Higher-rank Polymorphism: Type Inference and Extensions

by

Ningning Xie
(谢宁宁)

A thesis submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy
at The University of Hong Kong

February 2021

Abstract of thesis entitled

“Higher-rank Polymorphism: Type Inference and Extensions”

Submitted by

Ningning Xie

for the degree of Doctor of Philosophy
at The University of Hong Kong
in February 2021

Declaration

I declare that this thesis represents my own work, except where due acknowledgment is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

. . . .

Ningning Xie
February 2021

Acknowledgments

Contents

List of Figures

List of Tables

Part I

Prologue

1 Introduction

mention that in this thesis when we say “higher-rank polymorphism” we mean “predicative implicit higher-rank polymorphism”.

1.1 Contributions

In summary the contributions of this thesis are:

??

- ?? proposes a new design for type inference of higher-rank polymorphism.
  - We design a variant of bi-directional type checking where the inference mode is combined with a new, so-called, application mode. The application mode naturally propagates type information from arguments to the functions.
  - With the application mode, we give a new design for type inference of higher-rank polymorphism, which generalizes the HM type system, supports a polymorphic let as syntactic sugar, and infers higher rank types. We present a syntax-directed specification, an elaboration semantics to System F, and an algorithmic type system with completeness and soundness proofs.
- ?? presents a new approach for implementing unification.
  - We propose a process named *promotion*, which, given a unification variable and a type, promotes the type so that all unification variables in the type are well-typed with regard to the unification variable.
  - We apply promotion in a new implementation of the unification procedure in higher-rank polymorphism, and show that the new implementation is sound and complete.

??

- ?? extends higher-rank polymorphism with gradual types.
  - We define a framework for consistent subtyping with
    - ⋆ a new definition of consistent subtyping that subsumes and generalizes that of ? and can deal with polymorphism and top types;

        * and a syntax-directed version of consistent subtyping that is sound and complete with respect to our definition of consistent subtyping, but still guesses instantiations.

    - Based on consistent subtyping, we present he calculus GPC. We prove that our calculus satisfies the static aspects of the refined criteria for gradual typing [?], and is type-safe by a type-directed translation to λB [?].
    - We present a sound and complete bidirectional algorithm for implementing the declarative system based on the design principle of ?.

- **??** further explores the design of promotion in the context of kind inference for datatypes.
    - We formalize Haskell98's datatype declarations, providing both a declarative specification and syntax-driven algorithm for kind inference. We prove that the algorithm is sound and observe how Haskell98's technique of defaulting unconstrained kinds to $\star$ leads to incompleteness. We believe that ours is the first formalization of this aspect of Haskell98.
    - We then present a type and kind language that is unified and dependently typed, modeling the challenging features for kind inference in modern Haskell. We include both a declarative specification and a syntax-driven algorithm. The algorithm is proved sound, and we observe where and why completeness fails. In the design of our algorithm, we must choose between completeness and termination; we favor termination but conjecture that an alternative design would regain completeness. Unlike other dependently typed languages, we retain the ability to infer top-level kinds instead of relying on compulsory annotations.

Many metatheory in the paper comes with Coq proofs, including type safety, coherence, etc.[1]

1.2 Organization

This thesis is largely based on the publications by the author [?????], as indicated below.

??: Ningning Xie and Bruno C. d. S. Oliveira. 2018. "Let Arguments Go First". In *European Symposium on Programming (ESOP)*.

[1] For convenience, whenever possible, definitions, lemmas and theorems have hyperlinks (click ☞) to their Coq counterparts.

??: Ningning Xie and Bruno C. d. S. Oliveira. 2017. "Towards Unification for Dependent Types" (Extended abstract), In *Draft Proceedings of Trends in Functional Programming (TFP)*.

??: Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. 2018. "Consistent Subtyping for All". In *European Symposium on Programming (ESOP)*.

Ningning Xie, Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. "Consistent Subtyping for All". In *ACM Transactions on Programming Languages and Systems (TOPLAS)*.

??: Ningning Xie, Richard Eisenberg and Bruno C. d. S. Oliveira. 2020. "Kind Inference for Datatypes". In *Symposium on Principles of Programming Languages (POPL)*.

