Bidirectional polymorphism through greed and unions

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, even in the presence of subtyping. The first contribution is a bidirectional type system that supports first-class (higher-rank and impredicative) polymorphism but is complete for predicative polymorphism (including ML-style polymorphism and higher-rank polymorphism). This power comes from bidirectionality combined with a "greedy" method of finding polymorphic instances inspired by Cardelli's early work on System $F_{<:}$. The second contribution adds subtyping; combining bidirectional typechecking with intersection and union types fortuitously yields a simple but rather powerful system. Finally, I present a more powerful algorithm that forms intersections and unions automatically. This paper demonstrates that bidirectionality is a strong foundation for traditionally vexing features like first-class polymorphism and subtyping.

1. Introduction

To check programs in advanced type systems, it is often useful to split the traditional typing judgment $e : A$ into two forms, $e \Uparrow A$ read "e synthesizes type A" and $e \Downarrow A$ read "e checks against type A". 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). 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 $f : \forall\alpha.\, \alpha \to \alpha \to \alpha$, to find the right instantiation of α in the application $f\ x\ y$ we must look at x's type (and, for certain mixes of type system features, y's type as well). Clearly, we do not know how to instantiate α from the term f alone.

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

In this paper, I show how to use a "greedy method" 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 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 versatile "hint" mechanism.

I then show how to use intersection and union types to extend the method to systems with subtyping. Cardelli devised his greedy method for such systems, but without intersections and unions the method is weak: to find the instance of α for $f\ x\ y$ under subtyping, considering x alone may not suffice. Suppose $x : A$ and $y : B$ where A is a proper subtype of B. The first term visited (after f) is x, so we greedily take A as the instance of α. But y does not check against A, and typechecking fails. However, with union types the user can split α into two type variables, rewriting the type as

$$f : \forall\alpha_1, \alpha_2.\, \alpha_1 \to \alpha_2 \to (\alpha_1 \vee \alpha_2)$$

Now, in $f\ x\ y$ where $x : A$ and $y : B$, the variables α_1 and α_2 will be instantiated separately to A and B respectively, and the union of A and B is synthesized as the type of $f\ x\ y$. Alternatively, we can automatically "unionize" α, reducing the user's burden.

Taken together, these systems demonstrate that first-class polymorphism and subtyping, while tricky with type *inference*, are manageable in a bidirectional type*checking* approach. 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). Next, I explain in Section 3 a decidable version of that system (inspired by Cardelli's greedy algorithm), and show that it is complete, with respect to the Section 2 system, for typing derivations that use only predicative polymorphism. Section 4

A version of this paper was submitted to ICFP 2009. Version of 2009–04–07 plus bug fixes; recompiled 2020–11.

$$\begin{array}{rrcl}
\text{Type variables} & \alpha, \beta & & \\
\text{Atomic types} & A^{atomic} & ::= & \mathbf{1} \mid \alpha \mid \forall\alpha.\, A \\
\text{Types} & A, B, C & ::= & A^{atomic} \mid A \to B \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \mid \Gamma, \alpha \\
\text{Matches} & ms & ::= & \cdot \mid c(x) \Rightarrow e \parallel ms \\
\text{Annotations} & N & ::= & (\Gamma \vdash A) \\
\text{Terms} & e & ::= & x \mid () \mid \lambda x.\, e \mid e_1\, e_2 \\
 & & & \mid c(e) \mid \textbf{case}\ e\ \textbf{of}\ ms \mid (e : N) \\
\text{Values} & v & ::= & x \mid () \mid \lambda x.\, e \mid c(v) \mid (v : N) \\
\text{Evaluation contexts} & \mathcal{E} & ::= & [\,] \mid \mathcal{E}\, e \mid v\, \mathcal{E} \mid c(\mathcal{E}) \mid \textbf{case}\ \mathcal{E}\ \textbf{of}\ ms
\end{array}$$

$$\frac{e' \mapsto_R e''}{\mathcal{E}[e'] \mapsto \mathcal{E}[e'']} \qquad \begin{array}{rcl} (\lambda x.\, e)\, v & \mapsto_R & [v/x]e \\ \textbf{case}\ c(v)\ \textbf{of}\ \ldots c(x) \Rightarrow e \ldots & \mapsto_R & [v/x]e \end{array}$$

Figure 1: Grammar and operational semantics for System Bi

adds "real" subtyping, intersection types, and union types to the "magical" system, which Section 5 makes decidable. Section 6 describes its implementation. Finally, Section 7 presents an extension that automatically constructs intersections and unions within polymorphic instances where necessary, which is very helpful in certain situations.

2. System Bi

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

Figure 1 gives the syntax of terms, types, etc. For simplicity, we omit some constructs such as fixed point recursion $\textbf{fix}\ u.\, e$, which is easy to handle as in previous work (Dunfield and Pfenning 2004). We also gloss over datatypes $\vec{A}\ \delta$ where δ is the name of an n-argument inductive datatype and $\vec{A}$ is a sequence of n types. For example, given a base type int and the one-argument datatype list, we can write $\mathsf{int\ list}$. Term-level data constructors have *constructor type* $A^{con} = B \to \vec{\alpha}\ \delta$—no GADTs here. Datatypes are not particularly interesting in System Bi; while we give the syntax of case arms (matches ms) and constructors $c(e)$, we omit details such as the typing rules for case expressions.

The operational semantics (defined under type erasure) is straightforward, making use of evaluation contexts; $\mathcal{E}[e']$ is a term with e' in evaluation position.

Figure 2 has the rules for well-formedness of types and contexts. In general, we assume every context we write is well-formed, but tend to explicitly say when individual types are well-formed.

The bidirectional typing judgments are $\Gamma \vdash e \Uparrow A$, read "$e$ synthesizes A", and $\Gamma \vdash e \Downarrow B$, read "$e$ checks against B". (The arrows correspond to the flow of type information in an abstract syntax tree representation of e.) Figure 3 gives the typing rules. Introduction and elimination rules follow the pattern we introduced (Dunfield and Pfenning 2004): the conclusion of an introduction rule is checked against a given type, and the premise of an elimination rule—where the type being eliminated appears—synthesizes a type. This yields the smallest sensible set of rules, and means that annotations are needed only on redexes (including declarations of recursive functions).

The rule sub expresses a subsumption principle: if e synthesizes a type A that is at least as polymorphic as B—the known type that e is being checked against—then an A can be used where a B is expected. For example, a function of type $\forall\alpha.\, \alpha \to \alpha$ can be passed to a function expecting $\mathsf{int} \to \mathsf{int}$. We write this limited form of subtyping as $\Gamma \vdash A \leqq B$.

$$\boxed{\Gamma \vdash A\ \mathsf{wf}} \qquad \boxed{\Gamma\ \mathsf{wf}}$$

$$\frac{FV(A) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash A\ \mathsf{wf}} \qquad \frac{}{\cdot\ \mathsf{wf}} \qquad \frac{\Gamma\ \mathsf{wf} \quad x \notin \mathsf{dom}(\Gamma) \quad \Gamma \vdash A\ \mathsf{wf}}{\Gamma, x{:}A\ \mathsf{wf}}$$

Figure 2: Well-formedness of types and contexts

The rule anno is read as "if $N = (\Gamma' \vdash A')$ matches the typing $(\Gamma \vdash A)$ and e checks against A, then $(e : (\Gamma' \vdash A'))$ checks against A". The matching relation $\lesssim$ handles the renaming between Γ and Γ', and between A and A'. These *contextual annotations* are discussed below.

$\forall$I introduces a universal quantifier—with a value restriction, since my primary goal is a foundation for call-by-value languages with side effects. $\forall$E is an "oracular" elimination rule; it assumes someone has revealed to us the instance A'. Of course this is not practical—indeed, it begs the question that we want to answer—and we will address this in System Bi$^{\widehat{\alpha}}$.

Figure 3 gives the rules for the limited subtyping used by the rule sub. We continue to omit the rules for datatypes, and further assume that all polymorphic datatypes are covariant. In the rule $\forall L\leqq$, we write $[A'/\alpha]A$ to mean the substitution of A' for α in the type A. Following Dunfield and Pfenning (2003), reflexivity and transitivity are admissible and so need no explicit rules. For example, $\Gamma \vdash \forall\alpha.\, A \leqq \forall\beta.\, [\beta/\alpha]A$—which is the same as $\Gamma \vdash \forall\alpha.\, A \leqq \forall\alpha.\, A$—is derivable by (1) deriving $\Gamma, \beta \vdash [\beta/\alpha]A \leqq [\beta/\alpha]A$; (2) applying $\forall L\leqq$, giving $\Gamma, \beta \vdash \forall\alpha.\, A \leqq [\beta/\alpha]A$; (3) applying $\forall R\leqq$. (To prove transitivity, measure the derivations by the lexicographic ordering of (1) the number of $\forall L\leqq$ applications in the *second* derivation, with (2) the height of both derivations. This measure makes the $\forall R\leqq/\forall L\leqq$ case work.)

2.1 Contextual annotations

Annotations are *contextual* (Dunfield and Pfenning 2004): when checking $(e : (\Gamma' \vdash A'))$ under the context Γ, the context Γ' establishes the relationship between type variables declared in Γ and type variables used in A', the annotated type of e. For example, the following fragment uses the type of x to establish that the α in the inner annotation (on $\lambda y.\, \mathsf{Cons}(y, \mathsf{Nil})$) is the same as the α used in the outer annotation. The type variable α is bound by Γ', and its scope is $x{:}\alpha \vdash \ldots$, but the program variable x in $x{:}\alpha$ is free (and in the scope of λx).

$$\begin{array}{l}
\big(\lambda x.\, \lambda n.\, \ldots \\
\quad ((\lambda y.\, \mathsf{Cons}(y, \mathsf{Nil})) : (\alpha, x{:}\alpha \vdash \alpha \to \alpha\ \mathsf{list}))\ldots\big) \\
\qquad : \forall\alpha.\, \alpha \to \mathsf{int} \to \alpha\ \mathsf{list}
\end{array}$$

This avoids the need for a term-level binder for type variables. The tactic of writing

$$(\lambda x.\, \lambda n.\, \ldots (e : \underline{\underline{\alpha}}) \ldots) : \forall\alpha.\, \alpha \to \mathsf{int} \to \alpha\ \mathsf{list}$$

does not sit well: the underlined α is not within the most natural scope of α, which is just $\alpha \to \mathsf{int} \to \alpha\ \mathsf{list}$. Simply saying that α is in scope within the body of the annotated term breaks down when we add intersection types (Dunfield and Pfenning 2004).

In practice, Γ can be omitted where empty; the type variable declarations in Γ could be omitted (writing $(x{:}\alpha \vdash \alpha \to \alpha\ \mathsf{list})$ instead of $(\alpha, x{:}\alpha \vdash \alpha \to \alpha\ \mathsf{list})$; and (at least in a system without intersection types) one could even say that α is within the scope of its annotation. For this paper, contextual annotations' key virtue is robustness: they work well with or without intersection types, index refinements, and other features.

