not sit well: the underlined $\alpha$ is not within the most natural scope of $\alpha$, which is simply $\alpha \rightarrow \alpha \rightarrow \alpha$ list. Letting $\alpha$ be in scope within the body of the annotated term breaks down if we add intersection types (which aren’t in this paper, but we want a general mechanism).

Figure 4 gives the rules for deriving $(\Gamma' \vdash A') \subseteq (\Gamma \vdash A)$, where $(\Gamma' \vdash A')$ is the user’s typing from an annotation, $\Gamma$ is the “ambient” context under which the annotated term $(e : N)$ is being typed, and $A'$ is $A$, renamed as needed to match $\Gamma$. The only output is $A$. Rule $\subseteq$-empty allows closed types with an empty context, e.g. $(e : \vdash ((\forall \beta. \beta \rightarrow \beta) \vdash 1))$; in practice, the “$\vdash$” can be omitted. Rule $\subseteq$-pvar is used when the typing mentions a program variable, as $x: \alpha$ in the example; the premise $\Gamma(x) \equiv B_0$ denotes equality, modulo renaming of type variables. Rule $\subseteq$-tyvar allows $\alpha$-varying (no pun intended) of type variables. Note that as the rules traverse the left-hand context from left to right, the left-hand context in the judgment can become ill-formed, but the output (right-hand) context is always well-formed.

Contextual annotations’ major virtue is robustness: they work with or without intersection types, index refiners, and other features. The formalism can be simplified in practice—since we don’t regard type variables in the ambient context as being in scope in $\Gamma_0$, and the notation (and implementation) syntactically distinguish type variables from other things, the type declarations $\alpha$ could be omitted. Or, as long as we don’t have intersection types, we could declare that $\alpha$ is within the scope of its annotation, cutting out the nondeterministic choice of $\alpha$ in $\subseteq$-tyvar.

Contextual annotations also set the stage for System Bi, where we’ll add hint declarations $\text{hint} (\Gamma_\alpha \vdash A)$ in $e$. These are suggestions from the user to the typechecker: under a context $\Gamma$, when examining $e$, the typechecker can try $A$ when instantiating a quantifier $\forall \beta$, $B$—with the context $\Gamma_\alpha$ establishing the map from type variables in $A$ to type variables in $\Gamma$.

2.2 The metatheory of System Bi

Type safety can be proved in a three-step process:

1. Define a type assignment version of System Bi.
2. Show that every derivation in System Bi has a corresponding derivation in the type assignment system.
3. Prove a type safety theorem for the type assignment system, with respect to the operational semantics in Figure 1.

Step 1 is very easy: drop the rule anno and replace “$\vdash$” and “$\triangleright$” symbols in the typing judgments with “$\vdash$”. For example, $\Gamma \vdash e_1 e_2 : B$ in rule $\rightarrow$E becomes $\Gamma \vdash e_1 e_2 : B$.

For Step 2, we must show that given a derivation of $\Gamma \vdash e \triangleright A$ (or of $\Gamma \vdash e \vdash A$) in System Bi, we can construct a derivation of $\Gamma \vdash e : A$, where $e' = e$ with annotations erased. This is an easy proof by induction on the derivation, and I proved it in my dissertation [Dunfield, 2002], Ch. 2 for a similar (though richer) system. The only novelty here is parametric polymorphism, which presents no difficulties: the cases for $\forall I$ and $\forall E$ almost exactly follow the cases for $\Pi I$ and $\Pi E$ (the rules for universal index quantification, a type system feature omitted from this paper for simplicity).

Step 3 is not trivial, but it is an easy extension of the proof in my dissertation [Dunfield, 2002], Ch. 2. As in Step 2, the reasoning for $\forall I$ and $\forall E$ follows the reasoning for $\Pi I$ and $\Pi E$. In particular, there is no need to extend derivation rank and value definiteness [Dunfield, 2002], pp. 36–38, concepts needed for union types—which are not even present in System Bi.

3. System Bi: Explicit Existential Variables

Now let’s transform the declarative System Bi into an algorithmic System Bi (“bi e”) by adding existential variables for unsolved polymorphic instances. After extending the syntax, we explain the typing and subtyping rules, discuss the hint construct, and then prove (with respect to System Bi) soundness and a limited form of completeness.

Types

$A \ ::= \ldots | \hat{\alpha}$

Contexts

$\Gamma, \Omega \ ::= \ldots | \Gamma, \hat{\alpha} | \Gamma, \hat{\alpha}=A | \Gamma, \hat{\alpha}=\pi | \Gamma, \text{hint}(\Gamma' \vdash A')$

Terms

$e \ ::= \ldots | \text{hint} (\Gamma' \vdash A') \in e$

We write $\hat{\alpha}, \hat{\beta}$, and so on for existential type variables, created in situations corresponding to the $\forall I$ and $\forall L \subseteq$ rules of System Bi. We create $\hat{\alpha}$ by adding $\hat{\alpha}$ to the context $\Gamma$. When the system finds a solution (e.g. when trying to derive $\hat{\alpha} \leq 1$) the declaration $\hat{\alpha}$ is replaced by $\hat{\alpha}=1$, indicating that the solution of $\hat{\alpha}$ is $1$. Contexts are ordered: the position of the declaration $\hat{\alpha}$ determines which variables can appear in a solution: in the context $\Gamma_1, \hat{\alpha}=A, \Gamma_2$ the solution type $A$ must be well-formed under $\Gamma_1$, without using anything declared in $\Gamma_2$. This prevents circularity, and allows rules like $\forall I$ that add non-existential declarations to remove them without making dangling references. Similarly, $\hat{\alpha}, x: \hat{\alpha}$ is well-formed because $\hat{\alpha}$ is declared before $x: \hat{\alpha}$.

Since the rules need to add and replace things in $\Gamma$, we modify judgment forms like $\Gamma \vdash e \downarrow C$:

$$\Gamma \vdash e \downarrow \text{C becomes } \Gamma \vdash e \downarrow \text{C }\vdash \Gamma'$$

$$\Gamma \vdash e \uparrow C \becomes \Gamma \vdash e \uparrow C \vdash \Gamma'$$

$$\Gamma \vdash A \leq B \becomes \Gamma \vdash A \leq B \vdash \Gamma'$$

The output context $\Gamma'$ is like $\Gamma$ but may have more information, containing new $\hat{\alpha}$ and $\hat{\alpha}=A$ elements, and various $\hat{\beta}$ elements replaced by $\hat{\beta}=B$ elements. (I chose $\uparrow$ and $\uparrow$ to suggest the fact that $\Gamma$ and $\Gamma'$ are equivalent in a declarative sense: if all the $\hat{\alpha}, \hat{\alpha}=A, \hat{\alpha}=\pi$, $\text{hint}(\ldots)$ declarations are dropped from $\Gamma$ and $\Gamma'$, those contexts are equal.)

For the marker $\hat{\alpha}=\pi$, we can thank the proof of predicative completeness: one typing rule $(\forall L \subseteq)$ needs this marker to remove junk—$\hat{\alpha}$-variables that have gone out of scope—from the output context. Junk is harmless but would compromise the proof. Markers are ignored otherwise (and need not be implemented).

$$
\begin{align*}
FV(A) & \subseteq \text{dom}(\Gamma) \quad \forall \alpha \not\subseteq \text{dom}(\Gamma) \quad \Gamma_1, \hat{\alpha} \vdash \Gamma_2 \text{ wf} \\
\Gamma \vdash A \text{ wf} & \implies \Gamma_1 \vdash \hat{\alpha}, \Gamma_2 \text{ wf} \\
\hat{\alpha} \not\subseteq \text{dom}(\Gamma) & \implies \Gamma_1 \vdash A \text{ wf} \quad \Gamma_1, \hat{\alpha}=A \vdash \Gamma_2 \text{ wf} \\
\Gamma \vdash \text{ wf} & \implies \Gamma_1 \vdash \hat{\alpha}=A, \Gamma_2 \text{ wf}
\end{align*}
$$

Figure 5: Well-formedness of existential contexts and types
A context $\Gamma$ is well-formed, $\vdash \Gamma$ wf, if each variable occurs once in its domain (defined below) and each type in $\Gamma$ is well-formed under the declarations to its left.