2 Background

This chapter sets the stage for type systems in later chapters. **??** reviews the Hindley-Milner type system [**???**], a classical type system for the lambda calculus with parametric polymorphism. We first review its syntax and semantics, then discuss the property of principality, and finally talk briefly about its algorithmic system. **??** presents the Odersky-Läufer type system [**?**], which extends upon the Hindley-Milner type system by putting higher-rank type annotations to work. Finally in **??** we introduce the Dunfield-Krishnaswami type system, a bidirecitonal higher-rank type system.

2.1 The Hindley-Milner Type System

The global type-inference algorithms employed in modern functional languages such as ML, Haskell and OCaml, are derived from the Hindley-Milner type system. The Hindley-Milner type system, hereafter referred to as HM, is a polymorphic type discipline first discovered in **?**, later rediscovered by **?**, and also closely formalized by **?**.

2.1.1 Syntax

The syntax of HM is given in **??**. The expressions e include variables x, literals n, lambda abstractions $\lambda x.\ e$, applications $e_1\ e_2$ and $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$. Note here lambda abstractions have no type annotations, and the type information is to be reconstructed by the type system.

Types consist of polymorphic types σ and monomorphic types (monotypes) τ. A polymorphic type is a sequence of universal quantifications (which can be empty) followed by a monotype τ, which can be the integer type Int, type variables a and function types $\tau_1 \to \tau_2$.

A context Ψ tracks the type information for variables. We implicitly assume items in a context are distinct throughout the thesis.

2.1.2 Static Semantics

The declarative typing judgment $\Psi \vdash^{HM} e : \sigma$ derives the type σ of the expression e under the context Ψ. Rule hm-var fetches a polymorphic type $x : \sigma$ from the context. Literals always

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & x \mid n \mid \lambda x.\, e \mid e_1\, e_2 \mid \textbf{let}\, x = e_1\, \textbf{in}\, e_2 \\
\text{Types} & \sigma & ::= & \forall \overline{a}^{\,i}.\, \tau \\
\text{Monotypes} & \tau & ::= & \mathsf{Int} \mid a \mid \tau_1 \to \tau_2 \\
\text{Contexts} & \Psi & ::= & \bullet \mid \Psi, x : \sigma
\end{array}
$$

$\boxed{\Psi \vdash^{HM} e : \sigma}$ *(Typing)*

$$
\frac{(x : \sigma) \in \Psi}{\Psi \vdash^{HM} x : \sigma}\ \textsc{hm-var}
\qquad
\frac{}{\Psi \vdash^{HM} n : \mathsf{Int}}\ \textsc{hm-int}
\qquad
\frac{\Psi, x : \tau_1 \vdash^{HM} e : \tau_2}{\Psi \vdash^{HM} \lambda x.\, e : \tau_1 \to \tau_2}\ \textsc{hm-lam}
$$

$$
\frac{\Psi \vdash^{HM} e_1 : \tau_1 \to \tau_2 \qquad \Psi \vdash^{HM} e_2 : \tau_1}{\Psi \vdash^{HM} e_1\, e_2 : \tau_2}\ \textsc{hm-app}
\qquad
\frac{\Psi \vdash^{HM} e_1 : \sigma \qquad \Psi, x : \sigma \vdash^{HM} e_2 : \tau}{\Psi \vdash^{HM} \textbf{let}\, x = e_1\, \textbf{in}\, e_2 : \tau}\ \textsc{hm-let}
$$

$$
\frac{\overline{a}^{\,i} \notin \textsc{fv}\,(\Psi) \qquad \Psi \vdash^{HM} e : \tau}{\Psi \vdash^{HM} e : \forall \overline{a}^{\,i}.\, \tau}\ \textsc{hm-gen}
\qquad
\frac{\Psi \vdash^{HM} e : \forall \overline{a}^{\,i}.\, \tau}{\Psi \vdash^{HM} e : \tau[\overline{a_i \mapsto \tau_i}^{\,i}]}\ \textsc{hm-inst}
$$

Figure 2.1: Syntax and static semantics of the Hindley-Milner type system.

have the integer type (rule HM-INT). For lambdas (rule HM-LAM), since there is no type for the binder given, the system *guesses* a *monotype* τ_1 as the type of x, and derives the type τ_2 for the body e, returning a function $\tau_1 \to \tau_2$. Function types are eliminated by applications. In rule HM-APP, the type of the argument must match the parameter's type τ_1, and the whole application returns type τ_2.