Contextual annotations also set the stage for System Bi$^{\widehat{\alpha}}$ when we add *hint declarations* $\textbf{hint}\ (\Gamma_A \vdash A)\ \textbf{in}\ e$. These are suggestions from the user to the typechecker: under a context Γ, when exam-

$\boxed{\Gamma \vdash e \Downarrow A}$ e checks against type A

$\boxed{\Gamma \vdash e \Uparrow A}$ e synthesizes type A

$$\frac{\Gamma(x) = A}{\Gamma \vdash x \Uparrow A}\ \mathsf{var} \qquad \frac{\Gamma \vdash e \Uparrow A \qquad \Gamma \vdash A \leqq B}{\Gamma \vdash e \Downarrow B}\ \mathsf{sub}$$

$$\frac{N \lesssim (\Gamma \vdash A) \qquad \Gamma \vdash e \Downarrow A}{\Gamma \vdash (e : N) \Uparrow A}\ \mathsf{anno} \qquad \frac{}{\Gamma \vdash () \Downarrow \mathbf{1}}\ \mathbf{1}\mathrm{I}$$

$$\frac{\Gamma, x{:}A \vdash e \Downarrow B}{\Gamma \vdash \lambda x.\, e \Downarrow A \to B}\ {\to}\mathrm{I} \qquad \frac{\Gamma \vdash e_1 \Uparrow A \to B \qquad \Gamma \vdash e_2 \Downarrow A}{\Gamma \vdash e_1\, e_2 \Uparrow B}\ {\to}\mathrm{E}$$

$$\frac{\Gamma, \alpha \vdash v \Downarrow A}{\Gamma \vdash v \Downarrow \forall\alpha.\, A}\ \forall\mathrm{I} \qquad \frac{\Gamma \vdash e \Uparrow \forall\alpha.\, A \qquad \Gamma \vdash A'\ \mathsf{wf}}{\Gamma \vdash e \Uparrow [A'/\alpha]\, A}\ \forall\mathrm{E}$$

$\boxed{\Gamma \vdash A \leqq B}$ A is at least as polymorphic as B

$$\frac{}{\Gamma \vdash \mathbf{1} \leqq \mathbf{1}}\ \mathbf{1}{\leqq} \qquad \frac{\Gamma \vdash B_1 \leqq A_1 \qquad \Gamma \vdash A_2 \leqq B_2}{\Gamma \vdash A_1 \to A_2 \leqq B_1 \to B_2}\ {\to}{\leqq}$$

$$\frac{}{\Gamma \vdash \alpha \leqq \alpha}\ \alpha\mathrm{Refl}{\leqq}$$

$$\frac{\Gamma \vdash [A'/\alpha]A \leqq B \qquad \Gamma \vdash A'\ \mathsf{wf}}{\Gamma \vdash \forall\alpha.\, A \leqq B}\ \forall\mathrm{L}{\leqq} \qquad \frac{\Gamma, \beta \vdash A \leqq B}{\Gamma \vdash A \leqq \forall\beta.\, B}\ \forall\mathrm{R}{\leqq}$$

Figure 3: Typing and subtyping in System Bi

ining e, the typechecker can try A when instantiating a quantifier $\forall\beta.\ B$—with the typing annotation's context Γ_A establishing the mapping between free type variables in A and type variables in Γ.

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.

For Step 1, drop anno and change all "$\Uparrow$" and "$\Downarrow$" symbols to ":". Step 2 consists of erasing annotations and removing applications of anno, following Dunfield and Pfenning (2004).

We defer discussion of Step 3 until we discuss System Bi$^{\leq}$, which has richer subtyping, intersections, and unions. That examination will also apply to System Bi.

3. System Bi$^{\widehat{\alpha}}$: Explicit Existential Variables

In this section, we will add to System Bi existential variables that represent unsolved polymorphic instances, yielding System Bi$^{\widehat{\alpha}}$ ("bi ex"). 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.

$$\begin{array}{rrcl}
\text{Types} & A & ::= & \ldots \mid \widehat{\alpha} \\
\text{Contexts} & \Gamma, \Omega & ::= & \ldots \mid \Gamma, \widehat{\alpha} \mid \Gamma, \widehat{\alpha}{=}A \mid \Gamma, \mathsf{hint}(\Gamma' \vdash A') \\
\text{Terms} & e & ::= & \ldots \mid \textbf{hint}\ (\Gamma' \vdash A')\ \textbf{in}\ e
\end{array}$$

We write $\widehat{\alpha}$, $\widehat{\beta}$, and so on for existential type variables, created in situations corresponding to the $\forall$E and $\forall$L$\leqq$ rules of System Bi. We create $\widehat{\alpha}$ by adding $\widehat{\alpha}$ to the context Γ. When the system finds a solution (e.g. when trying to derive $\widehat{\alpha} \leqq A$) the declaration $\widehat{\alpha}$ is replaced by $\widehat{\alpha}{=}A$, indicating that the solution of $\widehat{\alpha}$ is A. Contexts are ordered: the position of the declaration $\widehat{\alpha}$ determines which variables can appear in a solution: in the context $\Gamma_1, \widehat{\alpha}{=}A, \Gamma_2$ the solution type A must be well-formed under Γ_1, without using anything declared in Γ_2. This prevents circularity, and allows rules like $\forall$I that add non-existential declarations to remove them without making dangling references. Similarly, $\widehat{\alpha}, x{:}\widehat{\alpha}$ is well-formed because $\widehat{\alpha}$ is declared before $x{:}\widehat{\alpha}$.

$$\frac{FV(A) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash A\ \mathsf{wf}} \qquad \frac{\widehat{\alpha} \notin \mathsf{dom}(\Gamma_1) \qquad \Gamma_1, \widehat{\alpha} \vdash \Gamma_2\ \mathsf{wf}}{\Gamma_1 \vdash \widehat{\alpha}, \Gamma_2\ \mathsf{wf}}$$

$$\frac{}{\Gamma \vdash \cdot\ \mathsf{wf}} \qquad \frac{\widehat{\alpha} \notin \mathsf{dom}(\Gamma_1) \qquad \Gamma_1 \vdash A\ \mathsf{wf} \qquad \Gamma_1, \widehat{\alpha}{=}A \vdash \Gamma_2\ \mathsf{wf}}{\Gamma_1 \vdash \widehat{\alpha}{=}A, \Gamma_2\ \mathsf{wf}}$$

Figure 4: Well-formedness of existential contexts and types

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

$$\begin{array}{lcl}
\Gamma \vdash e \Downarrow C & \text{becomes} & \Gamma \vdash e \Downarrow C \dashv \Gamma' \\
\Gamma \vdash e \Uparrow C & \text{becomes} & \Gamma \vdash e \Uparrow C \dashv \Gamma' \\
\Gamma \vdash A \leqq B & \text{becomes} & \Gamma \vdash A \leqq B \dashv \Gamma'
\end{array}$$

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

A context Γ is well-formed, $\cdot \vdash \Gamma\ \mathsf{wf}$, if each variable occurs once in its domain (defined below) and each type in Γ is well-formed under the declarations to its left.

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

$$\begin{array}{lcl}
\mathsf{dom}(\cdot) & = & \emptyset \\
\mathsf{dom}(\Gamma, x{:}A) & = & \mathsf{dom}(\Gamma) \cup \{x\} \\
\mathsf{dom}(\Gamma, \alpha) & = & \mathsf{dom}(\Gamma) \cup \{\alpha\} \\
\mathsf{dom}(\Gamma, \widehat{\alpha}) & = & \mathsf{dom}(\Gamma) \cup \{\widehat{\alpha}\} \\
\mathsf{dom}(\Gamma, \widehat{\alpha}{=}A) & = & \mathsf{dom}(\Gamma) \cup \{\widehat{\alpha}\} \\
\mathsf{dom}(\Gamma, \mathsf{hint}(\Gamma' \vdash A')) & = & \mathsf{dom}(\Gamma)
\end{array}$$

To prove properties of System Bi$^{\widehat{\alpha}}$ it will be useful to view existential contexts as iterated substitutions, so that

$$[\widehat{\alpha}{=}A, \widehat{\beta}{=}\widehat{\alpha}](\widehat{\alpha} \to \widehat{\beta}) = A \to A$$

The context is applied from the right, so first $\widehat{\alpha}$ replaces $\widehat{\beta}$, giving $\widehat{\alpha} \to \widehat{\alpha}$, and then A replaces $\widehat{\alpha}$, yielding $A \to A$.

We only apply contexts that complete the context in which the type lives, so all existential variables disappear: given $\widehat{\alpha} \to \widehat{\beta}$, well-formed in the context $(\widehat{\beta}{=}\widehat{\alpha}, \widehat{\alpha})$, applying $[\widehat{\beta}{=}\widehat{\alpha}, \widehat{\alpha}{=}\mathbf{1}]\ \widehat{\alpha} \to \widehat{\beta}$ yields $\mathbf{1} \to \mathbf{1}$. To apply a context Ω to another context Γ, the contexts must be the same except for Γ having more unsolved variables (and ignoring hints in both):

$$\begin{array}{rcl}
[\cdot]\cdot & = & \cdot \\
[\Omega, x{:}A](\Gamma, x{:}A) & = & [\Omega]\Gamma, x{:}[\Omega]A \\
[\Omega, \alpha](\Gamma, \alpha) & = & [\Omega]\Gamma, \alpha \\
[\Omega, \widehat{\alpha}{=}A](\Gamma, \widehat{\alpha}\quad) & = & [\Omega]\ ([A/\widehat{\alpha}]\Gamma) \\
[\Omega, \widehat{\alpha}{=}A](\Gamma, \widehat{\alpha}{=}A) & = & [\Omega]\ ([A/\widehat{\alpha}]\Gamma) \\
[\Omega](\Gamma, \mathsf{hint}(\Gamma' \vdash A')) & = & [\Omega]\Gamma \\
[\Omega, \mathsf{hint}(\Gamma' \vdash A')]\Gamma & = & [\Omega]\Gamma
\end{array}$$

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

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \textit{filter} \Uparrow \forall\alpha.\,(\alpha\to\text{bool})\to\alpha\;\text{list}\to\alpha\;\text{list} \qquad \Gamma\vdash\text{int wf}}{\Gamma\vdash\textit{filter}\Uparrow(\text{int}\to\text{bool})\to\text{int list}\to\text{int list}}\,\forall\text{E} \qquad \dfrac{\Gamma\vdash f\Uparrow\text{int}\to\text{bool} \qquad \dfrac{\Gamma\vdash\text{int}\leqq\text{int}\qquad\Gamma\vdash\text{bool}\leqq\text{bool}}{\Gamma\vdash\text{int}\to\text{bool}\leqq\text{int}\to\text{bool}}\,{\to}{\leqq}}{\Gamma\vdash f\Downarrow\text{int}\to\text{bool}}\,\text{sub}}{\Gamma\vdash\textit{filter}\;f\Uparrow\text{int list}\to\text{int list}}\,{\to}\text{E} \qquad \Gamma\vdash xs\Downarrow\text{int list}}{\Gamma\vdash\textit{filter}\;f\;xs\Uparrow\text{int list}}\,{\to}\text{E}$$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \textit{filter} \Uparrow \forall\alpha.\,(\alpha\to\text{bool})\to\alpha\;\text{list}\to\alpha\;\text{list}\dashv\Gamma}{\Gamma\vdash\textit{filter}\Uparrow(\widehat{\alpha}\to\text{bool})\to\widehat{\alpha}\;\text{list}\to\widehat{\alpha}\;\text{list}\dashv\Gamma,\widehat{\alpha}}\,\forall\text{E}\widehat{\alpha} \qquad \dfrac{\Gamma,\widehat{\alpha}\vdash f\Uparrow\text{int}\to\text{bool}\dashv\Gamma,\widehat{\alpha} \qquad \dfrac{\dfrac{\Gamma\vdash\text{int wf}}{\Gamma,\widehat{\alpha}\vdash\widehat{\alpha}\leqq\text{int}\dashv\Gamma,\widehat{\alpha}{=}\text{int}}\,\widehat{\alpha}^{=}\text{L}{\leqq}\qquad\Gamma,\widehat{\alpha}{=}\text{int}\vdash\text{bool}\leqq\text{bool}\dashv\Gamma,\widehat{\alpha}{=}\text{int}}{\Gamma,\widehat{\alpha}\vdash\text{int}\to\text{bool}\leqq\widehat{\alpha}\to\text{bool}\dashv\Gamma,\widehat{\alpha}{=}\text{int}}\,{\to}{\leqq}}{\Gamma,\widehat{\alpha}\vdash f\Downarrow\widehat{\alpha}\to\text{bool}\dashv\Gamma,\widehat{\alpha}{=}\text{int}}\,\text{sub}}{\Gamma\vdash\textit{filter}\;f\Uparrow\widehat{\alpha}\;\text{list}\to\widehat{\alpha}\;\text{list}\dashv\Gamma,\widehat{\alpha}{=}\text{int}}\,{\to}\text{E} \qquad \Gamma,\widehat{\alpha}{=}\text{int}\vdash xs\Downarrow\widehat{\alpha}\;\text{list}\dashv\Gamma,\widehat{\alpha}{=}\text{int}}{\Gamma\vdash\textit{filter}\;f\;xs\Uparrow\widehat{\alpha}\;\text{list}\dashv\Gamma,\widehat{\alpha}{=}\text{int}}\,{\to}\text{E}$$

Figure 5: Typing derivations for *filter* f xs in System Bi, above, and System Bi$^{\widehat{\alpha}}$, below

Definition 3. We write $\Gamma \subseteq \Omega$ if (1) each declaration in Γ is either in Ω or its solution is (e.g. $\widehat{\alpha}$ is in Γ while $\widehat{\alpha}{=}A$ is in Ω), and (2) if two declarations appear in Γ they appear in the same order in Ω.