**Definition 1** (Domain of $\Gamma$). The domain $\text{dom}(\Gamma)$ of a context $\Gamma$ is:

- $\text{dom}(\emptyset) = \emptyset$
- $\text{dom}(\Gamma, x:A) = \text{dom}(\Gamma) \cup \{x\}$
- $\text{dom}(\Gamma, \alpha) = \text{dom}(\Gamma) \cup \{\alpha\}$
- $\text{dom}(\Gamma, \alpha = A) = \text{dom}(\Gamma) \cup \{\alpha\}$
- $\text{dom}(\Gamma, \text{hint}(\Gamma' \vdash A')) = \text{dom}(\Gamma)$
- $\text{dom}(\Gamma, \bullet_{\alpha}) = \text{dom}(\Gamma)$

To prove properties of System Bi\(^2\), it’s useful to view existential contexts as iterated substitutions, so that

$$[\hat{\alpha} = : A, \hat{\beta} = : \alpha]([\hat{\alpha} = : \beta] = A \rightarrow A)$$

The context is applied from the right, so first $\hat{\alpha}$ replaces $\hat{\beta}$, giving $\hat{\alpha} \rightarrow \alpha$, and then $\hat{\beta}$ replaces $\hat{\alpha}$, resulting in $A \rightarrow A$.

We only apply contexts that complete the context in which the type lives, so all existential variables disappear: given $\hat{\alpha} \rightarrow \beta$, well-formed in the context $[\hat{\beta} = : \alpha, \hat{\alpha} = : \beta]$, applying $[\hat{\beta} = : \alpha, \hat{\alpha} = : \beta] \rightarrow \hat{\beta}$ yields $1 \rightarrow 1$. To apply a context $\Omega$ to another context $\Gamma$, the contexts must be the same except for $\Gamma$ having more unsolved variables (and ignoring hints and markers), and $\Omega$ having solutions for variables not even mentioned in $\Gamma$:

$$[\alpha, x:A] \vdash \Omega; [\Gamma, x:A] = [\Omega; [\Gamma, x:A]]$$
$$[\alpha, \alpha] \vdash \Omega; [\Gamma, \alpha] \quad = \quad \Omega; [\Gamma, \alpha]$$
$$[\alpha, \hat{\alpha} = A] \vdash \Omega; [\Gamma, \hat{\alpha} = A] \quad = \quad \Omega; [\Gamma, \hat{\alpha} = A]$$

$x$ is well-formed, $\forall \alpha [\alpha, x:A] \vdash \forall \beta [\alpha, \beta]$. Because $[\hat{\alpha} = : \alpha, \hat{\alpha} = : \beta] \dashv \vdash 1 = 1 = [\hat{\alpha} = : \alpha, \hat{\alpha} = : \beta]$

**Definition 2** (Solved contexts). A context $\Gamma'$ is solved if it contains no unsolved existentials $\hat{\alpha}$.

**Definition 3.** We write $\Gamma \subseteq \Omega$ if (1) for all $\hat{\alpha} \in \Gamma$ there is a solution $\hat{\alpha} = A$ in $\Omega$, and for all $\hat{\alpha} = A_r$ in $\Gamma$ there is $\hat{\alpha} = A_0$ such that $[\hat{\alpha} = A_0] = [\hat{\alpha} = A_r]$, and (2) declarations present in $\Omega$ appear in the same order in $\Gamma$.

**Definition 4** (Completion of contexts). A context $\Omega$ completes a context $\Gamma$ iff $\Gamma \subseteq \Omega$ and $\Omega$ is solved.

How these existential contexts behave is best shown with an example. Suppose that $\Gamma$ has $\vdash A \Rightarrow \text{bool}$. At the top of Figure\(^3\) is a derivation in System Bi, which “guesses” $\alpha = \text{int}$.

At the bottom of the figure is a derivation in System Bi\(^2\). It has three interesting parts: the names of the involved rules are shaded, along with changes in the existential context. Towards the left we apply $\forall\text{Ex}$, adding an unsolved existential $\hat{\alpha}$ to the output context. Along the upper right is a use of $\Rightarrow \text{L}_{\leq}$, which expresses the essence of the greedy method: if we need to satisfy $\hat{\alpha} \leq B$, take $B$ as the solution. In this example, $B$ is int. The premise of $\alpha \Rightarrow \text{int}$ checks that the solution is well-formed in the context to the left of $\hat{\alpha}$ in $\Gamma$, $\hat{\alpha}$.

Existential contexts flow “in-order”, starting in the conclusion on the left of the $\vdash$, up to the first premise (left of the $\vdash$), into the first premise’s derivation, then back into the first premise itself (right of the $\vdash$), over to the second premise (left of the $\vdash$), etc., and finally back to the conclusion on the right of the $\vdash$.

Finally, while omitted from the figure, within the subderivation of $\Gamma, \hat{\alpha} = \text{int}$ $\vdash \Gamma, \hat{\alpha} = \text{int}$ we apply a rule to replace $\hat{\alpha}$ with $\text{int}$; this is not done implicitly.

### 3.1 Hints

We could have an explicit instantiation construct $e[A']$, such that if $e \vdash \forall \alpha, A$, then $\forall[A'] \vdash [A'/\alpha]A$. In effect, this gives an explicit version of $\forall\alpha$. But we also have the subtyping rule $\forall\alpha \leq B$, which can be used on a deeply nested quantifier—and then where would we put the $A'$? We might write a type annotation $(e : [A'/\alpha]A)$, but this is verbose when $A$ is long.

So, instead of a construct that only works with $\forall\alpha$, let one that sets the user suggest an instance for $\forall\alpha$ or $\forall\alpha \leq B$. The syntax is

$$\text{hint}(\Gamma' \vdash A') \in e$$

When encountered, the typing $(\Gamma' \vdash A')$ is put in $\Gamma$:

$$\Gamma, \text{hint}(\Gamma' \vdash A') \vdash e \Downarrow C \quad \text{hint} \quad \Gamma \vdash \text{hint}(\Gamma' \vdash A') \in e \Downarrow C$$

The type is then available to the rules $\forall\alpha$-hint and $\forall\alpha \leq B$. As with contextual annotations, the context $\Gamma'$ guides the interpretation of $A'$. For example, $\text{hint}(\alpha, x : A \vdash \forall \beta, \alpha \rightarrow \beta)$ in $e$ constrains $\alpha$ to be the type variable that is the type of $x$. On the other hand, $\text{hint}(\alpha \vdash \forall \beta, \alpha \rightarrow \beta)$ in $e$ is unconstrained; $\alpha$ could be replaced by any available type variable. This is managed through the contextual subtyping rules in Figure\(^4\).