Rule HM-LET is the key rule for flexibility in HM, where a *polymorphic* expression can be defined, and later instantiated with different types in the call sites. In this rule, the expression e_1 has a polymorphic type σ, and the rule adds $x : \sigma$ into the context to type-check e_2.

Rule HM-GEN and rule HM-INST correspond to *generalization* and *instantiation* respectively. In rule HM-GEN, we can generalize over type variables $\overline{a}^{\,i}$ which are not bound in the type context Ψ. In rule HM-INST, we can instantiate the type variables with arbitrary *monotypes*.

2.1.3 Principal Type Scheme

One salient feature of HM is that the system enjoys the existence of *principal types*, without requiring any type annotations. Before we present the definition of principal types, let's first define the *subtyping* relation among types.

$\boxed{\vdash^{HM} \sigma_1 <: \sigma_2}$ *(Subtping)*

$$\frac{}{\vdash^{HM} \tau <: \tau}\ \textsc{hm-s-refl} \qquad \frac{a \notin \textsc{fv}(\sigma_1) \qquad \vdash^{HM} \sigma_1 <: \sigma_2}{\vdash^{HM} \sigma_1 <: \forall a.\, \sigma_2}\ \textsc{hm-s-forallR} \qquad \frac{\vdash^{HM} \sigma_1[a \mapsto \tau] <: \sigma_2}{\vdash^{HM} \forall a.\, \sigma_1 <: \sigma_2}\ \textsc{hm-s-forallL}$$

Figure 2.2: Subtyping in the Hindley-Milner type system.

The judgment $\vdash^{HM} \sigma_1 <: \sigma_2$, given in **??**, reads that σ_1 is a subtype of σ_2. The subtyping relation indicates that σ_1 is more *general* than σ_2: for any instantiation of σ_2, we can find an instantiation of σ_1 to make two types match. Rule HM-S-REFL is simply reflexive for monotypes. Rule HM-S-FORALLR has a polymorphic type $\forall a.\, \sigma_2$ on the right hand side. In order to prove the subtyping relation for *all* possible instantiation of a, we *skolemize* a, by making sure a does not appear in σ_1 (up to α-renaming). In this case, if σ_1 is still a subtype of σ_2, we are sure then whatever a can be instantiated to, σ_1 can be instantiated to match σ_2. In rule HM-S-FORALLL, by contrast, the a in $\forall a.\, \sigma_1$ can be instantiated to any monotype to match the right hand side. For example:

$$\begin{array}{lcl} \mathsf{Int} \to \mathsf{Int} & <: & \mathsf{Int} \to \mathsf{Int} \\ \forall a.\, a \to a & <: & \mathsf{Int} \to \mathsf{Int} \end{array}$$

Given the subtyping relation, now we can formally state that HM enjoys *principality*. That is, for every well-typed expression in HM, there exists one type for the expression, which is more general than any other types the expression can derive. Formally,

Theorem 2.1 (Principality for HM). *If $\Psi \vdash^{HM} e : \sigma$, then there exists σ' such that $\Psi \vdash^{HM} e : \sigma'$, and for all σ such that $\Psi \vdash^{HM} e : \sigma$, we have $\vdash^{HM} \sigma' <: \sigma$.*

Consider the expression $\lambda x.\, x$. It has a principal type $\forall a.\, a \to a$, which is more general than other options, e.g., $\mathsf{Int} \to \mathsf{Int}$, $(\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int})$, etc.

2.1.4 Algorithmic Type System

The declarative specification of HM given in **??** does not directly lead to an algorithm, because there are still many guesses in the system, such as in rule HM-LAM.

There exists a sound and complete type inference algorithm for HM [**?**], which has served as the basis for the type inference algorithm for many other systems [**??**], including the system presented in **??**. We will discuss more about it in **??**.

2.2 The Odersky-Läufer Type System

The HM system is simple, flexible and powerful. However, since the type annotations in lambda abstractions are always missing, HM only derives polymorphic types of *rank 1*. That is, universal quantifiers only appear at the top level. Polymorphic types are of *higher-rank*, if universal quantifiers can appear anywhere in a type.