Definition 4 (Completion of contexts)**.** A context Ω *completes* a context Γ iff $\mathsf{dom}(\Omega) = \mathsf{dom}(\Gamma)$ and $\Gamma \subseteq \Omega$ and Ω is solved.

How these existential contexts behave is best shown with an example. Suppose that $\Gamma(f) = \text{int} \to \text{bool}$. At the top of Figure 5 is a derivation in System Bi, which "guesses" $\alpha = \text{int}$.

At the bottom of the figure is a derivation in System Bi$^{\widehat{\alpha}}$. 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 $\forall\text{E}\widehat{\alpha}$, adding an unsolved existential $\widehat{\alpha}$ to the output context. Along the upper right is a use of $\widehat{\alpha}^{=}\text{L}{\leqq}$, which expresses the essence of the greedy method: if we need to satisfy $\widehat{\alpha} \leqq B$, take B as the solution. In this example, B is int. The premise of $\widehat{\alpha}^{=}\text{L}{\leqq}$ checks that the solution is well-formed in the context to the left of $\widehat{\alpha}$ in $\Gamma, \widehat{\alpha}$.

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 $\dashv$), over to the second premise (left of the $\vdash$), etc., and finally back to the conclusion on the right of the $\dashv$.

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

3.1 Hints

We could have an explicit instantiation construct $e\,[A']$, such that if $e \Uparrow \forall\alpha.\,A$, then $e\,[A'] \Uparrow [A'/\alpha]A$. In effect, this gives an explicit version of $\forall$E. But we also have the subtyping rule $\forall\text{L}{\leqq}$, 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 the type A is long.

So, instead of a construct that only works with $\forall$E, we add one that lets the user suggest an instance for $\forall$E or $\forall\text{L}{\leqq}$. The syntax is

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

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

$$\dfrac{}{(\cdot \vdash A) \lesssim (\Gamma \vdash A)}\;{\lesssim}\text{-empty}$$

$$\dfrac{\Gamma \vdash \Gamma(x) \leqq B_0 \qquad (\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)}{(x{:}B_0, \Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)}\;{\lesssim}\text{-pvar}$$

$$\dfrac{\Gamma \vdash \alpha' \text{ wf} \qquad ([\alpha'/\alpha]\Gamma_0 \vdash [\alpha'/\alpha]A_0) \lesssim (\Gamma \vdash A)}{(\alpha, \Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)}\;{\lesssim}\text{-tyvar}$$

$$\dfrac{(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)}{(\text{hint}(\Gamma' \vdash A'), \Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)}\;{\lesssim}\text{-hint}$$

Figure 6: Contextual matching, used in the hint mechanism and in type annotations

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

The type is then available to the rules $\forall$E-hint and $\forall\text{L-hint}{\leqq}$. As with contextual annotations, the context Γ' guides the interpretation of A'. For example, **hint** $(\alpha, x{:}\alpha \vdash \forall\beta.\,\alpha \to \beta)$ **in** e constrains α to be the type variable that is the type of x. On the other hand, **hint** $(\alpha \vdash \forall\beta.\,\alpha \to \beta)$ **in** e is unconstrained; α could be replaced by any type variable. This is managed through the contextual subtyping rules in Figure 6.

To ensure decidability, each hint can be used at most once in the derivation.[1]

3.2 Typing and subtyping rules

Many of the typing and subtyping rules of System Bi$^{\widehat{\alpha}}$ (Figure 7) are the same as System Bi, overlaid with existential contexts. We discuss typing first.

From the top, var, sub, anno, $\to$I, $\to$E and $\forall$I clearly correspond to the rules in Figure 3. Note that $\to$I and $\forall$I add declarations $x{:}A$ and α, respectively, and in their conclusions drop some existential declarations Γ_Z. Those declarations are out of scope, and since they appear on the right, nothing else refers to them. $\forall\text{E}\widehat{\alpha}$

[1] Otherwise, writing **hint** $(\cdot \vdash \forall\beta.\,\beta)$ **in** $f\;x$ where f has type $\forall\alpha.\,\alpha$ is fatal. Using the hint, we replace α with $\forall\beta.\,\beta$, resulting in $\forall\beta.\,\beta$, on which we can use the hint again, and again. . .

adds a fresh $\hat{\alpha}$ to the existential context and synthesizes $[\hat{\alpha}/\alpha]A$. The rules ExSubst$\Downarrow$ and ExSubst$\Uparrow$ apply the solution to $\hat{\alpha}$ in the checking and synthesizing direction, respectively. ExSubst$\Downarrow$ does not apply $[A/\hat{\alpha}]$ to Γ, because if we have, say, $y{:}\hat{\alpha}$ in Γ, we can apply ExSubst$\Uparrow$ after applying var. The rule $\to$I$\hat{\alpha}$ is syntax-directed: if checking a λ against $\hat{\alpha}$, then $\hat{\alpha} = \widehat{\alpha_1} \to \widehat{\alpha_2}$ for some new variables $\widehat{\alpha_1}$, $\widehat{\alpha_2}$. Rule $\to$E$\hat{\alpha}$ is dual.

In the subtyping rules, we change $\forall$L$\leqq$ as we changed $\forall$E, to add an $\hat{\alpha}$. As in $\to$I and $\forall$I, the declarations following the added $\hat{\alpha}$ declaration are dropped; the notation $\hat{\alpha}[\ldots]$ represents either an unsolved $\hat{\alpha}$ or a solved $\hat{\alpha}{=}A$. The subtyping rules ExSubst{L,R}$\leqq$ correspond to the typing rule ExSubst$\Uparrow$. When there is an arrow on one side and an existential variable on the other, $\to\hat{\alpha}$L$\leqq$/ $\to\hat{\alpha}$R$\leqq$ split the existential (similar to $\to$I$\hat{\alpha}$). Eventually an “atomic” type is reached, and $\hat{\alpha}^{=}$L$\leqq$/ $\hat{\alpha}^{=}$R$\leqq$ can be applied. These rules greedily instantiate the existential to the atomic type on the other side of $\leqq$. “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 $\hat{\alpha}^{=}$L$\leqq$ and $\hat{\alpha}^{=}$R$\leqq$ check that the solution is well-formed under the declarations that precede the variable.

3.2.1 Greed and contextual matching $\lesssim$

The rules for $\lesssim$ do *not* change from System Bi to System Bi$^{\hat{\alpha}}$, despite the $\leqq$ premise in $\lesssim$-pvar.

When we add richer subtyping in later systems, we will need several typings in a single annotation. For example, in the term $\big(e : ((x{:}\mathsf{pos} \vdash A), (x{:}\mathsf{neg} \vdash B))\big)$ we check e against A when $x{:}\mathsf{pos}$ is in the context, and against B when $x{:}\mathsf{neg}$ is in the context. But what if $x{:}\hat{\alpha}$ is in the context? Greedily mapping $\hat{\alpha}$ to pos would defer to a perhaps arbitrary ordering $(x{:}\mathsf{pos} \vdash A)$-before-$(x{:}\mathsf{neg} \vdash B)$. It seems best to refuse such curiosities, so that when you write a contextual annotation that depends on the type of x, you must make the type of x explicit, by writing an annotation elsewhere.

This leads to an embedding of System Bi’s subtyping system within System Bi$^{\hat{\alpha}}$, and questions arise about how we instantiate polymorphic types within that premise of $\lesssim$-pvar. Our simple solution is to weaken that embedded subtyping system by depriving it of $\forall$L$\leqq$ and $\forall$R$\leqq$, and replacing those two rules with

$$\frac{\Gamma, \beta \vdash [\beta/\alpha]A \leqq B}{\Gamma \vdash \forall\alpha.\ A \leqq \forall\beta.\ B}$$

This applies only for the $\leqq$ premise in $\lesssim$-pvar; the $\leqq$ premise of sub uses the rules in Figure 7.

3.3 Preliminaries

For the metatheory, we will use a function $\overline{\Gamma}$ that drops existential variable information and hints from Γ, yielding an “ordinary” Γ consisting only of variable declarations $x{:}A$ and type variables α.

$$\begin{array}{rclrcl} \overline{\cdot} & = & \cdot & \overline{\Gamma, \hat{\alpha}} & = & \overline{\Gamma} \\ \overline{\Gamma, x{:}A} & = & \overline{\Gamma}, x{:}A & \overline{\Gamma, \hat{\alpha}{=}A} & = & \overline{\Gamma} \\ \overline{\Gamma, \alpha} & = & \overline{\Gamma}, \alpha & \overline{\Gamma, \mathsf{hint}(\Gamma' \vdash A')} & = & \overline{\Gamma} \end{array}$$

The next three lemmas are by induction on the given derivation; Γ_2; and Γ_2, respectively. Subsequently omitted proofs are in the appendix.

Lemma 5. *If* $\Gamma_1 \vdash \mathcal{J} \dashv \Gamma_2$ *then* $\overline{\Gamma_1} = \overline{\Gamma_2}$.

Lemma 6. *If* $\overline{\Gamma_1, \Gamma_Z} = \overline{\Gamma_2}$ *where* Γ_Z *has the form* $x{:}A$ *or* α *then* $\Gamma_2 = \Gamma_{21}, \Gamma_Z, \Gamma_{22}$ *where* $\overline{\Gamma_{22}} = \cdot$.

Lemma 7. $(\Gamma_2 \vdash A) \lesssim (\Gamma_1, \Gamma_2 \vdash A)$.

Corollary 8 (Reflexivity)**.** $(\Gamma \vdash A) \lesssim (\Gamma \vdash A)$.

Lemma 9. *If* Ω *completes* Γ *then* $\mathsf{dom}([\Omega]\Gamma) \subseteq \mathsf{dom}(\Gamma)$.

Corollary 10. *Given a context* Ω *that completes* Γ, *if* $\Gamma \vdash A\ \mathsf{wf}$ *then* $[\Omega]\Gamma \vdash [\Omega]A\ \mathsf{wf}$.

Lemma 11. *Given a context* Ω *that completes* Γ, *if* $[\Omega]\Gamma \vdash [\Omega]A\ \mathsf{wf}$ *then* $\Gamma \vdash A\ \mathsf{wf}$.

Lemma 12 (Well-Formedness)**.** *If* $\mathcal{D} :: \Gamma \vdash \ldots \dashv \Gamma'$ *then for any solved* $\hat{\alpha} \in \mathsf{dom}(\Gamma)$, *it is the case that* $\Gamma = \Gamma_1, \hat{\alpha}{=}A, \Gamma_2$ *and* $\Gamma_1 \vdash A\ \mathsf{wf}$, *and likewise for any solved* $\hat{\alpha} \in \mathsf{dom}(\Gamma')$.

Lemma 13 (Monotonicity)**.** *If* $\mathcal{D} :: \Gamma \vdash \ldots \dashv \Gamma'$ *then for any* $\hat{\alpha} \in \mathsf{dom}(\Gamma')$, *one of the following holds:*

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

3.4 Decidability

We prove that System Bi$^{\hat{\alpha}}$ 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:

(i) $A_1 \prec A_2$ iff A_1 is a proper subexpression of A_2, or if, by replacing one or more $\hat{\alpha}$s with αs in A_1, we get a proper subexpression of A_2.

(ii) $\{C_1, C_2\} \prec \{D_1, D_2\}$ iff $C_k \not\succ D_\ell$ for all $k, \ell \in \{1, 2\}$, and there exist k, ℓ such that $C_k \prec D_\ell$.

