Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism

Joshua Dunfield Neelakantan R. Krishnaswami

Max Planck Institute for Software Systems
Kaiserslautern and Saarbrücken, Germany
{joshua,neelk}@mpi-sws.org

Abstract

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

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

Keywords bidirectional typechecking, higher-rank polymorphism

1. Introduction

Bidirectional typechecking (Pierce and Turner 2000) has become one of the most popular techniques for implementing typecheckers in new languages. This technique has been used for dependent types (Coquand 1996; Abel et al. 2008; Löh et al. 2008; Asperti et al. 2012); subtyping (Pierce and Turner 2000); 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; Dunfield 2009); refinement types for LF (Lovas 2010); contextual modal types (Pientka 2008); compiler intermediate representations (Chlipala et al. 2005); and object-oriented languages including C$^\sharp$ (Bierman et al. 2007) and Scala (Odersky et al. 2001). As can be seen, it scales well to advanced type systems; moreover, it is easy to implement, and yields relatively high-quality error messages (Peyton Jones et al. 2007).

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

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

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

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

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

Our algorithm is both simple to understand and simple to implement. The key data structure is an ordered context containing all bindings, including type variables, term variables, and existential variables denoting partial type information. By maintaining order, we are able to easily manage scope information, which is particu-

ICFP '13, September 25–27, 2013, Boston, MA, USA.

ACM 978-1-4503-2326-0/13/09…$15.00.
http://dx.doi.org/10.1145/2500365.2500582

$$\text{Terms} \quad e \quad ::= \quad x \mid () \mid \lambda x.\, e \mid e\; e \mid (e : A)$$

Figure 1. Source expressions

$$\begin{array}{llll}
\text{Types} & A, B, C & ::= & 1 \mid \alpha \mid \forall \alpha.\, A \mid A \to B \\
\text{Monotypes} & \tau, \sigma & ::= & 1 \mid \alpha \mid \tau \to \sigma \\
\text{Contexts} & \Psi & ::= & \cdot \mid \Psi, \alpha \mid \Psi, x : A
\end{array}$$

Figure 2. Syntax of declarative types and contexts

larly important in higher-rank systems, where different quantifiers may be instantiated with different sets of free variables. Furthermore, ordered contexts admit a notion of *extension* or *information increase*, which organizes and simplifies the soundness and completeness proofs of the algorithmic system with respect to the declarative one.

Contributions. We make the following contributions:

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

  As a result, we can explain where type annotations are needed, where they may be deleted, and why important code transformations are sound, all without reference to the implementation.

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

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

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

Lemmas and proofs. Proofs of the main results, as well as statements of all lemmas (and their proofs), can be found in the appendix, available from `www.cs.cmu.edu/~joshuad/bidir/`.

2. Declarative Type System

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

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

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

$\boxed{\Psi \vdash A}$ Under context Ψ, type A is well-formed

$$\frac{\alpha \in \Psi}{\Psi \vdash \alpha}\,\textsf{DeclUvarWF} \qquad \frac{}{\Psi \vdash 1}\,\textsf{DeclUnitWF}$$

$$\frac{\Psi \vdash A \qquad \Psi \vdash B}{\Psi \vdash A \to B}\,\textsf{DeclArrowWF} \qquad \frac{\Psi, \alpha \vdash A}{\Psi \vdash \forall \alpha.\, A}\,\textsf{DeclForallWF}$$

$\boxed{\Psi \vdash A \le B}$ Under context Ψ, type A is a subtype of B

$$\frac{\alpha \in \Psi}{\Psi \vdash \alpha \le \alpha}\,{\le}\textsf{Var} \qquad \frac{}{\Psi \vdash 1 \le 1}\,{\le}\textsf{Unit}$$

$$\frac{\Psi \vdash B_1 \le A_1 \qquad \Psi \vdash A_2 \le B_2}{\Psi \vdash A_1 \to A_2 \le B_1 \to B_2}\,{\le}{\to}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash [\tau/\alpha]A \le B}{\Psi \vdash \forall \alpha.\, A \le B}\,{\le}\forall\textsf{L} \qquad \frac{\Psi, \beta \vdash A \le B}{\Psi \vdash A \le \forall \beta.\, B}\,{\le}\forall\textsf{R}$$

Figure 3. Well-formedness of types and subtyping in the declarative system

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

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

To solve this problem, we turn to *focalization* (Andreoli 1992), the proof-theoretic foundation of bidirectional typechecking. In focused sequent calculi, it is natural to give terms in *spine form* (Cervesato and Pfenning 2003; Simmons 2012), sequences of applications to a head. So we view every application as really being a spine consisting of a series of type applications followed by a term application, and introduce an *application judgment* $\Psi \vdash A \bullet e \Rrightarrow C$, which says that if a term of type A is applied to argument e, the result has type C. Consequently, quantifiers will be instantiated exactly when needed to reveal a function type.

The application judgment lets us suppress explicit type applications, but to get the η law, we need more. Recall the example with $f : 1 \to \forall \alpha.\, \alpha$. In η-reducing $\lambda x.\, f\; x$ to f, we reduce the number of applications in the term. That is, we no longer have a syntactic position at which we can (implicitly) instantiate polymorphic quantifiers. To handle this, we follow Odersky and Läufer (1996) in modeling type instantiation using *subtyping*, where subtyping is defined as a "more-polymorphic-than" relation that guesses type instantiations arbitrarily deeply within types. As a result, $1 \to \forall \alpha.\, \alpha$ is a subtype of $1 \to 1$, and the η law holds.

Happily, subtyping *does* fit naturally into bidirectional systems (Davies and Pfenning 2000; Dunfield 2007; Lovas 2010), so we can give a declarative, bidirectional type system that guesses type instantiations, but is otherwise entirely syntax-directed. In particular, subsumption is confined to a single rule (which switches from checking to synthesis), and our use of an application judgment determines when to instantiate quantifiers. The resulting system is very well-behaved, and ensures that the expected typability results (such as typability being preserved by η-reductions) continue to hold. Furthermore, our declarative formulation makes it clear that

$\boxed{\Psi \vdash e \Leftarrow A}$ Under context Ψ, e checks against input type A

$\boxed{\Psi \vdash e \Rightarrow A}$ Under context Ψ, e synthesizes output type A

$\boxed{\Psi \vdash A \bullet e \Rrightarrow C}$ Under context Ψ, applying a function of type A to e synthesizes type C

$$\dfrac{(x : A) \in \Psi}{\Psi \vdash x \Rightarrow A}\ \textsf{DeclVar} \qquad \dfrac{\Psi \vdash e \Rightarrow A \qquad \Psi \vdash A \le B}{\Psi \vdash e \Leftarrow B}\ \textsf{DeclSub} \qquad \dfrac{\Psi \vdash A \qquad \Psi \vdash e \Leftarrow A}{\Psi \vdash (e : A) \Rightarrow A}\ \textsf{DeclAnno}$$

$$\dfrac{}{\Psi \vdash () \Leftarrow 1}\ \textsf{Decl1I} \qquad \dfrac{}{\Psi \vdash () \Rightarrow 1}\ \textsf{Decl1I}{\Rightarrow} \qquad \dfrac{\Psi, \alpha \vdash e \Leftarrow A}{\Psi \vdash e \Leftarrow \forall \alpha.\, A}\ \textsf{Decl}\forall\textsf{I} \qquad \dfrac{\Psi \vdash \tau \qquad \Psi \vdash [\tau/\alpha]A \bullet e \Rrightarrow C}{\Psi \vdash \forall \alpha.\, A \bullet e \Rrightarrow C}\ \textsf{Decl}\forall\textsf{App}$$

$$\dfrac{\Psi, x : A \vdash e \Leftarrow B}{\Psi \vdash \lambda x.\, e \Leftarrow A \to B}\ \textsf{Decl}{\to}\textsf{I} \qquad \dfrac{\Psi \vdash \sigma \to \tau \qquad \Psi, x : \sigma \vdash e \Leftarrow \tau}{\Psi \vdash \lambda x.\, e \Rightarrow \sigma \to \tau}\ \textsf{Decl}{\to}\textsf{I}{\Rightarrow} \qquad \dfrac{\Psi \vdash e_1 \Rightarrow A \qquad \Psi \vdash A \bullet e_2 \Rrightarrow C}{\Psi \vdash e_1\, e_2 \Rightarrow C}\ \textsf{Decl}{\to}\textsf{E}$$

$$\dfrac{\Psi \vdash e \Leftarrow A}{\Psi \vdash A \to C \bullet e \Rrightarrow C}\ \textsf{Decl}{\to}\textsf{App}$$

Figure 4. Declarative typing

the fundamental algorithmic problem in extending bidirectional typechecking to polymorphism is precisely the problem of figuring out what the missing type applications are.