Essentially higher-rank types enable much of the expressive power of System F, with the advantage of implicit polymorphism. Complete type inference for System F is known to be undecidable [?]. **?** proposed a type system, hereafter referred to as OL, which extends HM by allowing lambda abstractions to have explicit *higher-rank* types as type annotations. As a motivation, consider the following program[1]:

```
(\f. (f 1, f 'a')) (\x. x)
```

which is not typeable under HM because it fails to infer the type of `f`, since it is supposed to be polymorphic. With OL we can add the type annotation for `f`:

```
(\f : ∀a. a → a. (f 1, f 'a')) (\x. x)
```

Note that the first function now has a rank-2 type, as the polymorphic type `∀a. a → a` appears in the argument position of a function:

```
(\f : ∀a. a → a. (f 1, f 'a')) : (∀a. a → a) → (Int, Char)
```

2.2.1 Higher-rank Types

We define the rank of types as follows.

Definition 1 (Type rank). The *rank* of a type is the depth at which universal quantifiers appear contravariantly [?]. Formally,

$$\begin{array}{lcl} \mathsf{rank}(\tau) & = & 0 \\ \mathsf{rank}(\sigma_1 \to \sigma_2) & = & \mathsf{max}(\mathsf{rank}(\sigma_1) + 1, \mathsf{rank}(\sigma_2)) \\ \mathsf{rank}(\forall a.\, \sigma) & = & \mathsf{max}(1, \mathsf{rank}(\sigma)) \end{array}$$

Below we give some examples:

$$\begin{array}{lcl} \mathsf{rank}(\mathsf{Int} \to \mathsf{Int}) & = & 0 \\ \mathsf{rank}(\forall a.\, a \to a) & = & 1 \\ \mathsf{rank}(\mathsf{Int} \to (\forall a.\, a \to a)) & = & 1 \\ \mathsf{rank}((\forall a.\, a \to a) \to \mathsf{Int}) & = & 2 \end{array}$$

[1] For the purpose of better illustration, we assume basic constructs like booleans and pairs in examples.

Expressions	e	$::=$	$x \mid n \mid \lambda x : \sigma.\, e \mid \lambda x.\, e \mid e_1\, e_2 \mid \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$
Types	σ	$::=$	$\mathsf{Int} \mid a \mid \sigma_1 \to \sigma_2 \mid \forall a.\, \sigma$
Monotypes	τ	$::=$	$\mathsf{Int} \mid a \mid \tau_1 \to \tau_2$
Contexts	Ψ	$::=$	$\bullet \mid \Psi, x : \sigma \mid \Psi, a$

Figure 2.3: Syntax of the Odersky-Läufer type system.

$\boxed{\Psi \vdash^{OL} \sigma}$ *(Type Well-formedness)*

$$\text{OL-WF-INT}\quad \frac{}{\Psi \vdash^{OL} \mathsf{Int}} \qquad \text{OL-WF-TVAR}\quad \frac{a \in \Psi}{\Psi \vdash^{OL} a} \qquad \text{OL-WF-ARROW}\quad \frac{\Psi \vdash^{OL} \sigma_1 \qquad \Psi \vdash^{OL} \sigma_2}{\Psi \vdash^{OL} \sigma_1 \to \sigma_2} \qquad \text{OL-WF-FORALL}\quad \frac{\Psi, a \vdash^{OL} \sigma}{\Psi \vdash^{OL} \forall a.\, \sigma}$$

Figure 2.4: Well-formedness of types in the Odersky-Läufer type system.

From the definition, we can see that monotypes always have rank 0, and the polymorphic types in HM (σ in **??**) has at most rank 1.

2.2.2 Syntax

The syntax of OL is given in **??**. Comparing to HM, we observe the following differences.

First, expressions e include not only unannotated lambda abstractions $\lambda x.\, e$, but also annotated lambda abstractions $\lambda x : \sigma.\, e$, where the type annotation σ is a polymorphic type. Thus unlike HM, the argument type for a function is not limited to a monotype.