One new contextual subtyping rule is needed, to ignore hint declarations:

$$\frac{}{\text{hint}(\Gamma' \vdash A'), f \vdash A_0}{\forall\alpha \vdash B \leftrightarrow A} \leq \text{hint}$$

To ensure decidability, rules hint and $\forall\alpha \leq B$ remove hints as they use them. With no restriction, writing $\text{hint}(\alpha \vdash \forall \beta, \alpha \rightarrow \beta)$ in $e$ $f$ $x$, where $f$ has $\forall\alpha$. $\alpha$ is fatal: using the hint, we replace $\alpha$ with $\forall \beta, \alpha \rightarrow \beta$, resulting in $\forall \beta, \beta$, on which we can use the hint again, and again...

### 3.2 Typing and subtyping rules

Many of the typing and subtyping rules of System Bi\(^2\) (Figure\(^5\)) are the same as System Bi, overlaid with existential contexts. We’ll look at typing first.

From the top, var, sub, anno, $\rightarrow I$, $\rightarrow E$, and $\forall\alpha$ clearly correspond to the rules in Figure\(^6\). Note that $\rightarrow I$ and $\forall\alpha$ add declarations $x : A$ and $\alpha$, respectively, and in their conclusions drop some existential declarations $\Gamma_2$. Those declarations are out of scope, and since they appear on the right, nothing else refers to them. $\forall\alpha\Rightarrow$ adds a fresh $\alpha$ to the existential context and synthesizes $\hat{\alpha} = A$. The rules $\exists\text{Sub}_\Downarrow$ and $\exists\text{Sub}_\Uparrow$ apply the solution to $\hat{\alpha}$ in the checking and synthesizing direction, respectively. $\exists\text{Sub}_\Downarrow$ does not apply $[A/\alpha]$ to $\Gamma$, because if we have, say, $y : \alpha$ in $\Gamma$, we can apply $\exists\text{Sub}_\Uparrow$ after applying var. The rule $\rightarrow \Rightarrow$ is syntax-directed: if checking a $\lambda$ against $\alpha$, then $\hat{\alpha} = : \alpha \rightarrow \hat{\alpha}$ for some new “articulation” variables $\hat{\alpha}_1, \hat{\alpha}_2$. Rule $\rightarrow \text{Ex} \Rightarrow$ is dual.

In the subtyping rules, we change $\forall\alpha \leq B$ as we change $\forall\alpha$, to add an $\hat{\alpha}$:

$$\frac{}{\Gamma, \bullet_{\alpha}, \hat{\alpha} \vdash [\hat{\alpha} = : \alpha, \hat{\alpha} = : \alpha] \leq B \rightarrow \Gamma', \bullet_{\alpha}, \hat{\alpha} \vdash \forall\alpha \leq B \rightarrow \Gamma'}$$

As in $\rightarrow I$ and $\forall\alpha$, the declarations following the added $\hat{\alpha}$ declaration are dropped. Because $\rightarrow \text{L}_{\leq}$ and $\rightarrow \hat{\alpha} \leq B$ (below) can insert

---

\(^2\) My implementation imposes a looser restriction: an $\forall$ that came from a hint cannot be instantiated with another $\forall$ from a hint, but can be used more than once.
existential “articulation” variables just before \( \hat{\alpha} \), however, an explicit marker is needed to drop those declarations. The marker \( \hat{\alpha} \) separates the context it follows from \( \hat{\alpha} \)’s articulation variables, and anything else \((\Gamma')\) created after it. This bookkeeping prevents existential variables that won’t subsequently be used from building up in the context, making the completeness proof easier to manage. (Implementing junk is harmless, and mine doesn’t try to remove it.)

The subtyping rules \( \text{ExSubst}(L,R) \leq \) correspond to the typing rule \( \text{ExSubst} \uparrow \). When there is an arrow on one side and an existential variable on the other, \( \rightarrow \hat{\alpha}L \leq \rightarrow \hat{\alpha}R \leq \) split the existential (similar to \( \rightarrow \alpha \)). Eventually an “atomic” type is reached, and \( \rightarrow \hat{\alpha}L \leq \rightarrow \hat{\alpha}R \leq \) can be applied. These rules greedily instantiate the existential variable on the other side of \( \leq \). “Atomic” is a misnomer here: it could be a polytype \( \forall \alpha.A \); the point is to keep it from being an arrow, which would complicate the proof of predicative completeness. The premises of \( \rightarrow \hat{\alpha}L \leq \) and \( \rightarrow \hat{\alpha}R \leq \) check that the solution is well-formed under the declarations that precede the variable.

### 3.3 Contextual subtyping rules

Because \( \leq \)-pvar uses equality, modulo renaming, instead of the full \( = \) relation, the contextual subtyping rules from System Bi do not change (apart from the \( \leq \)-hint rule).

### 3.4 Preliminaries

For the metatheory, we will use a function \( \Gamma \) that drops existential variable information and hints from \( \Gamma \), yielding an “ordinary” \( \Gamma \) consisting only of variable declarations \( x:A \) and type variables \( \alpha \).

\[
\begin{align*}
\varepsilon &= \cdot \\
\Gamma, x:A &\vdash \Gamma, \hat{\alpha} = \Gamma \\
\Gamma, \hat{\alpha} = \Gamma &\vdash \Gamma, \text{hint}(\Gamma') \vdash A' \Rightarrow \Gamma \\
\Gamma, \hat{\alpha} = \Gamma &\vdash \Gamma, \hat{\alpha} = \Gamma
\end{align*}
\]

The proof of Lemma 5 is by induction on the given derivation; Lemmas 6 and 7 by induction on \( \gamma_2 \).

**Lemma 5.** If \( \Gamma_1 \vdash J \vdash \Gamma_2 \) then \( \Gamma_1 \vdash \Gamma_2 \).

**Lemma 6.** If \( \Gamma_1, \Gamma_2 = \Gamma_2 \) where \( \Gamma_2 \) has the form \( x:A \) or \( \alpha \) then \( \Gamma_2 = \Gamma_21, \Gamma_22 \) where \( \Gamma_22 = \cdot \).

**Lemma 7.** \( (\Gamma_2 \vdash A) \leq (\Gamma_1, \Gamma_2 \vdash A) \).

**Corollary 8 (Reflexivity).** \( (\Gamma \vdash A) \leq (\Gamma \vdash A) \).

**Lemma 9.** If \( \Omega \) completes \( \Gamma \) then \( \text{dom}(\Omega|\Gamma) \subseteq \text{dom}(\Gamma) \).

**Proof.** By induction on \( \Omega \). Since \( \Omega \) completes \( \Gamma \), the contexts are the same modulo hints and existential variables that are declared in both but only solved in \( \Omega \), or declared in \( \Omega \) only. In the case when \( \Omega = \Omega', \hat{\alpha} = A \) and \( \Gamma = \Gamma', \hat{\alpha} : \text{from the definition, } \Omega|\Gamma = \Omega'|\Gamma' \). By IH, \( \text{dom}(\Omega') \subseteq \text{dom}(\Omega|\Gamma) \), and substituting for \( \hat{\alpha} \) in \( \Gamma' \) does not change its domain at all.

**Lemma 10.** Given a context \( \Omega \) that completes \( \Gamma \), if \( \Gamma \vdash A \text{ wf} \) then \( \Omega|\Gamma \vdash \Omega|A \text{ wf} \).

**Proof.** By inversion on \( \Gamma \vdash A \text{ wf} \), \( FV(A) \subseteq \text{dom}(\Gamma) \). Since \( \Omega \) completes \( \Gamma \) and \( \Gamma \) is (implicitly) well-formed, all free variables of \( \Omega|\Gamma \) are \( \alpha \)-variables, and \( \text{dom}(\Gamma) \) and \( \text{dom}(\Omega|\Gamma) \) have the same \( \alpha \)-variables. So \( FV(\Omega|\Gamma) \subseteq \text{dom}(\Omega|\Gamma) \).