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

2.1 Typing in Detail

Language overview. In Figure 1, we give the grammar for our language. We have a unit term $()$, variables x, lambda-abstraction $\lambda x.\, e$, application $e_1\, e_2$, and type annotation $(e : A)$. We write A, B, C for types (Figure 2): types are the unit type 1, type variables α, universal quantification $\forall \alpha.\, A$, and functions $A \to B$. Monotypes τ and σ are the same, less the universal quantifier. Contexts Ψ are lists of type variable declarations, and term variables $x : A$, with the expected well-formedness condition. (We give a single-context formulation mixing type and term hypotheses to simplify the presentation.)

Checking, synthesis, and application. Our type system has three main judgments, given in Figure 4. The checking judgment $\Psi \vdash e \Leftarrow A$ asserts that e checks against the type A in the context Ψ. The synthesis judgment $\Psi \vdash e \Rightarrow A$ says that we can synthesize the type A for e in the context Ψ. Finally, an *application judgment* $\Psi \vdash A \bullet e \Rrightarrow C$ says that if a (possibly polymorphic) function of type A is applied to argument e, then the whole application synthesizes C for the whole application.

As is standard in the proof-theoretic presentations of bidirectional typechecking, each of the introduction forms in our calculus has a corresponding checking rule. The Decl1I rule says that $()$ checks against the unit type 1. The Decl→I rule says that $\lambda x.\, e$ checks against the function type $A \to B$ if e checks against B with the additional hypothesis that x has type A. The Decl∀I rule says that e has type $\forall \alpha.\, A$ if e has type A in a context extended with a fresh α.[1] Sums, products and recursive types can be added similarly (we leave them out for simplicity). Rule DeclSub mediates between synthesis and checking: it says that e can be checked against B, if e synthesizes A and A is a subtype of B (that is, A is at least as polymorphic as B).

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

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

In the following example, where we are applying some function polymorphic in α, Decl∀App instantiates the outer quantifier (to the unit type 1; we elide the premise $\Psi \vdash 1$), but leaves the inner quantifier over β alone.

$$\dfrac{\dfrac{\Psi \vdash x \Leftarrow (\forall \beta.\, \beta \to \beta)}{\Psi \vdash (\forall \beta.\, \beta \to \beta) \to 1 \to 1 \bullet x \Rrightarrow 1 \to 1}\ \textsf{Decl}{\to}\textsf{App}}{\Psi \vdash \big(\forall \alpha.\, (\forall \beta.\, \beta \to \beta) \to \alpha \to \alpha\big) \bullet x \Rrightarrow 1 \to 1}\ \textsf{Decl}\forall\textsf{App}$$

In the minimal proof-theoretic formulation of bidirectionality (Davies and Pfenning 2000; Dunfield and Pfenning 2004), introduction forms are checked and elimination forms synthesize, full stop. Even $()$ cannot synthesize its type! Actual bidirectional typecheckers tend to take a more liberal view, adding synthesis rules for at least some introduction forms. To show that our system can accommodate these kinds of extensions, we add the Decl1I⇒ and Decl→I⇒ rules, which synthesize a unit type for $()$ and a monomorphic function type for lambda expressions $\lambda x.\, e$. We examine other variations, including a purist bidirectional no-inference alternative, and a liberal Damas-Milner alternative, in Section 8.

Instantiating types. We express the fact that one type is a polymorphic generalization of another by means of the subtyping judgment given in Figure 3. One important aspect of the judgment is that types are compared relative to a context of free variables. This simplifies our rules, by letting us eliminate the awkward side conditions on sets of free variables that plague many presentations. Most of the subtyping judgment is typical: it proceeds structurally on types, with a contravariant twist for the arrow; all the real ac-

[1] Note that we do not require an explicit type abstraction operation. As a result, an implementation needs to use some technique like scoped type variables (Peyton Jones and Shields 2004) to mention bound type variables in annotations. This point does not matter to the abstract syntax, though.

tion is contained within the two subtyping rules for the universal quantifier.

The left rule, $\leq\forall\mathsf{L}$, says that a type $\forall\alpha. A$ is a subtype of B, if some instance $[\tau/\alpha]A$ is a subtype of B. This is what makes these rules only a declarative specification: $\leq\forall\mathsf{L}$ guesses the instantiation τ "out of thin air", and so the rules do not directly yield an algorithm.

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

The proof-theoretic intuition is simpler. The rules $\leq\forall\mathsf{L}$ and $\leq\forall\mathsf{R}$ are just the left and right rules for universal quantification in the sequent calculus. Type inference is a form of theorem proving, and our subtype relation gives some of the inference rules a theorem prover may use. Following good proof-theoretic hygiene enables us to leave the reflexivity and transitivity rules out of the subtype relation, since they are admissible properties (in sequent calculus terms, they are the identity and cut-admissibility properties). The absence of these rules (particularly, the absence of transitivity), in turn, simplifies a number of proofs. In fact, the rules are practically syntax-directed: the only exception is when both types are quantifiers, and either $\leq\forall\mathsf{L}$ or $\leq\forall\mathsf{R}$ could be tried. Since $\leq\forall\mathsf{R}$ is invertible, however, in practice one can apply it eagerly.

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

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

$$\frac{\Psi \vdash e \Rightarrow A \qquad \Psi, x : A \vdash e' \Leftarrow C}{\Psi \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' \Leftarrow C}$$

Note the absence of generalization in this rule.

2.2 Bidirectional Typing and Type Assignment System F

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

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

$\boxed{\Psi \vdash e : A}$ Under context Ψ, e has type A

$$\frac{(x : A) \in \Psi}{\Psi \vdash x : A}\ \mathsf{AVar} \qquad \frac{}{\Psi \vdash () : 1}\ \mathsf{AUnit}$$

$$\frac{\Psi, x : A \vdash e : B}{\Psi \vdash \lambda x. e : A \to B}\ \mathsf{A{\to}I} \qquad \frac{\Psi \vdash e_1 : A \to B \qquad \Psi \vdash e_2 : A}{\Psi \vdash e_1\ e_2 : B}\ \mathsf{A{\to}E}$$

$$\frac{\Psi, \alpha \vdash e : A}{\Psi \vdash e : \forall\alpha. A}\ \mathsf{A\forall I} \qquad \frac{\Psi \vdash e : \forall\alpha. A \qquad \Psi \vdash \tau}{\Psi \vdash e : [\tau/\alpha]A}\ \mathsf{A\forall E}$$

Figure 5. Type assignment rules for predicative System F

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

Theorem 2 (Soundness of Bidirectional Typing)**.** *If* $\Psi \vdash e \Leftarrow A$ *then there exists* e' *such that* $\Psi \vdash e' : A$ *and* $e' =_{\beta\eta} |e|$*.*

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

2.3 Robustness of Typing

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

Since our declarative system is bidirectional, the basic property is that type annotations are required only at redexes. Additionally, these typing rules can infer (actually, guess) all monomorphic types, so the answer to the question of where annotations are needed is: only on bindings of polymorphic type.[2] Where bidirectional typing really stands out is in its robustness under substitution. We can freely substitute and "unsubstitute" terms:

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

- *If* $\Psi, x : A \vdash e' \Leftarrow C$ *then* $\Psi \vdash [e/x]e' \Leftarrow C$*.*
- *If* $\Psi, x : A \vdash e' \Rightarrow C$ *then* $\Psi \vdash [e/x]e' \Rightarrow C$*.*

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

- *If* $\Psi \vdash [(e : A)/x]e' \Leftarrow C$ *then* $\Psi, x : A \vdash e' \Leftarrow C$*.*
- *If* $\Psi \vdash [(e : A)/x]e' \Rightarrow C$ *then* $\Psi, x : A \vdash e' \Rightarrow C$*.*

Substitution is stated in terms of synthesizing expressions, since any checking term can be turned into a synthesizing term by adding an annotation. Dually, inverse substitution allows extracting any checking term into a let-binding with a type annotation.[3] However, doing so indiscriminately can lead to a term with many redundant annotations, and so we also characterize when annotations can safely be removed:

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

- *If* $\Psi \vdash \big((\lambda x. e) : A\big) \Leftarrow C$ *then* $\Psi \vdash \lambda x. e \Leftarrow C$*.*
- *If* $\Psi \vdash (() : A) \Leftarrow C$ *then* $\Psi \vdash () \Leftarrow C$*.*
- *If* $\Psi \vdash e_1\ (e_2 : A) \Rightarrow C$ *then* $\Psi \vdash e_1\ e_2 \Rightarrow C$*.*
- *If* $\Psi \vdash (x : A) \Rightarrow A$ *then* $\Psi \vdash x \Rightarrow B$ *and* $\Psi \vdash B \leq A$*.*

[2] The number of annotations can be reduced still further; see Section 8 for how to infer the types of all terms typable under Damas-Milner.