(iii) The *weight* of an existential variable $\hat{\alpha}$ in Γ is the number of existential variables in Γ_L where $\Gamma = \Gamma_L, \hat{\alpha}[\ldots], \cdots$, plus itself. For example, the weight of $\hat{\beta}$ in $\hat{\alpha}, \hat{\beta}{=}\mathbf{1}$ is 2. Solved and unsolved variables are counted alike. Weights are natural numbers, ordered by $<$.

(iv) The *angst* that a type has with respect to Γ is the weight of its heaviest existential variable, again ordered by $<$.

The last two criteria are motivated by ExSubst$\Downarrow$, ExSubst$\Uparrow$, and ExSubst{L,R}$\leqq$. For example, the type in ExSubst$\Downarrow$’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 $\widehat{\alpha_1} \to \widehat{\alpha_2}$, which has two existential variables. But $\widehat{\alpha_1} \to \widehat{\alpha_2}$ *does* have less angst than $\hat{\alpha}$, because $\widehat{\alpha_1}$ and $\widehat{\alpha_2}$ must be declared before $\hat{\alpha}$ in Γ—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 $A \to B$, 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 a full definition of the ordering of judgments and a proof.

3.5 Soundness of System Bi$^{\hat{\alpha}}$

Each System Bi$^{\hat{\alpha}}$ 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 Bi$^{\hat{\alpha}}$ program won’t go wrong.

Theorem 14 (Soundness of System Bi$^{\hat{\alpha}}$)**.** *If* $\Gamma \vdash \mathcal{J} \dashv \Gamma'$ *and* Ω *completes* Γ' *then* $[\Omega]\Gamma' \vdash [\Omega]\mathcal{J}'$, *where* $\mathcal{J}'$ *is* $\mathcal{J}$ *with any* **hint** $\ldots$ **in** e *subterms replaced by* e *and hints in annotations removed.*

$$\boxed{\Gamma \vdash e \Downarrow A \dashv \Gamma' \qquad \Gamma \vdash e \Uparrow A \dashv \Gamma'}$$

$$\frac{\Gamma(x) = A}{\Gamma \vdash x \Uparrow A \dashv \Gamma}\ \mathsf{var} \qquad \frac{\Gamma_1 \vdash e \Uparrow A \dashv \Gamma_2 \qquad \Gamma_2 \vdash A \leqq B \dashv \Gamma_3}{\Gamma_1 \vdash e \Downarrow B \dashv \Gamma_3}\ \mathsf{sub} \qquad \frac{N \lesssim (\Gamma \vdash A) \qquad \Gamma \vdash e \Downarrow A \dashv \Gamma'}{\Gamma \vdash (e : N) \Uparrow A \dashv \Gamma'}\ \mathsf{anno}$$

$$\frac{}{\Gamma \vdash () \Downarrow \mathbf{1} \dashv \Gamma}\ \mathbf{1}\mathrm{I} \qquad \frac{\Gamma, x{:}A \vdash e \Downarrow B \dashv \Gamma', x{:}A, \Gamma_Z}{\Gamma \vdash \lambda x.\, e \Downarrow A \to B \dashv \Gamma'}\ {\to}\mathrm{I} \qquad \frac{\Gamma_1 \vdash e_1 \Uparrow A \to B \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 \Downarrow A \dashv \Gamma_3}{\Gamma_1 \vdash e_1\, e_2 \Uparrow B \dashv \Gamma_3}\ {\to}\mathrm{E}$$

$$\frac{\Gamma, \alpha \vdash v \Downarrow A \dashv \Gamma', \alpha, \Gamma_Z}{\Gamma \vdash v \Downarrow \forall\alpha.\, A \dashv \Gamma'}\ \forall\mathrm{I} \qquad \frac{\Gamma \vdash e \Uparrow \forall\alpha.\, A \dashv \Gamma'}{\Gamma \vdash e \Uparrow [\widehat{\alpha}/\alpha]\, A \dashv \Gamma', \widehat{\alpha}}\ \forall\mathrm{E}\widehat{\alpha}$$

$$\frac{\Gamma, \mathsf{hint}(\Gamma' \vdash A') \vdash e \Downarrow C \dashv \Gamma'}{\Gamma \vdash \textbf{hint}\,(\Gamma' \vdash A')\,\textbf{in}\, e \Downarrow C \dashv \Gamma'}\ \mathsf{hint} \qquad \frac{\Gamma_1 \vdash e \Uparrow \forall\alpha.\, A \dashv \Gamma_2 \quad \begin{array}{l}\Gamma_2 = \Gamma_L, \mathsf{hint}(\Gamma_0 \vdash A_0), \Gamma_R \\ (\Gamma_0 \vdash A_0) \lesssim (\Gamma_L, \Gamma_R \vdash A')\end{array}}{\Gamma_1 \vdash e \Uparrow [A'/\alpha]\, A \dashv \Gamma_L, \Gamma_R}\ \forall\text{E-hint}$$

$$\frac{\Gamma \vdash e \Downarrow \Gamma(\widehat{\alpha}) \dashv \Gamma'}{\Gamma \vdash e \Downarrow \widehat{\alpha} \dashv \Gamma'}\ \mathrm{ExSubst}{\Downarrow} \qquad \frac{\Gamma \vdash e \Uparrow \widehat{\alpha} \dashv \Gamma'}{\Gamma \vdash e \Uparrow \Gamma'(\widehat{\alpha}) \dashv \Gamma'}\ \mathrm{ExSubst}{\Uparrow}$$

$$\frac{\Gamma_1, \widehat{\alpha_1}, \widehat{\alpha_2}, \widehat{\alpha}{=}\widehat{\alpha_1}{\to}\widehat{\alpha_2}, \Gamma_2 \vdash \lambda x.\, e \Downarrow \widehat{\alpha} \dashv \Gamma'}{\Gamma_1, \widehat{\alpha}, \Gamma_2 \vdash \lambda x.\, e \Downarrow \widehat{\alpha} \dashv \Gamma'}\ {\to}\mathrm{I}\widehat{\alpha}$$

$$\frac{\Gamma_1, \widehat{\alpha}, \Gamma_2 \vdash e_1 \Uparrow \widehat{\alpha} \dashv \Gamma_1', \widehat{\alpha}, \Gamma_2' \qquad \Gamma_1', \widehat{\alpha_1}, \widehat{\alpha_2}, \widehat{\alpha}{=}\widehat{\alpha_1}{\to}\widehat{\alpha_2}, \Gamma_2' \vdash e_2 \Downarrow \widehat{\alpha_1} \dashv \Gamma'}{\Gamma_1, \widehat{\alpha}, \Gamma_2 \vdash e_1\, e_2 \Uparrow \widehat{\alpha_2} \dashv \Gamma'}\ {\to}\mathrm{E}\widehat{\alpha}$$

$$\boxed{\Gamma \vdash A \leqq B \dashv \Gamma'}$$

$$\frac{}{\Gamma \vdash \mathbf{1} \leqq \mathbf{1} \dashv \Gamma}\ \mathbf{1}{\leqq} \qquad \frac{\Gamma_1 \vdash B_1 \leqq A_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash A_2 \leqq B_2 \dashv \Gamma_3}{\Gamma_1 \vdash A_1 \to A_2 \leqq B_1 \to B_2 \dashv \Gamma_3}\ {\to}{\leqq} \qquad \frac{}{\Gamma \vdash \alpha \leqq \alpha \dashv \Gamma}\ \alpha\mathrm{Refl}{\leqq}$$

$$\frac{\Gamma, \widehat{\alpha} \vdash [\widehat{\alpha}/\alpha]A \leqq B \dashv \Gamma', \widehat{\alpha}[\ldots], \Gamma_Z}{\Gamma \vdash \forall\alpha.\, A \leqq B \dashv \Gamma'}\ \forall\mathrm{L}\widehat{\alpha}{\leqq} \qquad \frac{\Gamma, \beta \vdash A \leqq B \dashv \Gamma', \beta, \Gamma_Z}{\Gamma \vdash A \leqq \forall\beta.\, B \dashv \Gamma'}\ \forall\mathrm{R}{\leqq}$$

$$\frac{\begin{array}{l}\Gamma_1 = \Gamma_L, \mathsf{hint}(\Gamma_0 \vdash A_0), \Gamma_R \\ (\Gamma_0 \vdash A_0) \lesssim (\Gamma_L, \Gamma_R \vdash A') \qquad \Gamma_L, \Gamma_R \vdash [A'/\alpha]A \leqq B \dashv \Gamma_2\end{array}}{\Gamma_1 \vdash \forall\alpha.\, A \leqq B \dashv \Gamma_2}\ \forall\text{L-hint}{\leqq}$$

$$\frac{}{\Gamma \vdash \widehat{\alpha} \leqq \widehat{\alpha} \dashv \Gamma}\ \widehat{\alpha}\mathrm{Refl}{\leqq} \qquad \frac{\Gamma \vdash \Gamma(\widehat{\alpha}) \leqq B \dashv \Gamma'}{\Gamma \vdash \widehat{\alpha} \leqq B \dashv \Gamma'}\ \mathrm{ExSubstL}{\leqq} \qquad \frac{\Gamma \vdash A \leqq \Gamma(\widehat{\beta}) \dashv \Gamma'}{\Gamma \vdash A \leqq \widehat{\beta} \dashv \Gamma'}\ \mathrm{ExSubstR}{\leqq}$$

$$\frac{\Gamma_1, \widehat{\alpha_1}, \widehat{\alpha_2}, \widehat{\alpha}{=}\widehat{\alpha_1}{\to}\widehat{\alpha_2}, \Gamma_2 \vdash \widehat{\alpha} \leqq B_1 \to B_2 \dashv \Gamma'}{\Gamma_1, \widehat{\alpha}, \Gamma_2 \vdash \widehat{\alpha} \leqq B_1 \to B_2 \dashv \Gamma'}\ {\to}\widehat{\alpha}\mathrm{L}{\leqq} \qquad \frac{\Gamma, \widehat{\beta_1}, \widehat{\beta_2}, \widehat{\beta}{=}\widehat{\beta_1}{\to}\widehat{\beta_2} \vdash A_1 \to A_2 \leqq \widehat{\beta} \dashv \Gamma'}{\Gamma, \widehat{\beta} \vdash A_1 \to A_2 \leqq \widehat{\beta} \dashv \Gamma'}\ {\to}\widehat{\alpha}\mathrm{R}{\leqq}$$

$$\frac{\overbrace{\Gamma_1 \vdash B^{atomic}\ \mathsf{wf}}^{\text{prevents cycles}}}{\Gamma_1, \widehat{\alpha}, \Gamma_2 \vdash \widehat{\alpha} \leqq B^{atomic} \dashv \Gamma_1, \widehat{\alpha}{=}B^{atomic}, \Gamma_2}\ \widehat{\alpha}^{=}\mathrm{L}{\leqq} \qquad \frac{\Gamma_1 \vdash A^{atomic}\ \mathsf{wf}}{\Gamma_1, \widehat{\beta}, \Gamma_2 \vdash A^{atomic} \leqq \widehat{\beta} \dashv \Gamma_1, \widehat{\beta}{=}A^{atomic}, \Gamma_2}\ \widehat{\alpha}^{=}\mathrm{R}{\leqq}$$

Figure 7: Typing and subtyping rules of System Bi$^{\widehat{\alpha}}$

Note that Ω is an input to the theorem. Consider the System Bi$^{\widehat{\alpha}}$ derivation of $\cdot \vdash (\lambda x.\, x : (\forall\alpha.\, (\alpha{\to}\alpha) \to \alpha \to \alpha)\lambda y.\, y \dashv \widehat{\alpha}$. To create the corresponding System Bi derivation, one must create an Ω that instantiates $\widehat{\alpha}$. Fortunately, one can instantiate it to anything, including **1**.

3.6 Completeness of System Bi$^{\widehat{\alpha}}$

We will show that, with respect to System Bi, System Bi$^{\widehat{\alpha}}$ is incomplete for impredicative polymorphism, complete when **hint**s are added to the term, and complete for predicative polymorphism.