**Lemma 11 (Well-Formedness).** If \( D : \rightarrow \rightarrow \rightarrow \vdash \Gamma \) then for any solved \( \hat{\alpha} \in \text{dom}(\Gamma) \), it is the case that \( \Gamma \) = \( \Gamma_1, \hat{\alpha} = A_1 \), \( \Gamma_2 \) and \( \Gamma_1 \vdash A \text{ wf} \), and likewise for any solved \( \hat{\alpha} \in \text{dom}(\Gamma|\Gamma') \).

**Lemma 12 (Monotonicity).** If \( \Gamma \vdash \rightarrow \rightarrow \rightarrow \vdash \Gamma' \) then for any \( \hat{\alpha} \in \text{dom}(\Gamma') \), one of the following holds:

1. \( \hat{\alpha} \) is unsolved in both \( \Gamma \) and \( \Gamma' \); or
2. there exists \( A' \) such that \( \hat{\alpha} \) is unsolved in \( \Gamma \) and \( \Gamma' = \Gamma_1, \hat{\alpha} = A', \Gamma_2 \); or
3. there exists \( A' \) such that \( \Gamma \) = \( \Gamma_1, \hat{\alpha} = A' \), \( \Gamma_2 \) and \( \Gamma' = \Gamma_1, \hat{\alpha} = A', \Gamma_2 \).

Also, markers are preserved:

- if \( \Gamma = \Gamma_1, \hat{\alpha}, \Gamma_2 \) then \( \Gamma' = \Gamma_1, \hat{\alpha}, \Gamma_2' \).

### 3.5 Decidability

System Bi\( ^\alpha \) is decidable. To concisely define an ordering on judgments such that the premises of each rule are smaller than its conclusion, we need several definitions:

1. \( A_1 \prec A_2 \) if \( A_1 \) is a proper subexpression of \( A_2 \), or if, by replacing one or more \( \hat{\alpha} \)s with \( \hat{\alpha} \)s in \( A_1 \), we get a proper subexpression of \( A_2 \).
2. \( \{C_1, C_2\} \prec \{D_1, D_2\} \) if \( C_k \neq D_k \) for all \( k, \ell \in \{1, 2\} \), and there exist \( k, \ell \) such that \( C_k \prec D_\ell \).
(iii) The weight of an existential variable \( \hat{\alpha} \) in \( \Gamma \) is the number of existential variables in \( \Gamma_1 \), where \( \Gamma = \Gamma_1, \hat{\alpha}[\ldots], \ldots \), plus itself. For example, the weight of \( \hat{\beta} \) in \( \hat{\alpha}, \hat{\beta} = 1 \) is 2. Solved and unsolved variables are counted alike. Weights are natural numbers, ordered by \(<\).

(iv) A type's angst with respect to \( \Gamma \) is the weight of the type's heaviest existential variable, again ordered by \(<\).

The last two criteria are motivated by \( \text{ExSubst}^I, \text{ExSubst}^R \), and \( \text{ExSubst}^L,R \). For example, the type in \( \text{ExSubst}^I \)'s premise is \( \Gamma(\hat{\alpha}) \) while its conclusion has \( \hat{\alpha} \). In the sense of part (i), \( \Gamma(\hat{\alpha}) \) could be much larger than \( \hat{\alpha} \). Counting the number of free existentials in the type doesn't work, because \( \hat{\alpha} \)'s solution could be \( \hat{\alpha}_1 \rightarrow \hat{\alpha}_2 \), which has two existential variables. But \( \hat{\alpha}_1 \rightarrow \hat{\alpha}_2 \) does have less angst than \( \hat{\alpha} \), because \( \hat{\alpha}_1 \) and \( \hat{\alpha}_2 \) must be declared before \( \hat{\alpha} \) in \( \Gamma \)—otherwise they could not appear in \( \hat{\alpha} \)'s solution.

In each rule, a term gets smaller, a type gets smaller (in the ordinary sense, e.g. A smaller than \( A \rightarrow B \)), or in the sense of becoming less angstful), the set of available hints gets smaller, or we introduce a solution for an existential variable. When comparing two synthesis judgments we flip the ordering of types because the types are output rather than input. The appendix has the definitions of the orderings on subtyping and typing judgments, and proofs of decidability.

### 3.6 Soundness of System Bî

Each System Bî derivation corresponds to a Bi one. In combination with type safety for a type assignment version of System Bi, this means that a well-typed-in-System Bî program won't go wrong:

**Theorem 13** (Soundness of System Bî). If \( \Gamma \vdash \mathcal{F} \vdash \Gamma' \) and \( \Omega \) completes \( \mathcal{F}' \) then \([\Omega][\Gamma'] \vdash [\Omega]\mathcal{F}' \), where \( \mathcal{F}' \) is \( \mathcal{F} \) with any \textbf{hint} \ldots in e subterms replaced by e and hints in annotations removed.
3.7 Completeness of System Bi\(^\wedge\)

We will show that, with respect to System Bi, System Bi\(^\wedge\) is incomplete for impredicative polymorphism, complete when hints are added to the term, and complete for predicative polymorphism.

3.7.1 Impredicative incompleteness

A small example shows that System Bi\(^\wedge\) is incomplete for impredicative polymorphism. We abbreviate \(\forall\)\(\beta\), \(\beta\rightarrow\beta\) as ID. Let \(\Gamma = \vdash:\forall\alpha.\alpha\rightarrow\alpha\rightarrow I\), \(x:(\text{int}\rightarrow\text{int})\rightarrow I\), \(y:ID\rightarrow I\). The derivation in System Bi, shown at the top of Figure 8, has no hint-articulated derivation in System Bi. But the second constraint (shaded) can even when possible; the only way to guarantee that System Bi\(^\wedge\) is complete for impredicative polymorphism, complete when \(\exists\) \(\forall\)\(\alpha\). A. The proof of the next theorem is in the appendix. Some of the proof cases use a lemma saying that if A is at least as polymorphic as B in System Bi, but A and B are actually monotypes, then A = B.

**Lemma 16.** If \(\Gamma \vdash A \leq B\) where A and B contain no \(\forall\), then A = B.

**Proof.** By induction on the given Bi derivation.

**Theorem 17** (Predicative Completeness). For any \(\Omega\) and \(\Gamma'\) and predicative derivation \(\Gamma \vdash [\Omega]\|\\) in System Bi, provided that

\(\begin{align*}
(1) \Omega & \text{ is predicative (for any } \bar{\alpha} \text{, the type } \Omega[\bar{\alpha}] \text{ is monomorphic) and articulated, and} \\
(2) \Omega & \text{ completes } \Gamma'_1, \text{ and } [\Omega]\|\Gamma'_1 = \Gamma, \text{ then}
\end{align*}\)

\[\begin{align*}
[\Omega]\|\Gamma'_1 \vdash [\Omega]\|\alpha' \leq [\Omega]\|\beta' \\
\text{where } \Gamma'_1 \vdash \alpha' \text{ wfr and } \Gamma'_1 \vdash \beta' \text{ wfr} & \implies \Gamma'_1 \vdash \alpha' \leq \beta' + \Gamma'_2 \\
[\Omega]\|\Gamma'_1 \vdash e \downarrow \beta' & \implies \Gamma'_1 \vdash e \downarrow \beta' \vdash \Gamma'_2
\end{align*}\]

for some \(\Omega'\) such that \([\Omega, \Omega']\) completes \(\Gamma'_2\) and \([\Omega, \Omega']\|\Gamma'_2 = \Gamma\).