[3] The generalization of Theorem 4 to any synthesizing term—not just $(e : A)$—does not hold. For example, given $e = \lambda y. y$ and $e' = x$ and $\Psi \vdash \lambda y. y \Rightarrow 1 \to 1$ and $\Psi \vdash \lambda y. y \Leftarrow C_1 \to C_2$, we cannot derive $\Psi, x : 1 \to 1 \vdash x \Leftarrow C_1 \to C_2$ unless C_1 and C_2 happen to be 1.

$$\begin{array}{llcl}
\text{Types} & A, B, C & ::= & 1 \mid \alpha \mid \hat{\alpha} \mid \forall\alpha.\, A \mid A \to B \\
\text{Monotypes} & \tau, \sigma & ::= & 1 \mid \alpha \mid \hat{\alpha} \mid \tau \to \sigma \\
\text{Contexts} & \Gamma, \Delta, \Theta & ::= & \cdot \mid \Gamma, \alpha \mid \Gamma, x : A \\
 & & & \mid \Gamma, \hat{\alpha} \mid \Gamma, \hat{\alpha} = \tau \mid \Gamma, \blacktriangleright_{\hat{\alpha}} \\
\text{Complete Contexts} & \Omega & ::= & \cdot \mid \Omega, \alpha \mid \Omega, x : A \\
 & & & \mid \Omega, \hat{\alpha} = \tau \mid \Omega, \blacktriangleright_{\hat{\alpha}}
\end{array}$$

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

- *If* $\Psi \vdash \big((e_1\, e_2) : A\big) \Rightarrow A$ *then* $\Psi \vdash e_1\, e_2 \Rightarrow B$ *and* $\Psi \vdash B \leq A$.
- *If* $\Psi \vdash \big((e : B) : A\big) \Rightarrow A$ *then* $\Psi \vdash (e : B) \Rightarrow B$ *and* $\Psi \vdash B \leq A$.
- *If* $\Psi \vdash ((\lambda x.\, e) : \sigma \to \tau) \Rightarrow \sigma \to \tau$ *then* $\Psi \vdash \lambda x.\, e \Rightarrow \sigma \to \tau$.

We can also show that the expected η-laws hold:

Theorem 6 (Soundness of Eta)**.**
If $\Psi \vdash \lambda x.\, e\ x \Leftarrow A$ *and* $x \notin FV(e)$, *then* $\Psi \vdash e \Leftarrow A$.

3. Algorithmic Type System

Our declarative bidirectional system is a fine specification of how typing should behave, but it enjoys guessing entirely too much: the typing rules Decl∀App and Decl→I⇒ could only be implemented with the help of an oracle. The declarative subtyping rule ≤∀L has the same problem.

The first step in building our *algorithmic* bidirectional system will be to modify the three oracular rules so that, instead of guessing a type, they defer the choice by creating an existential type variable, to be solved later. However, our existential variables are not exactly unification variables; they are organized into *ordered algorithmic contexts* (Section 3.1), which define the variables' scope and controls the free variables of their solutions.

The algorithmic type system consists of subtyping rules (Figure 9, discussed in Section 3.2), instantiation rules (Figure 10, discussed in Section 3.3), and typing rules (Figure 11, discussed in Section 3.4). All of the rules manipulate the contexts in a way consistent with *context extension*, a metatheoretic notion described in Section 4; context extension is key in stating and proving decidability, soundness and completeness.

3.1 Algorithmic Contexts

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

Complete contexts Ω are the same as contexts, except that they cannot have unsolved variables.

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

$\boxed{\Gamma \vdash A}$ Under context Γ, type A is well-formed

$$\frac{}{\Gamma[\alpha] \vdash \alpha}\ \textsf{UvarWF} \qquad \frac{}{\Gamma \vdash 1}\ \textsf{UnitWF}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}\ \textsf{ArrowWF} \qquad \frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \forall\alpha.\, A}\ \textsf{ForallWF}$$

$$\frac{}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha}}\ \textsf{EvarWF} \qquad \frac{}{\Gamma[\hat{\alpha} = \tau] \vdash \hat{\alpha}}\ \textsf{SolvedEvarWF}$$

$\boxed{\Gamma\ \mathit{ctx}}$ Algorithmic context Γ is well-formed

$$\frac{}{\cdot\ \mathit{ctx}}\ \textsf{EmptyCtx} \qquad \frac{\Gamma\ \mathit{ctx} \qquad \alpha \notin \mathsf{dom}(\Gamma)}{\Gamma, \alpha\ \mathit{ctx}}\ \textsf{UvarCtx}$$

$$\frac{\Gamma\ \mathit{ctx} \qquad x \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash A}{\Gamma, x : A\ \mathit{ctx}}\ \textsf{VarCtx}$$

$$\frac{\Gamma\ \mathit{ctx} \qquad \hat{\alpha} \notin \mathsf{dom}(\Gamma)}{\Gamma, \hat{\alpha}\ \mathit{ctx}}\ \textsf{EvarCtx}$$

$$\frac{\Gamma\ \mathit{ctx} \qquad \hat{\alpha} \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \tau}{\Gamma, \hat{\alpha} = \tau\ \mathit{ctx}}\ \textsf{SolvedEvarCtx}$$

$$\frac{\Gamma\ \mathit{ctx} \qquad \blacktriangleright_{\hat{\alpha}} \notin \Gamma \qquad \hat{\alpha} \notin \mathsf{dom}(\Gamma)}{\Gamma, \blacktriangleright_{\hat{\alpha}}\ \mathit{ctx}}\ \textsf{MarkerCtx}$$

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

$$\begin{array}{lcl}
[\Gamma]\alpha & = & \alpha \\
[\Gamma]1 & = & 1 \\
\big[\Gamma[\hat{\alpha} = \tau]\big]\hat{\alpha} & = & \big[\Gamma[\hat{\alpha} = \tau]\big]\tau \\
\big[\Gamma[\hat{\alpha}]\big]\hat{\alpha} & = & \hat{\alpha} \\
[\Gamma](A \to B) & = & ([\Gamma]A) \to ([\Gamma]B) \\
[\Gamma](\forall\alpha.\, A) & = & \forall\alpha.\, [\Gamma]A
\end{array}$$

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

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

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

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

$$\Gamma = \Gamma_0[\Theta] \quad \text{means } \Gamma \text{ has the form } (\Gamma_L, \Theta, \Gamma_R)$$

For example, if $\Gamma = \Gamma_0[\hat{\beta}] = (\hat{\alpha}, \hat{\beta}, x : \hat{\beta})$, then $\Gamma_0[\hat{\beta} = \hat{\alpha}] = (\hat{\alpha}, \hat{\beta} = \hat{\alpha}, x : \hat{\beta})$. Since this notation is concise, we use it even in rules that do not replace declarations, such as the rules for type well-formedness in Figure 7.

Occasionally, we also need contexts with *two* ordered holes:

$$\Gamma = \Gamma_0[\Theta_1][\Theta_2] \quad \text{means } \Gamma \text{ has the form } (\Gamma_L, \Theta_1, \Gamma_M, \Theta_2, \Gamma_R)$$

Input and output contexts. Our declarative system used a subtyping judgment and three typing judgments: checking, synthesis, and application. Our algorithmic system includes similar judgment forms, except that we replace the declarative context Ψ with an algorithmic context Γ (the *input context*), and add an *output* context Δ written after a backwards turnstile: $\Gamma \vdash A <: B \dashv \Delta$ for subtyping, $\Gamma \vdash e \Leftarrow A \dashv \Delta$ for checking, and so on. Unsolved existential variables get solved when they are compared against a type. For example, $\hat{\alpha} <: \beta$ would lead to replacing the unsolved declaration $\hat{\alpha}$ with $\hat{\alpha} = \beta$ in the context (provided β is declared to the left of $\hat{\alpha}$). Input contexts thus evolve into output contexts that are "more solved".