3.6.1 Impredicative incompleteness

A small example shows that System Bi$^{\widehat{\alpha}}$ is incomplete for impredicative polymorphism. We abbreviate $\forall\beta.\ \beta \to \beta$ as **ID**. Let $\Gamma = f{:}\forall\alpha.\ \alpha \to \alpha \to \mathbf{1}, x{:}(\mathsf{int}{\to}\mathsf{int}) \to \mathbf{1}, y{:}\mathbf{ID} \to \mathbf{1}$. The derivation in System Bi, shown at the top of Figure 8, has no hint-free analogue in System Bi$^{\widehat{\alpha}}$. Below it, the failed derivation in System Bi$^{\widehat{\alpha}}$ makes the problem clear: the first constraint on $\widehat{\alpha}$ is that it be a supertype of x's type, $(\mathsf{int} \to \mathsf{int}) \to \mathbf{1}$, so that type is used, greedily, as the solution of $\widehat{\alpha}$. (For clarity, we substitute for $\widehat{\alpha}$ in the rest of the derivation.) But the second constraint (shaded) requires that $\mathsf{int} \to \mathsf{int}$ be a subtype of **ID**, which is false. All the choices of rules are fully determined, so no derivation exists.

3.6.2 Hinted completeness

The weakest completeness result says that for every System Bi derivation involving e, there exists a System Bi$^{\widehat{\alpha}}$ derivation involving e^+, where e^+ is e enclosed in **hint** declarations.

Theorem 15. *If* $\Gamma \vdash \mathcal{J}$ *in System Bi and* Γ_H *consists of hints, then* $\Gamma_H, \Gamma'_H, \Gamma \vdash \mathcal{J} \dashv \Gamma_H, \Gamma$ *in System Bi*$^{\widehat{\alpha}}$ *where* Γ'_H *consists of hints.*

Proof. By induction on the given derivation.

We show the $\forall$E case. Let $\Gamma_{HH} = (\Gamma_H, \mathsf{hint}(\Gamma \vdash A'))$. By IH, $\Gamma_{HH}, \Gamma'_H, \Gamma \vdash e \Uparrow \forall\alpha.\, A \dashv \Gamma_{HH}, \Gamma$. By Corollary 8, $(\Gamma \vdash A') \lesssim (\Gamma_H, \Gamma \vdash A')$. Finally, by $\forall$E-hint, $\Gamma_{HH}, \Gamma'_H, \Gamma \vdash e \Uparrow [A'/\alpha]A \dashv \Gamma_H, \Gamma$, which is $\Gamma_H, \mathsf{hint}(\Gamma \vdash A'), \Gamma'_H, \Gamma \vdash e \Uparrow [A'/\alpha]A \dashv \Gamma_H, \Gamma$, which was to be shown. ☐

Corollary 16 (Hinted Completeness)**.** *If* $\cdot \vdash e \Downarrow A$ *in System Bi then* $\cdot \vdash e^+ \Downarrow A \dashv \cdot$ *in System Bi*$^{\widehat{\alpha}}$*, where* $e^+ = \textbf{hint}\,(\Gamma_1 \vdash A_1)\ \textbf{in} \ldots \textbf{hint}\,(\Gamma_n \vdash A_n)\ \textbf{in}\ e$.

Proof. By Theorem 15, $\Gamma_H \vdash e \Downarrow A \dashv \cdot$ where Γ_H consists of n hints. The result follows by applying the hint rule n times. ☐

3.6.3 Predicative completeness

In this section, we show that System Bi$^{\widehat{\alpha}}$ is *predicatively complete*: given a derivation in System Bi in which all polymorphic instances A' used in $\forall$E and $\forall$L$\leqq$ are monomorphic (contain no $\forall$), we can derive the same judgment in System Bi$^{\widehat{\alpha}}$. Consequently, System Bi$^{\widehat{\alpha}}$ is complete for prenex or ML-style polymorphism, in which instances are monomorphic *and* $\forall$s appear only on the outside of types.

We show completeness by building a System Bi$^{\widehat{\alpha}}$ derivation from any System Bi one. Where we have a derivation in System Bi of $\Gamma \vdash \mathcal{J}'$, we create a derivation of $\Gamma'_1 \vdash \mathcal{J} \dashv \Gamma'_2$, where $\mathcal{J}$ is like $\mathcal{J}'$ but may have more existential variables. Specifically, $\mathcal{J}' = [\Omega]\mathcal{J}$ for some Ω representing solutions embedded in the System Bi derivation.

Moreover, Γ'_1 and Γ'_2 must correspond to Ω. The example derivations from Figure 5 give some intuition for this correspondence. Another example appears in the appendix.

When every arrow appearing in Ω has the form $\widehat{\alpha_1} \to \widehat{\alpha_2}$, we say that Ω is *articulated.* System Bi$^{\widehat{\alpha}}$ keeps contexts articulated by restricting $\widehat{\alpha}{=}\mathrm{L}{\leqq}$ and $\widehat{\alpha}{=}\mathrm{R}{\leqq}$, which instantiate existential variables, to non-arrows.

We define the articulation of $\widehat{\alpha}{=}A'$ as follows:

$$
\begin{array}{rcl}
\mathit{Artic}(\widehat{\alpha}{=}\mathbf{1}) & = & \widehat{\alpha}{=}\mathbf{1} \\
\mathit{Artic}(\widehat{\alpha}{=}\beta) & = & \widehat{\alpha}{=}\beta \\
\mathit{Artic}(\widehat{\alpha}{=}B_1{\to}B_2) & = & \widehat{\alpha}{=}\widehat{\beta_1}{\to}\widehat{\beta_2}, \mathit{Artic}(\widehat{\beta_1}{=}B_1), \mathit{Artic}(\widehat{\beta_2}{=}B_2)
\end{array}
$$

Since we require the System Bi$^{\widehat{\alpha}}$ derivation to be predicative, there is no need to define the articulation of $\forall\alpha.\ A$. The proof of the next theorem is in the appendix.

Theorem 17 (Predicative Completeness)**.** *For any* Ω *and* Γ'_1 *and predicative derivation* $\mathcal{D} :: \Gamma \vdash [\Omega]\mathcal{J}$ *in System Bi, provided that*

(1) Ω *is predicative (for any* $\widehat{\alpha}$*, the type* $\Omega(\widehat{\alpha})$ *is monomorphic) and articulated*

(2) Ω *completes* Γ'_1*, and* $[\Omega]\Gamma'_1 = \Gamma$

$$
\begin{array}{lll}
\textit{then}\quad [\Omega]\Gamma'_1 \vdash [\Omega]A' \leq [\Omega]B' & \Longrightarrow & \Gamma'_1 \vdash A' \leqq B' \dashv \Gamma'_2 \\
\phantom{\textit{then}}\quad [\Omega]\Gamma'_1 \vdash e \Downarrow [\Omega]A' & \Longrightarrow & \Gamma'_1 \vdash e \Downarrow A' \dashv \Gamma'_2 \\
\phantom{\textit{then}}\quad [\Omega]\Gamma'_1 \vdash e \Uparrow C & \Longrightarrow & \Gamma'_1 \vdash e \Uparrow C' \dashv \Gamma'_2 \\
 & & \textit{for some } C' \textit{ such that} \\
 & & C = [\Omega]C'
\end{array}
$$

4. System Bi$^{\leq}$ with subtyping, intersection types, and union types

In this section, we extend System Bi with intersection and union types, replacing the weak $\leqq$ relation with a richer subtyping relation $\leq$. The atomic subtyping relation on datatypes is defined using a datasort relation $\delta_1 \preceq \delta_2$. For example, we can define nonempty lists as a subsort of all lists: $\mathsf{nonempty} \preceq \mathsf{list}$. Then $\mathsf{int\ nonempty} \leq \mathsf{int\ list}$. (In this paper, we assume that all datatypes are covariant, so $A\ \delta$ is a subtype of $B\ \delta$ iff A is a subtype of B; extension to co-, contra-, and bivariant type arguments is straightforward.)

A value has intersection type $A \wedge B$ if it has 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); here, their central role is to express combinations of constraints on existential type variables. Where we need both $\widehat{\alpha} \leq B_1$ and $\widehat{\alpha} \leq B_2$, we can replace α with the intersection of new type variables $\alpha_1 \wedge \alpha_2$. Then we must satisfy $\widehat{\alpha_1} \wedge \widehat{\alpha_2} \leq B_1$ and $\widehat{\alpha_1} \wedge \widehat{\alpha_2} \leq B_2$; greedy instantiation yields $\widehat{\alpha_1} = B_1$ and $\widehat{\alpha_2} = B_2$. (In Section 7, we formulate a system that instantiates $\widehat{\alpha}$ to $B_1 \wedge B_2$ automatically.)

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). Here, they serve a symmetric purpose: where we need $A_1 \leq \widehat{\beta}$ and $A_2 \leq \widehat{\beta}$, we can instead replace $\widehat{\beta}$ with $\widehat{\beta_1} \vee \widehat{\beta_2}$, leading to the obligations $A_1 \leq \widehat{\beta_1} \vee \widehat{\beta_2}$ and $A_2 \leq \widehat{\beta_1} \vee \widehat{\beta_2}$, satisfied by instantiating $\widehat{\beta_1}$ to A_1 and $\widehat{\beta_2}$ to A_2.

The typing rules for intersection and union types are the same as in Dunfield and Pfenning (2004). As before, introduction rules check and elimination rules synthesize. The rule $\vee$E reasons by cases: if $e' \Uparrow A \vee B$, then $\mathcal{E}[e'] \Downarrow C$ if, assuming $x{:}A$, the term $\mathcal{E}[x]$—$\mathcal{E}[e']$ with x in place of e—checks against C and also, assuming $y{:}B$, the term $\mathcal{E}[y]$ checks against C. The "tridirectional rule" direct, a unary version of $\vee$E, is needed for technical reasons (Dunfield 2007b, p. 60) not especially pertinent here.

$$\dfrac{\dfrac{f \Uparrow \forall\alpha.\,\alpha \to \alpha \to \mathbf{1}}{f \Uparrow [(\mathbf{ID}\to\mathbf{1})/\alpha](\alpha\to\alpha\to\mathbf{1})}\,\forall\text{E} \quad \dfrac{x \Uparrow \ldots \quad \dfrac{\mathbf{ID} \leqq \mathsf{int}\to\mathsf{int} \quad \mathbf{1} \leqq \mathbf{1}}{(\mathsf{int}\to\mathsf{int}) \to \mathbf{1} \leqq \mathbf{ID}\to\mathbf{1}}\,{\to}{\leqq}}{x \Downarrow \mathbf{ID}\to\mathbf{1}}\,\mathsf{sub}}{f\,x \Uparrow (\mathbf{ID}\to\mathbf{1}) \to \mathbf{1}}\,{\to}\text{E} \quad \dfrac{y \Uparrow \mathbf{ID}\to\mathbf{1} \quad \mathbf{ID}\to\mathbf{1} \leqq \mathbf{ID}\to\mathbf{1}}{y \Downarrow \mathbf{ID}\to\mathbf{1}}\,\mathsf{sub}}{f\,x\,y \Uparrow \mathbf{1}}\,{\to}\text{E}$$

$$\dfrac{\dfrac{\dfrac{f \Uparrow \forall\alpha.\,\alpha\to\alpha\to\mathbf{1}}{f \Uparrow \widehat{\alpha}\to\widehat{\alpha}\to\mathbf{1}}\,\forall\text{E}\widehat{\alpha} \quad \dfrac{x \Uparrow \ldots \quad (\mathsf{int}\to\mathsf{int}) \to \mathbf{1} \leqq \widehat{\alpha}}{x \Downarrow \widehat{\alpha}}\,\mathsf{sub}}{f\,x \Uparrow ((\mathsf{int}\to\mathsf{int})\to\mathbf{1}) \to \mathbf{1}}\,{\to}\text{E} + \text{ExSubst}{\Uparrow} \quad \dfrac{y \Uparrow \mathbf{ID}\to\mathbf{1} \quad \dfrac{\mathsf{int}\to\mathsf{int} \not\leqq \mathbf{ID} \quad \mathbf{1} \leqq \mathbf{1}}{\mathbf{ID}\to\mathbf{1} \not\leqq ((\mathsf{int}\to\mathsf{int})\to\mathbf{1}}\,{\to}{\leqq}}{y \not\Downarrow ((\mathsf{int}\to\mathsf{int})\to\mathbf{1})}\,\mathsf{sub}}{f\,x\,y \not\Uparrow}\,{\to}\text{E}$$

Figure 8: Derivation in System Bi using impredicative polymorphism (above), and a failed derivation in System Bi$^{\widehat{\alpha}}$ (below)

5. System Bi$^{\leq\widehat{\alpha}}$: System Bi$^{\leq}$ with existential contexts

Just as we added existential contexts to System Bi to get System Bi$^{\widehat{\alpha}}$, we add existential contexts to System Bi$^{\leq}$ to obtain System Bi$^{\leq\widehat{\alpha}}$. The rules are given in Figure 11. This step is easy; however, while there are various new rules for intersections and unions, there are no new rules corresponding to $\to$I$\widehat{\alpha}$/$\to$E$\widehat{\alpha}$. The reason is that intersection and union introduction are type-directed, not syntax-directed. We can have $\to$I$\widehat{\alpha}$ concluding $\lambda x.\, e \Downarrow \widehat{\alpha}$ because the λ is a marker saying that $\widehat{\alpha}$ should be an arrow, but our language has no syntax to mark where intersections and unions should be introduced.