Second, the polymorphic types σ now include the integer type Int, type variables a, functions $\sigma_1 \to \sigma_2$ and universal quantifications $\forall a.\, \sigma$. Since the argument type in a function can be polymorphic, we see that OL supports *arbitrary* rank of types. The definition of monotypes remains the same. Obviously polymorphic types still subsume monotypes.

Finally, in addition to variable types, the contexts Ψ now also keep track of type variables. Note that in the original work in **?**, the system, much like HM, does not track type variables; instead, it explicitly checks that type variables are fresh with respect to a context or a type when needed. Here we include type variables in contexts, as it sets us well for the Dunfield-Krishnaswami type system to be introduced in the next section. Moreover, the differences do not change the essence of the type system, and it provides a complete view of the possible formalism of contexts in a type system with generalization. As before, we assume all items in a context are distinct.

Now for a type to be well-formedness, it must have all its free variable bound in the context. The type well-formedness rules are given in **??**. All rules are straightforward.

$\boxed{\Psi \vdash^{OL} e : \sigma}$ *(Typing)*

$$\frac{(x : \sigma) \in \Psi}{\Psi \vdash^{OL} x : \sigma}\ \textsc{ol-var} \qquad \frac{}{\Psi \vdash^{OL} n : \mathsf{Int}}\ \textsc{ol-int} \qquad \frac{\Psi, x : \sigma_1 \vdash^{OL} e : \sigma_2}{\Psi \vdash^{OL} \lambda x : \sigma_1 .\, e : \sigma_1 \to \sigma_2}\ \textsc{ol-lamann}$$

$$\frac{\Psi \vdash^{OL} \tau \qquad \Psi, x : \tau \vdash^{OL} e : \sigma}{\Psi \vdash^{OL} \lambda x.\, e : \tau \to \sigma}\ \textsc{ol-lam} \qquad \frac{\Psi \vdash^{OL} e_1 : \sigma_1 \to \sigma_2 \qquad \Psi \vdash^{OL} e_2 : \sigma_1}{\Psi \vdash^{OL} e_1\, e_2 : \sigma_2}\ \textsc{ol-app}$$

$$\frac{\Psi \vdash^{OL} e_1 : \sigma_1 \qquad \Psi, x : \sigma_1 \vdash^{OL} e_2 : \sigma_2}{\Psi \vdash^{OL} \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \sigma_2}\ \textsc{ol-let} \qquad \frac{\Psi, a \vdash^{OL} e : \sigma}{\Psi \vdash^{OL} e : \forall a.\, \sigma}\ \textsc{ol-gen}$$

$$\frac{\Psi \vdash^{OL} e : \sigma_1 \qquad \Psi \vdash \sigma_1 <: \sigma_2}{\Psi \vdash^{OL} e : \sigma_2}\ \textsc{ol-sub}$$

$\boxed{\Psi \vdash^{OL} \sigma_1 <: \sigma_2}$ *(Subtyping)*

$$\frac{a \in \Psi}{\Psi \vdash^{OL} a <: a}\ \textsc{ol-s-tvar} \qquad \frac{}{\Psi \vdash^{OL} \mathsf{Int} <: \mathsf{Int}}\ \textsc{ol-s-int} \qquad \frac{\Psi \vdash^{OL} \sigma_3 <: \sigma_1 \qquad \Psi \vdash^{OL} \sigma_2 <: \sigma_4}{\Psi \vdash^{OL} \sigma_1 \to \sigma_2 <: \sigma_3 \to \sigma_4}\ \textsc{ol-s-arrow}$$

$$\frac{\Psi \vdash^{OL} \tau \qquad \Psi \vdash^{OL} \sigma[a \mapsto \tau] <: \sigma_2}{\Psi \vdash^{OL} \forall a.\, \sigma_1 <: \sigma_2}\ \textsc{ol-s-forall}\mathrm{L} \qquad \frac{\Psi, a \vdash^{OL} \sigma_1 <: \sigma_2}{\Psi \vdash^{OL} \sigma_1 <: \forall a.\, \sigma_2}\ \textsc{ol-s-forall}\mathrm{R}$$

Figure 2.5: Static semantics of the Odersky-Läufer type system.

2.2.3 Static Semantics

The static semantics of OL is given in ??.