**Proof.** See the appendix.

4. Datatypes

Supporting datatypes in System Bi is straightforward: we need only add two typing rules and one subtyping rule. The existential-context versions of these rules, in System Bi\(^\wedge\), follow the pattern of \(\rightarrow\). We articulate as with \(\beta\rightarrow\); the analogy is clear if we think of \(\rightarrow\) as just a two-argument datatype \(\langle(\alpha, \beta)\rightarrow\rangle\). So we have \(\delta\leq\leq\) and \(\delta\leq\leq\) following \(\rightarrow\leq\leq\) and \(\let\rightarrow\let\), and a rule \(\delta\leftarrow\) following \(\rightarrow\leftarrow\). But I have no \(\delta\leftarrow\), which would solve \(\rightarrow\) appropriately when checking case of ms where \(e\uparrow\alpha\); such a rule could work in some cases, such as case e of c(x) \(\Rightarrow\ldots\), because \(\delta\) can be inferred from the pattern c(x), but not in general. See Figure 9.

I assume covariant type arguments, but handling contravariant and nonvariant type arguments is easy: the rule \(\delta\rightarrow\) just needs to flip subtyping judgments (for contravariance), or add flipped judgments (for nonvariance). Actually determining the variance of type arguments is outside this paper’s scope.

5. Subtyping with Intersection and Union Types

By adding intersection and union types, we can extend System Bi and, more importantly, System Bi\(^\wedge\) to subtyping, replacing the weak “at least as polymorphic as” relation \(\leq\) with a richer \(\leq\).

A value has intersection type \(A \cap B\) if it has both type \(A\) and type \(B\). Intersection types can express combinations of properties of functions and data constructors (Reynolds 1996; Davies 2005; Dunfield and Pfenning 2004). Union types (Pierce 1991; Dunfield and Pfenning 2004; Dunfield 2007b) are dual to intersection types; a value has type \(A \vee B\) if it has type \(A\) or type \(B\) (or possibly both). Given an atomic subtyping relation on datatypes \(\delta_1 \leq \delta_2\),
simply adding to System Bi\textsuperscript{δ} typing and subtyping rules for intersections and unions (as we might in a setting without parametric polymorphism) delivers a reasonable system.

A further enhancement uses intersection and union types to refine the greedy approach itself. The idea is that, when the system tries to derive $\alpha \leq \text{nat}$ (with $\alpha$ as yet unknown), don’t instantiate $\alpha$ to nat permanently as above; instead, instantiate it provisionally. So, if we then see $\alpha \leq \text{int}$, we add $\text{int}$, yielding $\alpha = \text{nat} \land \text{int}$: the type nat $\land$ int, being the intersection of nat and int, is included in both, so nat $\land$ int $\leq$ nat and nat $\land$ int $\leq$ int. Dually, if we need to derive nat $\leq \beta$ and then int $\leq \beta$, we end up with $\beta = \text{nat} \lor \text{int}$, because nat $\lor$ int is a supertype of both nat and int.

For a fuller account of these systems, see [Dunfield 2009] (which I expect to revise). I mention them here to suggest that my approach is robust. It does not break when a supposedly tricky feature, subtyping, is added, and there is an actual synergy between the greedy method and intersection/union types.

### 6. Implementation

I implemented a version of System Bi\textsuperscript{δ} (with the datatype rules in Figure 9) as an extension of Stardust [Dunfield 2007], a typechecker for a subset of Standard ML with intersection types, union types, datatypes, and index refinements. The implemented system is much richer than System Bi\textsuperscript{δ}, but the examples here don’t use the extra features; unlike the extension mentioned in Section 5, the implementation does not automatically create intersections and unions.

The example in Figure 10 begins with a simple application of higher-rank predicative polymorphism, used in short-cut deforestation [Gill et al. 1993]. Types are quantified explicitly in the function type annotations ($\times \ldots \ast$). foldr uses only prenex polymorphism and can of course be written in SML, but br12d uses rank-2 polymorphism. The rest is adapted from [Leijen 2004], showing impredicative polymorphism.

#### 6.1 Complexity of typechecking

If hints are used, typechecking a function can be exponential in the number of hints: at each opportunity to apply $\forall \text{E} \Gamma$ or $\forall \text{E}$-hint, there is a choice between applying $\forall \text{E} \Gamma$ or applying $\forall \text{E}$-hint with the first available hint, with the second, etc. However, we can show the complexity is exponential even if $\forall \text{E} \Gamma$ is never used: As formulated, $\forall \text{E}$-hint drops a hint after use. First there are $n$ hints and $n$ choices; at the next opportunity to apply $\forall \text{E}$-hint there are $n - 1$ hints and $n - 1$ choices; and so on. If the last sequence of hints chosen is the only one to yield a valid derivation, we have done work proportional to $n \cdot (n - 1) \cdot \ldots \cdot 2$, or roughly $n^n$. 

---

**Type variable sequences** $\bar{\alpha}, \bar{\beta} := \cdot | \alpha | (\alpha_1, \ldots, \alpha_n)$

**Types** $A, B, C := A \rightarrow \alpha | \forall \alpha. A | \bar{B} \delta$

**Add to the System Bi rules in Figure 3**

\[
\frac{\Gamma \vdash c : A \rightarrow \overline{\delta} \beta \quad \Gamma \vdash e \downarrow A}{\Gamma \vdash c(e) \downarrow \overline{\beta} \overline{\delta}} \ \delta i
\]

**Add to the System Bi\textsuperscript{δ} rules in Figure 7**

\[
\frac{\Gamma_1 \vdash c : A \rightarrow \overline{\delta} \beta \quad \Gamma_2 \vdash e \downarrow A \vdash \Gamma_3}{\Gamma_1 \vdash c(e) \downarrow \overline{\beta} \overline{\delta} \vdash \Gamma_3} \ \delta i
\]

**Figure 8:** Derivation in System Bi using impredicative polymorphism (top), and a failed derivation in System Bi\textsuperscript{δ} (bottom)

**Figure 9:** Extending System Bi and System Bi\textsuperscript{δ} with datatypes
I have not analyzed the complexity of typechecking, but consider the function $\text{fun \ nonlinear}_{\alpha \beta \alpha'}(\alpha \beta \alpha' \beta') \equiv ()$ with type annotation $\forall \alpha \beta \alpha' \beta. \alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta \rightarrow \beta$. Given the context $id : \forall \delta \rightarrow \delta; uf : \alpha \rightarrow 1$, synthesizing a type for the application $\text{nonlinear}_{\alpha \beta \alpha'}(\alpha \beta \alpha' \beta')$ involves several nondeterministic choices of when to instantiate each of the $\alpha$ types. Still, this can be checked with only a few calls to the attempts that derive a subtyping judgment, and this continues to hold as we add arguments according to the same pattern. But if we introduce a type error, even an obvious one like an extra argument $\text{nonlinear}_{\alpha \beta \alpha'}(\alpha \beta \alpha' \beta')$, then by the time we reach the 14-argument function $\text{nonlinear}_{\alpha \beta \alpha'}$ it takes $87,000$ subtyping ($\leq$) calls and $49$ seconds to reject the program. This is a contrived example, and I have not yet found a real example that makes typechecking unacceptably slow.

Note that intersection types make these systems PSPACE-hard (Reynolds [1999]), even if parametric polymorphism is never used, and typechecking can be very slow when intersections and unions are used extensively (Dunfield [2009]).