5.1 Type safety

To prove type safety, we:

1. Define "System F$^{\leq}$", a type assignment version of System Bi$^{\leq}$.
2. Show that System Bi$^{\leq}$ is sound with respect to System F$^{\leq}$.
3. Prove type safety (and several lemmas) for System F$^{\leq}$ with respect to the operational semantics.

The first task is very easy: remove the rule `anno` and replace all "$\Uparrow$" and "$\Downarrow$" in the typing judgments with ":". For example, $\Gamma \vdash e_1\, e_2 \Uparrow B$ becomes $\Gamma \vdash e_1\, e_2 : B$.

For the second task, soundness, we must show that given a derivation of $\Gamma \vdash e \Downarrow A$ (or of $\Gamma \vdash e \Uparrow A$) in System Bi, we can construct a derivation of $\Gamma \vdash e' : A$ in System F$^{\leq}$, where e' is e with annotations erased. This is an easy proof by induction on the derivation, and I proved it in my dissertation Dunfield (2007b, Ch. 2) for a system similar to System F$^{\leq}$ and System Bi. The only novelty here is parametric polymorphism. But that presents no difficulties—in fact, the cases for $\forall$I and $\forall$E almost exactly follow the cases for ΠI and ΠE (those are the rules for universal index quantification, a feature I omit here to avoid distraction).

The third task is not easy to undertake from scratch, but it *is* an easy extension of the proof in my dissertation (Dunfield 2007b, Ch. 2). Again the reasoning for $\forall$I and $\forall$E follows the reasoning for ΠI and ΠE. In particular, there is no need to extend *derivation rank* and *value definiteness* (Dunfield 2007b, pp. 36–38), nor the other concepts used in the type safety proof; all of those are tied in with so-called *indefinite* types ($\bot$, $\vee$), and $\forall$ is not among those.

5.2 Decidability

Decidability of subtyping in System Bi$^{\leq\widehat{\alpha}}$ can be proved as before in System Bi$^{\widehat{\alpha}}$. Decidability of typing can also be proved as before, with one exception: in `direct`, if e' is itself some variable y, the second premise is not smaller than the conclusion. It should be straightforward to obtain a proof of decidability by designing a "left tridirectional" system following Dunfield and Pfenning (2004) (and proving such a system equivalent to System Bi$^{\leq\widehat{\alpha}}$). Informally, it suffices to observe that if we never apply `direct` when e' is a variable, System Bi$^{\leq\widehat{\alpha}}$ is decidable by the same reasoning as System Bi$^{\widehat{\alpha}}$.

6. Implementation

I implemented a version of System Bi$^{\leq\widehat{\alpha}}$ as an extension of Stardust (Dunfield 2007a), a typechecker for a subset of Standard ML with intersection types, union types, datasort refinements, and index refinements.

The example in Figure 12 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 `(*[ ... ]*)`. `foldr` uses only prenex polymorphism and can of course be written in SML, but `build` uses rank-2 polymorphism. The rest of the example is adapted from Leijen (2009), 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 $\forall$E$\widehat{\alpha}$ or $\forall$E-hint, there is a choice between applying $\forall$E$\widehat{\alpha}$, applying $\forall$E-hint with the first available hint, with the second, etc. However, we can show the complexity is exponential even if $\forall$E$\widehat{\alpha}$ is never used: As formulated, $\forall$E-hint drops a hint after use. First there are n hints and n choices; at the next opportunity to apply $\forall$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.

I have not analyzed the complexity of typechecking, but consider the function **fun** *nonlinear2* $a\ b\ a'\ b'$ = `()` with type annotation $\forall\alpha, \beta.\ \alpha \to \beta \to \alpha \to \beta \to \mathbf{1}$. Given the context *id* : $\forall\delta.\ \delta\to\delta$, *uf* : $\mathbf{1} \to \mathbf{1}$, synthesizing a type for the application *nonlinear2 uf uf id id* involves several nondeterministic choices of when to instantiate each of the *id* types. Still, this can be checked with only a few calls to the function that attempts to 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 *nonlinear2 uf uf id id id*, then by the time we reach the 14-argument function *nonlinear7* it takes 87,000 subtyping calls and 49 seconds to reject the program. This is a contrived example, and I have not yet tried a real example that makes typechecking unacceptably slow.

Note that intersection types make these systems PSPACE-hard (Reynolds 1996), even if parametric polymorphism is never used,

$$
\begin{array}{rlcl}
\text{Type variable sequences} & \vec{\alpha}, \vec{\beta} & ::= & \cdot \mid \alpha \mid (\alpha_1, \dots, \alpha_n) \\
\text{Datatype names} & \delta & & \\
\text{Types} & A, B, C & ::= & A \to B \mid \alpha \mid \forall \alpha.\, A \mid \vec{\alpha}\, \delta \mid A \wedge B \mid A \vee B \\
\text{Constructor types} & A^{con} & ::= & A \to \delta \mid A_1^{con} \wedge A_2^{con}
\end{array}
$$

$\boxed{\Gamma \vdash e \Downarrow A}$ e checks against type A $\qquad$ $\boxed{\Gamma \vdash e \Uparrow A}$ e synthesizes type A

$$
\frac{\Gamma(x) = A}{\Gamma \vdash x \Uparrow A}\,\mathsf{var}
\qquad
\frac{\Gamma \vdash e \Uparrow A \qquad \Gamma \vdash A \le B}{\Gamma \vdash e \Downarrow B}\,\mathsf{sub}
\qquad
\frac{N \lesssim (\Gamma \vdash A) \qquad \Gamma \vdash e \Downarrow A}{\Gamma \vdash (e : N) \Uparrow A}\,\mathsf{anno}
$$

$$
\frac{\Gamma, x{:}A \vdash e \Downarrow B}{\Gamma \vdash \lambda x.\, e \Downarrow A \to B}\,{\to}\mathrm{I}
\qquad
\frac{\Gamma \vdash e_1 \Uparrow A \to B \qquad \Gamma \vdash e_2 \Downarrow A}{\Gamma \vdash e_1\, e_2 \Uparrow B}\,{\to}\mathrm{E}
$$

$$
\frac{\overline{\Gamma} \vdash c : A \to \vec{B}\,\delta \qquad \Gamma \vdash e \Downarrow A}{\Gamma \vdash c(e) \Downarrow \vec{B}\,\delta}\,\delta\mathrm{I}
\qquad
\frac{\Gamma \vdash e \Uparrow \vec{B}\,\delta \qquad \Gamma \vdash ms \Downarrow_{\vec{B}\,\delta} C}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ ms \Downarrow C}\,\delta\mathrm{E}
$$

$$
\frac{\Gamma \vdash v \Downarrow A_1 \qquad \Gamma \vdash v \Downarrow A_2}{\Gamma \vdash v \Downarrow A_1 \wedge A_2}\,\wedge\mathrm{I}
\qquad
\frac{\Gamma \vdash e \Uparrow A_1 \wedge A_2}{\Gamma \vdash e \Uparrow A_1}\,\wedge\mathrm{E}_1
\qquad
\frac{\Gamma \vdash e \Uparrow A_1 \wedge A_2}{\Gamma \vdash e \Uparrow A_2}\,\wedge\mathrm{E}_2
$$

$$
\frac{\Gamma \vdash e \Downarrow A}{\Gamma \vdash e \Downarrow A \vee B}\,\vee\mathrm{I}_1
\qquad
\frac{\Gamma \vdash e \Downarrow B}{\Gamma \vdash e \Downarrow A \vee B}\,\vee\mathrm{I}_2
\qquad
\frac{\Gamma \vdash e' \Uparrow A \vee B \qquad \begin{array}{l}\Gamma, x{:}A \vdash \mathcal{E}[x] \Downarrow C \\ \Gamma, y{:}B \vdash \mathcal{E}[y] \Downarrow C\end{array}}{\Gamma \vdash \mathcal{E}[e'] \Downarrow C}\,\vee\mathrm{E}
$$

$$
\frac{\Gamma \vdash e' \Uparrow A \qquad \Gamma, x{:}A \vdash \mathcal{E}[x] \Downarrow C}{\Gamma \vdash \mathcal{E}[e'] \Downarrow C}\,\mathsf{direct}
\qquad
\frac{\Gamma, \alpha \vdash v \Downarrow A}{\Gamma \vdash v \Downarrow \forall \alpha.\, A}\,\forall\mathrm{I}
\qquad
\frac{\Gamma \vdash e \Uparrow \forall \alpha.\, A \qquad \Gamma \vdash A'\ \mathsf{wf}}{\Gamma \vdash e \Uparrow [A'/\alpha]\, A}\,\forall\mathrm{E}
$$

Figure 9: Typing rules in System $\mathrm{Bi}^{\le}$

$\boxed{\Gamma \vdash A \le B}$ A is a subtype of B

$$
\frac{\Gamma \vdash B_1 \le A_1 \qquad \Gamma \vdash A_2 \le B_2}{\Gamma \vdash A_1 \to A_2 \le B_1 \to B_2}\,{\to}{\le}
\qquad
\frac{\Gamma \vdash A_1 \le B_1 \quad \cdots \quad \Gamma \vdash A_n \le B_n \qquad \delta_1 \preceq \delta_2}{\Gamma \vdash (A_1, \dots, A_n)\, \delta_1 \le (B_1, \dots, B_n)\, \delta_2}\,\delta\alpha
$$

$$
\frac{\Gamma \vdash A \le B_1 \qquad \Gamma \vdash A \le B_2}{\Gamma \vdash A \le B_1 \wedge B_2}\,\wedge\mathrm{R}{\le}
\qquad
\frac{\Gamma \vdash A_1 \le B}{\Gamma \vdash A_1 \wedge A_2 \le B}\,\wedge\mathrm{L}_1{\le}
\qquad
\frac{\Gamma \vdash A_2 \le B}{\Gamma \vdash A_1 \wedge A_2 \le B}\,\wedge\mathrm{L}_2{\le}
$$

$$
\frac{\Gamma \vdash A_1 \le B \qquad \Gamma \vdash A_2 \le B}{\Gamma \vdash A_1 \vee A_2 \le B}\,\vee\mathrm{L}{\le}
\qquad
\frac{\Gamma \vdash A \le B_1}{\Gamma \vdash A \le B_1 \vee B_2}\,\vee\mathrm{R}_1{\le}
\qquad
\frac{\Gamma \vdash A \le B_2}{\Gamma \vdash A \le B_1 \vee B_2}\,\vee\mathrm{R}_2{\le}
$$

$$
\frac{}{\Gamma \vdash \alpha \le \alpha \dashv \Gamma}\,\alpha\mathrm{Refl}
\qquad
\frac{\Gamma \vdash [A'/\alpha]A \le B \qquad \Gamma \vdash A'\ \mathsf{wf}}{\Gamma \vdash \forall \alpha.\, A \le B}\,\forall\mathrm{L}{\le}
\qquad
\frac{\Gamma, \beta \vdash A \le B}{\Gamma \vdash A \le \forall \beta.\, B}\,\forall\mathrm{R}{\le}
$$

Figure 10: Subtyping rules in System $\mathrm{Bi}^{\le}$

and typechecking can be very slow when intersections and unions are used extensively (Dunfield 2007a).