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

3.2 Algorithmic Subtyping

The first four subtyping rules in Figure 9 do not directly manipulate the context, but do illustrate how contexts are propagated.

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

Rule $<:\!\rightarrow$ is a bit more interesting: it has two premises, where the first premise has an output context Θ, which is used as the input context to the second (subtyping) premise; the second premise has output context Δ, which is the output of the conclusion.[4] Note that in $<:\!\rightarrow$'s second premise, we do not simply check that $A_2 <: B_2$, but apply the first premise's output Θ to those types:

$$\Theta \vdash [\Theta]A_2 <: [\Theta]B_2 \dashv \Delta$$

This maintains a general invariant: whenever we try to derive $\Gamma \vdash A <: B \dashv \Delta$, the types A and B are already fully applied under Γ. That is, they contain no existential variables already solved in Γ. On balance, this invariant simplifies the system: the extra applications of Θ in $<:\!\rightarrow$ avoid the need for extra rules for replacing solved variables with their solutions.

All the rules discussed so far have been natural extensions of the declarative rules, with $<:$Exvar being a logical way to extend reflexivity to types containing existentials. Rule $<:\forall$L diverges significantly from the corresponding declarative rule $\leq\forall$L. Instead of replacing the type variable α with a guessed τ, rule $<:\forall$L replaces α with a new existential variable $\hat{\alpha}$, which it adds to the premise's input context: $\Gamma, \blacktriangleright_{\hat{\alpha}}, \hat{\alpha} \vdash [\hat{\alpha}/\alpha]A <: B \dashv \Delta, \blacktriangleright_{\hat{\alpha}}, \Theta$. The peculiar-looking $\blacktriangleright_{\hat{\alpha}}$ is a *scope marker*, pronounced "marker $\hat{\alpha}$", which will delineate existentials created by *articulation* (the step of solving $\hat{\alpha}$ to $\hat{\alpha}_1 \rightarrow \hat{\alpha}_2$, discussed in the next subsection). The output context $(\Delta, \blacktriangleright_{\hat{\alpha}}, \Theta)$ allows for some additional (existential) variables to appear after $\blacktriangleright_{\hat{\alpha}}$, in a trailing context Θ. These existential variables could mention $\hat{\alpha}$, or (if they appear between $\blacktriangleright_{\hat{\alpha}}$ and $\hat{\alpha}$) could be mentioned *by* $\hat{\alpha}$; since $\hat{\alpha}$ goes out of scope in the conclusion, we drop such "trailing existentials" from the concluding output context, which is simply Δ.[5]

Rule $<:\forall$R is fairly close to the declarative version, but for scoping reasons similar to $<:\forall$L, it also drops Θ, the part of the context to the right of the universal type variable α. (Articulation makes no sense for universal variables, so α can act as its own marker.)

The last two rules are essential: they derive subtypings with an unsolved existential on one side, and an arbitrary type on the other. Rule $<:$InstantiateL derives $\hat{\alpha} <: A$, and $<:$InstantiateR derives $A <: \hat{\alpha}$. These rules do not directly change the output context; they just do an "occurs check" $\hat{\alpha} \notin FV(A)$ to avoid circularity, and leave all the real work to the instantiation judgment.

3.3 Instantiation

Two almost-symmetric judgments instantiate unsolved existential variables: $\Gamma \vdash \hat{\alpha} :\leqq A \dashv \Delta$ and $\Gamma \vdash A \leqq: \hat{\alpha} \dashv \Delta$. The symbol $:\leqq$ suggests assignment of the variable to its left, but also subtyping: the subtyping rule $<:$InstantiateL moves from instantiation $\hat{\alpha} :\leqq A$, read "instantiate $\hat{\alpha}$ to a subtype of A", to subtyping $\hat{\alpha} <: A$. The symmetric judgment $A \leqq: \hat{\alpha}$ can be read "instantiate $\hat{\alpha}$ to a supertype of A".

The first instantiation rule in Figure 10, InstLSolve, sets $\hat{\alpha}$ to τ in the output context: its conclusion is $\Gamma, \hat{\alpha}, \Gamma' \vdash \hat{\alpha} :\leqq \tau \dashv \Gamma, \hat{\alpha} = \tau, \Gamma'$. The premise $\Gamma \vdash \tau$ checks that the monotype τ is well-formed under the prefix context Γ. To check the soundness of this rule, we can take the conclusion $\hat{\alpha} :\leqq \tau$, substitute our new solution for $\hat{\alpha}$, and check that the resulting subtyping makes sense. Since $[\Gamma, \hat{\alpha} = \tau, \Gamma']\hat{\alpha} = \tau$, we ask whether $\tau <: \tau$ makes sense, and of course it does through reflexivity.

Rule InstLArr can be applied when the type A in $\hat{\alpha} :\leqq A$ has the form $A_1 \rightarrow A_2$. It follows that $\hat{\alpha}$'s solution must have the form $\cdots \rightarrow \cdots$, so we "articulate" $\hat{\alpha}$, giving it the solution $\hat{\alpha}_1 \rightarrow \hat{\alpha}_2$ where the $\hat{\alpha}_k$ are fresh existentials. We insert their declarations just before $\hat{\alpha}$—they must be to the left of $\hat{\alpha}$ so they can be mentioned in its solution, but they must be close enough to $\hat{\alpha}$ that they appear to the *right* of the marker $\blacktriangleright_{\hat{\alpha}}$ introduced by $<:\forall$L. Note that the first premise $A_1 \leqq: \hat{\alpha}_1$ switches to the other instantiation judgment. Also, the second premise $\Theta \vdash \hat{\alpha}_2 :\leqq [\Theta]A_2 \dashv \Delta$ applies Θ to A_2, to apply any solutions found in the first premise.

The other rules are somewhat subtle. Rule InstLReach derives

$$\Gamma[\hat{\alpha}][\hat{\beta}] \vdash \hat{\alpha} :\leqq \hat{\beta} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]$$

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

Rule InstLAllR is the instantiation version of $<:\forall$R. Since our polymorphism is predicative, we can't assign $\forall\beta.\, B$ to $\hat{\alpha}$, but we can decompose the quantifier in the same way that subtyping does.

The rules for the second judgment $A \leqq: \hat{\alpha}$ are similar: InstRSolve, InstRReach and InstRArr are direct analogues of the first three $\hat{\alpha} :\leqq A$ rules, and InstRAllL is the instantiation version of $<:\forall$L.

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

[4] Rule $<:\!\rightarrow$ enforces that the function domains B_1, A_1 are compared first: Θ is an input to the second premise. But this is an arbitrary choice; the system would behave the same if we chose to check the codomains first.

[5] In our setting, it is safe to drop trailing existentials that are unsolved: such variables are unconstrained, and we can treat them as having been instantiated to any well-formed type, such as 1. In a dependently typed setting, we would need to check that at least one solution exists.

$\boxed{\Gamma \vdash A <: B \dashv \Delta}$ Under input context Γ, type A is a subtype of B, with output context Δ

$$\frac{}{\Gamma[\alpha] \vdash \alpha <: \alpha \dashv \Gamma[\alpha]}\;{<:}\mathsf{Var} \qquad \frac{}{\Gamma \vdash 1 <: 1 \dashv \Gamma}\;{<:}\mathsf{Unit} \qquad \frac{}{\Gamma[\hat\alpha] \vdash \hat\alpha <: \hat\alpha \dashv \Gamma[\hat\alpha]}\;{<:}\mathsf{Exvar}$$

$$\frac{\Gamma \vdash B_1 <: A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 <: [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \to A_2 <: B_1 \to B_2 \dashv \Delta}\;{<:}{\to}$$

$$\frac{\Gamma, \blacktriangleright_{\hat\alpha}, \hat\alpha \vdash [\hat\alpha/\alpha]A <: B \dashv \Delta, \blacktriangleright_{\hat\alpha}, \Theta}{\Gamma \vdash \forall\alpha.\, A <: B \dashv \Delta}\;{<:}\forall\mathsf{L} \qquad \frac{\Gamma, \alpha \vdash A <: B \dashv \Delta, \alpha, \Theta}{\Gamma \vdash A <: \forall\alpha.\, B \dashv \Delta}\;{<:}\forall\mathsf{R}$$

$$\frac{\hat\alpha \notin FV(A) \qquad \Gamma[\hat\alpha] \vdash \hat\alpha :\leqq A \dashv \Delta}{\Gamma[\hat\alpha] \vdash \hat\alpha <: A \dashv \Delta}\;{<:}\mathsf{InstantiateL} \qquad \frac{\hat\alpha \notin FV(A) \qquad \Gamma[\hat\alpha] \vdash A \leqq: \hat\alpha \dashv \Delta}{\Gamma[\hat\alpha] \vdash A <: \hat\alpha \dashv \Delta}\;{<:}\mathsf{InstantiateR}$$