7. Related Work

For impredicative System $F$ without annotations, type inference is undecidable (Wells [1999]); it becomes decidable if quantifiers are restricted to rank 2 or less (Kfoury and Wells [1994]).

Peyton Jones et al. (2007) developed a bidirectional system that supports arbitrary-rank, but predicative, polymorphism (quantifiers can appear anywhere in types, but polymorphic instances must be monotypic). Their system does not support subtyping, except for “at least as polymorphic as” subtyping (which we write as $\leq$).

My bidirectional systems are strictly weaker than Damas-Milner: they require annotations on redexes (though that requirement could be weakened by adding synthesis rules for some syntactic forms), and don’t manufacture quantifiers by generalizing type variables. $\text{ML}^\delta$ (Le Botlan and Rémy [2003]), a type inference system in the Damas-Milner tradition, supports impredicative polymorphism, with annotations needed only for impredicative instantiations (similar to my predicative completeness). $\text{ML}^F$ is more powerful than my systems, but appears substantially more complicated, even in its revised form (Rémy and Yakobowski [2008]).

$\text{HML}$ (Leijen [2009]) extends Damas-Milner and has similar goals to $\text{ML}^F$. $\text{HML}$ infers flexible types, polymorphic types that are bounded as below, where $\forall \beta \geq \forall (\alpha \rightarrow \alpha), \beta \rightarrow \beta$.$\text{HML}$ requires annotations only on polymorphic arguments, and is a good deal simpler than $\text{ML}^F$. It is robust under many simple transformations, such as $\text{revapp}_{\epsilon_2 \epsilon_1}$ in place of $\epsilon_1 \epsilon_2$ (where $\text{revapp}_{\epsilon_2 \epsilon_1}$ has type $\forall \alpha \beta, \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$. In contrast, System $\text{Bi}^\delta$ is sensitive to the ordering of terms when impredicative polymorphism is used; in the failed derivation in Figure 8, swapping the arguments x and y results in success.

In systems with subtyping, several approaches to inferring polymorphic instances have been presented.

In local type inference (Pierce and Turner [2000]), instances are found by computing upper and lower bounds on types, using information propagated locally within the program.

Colored local type inference (Odersky et al. [2001]) is akin to Pierce and Turner’s approach, but also allows different parts of type expressions to be propagated in different directions. My approach gets a similar effect by manipulating type expressions with $\alpha$-variables, which allows us to fix part of the type expression (the part that is not $\alpha$) while $\alpha$ remains flexible.

Davies’ Refinement $\text{ML}$ (Davies [2005]), an extension of Standard $\text{ML}$ with intersection types and subtyping, has a refinement restriction: the intersection $\alpha \land \beta$ can be formed only if $\alpha$ and $\beta$ are refinements (subtypes) of the same simple type. It is thus possible in his setting to do ordinary Damas-Milner $\text{SML}$ type inference to find simple-type instances of polymorphic variables. In his system, there are only finitely many subtypes of a given simple type, so the one that will make typechecking succeed can be found (in theory, and often in practice), by exhaustive search.

8. Conclusion

I have presented a new approach to inferring polymorphic instances in a bidirectional setting. This paper applies this approach to first-class polymorphism, without subtyping. At first, my goal was simply to add parametric polymorphism to the type systems described in my dissertation—the application to first-class polymorphism was a pleasant surprise. System $\text{Bi}^\delta$ is a “light” version of a rich System $\text{Bi}^{\leq \delta}$ with subtyping, intersection, and union types. As describing both systems would be (or, shall we say, was) on the long side for a conference, I intend to write a journal article; in the meantime, see Dunfield [2009].

The type systems in this paper might seem odd at first. System $\text{Bi}^{\leq \delta}$, which is not inherently exoteric—it lacks intersections and unions—looks quite different from previous approaches to first-class polymorphism. Even those that use bidirectionality, such as Peyton Jones et al. (2007), are rooted in the Damas-Milner type inference tradition. My work here is rooted elsewhere (Dunfield and Pfennning 2004). I attribute the virtues of my work to the essential simplicity of bidirectional typechecking.

The systems in this paper, like those in its immediate ancestors (my dissertation and the works of Xi, Davies, Pfennning), are meant for typechecking, not elaboration/compilation: they do not insert explicit polymorphic abstractions and applications. Reformulating System $\text{Bi}^{\leq \delta}$ in an elaboration style looks straightforward, though.

In addition to investigating elaboration and compilation, I plan to extend this work to GADTs. With bidirectionality and existential type variables, I expect this to be relatively easy.

To designers of languages and type systems, consider bidirectional typechecking: as your type system becomes more powerful, you will likely outrun Damas-Milner inference, and making it
bidirectional from the beginning should lead to a cleaner and more logical system than what you get after retrofitting bidirectionality. If you don’t need subtyping, polymorphism is nearly free with your purchase of bidirectionality; if you do need subtyping, polymorphism is nearly free with your purchase of intersections and unions (stay tuned).

Acknowledgments. Thanks to Frank Pfenning, Sungwoo Park, Brigitte Pientka, an anonymous ICFP 2009 reviewer, and the ML Workshop reviewers for (variously) discussions, encouragement and useful comments on several drafts of this paper. A few parts of Stardust are based on code by Tolmach and Oliva [1998].

References

Appendix: Decidability, Soundness, Completeness

Lemma 1 (Well-Formedness). Proof. By induction on D. In the 6 rules that introduce existential solutions, the well-formedness of the solution is either explicit (α ∈ L ⪯ L ⪯ α ∈ R ⪯ α R) or is evident from the context (→α ∈ L, →α ∈ R ⪯ →α R).

Theorem 2. Proof. By induction on the given derivation. We show the ∀E case. Let Γ 1 H11 = {Γ 1 H11 hint(Γ 1 ⊢ A)}. By IH, Γ 1 H11, Γ 1 H11, Γ 1 E ⪯ e ⪯ ∃α. A ⊢ Γ 1 H11, Γ 1, By Corollary 8, [Γ 1 ⊢ A] ⪯ [Γ 1 H11, Γ 1 ⊢ A]. Finally, by ∀E-hint, Γ 1 H11, Γ 1 E ⪯ e ⪯ [A′/α]A ⊢ Γ 1 H11, Γ 1, which is the same, hint(Γ 1 ⊢ A′), Γ 1 H11, Γ 1 ⪯ e ⪯ [A′/α]A ⊢ Γ 1 H11, Γ 1, which was to be shown.

Corollary 3. Proof. By Theorem 2, Γ 1 H11 ⊢ e ∥ A ⊢ · where Γ 1 H11 consists of n hints. The result follows by applying the hint rule n times.

Definition 4 (Ordering of subtyping judgments). Given Γ 1 F 1 = Γ 1 B 1 ⪯ B 1 ⊢ · and Γ 2 F 2 = Γ 2 B 2 ⪯ B 2 ⊢ ·, the order < is defined lexicographically by

1. the numbers of hints in Γ 1 and in Γ 2, under <;
2. if B 1 = B 2 and Γ 1 = Γ 2, the angst of A 1 versus A 2; or, if A 1 = A 2 and Γ 1 = Γ 2, the angst of B 1 versus B 2;
3. [A 1, B 1] < [A 2, B 2];
4. A 1 = A 2 and B 1 = B 2 where all existential variables in A 1 (= A 2) are solved in Γ 1 but not in Γ 2; or, the same, swapping B 1 and B 2 for A 1 and A 2.
Definition 19 (Ordering of typing judgments).