7. A More Automatic System

A shortcoming of System $\mathrm{Bi}^{\le\hat{\alpha}}$ is that the user must often rewrite types, adding unions and intersections, as in our introductory example: *choose* : $\forall \alpha.\, \alpha \to \alpha \to \alpha$ must be changed to

$$\textit{choose} : \forall \alpha_1, \alpha_2.\, \alpha_1 \to \alpha_2 \to \alpha_1 \vee \alpha_2$$

There is some compensation: the rewritten type seems more informative (to the user) than the original. Moreover, the system enjoys a subformula property (Gentzen 1969, p. 87) because it does not fabricate intersections and unions.

But while in first-order functions such as *choose* the user can rewrite the type once and for all, higher-order functions are less cooperative. Say we want to compose $g : (B_1 \to C_1) \wedge (B_2 \to C_2)$ and $f : A \to B_1 \vee B_2$. These types do not fit the usual type of *compose*:

$$\textit{compose} : \forall \alpha, \beta, \gamma.\, (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

In fact, System $\mathrm{Bi}^{\le}$ would instantiate β to B_1 and γ to C_1, not matching the $B_2 \to C_2$ part of g's type, and typechecking would fail.

Now, one can rewrite the type of *compose* to handle the above situation:

$$
\begin{aligned}
\textit{compose} : \forall \alpha, \beta_1, \beta_2, \gamma_1, \gamma_2.&\\
&(\beta_1 \to \gamma_1 \wedge \beta_2 \to \gamma_2)\\
&\to (\alpha \to \beta_1 \vee \beta_2) \to \alpha \to (\gamma_1 \vee \gamma_2)
\end{aligned}
$$

But this is only a (verbose) stopgap: it breaks down for $g' : (B_1 \to C_1) \wedge (B_2 \to C_2) \wedge (B_3 \to C_3)$. Thus, manually rewriting types is not sustainable.

We can address this by automatically conjoining unions and intersections to solutions. This yields System $\mathrm{Bi}^{\le\hat{\alpha}Auto}$ (Figure 13).

$\boxed{\Gamma \vdash e \Downarrow A \dashv \Gamma' \qquad \Gamma \vdash e \Uparrow A \dashv \Gamma'}$ All System Bi$^{\widehat{\alpha}}$ typing rules (Figure 7), with $\le$ instead of $\leqq$ in sub, plus:

$$\frac{\Gamma_1 \vdash c : A \to \vec{B}\,\delta \dashv \Gamma_2 \qquad \Gamma_2 \vdash e \Downarrow A \dashv \Gamma_3}{\Gamma_1 \vdash c(e) \Downarrow \vec{B}\,\delta \dashv \Gamma_3}\,\delta\text{I} \qquad \frac{\Gamma_1 \vdash e \Uparrow \vec{B}\,\delta \dashv \Gamma_2 \qquad \Gamma_2 \vdash ms \Downarrow_{\vec{B}\,\delta} C \dashv \Gamma_3}{\Gamma_1 \vdash \textbf{case}\ e\ \textbf{of}\ ms \Downarrow C \dashv \Gamma_3}\,\delta\text{E}$$

$$\frac{\Gamma_1 \vdash v \Downarrow A_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash v \Downarrow A_2 \dashv \Gamma_3}{\Gamma_1 \vdash v \Downarrow A_1 \wedge A_2 \dashv \Gamma_3}\,\wedge\text{I} \qquad \frac{\Gamma \vdash e \Uparrow A_1 \wedge A_2 \dashv \Gamma'}{\Gamma \vdash e \Uparrow A_1 \dashv \Gamma'}\,\wedge\text{E}_1 \qquad \frac{\Gamma \vdash e \Uparrow A_1 \wedge A_2 \dashv \Gamma'}{\Gamma \vdash e \Uparrow A_2 \dashv \Gamma'}\,\wedge\text{E}_2$$

$$\frac{\Gamma \vdash e \Downarrow A \dashv \Gamma'}{\Gamma \vdash e \Downarrow A \vee B \dashv \Gamma'}\,\vee\text{I}_1 \qquad \frac{\Gamma \vdash e \Downarrow B \dashv \Gamma'}{\Gamma \vdash e \Downarrow A \vee B \dashv \Gamma'}\,\vee\text{I}_2 \qquad \frac{\Gamma_1 \vdash e' \Uparrow A \vee B \dashv \Gamma_2 \qquad \begin{array}{l}\Gamma_2, x{:}A \vdash \mathcal{E}[x] \Downarrow C \dashv \Gamma_3 \\ \Gamma_3, y{:}B \vdash \mathcal{E}[y] \Downarrow C \dashv \Gamma_4\end{array}}{\Gamma_1 \vdash \mathcal{E}[e'] \Downarrow C \dashv \Gamma_4}\,\vee\text{E}$$

$$\frac{\Gamma_1 \vdash e' \Uparrow A \dashv \Gamma_2 \qquad \Gamma_2, x{:}A \vdash \mathcal{E}[x] \Downarrow C \dashv \Gamma_3, x{:}A, \Gamma_Z}{\Gamma_1 \vdash \mathcal{E}[e'] \Downarrow C \dashv \Gamma_3}\,\mathsf{direct}$$

$\boxed{\Gamma \vdash A \le B}$ A is a subtype of B All System Bi$^{\widehat{\alpha}}$ $\leqq$ rules (Figure 7), with $\le$ instead of $\leqq$, plus:

$$\frac{\Gamma_1 \vdash A_1 \le B_1 \dashv \Gamma_2 \quad \cdots \quad \Gamma_n \vdash A_n \le B_n \dashv \Gamma_{n+1} \qquad \delta_1 \preceq \delta_2}{\Gamma_1 \vdash (A_1, \ldots, A_n)\,\delta_1 \le (B_1, \ldots, B_n)\,\delta_2 \dashv \Gamma_{n+1}}\,\delta\alpha$$

$$\frac{\Gamma_1 \vdash A \le B_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash A \le B_2 \dashv \Gamma_3}{\Gamma_1 \vdash A \le B_1 \wedge B_2 \dashv \Gamma_3}\,\wedge\text{R}{\le} \qquad \frac{\Gamma \vdash A_1 \le B \dashv \Gamma'}{\Gamma \vdash A_1 \wedge A_2 \le B \dashv \Gamma'}\,\wedge\text{L}_1{\le} \qquad \frac{\Gamma \vdash A_2 \le B \dashv \Gamma'}{\Gamma \vdash A_1 \wedge A_2 \le B \dashv \Gamma'}\,\wedge\text{L}_2{\le}$$

$$\frac{\Gamma_1 \vdash A_1 \le B \dashv \Gamma_2 \qquad \Gamma_2 \vdash A_2 \le B \dashv \Gamma_3}{\Gamma_1 \vdash A_1 \vee A_2 \le B \dashv \Gamma_3}\,\vee\text{L}{\le} \qquad \frac{\Gamma \vdash A \le B_1 \dashv \Gamma'}{\Gamma \vdash A \le B_1 \vee B_2 \dashv \Gamma'}\,\vee\text{R}_1{\le} \qquad \frac{\Gamma \vdash A \le B_2 \dashv \Gamma'}{\Gamma \vdash A \le B_1 \vee B_2 \dashv \Gamma'}\,\vee\text{R}_2{\le}$$

Figure 11: System Bi$^{\le\widehat{\alpha}}$

```
datatype 'a list = Nil | Cons of 'a * 'a list ;

(*[ val foldr : -all 'a,'b- ('a*'b → 'b)
                   → 'b → 'a list → 'b   ]*)
fun foldr f u xs = case xs of
    Nil ⇒ u  |  Cons(x, xs) ⇒ f (x, foldr f u xs)

(*[ val build : -all 'a- (-all 'b- ('a*'b→'b)→'b→'b)
                  → 'a list                         ]*)
fun build f = f Cons Nil

(*[ val map : -all 'a,'b- ('a→'b)→'a list→'b list ]*)
fun map f xs = build (fn c ⇒ fn n ⇒ foldr
                     (fn (x,ys) ⇒ c (f x, ys)) n xs)


(*[ val id : -all 'a- 'a → 'a  ]*)   fun id x = x
(*[ val inc : int → int ]*)          fun inc x = x + 1

(*[ val poly : (-all 'a- 'a→'a) → int * bool  ]*)
fun poly f = (f 1, f true)

(*[ val single : -all 'a- 'a → 'a list ]*)
fun single x = Cons(x, Nil)

(*[ val append : -all 'a- 'a list→'a list→'a list ]*)
fun append xs ys = ...

val _ = poly id
val _ = poly (fn x ⇒ x)
val ids = single id
val _ = map poly ids
val _ = append (single inc) ids
```

Figure 12: Example of first-class polymorphism

I have not yet proved any soundness or completeness properties for this system.

The basic idea is to *provisionally* set type variables to solutions. Instead of only having unsolved variables $\widehat{\alpha}$ and solved variables $\widehat{\alpha}{=}B$, we have four classes of elements in existential contexts:

- (completely) unsolved variables $\widehat{\alpha}$, as before;
- open intersection solutions $\widehat{\alpha}^{\wedge}{=}(B_1 \wedge \ldots \wedge B_n)$;
- open union solutions $\widehat{\alpha}^{\vee}{=}(B_1 \wedge \ldots \wedge B_n)$;
- closed solutions $\widehat{\alpha}{=}B$, as before.

Traversing derivations in-order, the "transition relation" for the existential context element for some $\widehat{\alpha}$ is no longer simply from $\widehat{\alpha}$ to $\widehat{\alpha}{=}B$ as it was in the earlier systems, but is now

$$\begin{tikzcd}
 & \widehat{\alpha}^{\wedge}{=}\ldots \arrow[loop above, "\widehat{\alpha} \le"] \arrow[dr, bend left, "\le \widehat{\alpha}"] & \\
\widehat{\alpha} \arrow[ur, bend left, "\widehat{\alpha} \le"] \arrow[dr, bend right, "\le \widehat{\alpha}"'] & & \widehat{\alpha}{=}\ldots \\
 & \widehat{\alpha}^{\vee}{=}\ldots \arrow[loop below, "\le \widehat{\alpha}"] \arrow[ur, bend right, "\widehat{\alpha} \le"'] &
\end{tikzcd}$$

where the loops indicate additional types being intersected or unioned with an open solution, and the two rightmost arrows represent what happens when we hit a subtyping obligation that goes against the grain of the previous obligations. For example, $\widehat{\alpha} \le A_1$ followed by $\widehat{\alpha} \le A_2$ results in $\widehat{\alpha}^{\wedge}{=}A_1 \wedge A_2$; these are upper bounds on the instantiation. If we then need to satisfy a lower bound like $B \le \widehat{\alpha}$, we "close" the solution and end up in the rightmost node. This is necessary in the following situation. Suppose we encounter $\widehat{\alpha} \le A_1$, then $\widehat{\alpha} \le A_2$, then $A_1 \wedge A_2 \le \widehat{\alpha}$, and

$$\frac{\Gamma_1 \vdash B\ \mathsf{wf}}{\Gamma_1, \widehat{\alpha}, \Gamma_2 \vdash \widehat{\alpha} \leq B \dashv \Gamma_1, \widehat{\alpha}^{\wedge}{=}B, \Gamma_2}\ \widehat{\alpha}^{=}\mathrm{L}\,\mathit{Auto}$$