Figure 9. Algorithmic subtyping

$\boxed{\Gamma \vdash \hat\alpha :\leqq A \dashv \Delta}$ Under input context Γ, instantiate $\hat\alpha$ such that $\hat\alpha <: A$, with output context Δ

$$\frac{\Gamma \vdash \tau}{\Gamma, \hat\alpha, \Gamma' \vdash \hat\alpha :\leqq \tau \dashv \Gamma, \hat\alpha = \tau, \Gamma'}\;\mathsf{InstLSolve} \qquad \frac{}{\Gamma[\hat\alpha][\hat\beta] \vdash \hat\alpha :\leqq \hat\beta \dashv \Gamma[\hat\alpha][\hat\beta = \hat\alpha]}\;\mathsf{InstLReach}$$

$$\frac{\Gamma[\hat\alpha_2, \hat\alpha_1, \hat\alpha = \hat\alpha_1 \to \hat\alpha_2] \vdash A_1 \leqq: \hat\alpha_1 \dashv \Theta \qquad \Theta \vdash \hat\alpha_2 :\leqq [\Theta]A_2 \dashv \Delta}{\Gamma[\hat\alpha] \vdash \hat\alpha :\leqq A_1 \to A_2 \dashv \Delta}\;\mathsf{InstLArr} \qquad \frac{\Gamma[\hat\alpha], \beta \vdash \hat\alpha :\leqq B \dashv \Delta, \beta, \Delta'}{\Gamma[\hat\alpha] \vdash \hat\alpha :\leqq \forall\beta.\, B \dashv \Delta}\;\mathsf{InstLAllR}$$

$\boxed{\Gamma \vdash A \leqq: \hat\alpha \dashv \Delta}$ Under input context Γ, instantiate $\hat\alpha$ such that $A <: \hat\alpha$, with output context Δ

$$\frac{\Gamma \vdash \tau}{\Gamma, \hat\alpha, \Gamma' \vdash \tau \leqq: \hat\alpha \dashv \Gamma, \hat\alpha = \tau, \Gamma'}\;\mathsf{InstRSolve} \qquad \frac{}{\Gamma[\hat\alpha][\hat\beta] \vdash \hat\beta \leqq: \hat\alpha \dashv \Gamma[\hat\alpha][\hat\beta = \hat\alpha]}\;\mathsf{InstRReach}$$

$$\frac{\Gamma[\hat\alpha_2, \hat\alpha_1, \hat\alpha = \hat\alpha_1 \to \hat\alpha_2] \vdash \hat\alpha_1 :\leqq A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \leqq: \hat\alpha_2 \dashv \Delta}{\Gamma[\hat\alpha] \vdash A_1 \to A_2 \leqq: \hat\alpha \dashv \Delta}\;\mathsf{InstRArr} \qquad \frac{\Gamma[\hat\alpha], \blacktriangleright_{\hat\beta}, \hat\beta \vdash [\hat\beta/\beta]B \leqq: \hat\alpha \dashv \Delta, \blacktriangleright_{\hat\beta}, \Delta'}{\Gamma[\hat\alpha] \vdash \forall\beta.\, B \leqq: \hat\alpha \dashv \Delta}\;\mathsf{InstRAllL}$$

Figure 10. Instantiation

$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$ Under input context Γ, e checks against input type A, with output context Δ

$\boxed{\Gamma \vdash e \Rightarrow A \dashv \Delta}$ Under input context Γ, e synthesizes output type A, with output context Δ

$\boxed{\Gamma \vdash A \bullet e \Rrightarrow C \dashv \Delta}$ Under input context Γ, applying a function of type A to e synthesizes type C, with output context Δ

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \dashv \Gamma}\;\mathsf{Var} \qquad \frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A <: [\Theta]B \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}\;\mathsf{Sub} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash (e : A) \Rightarrow A \dashv \Delta}\;\mathsf{Anno}$$

$$\frac{}{\Gamma \vdash () \Leftarrow 1 \dashv \Gamma}\;1\mathsf{I} \qquad \frac{}{\Gamma \vdash () \Rightarrow 1 \dashv \Gamma}\;1\mathsf{I}{\Rightarrow} \qquad \frac{\Gamma, \alpha \vdash e \Leftarrow A \dashv \Delta, \alpha, \Theta}{\Gamma \vdash e \Leftarrow \forall\alpha.\, A \dashv \Delta}\;\forall\mathsf{I} \qquad \frac{\Gamma, \hat\alpha \vdash [\hat\alpha/\alpha]A \bullet e \Rrightarrow C \dashv \Delta}{\Gamma \vdash \forall\alpha.\, A \bullet e \Rrightarrow C \dashv \Delta}\;\forall\mathsf{App}$$

$$\frac{\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x.\, e \Leftarrow A \to B \dashv \Delta}\;{\to}\mathsf{I} \qquad \frac{\Gamma, \hat\alpha, \hat\beta, x : \hat\alpha \vdash e \Leftarrow \hat\beta \dashv \Delta, x : \hat\alpha, \Theta}{\Gamma \vdash \lambda x.\, e \Rightarrow \hat\alpha \to \hat\beta \dashv \Delta}\;{\to}\mathsf{I}{\Rightarrow} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A \bullet e_2 \Rrightarrow C \dashv \Delta}{\Gamma \vdash e_1\, e_2 \Rightarrow C \dashv \Delta}\;{\to}\mathsf{E}$$

$$\frac{\Gamma[\hat\alpha_2, \hat\alpha_1, \hat\alpha = \hat\alpha_1 \to \hat\alpha_2] \vdash e \Leftarrow \hat\alpha_1 \dashv \Delta}{\Gamma[\hat\alpha] \vdash \hat\alpha \bullet e \Rrightarrow \hat\alpha_2 \dashv \Delta}\;\hat\alpha\mathsf{App} \qquad \frac{\Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash A \to C \bullet e \Rrightarrow C \dashv \Delta}\;{\to}\mathsf{App}$$

Figure 11. Algorithmic typing

$$\dfrac{\dfrac{}{\Gamma[\hat\beta], \blacktriangleright_{\hat\alpha}, \hat\alpha \vdash \hat\alpha \leqq: \hat\beta \dashv \Gamma[\hat\beta], \blacktriangleright_{\hat\alpha}, \hat\alpha = \hat\beta}\,\textsf{InstRReach}}{\Gamma[\hat\beta] \vdash \forall\alpha.\,\alpha \leqq: \hat\beta \dashv \Gamma[\hat\beta]}\,\textsf{InstRAllL}$$

Here, we introduce a new variable $\hat\alpha$ to go under the universal quantifier; then, instantiation applies InstRReach to set $\hat\alpha$, not $\hat\beta$. Hence, $\hat\beta$ is, correctly, *not constrained* by this subtyping problem.

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

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

3.4 Algorithmic Typing

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

Unit and $\forall$. In the second row of typing rules, 1I and 1I⇒ generate no new information and simply propagate the input context.

∀I is more interesting: Like the declarative rule Decl∀I, it adds a universal type variable α to the (input) context. The output context of the premise $\Gamma, \alpha \vdash e \Leftarrow A \dashv \Delta, \alpha, \Theta$ allows for some additional (existential) variables to appear after α, in a trailing context Θ. These existential variables could depend on α; since α goes out of scope in the conclusion, we must drop them from the concluding output context, which is just Δ: the part of the premise's output context that cannot depend on α.

The application-judgment rule ∀App serves a similar purpose to the subtyping rule <:∀L, but does *not* place a marker before $\hat\alpha$: the variable $\hat\alpha$ may appear in the output type C, so $\hat\alpha$ must survive in the output context Δ.

Functions. In the third row of typing rules, rule →I follows the same scheme: the declarations Θ following $x : A$ are dropped in the conclusion's output context.