Rule OL-VAR and rule OL-INT are the same as that of HM. Rule OL-LAMANN type-checks annotated lambda abstractions, by simply putting $x : \sigma$ into the context to type the body. For unannotated lambda abstractions in rule OL-LAM, the system still guesses a mere monotype. That is, the system never guesses a polymorphic type for lambdas; instead, an explicit polymorphic type annotation is required. Rule OL-APP, rule OL-LET are similar as HM, except that polymorphic types may appear in return types. In the generalization rule OL-GEN, we put a new type variable a into the context, and the return type σ is then generalized over a, returning $\forall a. \sigma$.

The subsumption rule OL-SUB is crucial for OL, which allows an expression of type σ_1 to have type σ_2 with σ_1 being a subtype of σ_2 ($\Psi \vdash \sigma_1 <: \sigma_2$). Note that the instantiation rule HM-INST in HM is a special case of rule OL-SUB, as we have $\forall \overline{a}^{\,i}. \tau <: \tau[\overline{a_i \mapsto \tau_i}^{\,i}]$ by applying rule HM-S-FORALLL repeatedly.

The subtyping relation of OL $\Psi \vdash^{OL} \sigma_1 <: \sigma_2$ also generalizes the subtyping relation of HM. In particular, in rule OL-S-ARROW, functions are *contravariant* on arguments, and *covariant* on return types. This rule allows us to compare higher-rank polymorphic types, rather than just polymorphic types with universal quantifiers only at the top level. For example,

$$
\begin{array}{lcl}
\Psi \vdash^{OL} \forall a. a \to a & <: & \mathsf{Int} \to \mathsf{Int} \\
\Psi \vdash^{OL} \mathsf{Int} \to (\forall a. a \to a) & <: & \mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int}) \\
\Psi \vdash^{OL} (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int} & <: & (\forall a. a \to a) \to \mathsf{Int}
\end{array}
$$

Predicativity. In a system with high-ranker types, one important design decision to make is whether the system is *predicative* or *impredicative*. A system is predicative, if the type variable bound by a universal quantifier is only allowed to be substituted by a monotype; otherwise it is impredicative. It is well-known that general type inference for impredicativity is undecidable [?]. OL is predicative, which can be seen from rule OL-S-FORALLL. We focus only on predicative type systems throughout the thesis.

2.2.4 Relating to HM

It can be proved that OL is a conservative extension of HM. That is, every well-typed expression in HM is well-typed in OL, modulo the different representation of contexts.

Theorem 2.2 (Odersky-Läufer type system conservative over Hindley-Milner type system). *If $\Psi \vdash^{HM} e : \sigma$, suppose Ψ' is Ψ extended with type variables in Ψ and σ, then $\Psi' \vdash^{OL} e : \sigma$.*

Moreover, since OL is predicative and only guesses monotypes for unannotated lambda abstractions, its algorithmic system can be implemented as a direct extension of the one for HM.

2.3 The Dunfield-Krishnaswami Type System

Both HM and OL derive only monotypes for unannotated lambda abstractions. OL improves on HM by allowing polymorphic lambda abstractions but requires the polymorphic type annotations are given explicitly. The Dunfield-Krishnaswami type system [**?**], hereafter refered to as DM, give a *bidirectional* account of higher-rank polymorphism, where type information can be propagated through the syntax tree. Therefore, it is possible for a variable bound in a lambda abstraction without explicit type annotations to get a polymorphic type.

2.3.1 Bidirectional Type Checking

Bidirectional type checking has been known in the folklore of type systems for a long time. It was popularized by Pierce and Turner's work on local type inference [**?**]. Local type inference was introduced as an alternative to HM type systems, which could easily deal with polymorphic languages with subtyping. The key idea in local type inference is simple.

> *"... are local in the sense that missing annotations are recovered using only information from adjacent nodes in the syntax tree, without long-distance constraints such as unification variables."*

Bidirectional type checking is one component of local type inference that, aided by some type annotations, enables type inference in an expressive language with polymorphism and subtyping. In its basic form typing is split into *inference* and *checking* modes. The most salient feature of a bidirectional type-checker is when information deduced from inference mode is used to guide checking of an expression in checked mode.

Since Pierce and Turner's work, various other authors have proved the effectiveness of bidirectional type checking in several other settings, including many different systems with subtyping [**??**], systems with dependent types [**????**], etc.