Given \( J_1 = \Gamma \vdash e_1 \uparrow \uparrow \downarrow; C_1 \rightarrow \Gamma'_1 \)
and \( J_2 = \Gamma_2 \vdash e_2 \uparrow \uparrow \downarrow; C_2 \rightarrow \Gamma'_2 \),
we define \( J_1 \leq J_2 \) by the lexicographic ordering of:

1. \( e_1 \) and \( e_2 \) (subterm ordering);
2. the directions, considering \( \uparrow \) smaller than \( \downarrow \);
3a. If both are checking judgments:
   - (i) \( C_1 \preceq C_2 \);
   - (ii) \( \Gamma_2 = \Gamma_1 \) and \( C_1 \) has less angst then \( C_2 \); or
   - (iii) all \( \alpha \)-variables in \( C_1 (= C_2) \) are solved in \( \Gamma_1 \) but not in \( \Gamma_2 \).
3b. If both are synthesis judgments:
   - (i) the number of hints in \( \Gamma'_1 \) versus \( \Gamma'_2 \); if equal,
   - (ii) \( C_2 \preceq C_1 \);
   - (iii) \( \Gamma_2 \) has less angst with respect to \( \Gamma'_2 \) than \( C_1 \) w.r.t. \( \Gamma'_1 \).

Theorem (Decidability of Subtyping and Contextual Matching).

Given \( \Gamma, \Lambda, \) and \( B, \) the existence of \( \Gamma' \) such that \( \Gamma \vdash A \subseteq B \rightarrow \Gamma' \)
in System Bi\(^{\beta} \) is decidable. Moreover, given \( \Gamma_0, \Lambda_0, \) and \( \Gamma, \) the existence of \( A \) such that \( \Gamma_0 \vdash \Lambda_0 \subseteq \Gamma \rightarrow A \) is also in \( \Omega. \) Thus, we have \( \cdots \downarrow \subseteq \downarrow \Omega \), which was to be shown.

In the \( \downarrow \Omega \) case, we have \( \Gamma = B \in \Gamma. \) By Lemma 12 \([ \downarrow \Omega = B \in \Omega \) (\( \uparrow \uparrow \downarrow \Omega \)) \] is in \( \Omega \), so in fact \( \downarrow \Omega \rightarrow \Gamma \). The result follows by reflexivity of \( \subseteq \). The \( \downarrow \Omega \subseteq \downarrow \) is symmetric.

The \( \downarrow \Omega \subseteq \downarrow \Omega \subseteq \downarrow \Omega \) cases use similar reasoning as the \( \downarrow \Omega \rightarrow \Gamma \) case. The remaining cases are straightforward.

Theorem 17 (Predicate Completeness).

Proof. By induction on \( D. \) Note that the type \( C' \) in the consequent is well-formed under \( \Gamma_2 \)—and not necessarily under \( \Gamma_1, \) as \( \Gamma_2 \) may have existential type variables that \( \Gamma_1 \) does not.

- **Case \( \rightarrow \subseteq :**

  \[
  \Gamma \vdash B_1 \subseteq A_1 \quad \Gamma \vdash A_2 \subseteq B_2
  \]

  We know that \( \Downarrow \Omega[A'] = A_1 \rightarrow A_2. \) Either \( \rightarrow \) (case) \( A' = A_1 \rightarrow A_2 \) (so \( \Downarrow \Omega[A'] = \Downarrow \Omega[A_1] \rightarrow \Downarrow \Omega[A_2] = A_1 \rightarrow A_2 \)) or \( (\downarrow \downarrow \Omega) \) case \( A = \downarrow \downarrow \Omega[A] = \Downarrow \Omega[A]. \)

  Similarly, we distinguish \( \rightarrow \) (case) and \( \downarrow \downarrow \Omega \) depending on whether \( B' \) is \( B_1 \rightarrow B_2 \) or \( B. \) (Note that possibly \( \beta = \beta. \))

  - \( \leftarrow \) (case) and \( \downarrow \downarrow \Omega :\)
    - By IH, \( \Gamma' \vdash B_1 \subseteq A' \rightarrow \Gamma' \) and again, \( \Gamma' \vdash A'_1 \subseteq B_2 \rightarrow \Gamma' \). By \( \rightarrow \subseteq \), \( \Gamma' \vdash A'_1 \rightarrow A_2 \subseteq B_1 \rightarrow \Gamma' \). By \( \rightarrow \subseteq \).
    - If \( \Gamma' \) includes a solution for \( \alpha \), then:
  
    - \( \Gamma' \vdash B_1 \subseteq A'_1 \rightarrow B_1 \rightarrow \Gamma' \) by \( \rightarrow \subseteq \).
    - Otherwise, \( \Gamma' \) does not include a solution for \( \alpha. \) \( \Omega[\alpha] = \Omega[A'] = A_1 \rightarrow A_2. \) The types \( A \rightarrow \Omega[A] = \Downarrow \Omega[A]. \) We assumed that \( \Gamma' \) does not include a solution for \( \alpha, \) so \( \Gamma' = \Gamma_1, \uparrow \uparrow \downarrow \Omega. \) Let \( \Gamma_1 = \Gamma_1, \uparrow \uparrow \downarrow \Omega, \Gamma_2, \alpha = \alpha \rightarrow \rightarrow \alpha \rightarrow \rightarrow \alpha \). Then:

  \[
  \Gamma_1 \vdash B_1 \subseteq A_1 \rightarrow \Gamma_2
  \]

  By IH on \( \Gamma_1 \vdash B_1 \subseteq A_1, \) taking \( \Gamma_1, \Gamma_2, \alpha = \alpha \rightarrow \rightarrow \alpha \).

  \[
  \Gamma_2 \vdash A_2 \subseteq B_2 \rightarrow \Gamma_2
  \]

  By \( \rightarrow \subseteq \).

  \[
  \Gamma_2 \vdash B_2 \subseteq A_2 \rightarrow \Gamma_2
  \]

  By \( \rightarrow \subseteq \).

  \[
  \Gamma_2 \vdash B_2 \subseteq A_2 \rightarrow \Gamma_2
  \]

  By \( \rightarrow \subseteq \).

  \[
  \rightarrow \downarrow \downarrow \Omega \subseteq \rightarrow \downarrow \downarrow \Omega \subseteq \rightarrow \downarrow \downarrow \Omega
  \]

  - \( \rightarrow \) (case) and \( \downarrow \downarrow \Omega \) : Symmetric to the previous.
    - \( \rightarrow \) (case) and \( \downarrow \downarrow \Omega \) : If either \( \alpha \) or \( \beta \) is solved in \( \Gamma_1 \), then the solution in \( \Gamma_1 \) has an \( \alpha \) at its head (since the solution in \( \Omega \) does). Using suitably articulated contexts, use the IH, then use \( \rightarrow \downarrow \downarrow \Omega \) and \( \rightarrow \downarrow \downarrow \Omega \) as needed.