Rule →I⇒ corresponds to Decl→I⇒, one of the guessing rules, so we create new existential variables $\hat\alpha$ (for the function domain) and $\hat\beta$ (for the codomain) and check the function body against $\hat\beta$. As in ∀App, we do not place a marker before $\hat\alpha$, because $\hat\alpha$ and $\hat\beta$ appear in the output type $(\lambda x.\, e \Rightarrow \hat\alpha \to \hat\beta)$.

Rule →E is the expected analogue of Decl→E; like other rules with two premises, it applies the intermediate context Θ.

On the last row of typing rules, $\hat\alpha$App derives $\hat\alpha \bullet e \Rightarrow\!\!\Rightarrow \hat\alpha_2$ where $\hat\alpha$ is unsolved in the input context. Here we have an application judgment, which is supposed to synthesize a type for an application $e_1\ e$ where e_1 has type $\hat\alpha$. We know that e_1 should have function type; similarly to InstLArr/InstRArr, we introduce $\hat\alpha_1$ and $\hat\alpha_2$ and add $\hat\alpha = \hat\alpha_1{\to}\hat\alpha_2$ to the context. (Rule $\hat\alpha$App is the only algorithmic typing rule that does not correspond to a declarative rule.)

Finally, rule →App is analogous to Decl→App.

4. Context Extension

We motivated the algorithmic rules by saying that they evolved input contexts to output contexts that were "more solved". To state and prove the metatheoretic results of decidability, soundness and completeness (Sections 5–7), we introduce a context extension judgment $\Gamma \longrightarrow \Delta$. This judgment captures a notion of information increase from an input context Γ to an output context Δ, and relates algorithmic contexts Γ and Δ to completely solved extensions Ω, which correspond—via the context application described in Section 4.1—to declarative contexts Ψ.

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

The rules deriving the context extension judgment (Figure 13) say that the empty context extends the empty context (⟶ID); a term variable typing $x : A'$ extends $x : A$ if applying the extending context Δ to A and A' yields the same type (⟶Var); universal type variables must match (⟶Uvar); scope markers must match (⟶Marker); and, existential variables may:

- appear unsolved in both contexts (⟶Unsolved),
- appear solved in both contexts, if applying the extending context Δ makes the solutions τ and τ' equal (⟶Solved),
- get solved by the extending context (⟶Solve),
- be added by the extending context, either without a solution (⟶Add) or with a solution (⟶AddSolved);

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

$$(\hat\alpha, \hat\beta = \hat\alpha) \longrightarrow (\hat\alpha = 1, \hat\beta = \hat\alpha)$$

directly increases information about $\hat\alpha$, and indirectly increases information about $\hat\beta$. Perhaps more interestingly, the extension

$$\underbrace{(\hat\alpha = 1, \hat\beta = \hat\alpha)}_{\Delta} \longrightarrow \underbrace{(\hat\alpha = 1, \hat\beta = 1)}_{\Omega}$$

also holds: while the solution of $\hat\beta$ in Ω is different, in the sense that Ω contains $\hat\beta = 1$ while Δ contains $\hat\beta = \hat\alpha$, applying Ω to the two solutions gives the same thing: applying Ω to Δ's solution of $\hat\beta$ gives $[\Omega]\hat\alpha = [\Omega]1 = 1$, while applying Ω to Ω's own solution for $\hat\beta$ also gives 1, because $[\Omega]1 = 1$.

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

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

4.1 Context Application

A complete context Ω (Figure 6) has no unsolved variables, so applying it to a (well-formed) type yields a type $[\Omega]A$ with no existen-

$$\Delta = \Gamma[\hat\beta_2, \hat\beta_1{=}\hat\beta_2, \hat\alpha{=}\hat\beta_1{\to}\hat\beta_2]$$

$$\dfrac{\dfrac{\dfrac{\overbrace{\Gamma', \blacktriangleright_{\hat\beta}}^{\text{context to the left of } \hat\beta} \vdash \hat\beta_1}{\Gamma', \blacktriangleright_{\hat\beta}, \hat\beta \vdash \hat\beta_2 \mathrel{:\leqq} \hat\beta \dashv \Gamma', \blacktriangleright_{\hat\beta}, \hat\beta{=}\hat\beta_1}\;\mathsf{InstLReach} \qquad \dfrac{\overbrace{\ldots, \hat\beta_2}^{\text{context to the left of } \hat\beta_1} \vdash \hat\beta_2}{\Gamma', \blacktriangleright_{\hat\beta}, \hat\beta{=}\hat\beta_1 \vdash \hat\beta_2 \mathrel{\leqq:} \hat\beta_1 \dashv \Delta, \blacktriangleright_{\hat\beta}, \hat\beta{=}\hat\beta_1}\;\mathsf{InstRSolve}}{\Gamma[\hat\alpha], \blacktriangleright_{\hat\beta}, \hat\beta \vdash \hat\beta{\to}\hat\beta \mathrel{\leqq:} \hat\alpha \dashv \Delta, \blacktriangleright_{\hat\beta}, \hat\beta{=}\hat\beta_1}\;\mathsf{InstRArr}}{\Gamma[\hat\alpha] \vdash (\forall\beta.\, \beta{\to}\beta) \mathrel{\leqq:} \hat\alpha \dashv \Delta}\;\mathsf{InstRAllL}$$

$$\Gamma' = \Gamma[\hat\beta_2, \hat\beta_1, \hat\alpha{=}\hat\beta_1{\to}\hat\beta_2]$$

Figure 12. Example of instantiation

$\boxed{\Gamma \longrightarrow \Delta}$ Γ is extended by Δ

$$\dfrac{}{\cdot \longrightarrow \cdot}\;{\longrightarrow}\mathsf{ID} \qquad \dfrac{\Gamma \longrightarrow \Delta \qquad [\Delta]A = [\Delta]A'}{\Gamma, x : A \longrightarrow \Delta, x : A'}\;{\longrightarrow}\mathsf{Var} \qquad \dfrac{\Gamma \longrightarrow \Delta}{\Gamma, \alpha \longrightarrow \Delta, \alpha}\;{\longrightarrow}\mathsf{Uvar}$$

$$\dfrac{\Gamma \longrightarrow \Delta}{\Gamma, \hat\alpha \longrightarrow \Delta, \hat\alpha}\;{\longrightarrow}\mathsf{Unsolved} \qquad \dfrac{\Gamma \longrightarrow \Delta \qquad [\Delta]\tau = [\Delta]\tau'}{\Gamma, \hat\alpha = \tau \longrightarrow \Delta, \hat\alpha = \tau'}\;{\longrightarrow}\mathsf{Solved} \qquad \dfrac{\Gamma \longrightarrow \Delta}{\Gamma, \hat\alpha \longrightarrow \Delta, \hat\alpha = \tau}\;{\longrightarrow}\mathsf{Solve} \qquad \dfrac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \hat\alpha}\;{\longrightarrow}\mathsf{Add}$$

$$\dfrac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \hat\alpha = \tau}\;{\longrightarrow}\mathsf{AddSolved} \qquad \dfrac{\Gamma \longrightarrow \Delta}{\Gamma, \blacktriangleright_{\hat\alpha} \longrightarrow \Delta, \blacktriangleright_{\hat\alpha}}\;{\longrightarrow}\mathsf{Marker}$$

Figure 13. Context extension

$$\begin{array}{lcll}
[\cdot]\cdot & = & \cdot & \\
[\Omega, x : A](\Gamma, x : A_\Gamma) & = & [\Omega]\Gamma,\ x : [\Omega]A & \text{if } [\Omega]A = [\Omega]A_\Gamma \\
[\Omega, \alpha](\Gamma, \alpha) & = & [\Omega]\Gamma,\ \alpha & \\
[\Omega, \hat\alpha = \tau](\Gamma, \hat\alpha) & = & [\Omega]\Gamma & \\
[\Omega, \hat\alpha = \tau](\Gamma, \hat\alpha = \tau_\Gamma) & = & [\Omega]\Gamma & \text{if } [\Omega]\tau = [\Omega]\tau_\Gamma \\
[\Omega, \hat\alpha = \tau]\Gamma & = & [\Omega]\Gamma & \text{if } \hat\alpha \notin \mathsf{dom}(\Gamma) \\
[\Omega, \blacktriangleright_{\hat\alpha}](\Gamma, \blacktriangleright_{\hat\alpha}) & = & [\Omega]\Gamma &
\end{array}$$

Figure 14. Applying a complete context Ω to a context

tials. Such a type is well-formed under a *declarative* context—with just α and $x : A$ declarations—obtained by dropping all the existential declarations and applying Ω to declarations $x : A$ (to yield $x : [\Omega]A$). We can think of this context as the result of applying Ω to itself: $[\Omega]\Omega$.