In particular, bidirectional type checking has also been combined with HM-style techniques for providing type inference in the presence of higher-rank type, including DK and **?**. Let's revisit the example in **??**:

```
(\f. (f 1, f 'a')) (\x. x)
```

Expressions	e	$::=$	$x \mid n \mid \lambda x : \sigma.\, e \mid \lambda x.\, e \mid e_1\, e_2 \mid e : \sigma$
Types	σ	$::=$	$\mathsf{Int} \mid a \mid \sigma_1 \to \sigma_2 \mid \forall a.\, \sigma$
Monotypes	τ	$::=$	$\mathsf{Int} \mid a \mid \tau_1 \to \tau_2$
Contexts	Ψ	$::=$	$\bullet \mid \Psi, x : \sigma \mid \Psi, a$

Figure 2.6: Syntax of the Dunfield-Krishnaswami Type System

which is not typeable in HM as it they fail to infer the type of `f`. In OL, it can be type-checked by adding a polymorphic type annotation on `f`. In DK, we can also add a polymorphic type annotation on `f`. But with bi-directional type checking, the type annotation can be propagated from somewhere else. For example, we can rewrite this program as:

```
((\f. (f 1, f 'c')) : (∀a. a → a) → (Int, Char)) (\x . x)
```

Here the type of `f` can be easily derived from the type signature using checking mode in bi-directional type checking.

2.3.2 Syntax

The syntax of the DK is given in **??**. Comparing to OL, only the definition of expressions slightly differs. First, the expressions e in DK have no let expressions. **?** omitted let-binding from the formal development, but argued that restoring let-bindings is easy, as long as they get no special treatment incompatible with substitution (e.g., a syntax-directed HM does polymorphic generalization only at let-bindings). Second, DK has annotated expressions $e : \sigma$, in which the type annotation can be propagated inward the expression, as we will see shortly.

The definitions of types and contexts are the same as in OL. Thus, DK also shares the same well-formedness definition as in OL (**??**). We thus omit the definitions, but use $\vdash^{DK}$ to denote the corresponding judgment in DK.

2.3.3 Static Semantics

?? presents the typing rules for DK. The system uses bidirectional type checking to accommodate polymorphism. Traditionally, two modes are employed in bidirectional systems: the inference mode $\Psi \vdash^{DK} e \Rightarrow \sigma$, which takes a term e and produces a type σ, similar to the judgment $\Psi \vdash^{HM} e : \sigma$ or $\Psi \vdash^{OL} e : \sigma$ in previous systems; the checking mode $\Psi \vdash^{DK} e \Leftarrow \sigma$, which takes a term e and a type σ as input, and ensures that the term e checks against σ. We first discuss rules in the inference mode.

$\boxed{\Psi \vdash^{DK} e \Rightarrow \sigma}$ *(Type Inference)*

$$\frac{(x : \sigma) \in \Psi}{\Psi \vdash^{DK} x \Rightarrow \sigma}\ \textsc{dk-inf-var} \qquad \frac{}{\Psi \vdash^{DK} n \Rightarrow \mathsf{Int}}\ \textsc{dk-inf-int} \qquad \frac{\Psi \vdash^{DK} \tau_1 \rightarrow \tau_2 \qquad \Psi, x : \tau_1 \vdash^{DK} e \Rightarrow \tau_2}{\Psi \vdash^{DK} \lambda x.\, e \Rightarrow \tau_1 \rightarrow \tau_2}\ \textsc{dk-inf-lam}$$

$$\frac{\Psi \vdash^{DK} e_1 \Rightarrow \sigma \qquad \Psi \vdash^{DK} \sigma \rhd \sigma_1 \rightarrow \sigma_2 \qquad \Psi \vdash^{DK} e_2 \Leftarrow \sigma_1}{\Psi \vdash^{DK} e_1\, e_2 \Rightarrow \sigma_2}\ \textsc{dk-inf-app} \qquad \frac{\Psi \vdash^{DK} e \Leftarrow \sigma}{\Psi \vdash^{DK} e : \sigma \Rightarrow \sigma}\ \textsc{dk-inf-anno}$$