  If neither is solved and \( \alpha = \beta \), then the result follows by \( \rightarrow \downarrow \downarrow \Omega \). Otherwise, neither is solved and \( \alpha \neq \beta \). So add a solution for whichever of \( \alpha \) and \( \beta \) is declared last in \( \Gamma_1 \). Suppose without loss of generality that \( \Gamma_1 = \Gamma_1, \Gamma_2, \Gamma_3, \beta = \beta. \) Then:

  \[
  \Gamma_1 \vdash B_1 \subseteq A_1 \rightarrow \Gamma_2
  \]

  By \( \rightarrow \downarrow \downarrow \Omega \).

  \[
  \Gamma_1 \vdash B_2 \subseteq A_2 \rightarrow \Gamma_2
  \]

  By \( \rightarrow \downarrow \downarrow \Omega \).

  \[
  \Gamma_2 \vdash A_2 \subseteq B_2 \rightarrow \Gamma_2
  \]

  By \( \rightarrow \downarrow \downarrow \Omega \).

  \[
  \rightarrow \downarrow \downarrow \Omega \subseteq \rightarrow \downarrow \downarrow \Omega \subseteq \rightarrow \downarrow \downarrow \Omega
  \]

  - **Case \( \rightarrow \) (case):**

    \( \rightarrow \) (case) and \( \downarrow \downarrow \Omega \) : We have \( \alpha = \Omega[A'] = \downarrow \downarrow \Omega[B']. \) The types \( A' \) and \( B' \) can each be \( \alpha \) or various existential variables.
If $A' = B' = \alpha$, the result follows by $\alphaRef{\leq}$, giving $\Gamma_i \vdash \alpha \leq \alpha \vdash \Gamma_i$. If $A' = \alpha$ and $B'$ is some solved $\beta$, the result follows by $\alphaRef{\leq}$, yielding $\Gamma_i' \vdash \alpha \leq \alpha \vdash \Gamma_i'$ then $\text{ExSubstR}{\leq}$ for $\Gamma_i' \vdash \alpha \leq \beta \vdash \Gamma_i'$. If $\beta$ is unsolved: $\beta$ is well-formed in $\Gamma_i'$, so $\Gamma_i' = \Gamma_1, \beta, \Gamma_k$. Applying $\alpha \mapsto \Gamma_i' \vdash \alpha \leq \beta \vdash \Gamma_i'$, $\Gamma_i = \Gamma_1, \beta, \Gamma_k$. Substituting gives $\Gamma_i' \vdash \alpha \leq \beta \vdash \Gamma_i'$, which was to be shown. The subcases where $B' = \alpha$ and $A'$ is some solved $\beta$ are symmetric to the last two.

If $A' = \gamma$ and $B' = \beta$, first apply $\alphaRef{\leq}$, then:

- If both are solved in $\Gamma_i'$, apply $\text{ExSubstL}{\leq}$ then $\text{ExSubstR}{\leq}$.
- If only $\gamma$ is solved, apply $\text{ExSubstL}{\leq}$ then $\alpha \mapsto \Gamma_i'$.
- If neither is solved: Both $\gamma$ and $\beta$ are well-formed under $\Gamma_i'$.

Either $\gamma$ comes first or $\beta$ comes first. Suppose $\beta$ comes first. Then $\alpha \mapsto \Gamma_i' \vdash \gamma \vdash \beta \vdash \ldots, \alpha = \beta = \ldots$.

**Case 1**: Similar to the previous case.

$$D := \Gamma \vdash [C/\alpha]A_0 \leq B \quad \Gamma \vdash C \text{ wf}$$

We know that $[\Omega]A' = \forall \alpha, A_0$. Either $\forall \alpha A' := \forall \alpha, A_0$, so $[\Omega]A' = \forall \alpha, A_0$, or $\forall \alpha A' := \forall \alpha, A_0$, where $[\Omega]A' = \forall \alpha, A_0$, which is impossible by the assumption that $\Omega$ is predicative.

- $\forall \alpha A'$: Choose a fresh $\alpha$. Let $\Omega' := Artic(\alpha, C)$.

$$\begin{align*}
A_0 &= [\alpha]A_0' \\
[C/\alpha]A_0' &= [\alpha]C_0 \\
\text{Applying $[\alpha]C_0$} &= \Omega'[\alpha]A_0' \\
\text{Permutation} &= \Omega'[\alpha]A_0' \\
\text{Def. of $[\alpha]A_0'$} &= \Omega'[\alpha]A_0'
\end{align*}$$

So $[\alpha]A_0' = [\alpha]C_0 \vdash A_0' \leq B' \vdash \Gamma_k$.

By IH w/ $\Gamma_k' \vdash \Omega, \Omega'$.

**Case 2**: Similar to the previous case.

$$D := \Gamma \vdash B \leq \forall \alpha, B_0$$

We know that $[\Omega]B' = \forall \beta, B_0$. Either $\forall \beta B'$ (so $[\Omega]B' = \forall \beta, B_0$) or $\forall \beta B'$ case $B'$ is $\gamma$.

- $\forall \beta B'$ case:

$$\begin{align*}
\Gamma_i', \alpha \vdash \beta' \leq B' \vdash \Gamma_k & \text{ By IH} \\
\Gamma_i' & \vdash \alpha \leq A' \vdash \Gamma_i' \\
\Gamma_i' & \vdash \alpha \leq \beta \vdash \Gamma_i'
\end{align*}$$

By Lemma [2]

**Case var**: $\Gamma = [\Omega]i'$, therefore $\Gamma(x) = [\Omega](\Gamma_i'(x))$. So $\Gamma_i(x) = A'$, where $[\Omega]A' = A$. The result, $\Gamma_i \vdash x \vdash A' \vdash \Gamma_i$, follows by var.

**Case sub**: $\Gamma \vdash e \vdash B \vdash A \leq A$.

By IH, $\Gamma_i \vdash e \vdash B \vdash \Gamma_m$ where $[\Omega]B = B$. We have $[\Omega]A' = A$. By IH, $\Gamma_m \vdash B' \leq A' \vdash \Gamma_i'$.

The result follows by the IH and anno. (The $\leq$ premise of anno in System $Bi^2$ does not involve existential contexts.)

**Case anno**: $\Gamma \vdash (e : N) \vdash A$.

$$D := \Gamma \vdash e : A \leq A$$

**Case 1**: Similar to the previous case.

$$D := \Gamma \vdash \alpha \leq \forall \alpha, A_0$$

By IH, $\Gamma_i \vdash e \vdash C' \vdash \Gamma_m$ where $[\Omega]C' = B \vdash A'$. If $C' \vdash A'$ then $\forall \alpha A' := \forall \alpha, A_0$. By IH, $\Gamma_m \vdash e \vdash B' \vdash \Gamma_i'$.

By Lemma [1]

**Case 2**: Similar to the previous case.

$$D := \Gamma \vdash e \vdash A_0$$

$\forall \alpha A'$ is either $\forall \alpha, A_0$ or $\beta$. But if $\beta$ then $\forall \alpha, A_0 = \forall \alpha, A_0$.

Therefore $\forall \alpha A'$ is either $\forall \alpha, A_0$, or $[\Omega]A_0 = A_0$. By $\forall \exists \alpha$ $\Gamma_i \vdash e \vdash \forall \alpha, A_0 \vdash \Gamma_i'$. By IH.

**Case 3**: Similar to the previous case.

$$D := \Gamma \vdash \alpha \vdash \forall \alpha, A_0$$

Let $\Omega' := Artic(\alpha = B)$. By IH with $(\Omega, \Omega')$, we have $\Gamma_i \vdash e \vdash A' \vdash \Gamma_i'$ where $\Omega, \Omega' = \forall \alpha, A_0$. Since $\Omega$ is predicative, $A'$ must have the form $\forall \alpha, A_0$ where $[\Omega]A_0 = A_0$. By $\forall \exists \alpha$ $\Gamma_i \vdash e \vdash \forall \alpha, A_0 \vdash \Gamma_i'$.

The context $\Omega, \Omega'$ includes the articulation of $\alpha = B$, so $[\Omega, \Omega'] = \forall \alpha, A_0$. Then $\forall \alpha, A_0 \vdash \forall \alpha, A_0 = [\beta/\alpha]A_0$.