$$\frac{\Gamma_1 \vdash A\ \mathsf{wf}}{\Gamma_1, \widehat{\beta}, \Gamma_2 \vdash A \leq \widehat{\beta} \dashv \Gamma_1, \widehat{\beta}^{\vee}{=}A, \Gamma_2}\ \widehat{\alpha}^{=}\mathrm{R}\,\mathit{Auto}$$

$$\frac{\Gamma_1 \vdash B\ \mathsf{wf}}{\Gamma_1, \widehat{\alpha}^{\wedge}{=}A, \Gamma_2 \vdash \widehat{\alpha} \leq B \dashv \Gamma_1, \widehat{\alpha}^{\wedge}{=}A \wedge B, \Gamma_2}\ \widehat{\alpha}^{\wedge}\mathrm{L}\,\mathit{Auto}$$

$$\frac{\Gamma_1 \vdash A\ \mathsf{wf}}{\Gamma_1, \widehat{\beta}^{\vee}{=}A, \Gamma_2 \vdash A \leq \widehat{\beta} \dashv \Gamma_1, \widehat{\beta}^{\vee}{=}A \vee B, \Gamma_2}\ \widehat{\alpha}^{\vee}\mathrm{R}\,\mathit{Auto}$$

$$\frac{\Gamma_1, \widehat{\alpha}{=}A, \Gamma_2 \vdash e \Downarrow A \dashv \Gamma'}{\Gamma_1, \widehat{\alpha}^{\wedge\vee}{=}A, \Gamma_2 \vdash e \Downarrow \widehat{\alpha} \dashv \Gamma'}\ \mathrm{ExSubst}{\Downarrow}\ \mathit{Close}$$

$$\frac{\Gamma_1, \widehat{\alpha}^{\wedge\vee}{=}A, \Gamma_2 \vdash e \Uparrow \widehat{\alpha} \dashv \Gamma'}{\Gamma_1, \widehat{\alpha}{=}A, \Gamma_2 \vdash e \Uparrow A \dashv \Gamma'}\ \mathrm{ExSubst}{\Uparrow}\ \mathit{Close}$$

$$\frac{\Gamma_1, \widehat{\alpha}{=}A, \Gamma_2 \vdash A \leq B \dashv \Gamma'}{\Gamma_1, \widehat{\alpha}^{\wedge\vee}{=}A, \Gamma_2 \vdash \widehat{\alpha} \leq B \dashv \Gamma'}\ \mathrm{ExSubstL}{\leq}\ \mathit{Close}$$

$$\frac{\Gamma_1, \widehat{\beta}{=}B, \Gamma_2 \vdash A \leq B \dashv \Gamma'}{\Gamma_1, \widehat{\beta}^{\wedge\vee}{=}B, \Gamma_2 \vdash A \leq \widehat{\beta} \dashv \Gamma'}\ \mathrm{ExSubstR}{\leq}\ \mathit{Close}$$

Figure 13: Rules of the "more automatic" System Bi$^{\leq\widehat{\alpha}\mathit{Auto}}$

finally $\widehat{\alpha} \leq A_3$. If we left the solution $\widehat{\alpha}^{\wedge}{=}A_1 \wedge A_2$ open, the $A_1 \wedge A_2 \leq \widehat{\alpha}$ obligation could be "satisfied" despite the later constraint $\widehat{\alpha} \leq A_3$. Closing the solution cuts off this source of unsoundness.

The rules ExSubst⇓, ExSubst⇑, ExSubst{L,R}≤ remain, with the understanding that $\Gamma(\widehat{\alpha})$ is defined only for $\Gamma = \Gamma_1, \widehat{\alpha}{=}A$. The new rules ExSubst⇓ *Close*, etc. handle the $\Gamma = \Gamma_1, \widehat{\alpha}^{\wedge}{=}A, \Gamma_2$ and $\Gamma = \Gamma_1, \widehat{\alpha}^{\vee}{=}A, \Gamma_2$ cases.

This mechanism has a slight resemblance to *expansion* in pure type inference in intersection type systems (Ronchi Della Rocca and Venneri (1984); Carlier and Wells (2004), for instance), which creates intersections with fresh type variables, e.g. $\alpha \to \alpha$ becomes $(\alpha_1 \to \alpha_1) \wedge (\alpha_2 \to \alpha_2)$.

8. Related Work

8.1 Systems without subtyping and intersections

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 monotypes). Their system does not support subtyping, except for "at least as polymorphic as" subtyping (which we write as $\leqq$).

ML^{F} (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 the predicative completeness of our system). ML^{F} is more powerful than our systems, in the sense that our bidirectional approach requires annotations on more terms (including all function declarations), but appears substantially more complicated, even in its revised form (Rémy and Yakobowski 2008).

HML (Leijen 2009) extends Damas-Milner and has similar goals to ML^{F}. HML infers *flexible types*, polymorphic types that are bounded below, as $\forall(\beta \geq \forall\alpha.\ \alpha{\to}\alpha).\ \beta \to \beta$. HML requires annotations only on polymorphic arguments, and is a good deal simpler than ML^{F}. It is robust under many simple transformations, such as *revapp* $e_2\ e_1$ in place of $e_1\ e_2$ (where *revapp* has type $\forall\alpha, \beta.\ \alpha \to (\alpha{\to}\beta) \to \beta$). In contrast, System Bi$^{\widehat{\alpha}}$ is sensitive to the ordering of terms when impredicative polymorphism is used; in the failed derivation in Figure 8, switching the arguments x and y would result in success.

8.2 Polymorphic instantiation under subtyping

In systems with subtyping, several approaches to inferring polymorphic instances have been presented. The most important difference from the subtyping systems in this paper is the lack of intersection and union types (Davies' work has intersections, but not unions, and both are essential to System Bi$^{\leq\widehat{\alpha}}$). We discuss some of these approaches here.

- 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 broadly similar 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 $\widehat{\alpha}$-variables, which allows us to fix part of the type expression (the part that is not $\widehat{\alpha}$) while $\widehat{\alpha}$ remains flexible.
- Davies' Refinement ML (Davies 2005), an extension of Standard ML with intersection types (but not union types) and datasort refinements (but not index refinements), has a *refinement restriction*: $A \wedge B$ can be formed only if A and B are refinements of the same simple type. It is thus possible in his setting to do ordinary SML type inference to find simple-type instances of polymorphic variables. There are only finitely many datasort refinements of a given simple type, and therefore finitely many subtypes of it, so the instance that will make typechecking succeed can be found, in theory, by exhaustive search. (In practice, there can be too many refinements for an exhaustive search, so in some cases an explicit annotation is needed.)

  Davies also sketches a proposal for *modes*, in which types can be marked with the desired direction of typechecking, synthesis or checking. This would lead to somewhat verbose types; writing $\overset{\uparrow}{\to}$ for functions in which the argument is to be synthesized, the type of *choose* would be $\forall\alpha.\ (\alpha \overset{\uparrow}{\to} \alpha \to \alpha) \wedge (\alpha \to \alpha \overset{\uparrow}{\to} \alpha)$. In the second part, $\alpha \to \alpha \overset{\uparrow}{\to} \alpha$, we must somehow skip the first argument so we can get to the second, but Davies does not propose creating an existential variable. Instead, it seems necessary to use a two-step process, in which the first step is simple Damas-Milner inference, giving an approximation of the first argument's type.

9. Conclusion

I have presented a new approach to inferring polymorphic instances in bidirectional type systems. The simplest application of this approach is to first-class polymorphism, without subtyping. When intersection and union types are available, the approach can be readily extended to systems with subtyping.

The type systems in this paper might seem odd at first. System Bi$^{\widehat{\alpha}}$, which is not inherently exotic—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 inference tradition. My work here is rooted elsewhere (Dunfield and Pfenning 2004). I would attribute the virtues of my work to the essential simplicity of bidirectional typechecking, plus dumb luck: I had no inkling that intersections and unions (together) would help with the problem of finding polymorphic instances.

The systems in this paper, like those in its immediate ancestors (my dissertation and the works of Xi, Davies, Pfenning), are meant for typechecking, not elaboration/compilation. They do not insert explicit polymorphic abstractions and applications. It seems easy to change System $\mathrm{Bi}^{\hat{\alpha}}$ into an elaboration system, but for System $\mathrm{Bi}^{\leq\hat{\alpha}}$ we would need to elaborate intersections and unions.

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 straightforward.

At first, my goal was simply to add parametric polymorphism to the type systems described in my dissertation. For simplicity, I have omitted from this paper many interesting features of those "full" type systems. I have proved type safety for the full system, but I have not proved most of the other properties, e.g. those in respect of let-normal form (Dunfield 2007b, Ch. 5).

To designers of languages and type systems, consider bidirectional typechecking; as your type system becomes more powerful, you will likely outgrow 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.

References

Andreas Abel. Termination checking with types. *RAIRO—Theoretical Informatics and Applications*, 38(4):277–319, 2004. Special Issue: Fixed Points in Computer Science (FICS'03).

Andreas Abel, Thierry Coquand, and Peter Dybjer. Verifying a semantic $\beta\eta$-conversion test for Martin-Löf type theory. In *Mathematics of Program Construction (MPC'08)*, volume 5133 of *LNCS*, pages 29–56, 2008.

Luca Cardelli. An implementation of $\mathsf{F}_{<:}$. Research report 97, DEC/Compaq Systems Research Center, February 1993.

Sébastien Carlier and J. B. Wells. Expansion: the crucial mechanism for type inference with intersection types: A survey and explanation. In *Workshop on Intersection Types and Related Systems (ITRS '04)*, pages 173–202, 2004.

Adam Chlipala, Leaf Petersen, and Robert Harper. Strict bidirectional type checking. In *Workshop on Types in Language Design and Impl. (TLDI '05)*, pages 71–78, 2005.

Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1–3):167–177, 1996.

Rowan Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, 2005. CMU-CS-05-110.

Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ICFP*, pages 198–208, 2000.

Jana Dunfield. Refined typechecking with Stardust. In *Programming Languages meets Programming Verification (PLPV '07)*, 2007a.

Jana Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007b. CMU-CS-07-129.

Jana Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Found. Software Science and Computation Structures (FOSSACS '03)*, pages 250–266, 2003.

Jana Dunfield and Frank Pfenning. Tridirectional typechecking. In *POPL*, pages 281–292, January 2004.

Gerhard Gentzen. Investigations into logical deduction. In M. Szabo, editor, *Collected papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969.

Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *ACM Conf. Functional Programming and Computer Architecture*, pages 223–232. ACM Press, 1993.

Jean-Yves Girard. The system *F* of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192, 1986.

A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *LISP and Functional Programming*, pages 196–207, 1994.

Didier Le Botlan and Didier Rémy. ML^{F}: raising ML to the power of System F. In *ICFP*, pages 27–38, 2003.

Daan Leijen. Flexible types: robust type inference for first-class polymorphism. In *POPL*, pages 66–77, January 2009.

Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. Unpublished draft, `http://people.cs.uu.nl/andres/LambdaPi/index.html`, 2008.

William Lovas and Frank Pfenning. A bidirectional refinement type system for LF. In *Int'l Workshop on Logical Frameworks and Meta-languages*, Electronic Notes in Theoretical Computer Science, pages 113–128. Elsevier, July 2007.

Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Dept. of Computer Science and Engineering, Chalmers University of Technology, 2007.

Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *POPL*, pages 41–53, 2001.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *J. Functional Programming*, 17(1):1–82, 2007.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL*, pages 371–382, 2008.

Brigitte Pientka and Jana Dunfield. Programming with proofs and explicit contexts. In *Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173, July 2008.

Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Programming Languages and Systems*, 22:1–44, 2000.

Didier Rémy and Boris Yakobowski. From ML to ML^{F}: graphic type constraints with efficient type inference. In *ICFP*, pages 63–74, September 2008.

John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *LNCS*, pages 408–425. Springer, 1974. `http://www.cs.cmu.edu/afs/cs/user/jcr/ftp/theotypestr.pdf`.

John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.

Simona Ronchi Della Rocca and Betti Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28:151–169, 1984.

J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98:111–156, 1999.

Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

Abbrev.:
ICFP = Int'l Conf. Functional Programming;
POPL = Principles of Programming Languages.