More generally, we can apply Ω to any context Γ that it extends. This operation of context application $[\Omega]\Gamma$ is given in Figure 14. The application $[\Omega]\Gamma$ is defined if and only if $\Gamma \longrightarrow \Omega$, and applying Ω to any such Γ yields the same declarative context $[\Omega]\Omega$:

Lemma (Stability of Complete Contexts)**.** *If* $\Gamma \longrightarrow \Omega$ *then* $[\Omega]\Gamma = [\Omega]\Omega$.

5. Decidability

Our algorithmic type system is decidable. Since the typing rules (Figure 11) depend on the subtyping rules (Figure 9), which in turn depend on the instantiation rules (Figure 10), showing that the typing judgments (checking, synthesis and application) are decidable requires that we show that the instantiation and subtyping judgments are decidable.

5.1 Decidability of Instantiation

As discussed in Section 3.3, deriving $\Gamma \vdash \hat\alpha \mathrel{:\leqq} A \dashv \Delta$ does not necessarily instantiate any existential variable (unless A is a monotype). However, the instantiation rules do preserve the size of (substituted) types:

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

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

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

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

(1) Either there exists Δ *such that* $\Gamma_0[\hat\alpha] \vdash \hat\alpha \mathrel{:\leqq} A \dashv \Delta$, *or not.*

(2) Either there exists Δ *such that* $\Gamma_0[\hat\alpha] \vdash A \mathrel{\leqq:} \hat\alpha \dashv \Delta$, *or not.*

5.2 Decidability of Algorithmic Subtyping

To prove decidability of the subtyping system in Figure 9, measure judgments $\Gamma \vdash A <: B \dashv \Delta$ lexicographically by

(S1) the number of ∀ quantifiers in A and B;

(S2) $|\mathsf{unsolved}(\Gamma)|$, the number of unsolved existentials in Γ;

(S3) $|\Gamma \vdash A| + |\Gamma \vdash B|$.

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

Definition (Contextual Size)**.**

$$\begin{array}{lcll}
|\Gamma \vdash \alpha| & = & 1 & \\
|\Gamma[\hat\alpha] \vdash \hat\alpha| & = & 1 & \\
|\Gamma[\hat\alpha = \tau] \vdash \hat\alpha| & = & 1 + |\Gamma[\hat\alpha = \tau] \vdash \tau| & (*) \\
|\Gamma \vdash \forall\alpha.\, A| & = & 1 + |\Gamma, \alpha \vdash A| & \\
|\Gamma \vdash A \to B| & = & 1 + |\Gamma \vdash A| + |\Gamma \vdash B| &
\end{array}$$

For example, if $\Gamma = (\beta, \hat\alpha = \beta)$ then $|\Gamma \vdash \hat\alpha| = 1 + |\Gamma \vdash \beta| = 1 + 1 = 2$, whereas the plain size of $\hat\alpha$ is simply 1.

The connection between (S1) and (S2) may be clarified by examining rule $<:\to$, whose conclusion says that $A_1 \to A_2$ is a subtype of $B_1 \to B_2$. If A_2 or B_2 is polymorphic, then the first premise on $A_1 \to A_2$ is smaller by (S1). Otherwise, the first premise has the same input context as the conclusion, so it has the same (S2), but is smaller by (S3). If B_1 or A_1 is polymorphic, then the second premise is smaller by (S1). Otherwise, we use the property that instantiating a monotype always solves an existential:

Lemma (Monotypes Solve Variables)**.** *If* $\Gamma \vdash \hat\alpha :\leqq \tau \dashv \Delta$ *or* $\Gamma \vdash \tau \leqq: \hat\alpha \dashv \Delta$, *then if* $[\Gamma]\tau = \tau$ *and* $\hat\alpha \notin FV([\Gamma]\tau)$, *we have* $|\mathsf{unsolved}(\Gamma)| = |\mathsf{unsolved}(\Delta)| + 1$.

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

Lemma (Monotype Monotonicity)**.**
If $\Gamma \vdash \tau_1 <: \tau_2 \dashv \Delta$ *then* $|\mathsf{unsolved}(\Delta)| \le |\mathsf{unsolved}(\Gamma)|$.

Lemma (Substitution Decreases Size)**.**
If $\Gamma \vdash A$ *then* $|\Gamma \vdash [\Gamma]A| \le |\Gamma \vdash A|$.

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

5.3 Decidability of Algorithmic Typing

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

(i) Synthesis: *Given a context* Γ *and a term* e, *it is decidable whether there exist a type* A *and a context* Δ *such that* $\Gamma \vdash e \Rightarrow A \dashv \Delta$.

(ii) Checking: *Given a context* Γ, *a term* e, *and a type* B *such that* $\Gamma \vdash B$, *it is decidable whether there is a context* Δ *such that* $\Gamma \vdash e \Leftarrow B \dashv \Delta$.

(iii) Application: *Given a context* Γ, *a term* e, *and a type* A *such that* $\Gamma \vdash A$, *it is decidable whether there exist a type* C *and a context* Δ *such that* $\Gamma \vdash A \bullet e \Rightarrow\!\!\!\!\Rightarrow C \dashv \Delta$.

The following induction measure suffices to prove decidability:

$$\left\langle e,\ \begin{array}{ll} \Rightarrow & \\ \Leftarrow, & |\Gamma \vdash B| \\ \Rightarrow\!\!\!\!\Rightarrow, & |\Gamma \vdash A| \end{array} \right\rangle$$

where $\langle \dots \rangle$ denotes lexicographic order, and where (when comparing two judgments typing the same term e) the synthesis judgment (top line) is considered smaller than the checking judgment (second line), which in turn is considered smaller than the application judgment (bottom line). That is, $\Rightarrow \prec \Leftarrow \prec \Rightarrow\!\!\!\!\Rightarrow$. In Sub, this makes the synthesis premise smaller than the checking conclusion; in $\to$App and $\hat\alpha$App, this makes the checking premise smaller than the application conclusion.

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

6. Soundness

We want the algorithmic specifications of subtyping and typing to be sound with respect to the declarative specifications. Roughly, given a derivation of an algorithmic judgment with input context Γ and output context Δ, and some complete context Ω that extends Δ (which therefore extends Γ), applying Ω throughout the given algorithmic judgment should yield a derivable declarative judgment. Let's make that rough outline concrete for instantiation, showing that the action of the instantiation rules is consistent with declarative subtyping:

Theorem 10 (Instantiation Soundness)**.**
Given $\Delta \longrightarrow \Omega$ *and* $[\Gamma]B = B$ *and* $\hat\alpha \notin FV(B)$*:*

(1) If $\Gamma \vdash \hat\alpha :\leqq B \dashv \Delta$ *then* $[\Omega]\Delta \vdash [\Omega]\hat\alpha \le [\Omega]B$.

(2) If $\Gamma \vdash B \leqq: \hat\alpha \dashv \Delta$ *then* $[\Omega]\Delta \vdash [\Omega]B \le [\Omega]\hat\alpha$.

Note that the declarative derivation is under $[\Omega]\Delta$, which is Ω applied to the algorithmic output context Δ.

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

Theorem 11 (Soundness of Algorithmic Subtyping)**.**
If $\Gamma \vdash A <: B \dashv \Delta$ *where* $[\Gamma]A = A$ *and* $[\Gamma]B = B$ *and* $\Delta \longrightarrow \Omega$ *then* $[\Omega]\Delta \vdash [\Omega]A \le [\Omega]B$.

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

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

(i) If $\Gamma \vdash e \Leftarrow A \dashv \Delta$ *then* $[\Omega]\Delta \vdash e \Leftarrow [\Omega]A$.

(ii) If $\Gamma \vdash e \Rightarrow A \dashv \Delta$ *then* $[\Omega]\Delta \vdash e \Rightarrow [\Omega]A$.

(iii) If $\Gamma \vdash A \bullet e \Rightarrow\!\!\!\!\Rightarrow C \dashv \Delta$ *then* $[\Omega]\Delta \vdash [\Omega]A \bullet e \Rightarrow\!\!\!\!\Rightarrow [\Omega]C$.

The proofs need several lemmas, including this one:

Lemma (Typing Extension)**.**
If $\Gamma \vdash e \Leftarrow A \dashv \Delta$ *or* $\Gamma \vdash e \Rightarrow A \dashv \Delta$ *or* $\Gamma \vdash A \bullet e \Rightarrow\!\!\!\!\Rightarrow C \dashv \Delta$ *then* $\Gamma \longrightarrow \Delta$.

7. Completeness

Completeness of the algorithmic system is something like soundness in reverse: given a declarative derivation of $[\Omega]\Gamma \vdash [\Omega]\cdots$, we want to get an algorithmic derivation of $\Gamma \vdash \cdots \dashv \Delta$.

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

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

Theorem 13 (Instantiation Completeness)**.** *Given* $\Gamma \longrightarrow \Omega$ *and* $A = [\Gamma]A$ *and* $\hat\alpha \in \mathsf{unsolved}(\Gamma)$ *and* $\hat\alpha \notin FV(A)$*:*

(1) If $[\Omega]\Gamma \vdash [\Omega]\hat\alpha \le [\Omega]A$ *then there are* Δ, Ω' *such that* $\Omega \longrightarrow \Omega'$ *and* $\Delta \longrightarrow \Omega'$ *and* $\Gamma \vdash \hat\alpha :\leqq A \dashv \Delta$.

(2) If $[\Omega]\Gamma \vdash [\Omega]A \le [\Omega]\hat\alpha$ *then there are* Δ, Ω' *such that* $\Omega \longrightarrow \Omega'$ *and* $\Delta \longrightarrow \Omega'$ *and* $\Gamma \vdash A \leqq: \hat\alpha \dashv \Delta$.

Theorem 14 (Generalized Completeness of Subtyping)**.**
If $\Gamma \longrightarrow \Omega$ *and* $\Gamma \vdash A$ *and* $\Gamma \vdash B$ *and* $[\Omega]\Gamma \vdash [\Omega]A \le [\Omega]B$ *then there exist* Δ *and* Ω' *such that* $\Delta \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash [\Gamma]A <: [\Gamma]B \dashv \Delta$.

Theorem 15 (Completeness of Algorithmic Typing)**.**
Given $\Gamma \longrightarrow \Omega$ *and* $\Gamma \vdash A$*:*

(i) If $[\Omega]\Gamma \vdash e \Leftarrow [\Omega]A$
then there exist Δ *and* Ω'
such that $\Delta \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash e \Leftarrow [\Gamma]A \dashv \Delta$*.*

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

(iii) If $[\Omega]\Gamma \vdash [\Omega]A \bullet e \Rrightarrow C$
then there exist Δ*,* Ω'*, and* C'
such that $\Delta \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$
and $\Gamma \vdash [\Gamma]A \bullet e \Rrightarrow C' \dashv \Delta$ *and* $C = [\Omega']C'$*.*

8. Design Variations

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

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

$$\frac{}{\Gamma[\hat\alpha] \vdash () \Leftarrow \hat\alpha \dashv \Gamma[\hat\alpha = 1]}\;1\mathsf{I}\hat\alpha$$

$$\frac{\Gamma[\hat\alpha_2, \hat\alpha_1, \hat\alpha = \hat\alpha_1 \to \hat\alpha_2], x : \hat\alpha_1 \vdash e \Leftarrow \hat\alpha_2 \dashv \Delta, x : \hat\alpha_1, \Delta'}{\Gamma[\hat\alpha] \vdash \lambda x.\, e \Leftarrow \hat\alpha \dashv \Delta}\;{\to}\mathsf{I}\hat\alpha$$

With these two rules replacing $1\mathsf{I}{\Rightarrow}$ and ${\to}\mathsf{I}{\Rightarrow}$, we have a complete algorithm for the no-inference bidirectional system.

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

$$\frac{\begin{array}{c}\Gamma, \blacktriangleright_{\hat\alpha}, \hat\alpha, \hat\beta, x : \hat\alpha \vdash e \Leftarrow \hat\beta \dashv \Delta, \blacktriangleright_{\hat\alpha}, \Delta' \\ \tau = [\Delta'](\hat\alpha \to \hat\beta) \qquad \vec{\hat\alpha} = \mathsf{unsolved}(\Delta')\end{array}}{\Gamma \vdash \lambda x.\, e \Rightarrow \forall\vec\alpha.\ [\vec\alpha/\vec{\hat\alpha}]\tau \dashv \Delta}\;{\to}\mathsf{I}{\Rightarrow}'$$

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

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

9. Related Work and Discussion

9.1 Type Inference for System F

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

As discussed in Section 2, for typability under η-reductions, it is necessary for subtyping to instantiate deeply: that is, we must allow instantiation of quantifiers to the right of an arrow. However, Tiuryn and Urzyczyn (1996) and Chrząszcz (1998) showed that the subtyping relation for impredicative System F is undecidable. As a result, if we want η *and* a complete algorithm, then either the polymorphic instantiations must be predicative, or a different language of types must be used.

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

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

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

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

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

History of our approach. Several of the ideas used in the present paper descend from Dunfield (2009), an approach to first-class polymorphism (including impredicativity) also based on ordered contexts with existential variables instantiated via subtyping. In

System	η-laws?	Impredicative?	System F type language?
$\mathsf{ML^F}$	yes	yes	no
FPH	no	yes	yes
HML	no	yes	yes
Peyton Jones et al. (2007)	yes	no	yes
This paper	yes	no	yes

Figure 15. Comparison of type inference algorithms

fact, the present work began as an attempt to extend Dunfield (2009) with type-level computation. During that attempt, we found several shortcomings and problems. The most serious is that the decidability and completeness arguments were not valid. These problems may be fixable, but instead we started over, reusing several of the high-level ideas in different technical forms.

9.2 Other Type Systems

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

9.3 Our Algorithm

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

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

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

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

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

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

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

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

Acknowledgments

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

References

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 βη-conversion test for Martin-Löf type theory. In *Mathematics of Program Construction (MPC'08)*, volume 5133 of *LNCS*, pages 29–56, 2008.

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Logic and Computation*, 2(3):297–347, 1992.

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

Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost in translation: formalizing proposed extensions to C$^\sharp$. In *OOPSLA*, 2007.

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

Iliano Cervesato and Frank Pfenning. A linear spine calculus. *J. Logic and Computation*, 13(5):639–688, 2003.

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.

Jacek Chrząszcz. Polymorphic subtyping without distributivity. In *Mathematical Foundations of Computer Science*, volume 1450 of *LNCS*, pages 346–355. Springer, 1998.

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

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212. ACM, 1982.

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

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

Joshua Dunfield. Greedy bidirectional polymorphism. In *ML Workshop*, pages 15–26, 2009. `http://www.cs.cmu.edu/~joshuad/papers/poly/`.

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

Adam Gundry, Conor McBride, and James McKinna. Type inference in context. In *Mathematically Structured Functional Programming (MSFP)*, 2010.

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

Didier Le Botlan and Didier Rémy. Recasting ML^F. *Information and Computation*, 207:726–785, 2009.

Daan Leijen. Flexible types: robust type inference for first-class polymorphism. In *POPL*, pages 66–77, 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. *Refinement Types for Logical Frameworks*. PhD thesis, Carnegie Mellon University, 2010. CMU-CS-10-138.

Dale Miller. Unification under a mixed prefix. *J. Symbolic Computation*, 14:321–358, 1992.

John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76:211–249, 1988.

Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *POPL*, 1996.

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

Simon Peyton Jones and Mark Shields. Lexically scoped type variables. Technical report, Microsoft Research, 2004.

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.

Frank Pfenning. Church and Curry: Combining intrinsic and extrinsic typing. In *Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday*. College Publications, 2008.

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

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Prog. Lang. Sys.*, 22:1–44, 2000.

Didier Rémy. Extension of ML type system with a sorted equational theory on types. Research Report 1766, INRIA, 1992.

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

Robert J. Simmons. Structural focalization. arXiv:`1109.6273v4 [cs.LO]`, 2012.

Jerzy Tiuryn and Paweł Urzyczyn. The subtyping problem for second-order types is undecidable. In *LICS*, 1996.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. Boxy types: inference for higher-rank types and impredicativity. In *ICFP*, pages 251–262, 2006.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. FPH: First-class polymorphism for Haskell. In *ICFP*, pages 295–306, 2008.

Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. Let should not be generalised. In *Workshop on Types in Language Design and Impl. (TLDI '10)*, pages 39–50, 2010.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In *Types for Proofs and Programs*, pages 355–377. Springer-Verlag LNCS 3085, 2004.

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.