$\boxed{\Psi \vdash^{DK} e \Leftarrow \sigma}$ *(Type Checking)*

$$\frac{}{\Psi \vdash^{DK} n \Leftarrow \mathsf{Int}}\ \textsc{dk-chk-int} \qquad \frac{\Psi, x : \sigma_1 \vdash^{DK} e \Leftarrow \sigma_2}{\Psi \vdash^{DK} \lambda x.\, e \Leftarrow \sigma_1 \rightarrow \sigma_2}\ \textsc{dk-chk-lam} \qquad \frac{\Psi, a \vdash^{DK} e \Leftarrow \sigma}{\Psi \vdash^{DK} e \Leftarrow \forall a.\, \sigma}\ \textsc{dk-chk-gen}$$

$$\frac{\Psi \vdash^{DK} e \Rightarrow \sigma_1 \qquad \Psi \vdash^{DK} \sigma_1 <: \sigma_2}{\Psi \vdash^{DK} e \Leftarrow \sigma_2}\ \textsc{dk-chk-sub}$$

$\boxed{\Psi \vdash^{DK} \sigma_1 \rhd \sigma_2}$ *(Matching)*

$$\frac{\Psi \vdash^{DK} \tau \qquad \Psi \vdash^{DK} \sigma[a \mapsto \tau] \rhd \sigma_1 \rightarrow \sigma_2}{\Psi \vdash^{DK} \forall a.\, \sigma \rhd \sigma_1 \rightarrow \sigma_2}\ \textsc{dk-m-forall} \qquad \frac{}{\Psi \vdash^{DK} \sigma_1 \rightarrow \sigma_2 \rhd \sigma_1 \rightarrow \sigma_2}\ \textsc{dk-m-arr}$$

Figure 2.7: Static semantics of the Dunfield-Krishnaswami type system.

Type Inference. Rule DK-INF-VAR and rule DK-INF-INT are straightforward. To infer unannotated lambdas, rule DK-INF-LAM guesses a monotype. For an application $e_1\, e_2$, rule DK-INF-APP first infers the type σ of the expression e_1. Then, because e_1 is applied to an argument, the type σ is decomposed into a function type $\sigma_1 \to \sigma_2$, using the matching judgment (discussed shortly). Now since the function expects an argument of type σ_1, the rule proceeds by checking e_2 against σ_1. Similarly, for an annotated expression $e : \sigma$, rule DK-INF-ANNO simply checks e against σ. Both rules (rule DK-INF-APP and rule DK-INF-ANNO) have mode switched from inference to checking.

Type Checking. Now we turn to the checking mode. When an expression is checked against a type, the expression is expected to have that type. More importantly, the checking mode allows us to push the type information into the expressions.

Rule DK-CHK-INT checks literals again the integer type Int. Rule DK-CHK-LAM is where the system benefits from bidirectional type checking: the type information gets pushed inside an lambda. For an unannotated lambda abstraction $\lambda x.\, e$, recall that in the inference mode, we can only guess a monotype for x. With the checking mode, when $\lambda x.\, e$ is checked against $\sigma_1 \to \sigma_2$, we do not need to guess any type. Instead, x gets directly the (possibly polymorphic) argument type σ_1. Then the rule proceeds by checking e with σ_2, allowing the type information to be pushed further inside. Note how rule DK-CHK-LAM improves over HM and OL, by allowing lambda abstractions to have a polymorphic argument type without requiring type annotations.

Rule DK-CHK-GEN deals with a polymorphic type $\forall a.\, \sigma$, by putting the (fresh) type variable a into the context to check e against σ. Rule DK-CHK-SUB switches the mode from checking to inference: an expression e can be checked against σ_2, if e infers the type σ_1 and σ_1 is a subtype of σ_2.

Matching. In rule DK-INF-APP where we type-check an application $e_1\, e_2$, we derive that e_1 has type σ, but e_1 must have a function type so that it can be applied to an argument. The *matching* judgment instantiates σ into a function.

Matching has two straightforward rules: rule DK-M-FORALL instantiates a polymorphic type, by substituting a with a well-formed monotype τ, and continues matching on $\sigma[a \mapsto \tau]$; rule DK-M-ARR returns the function type directly.

In **?**, they use an *application judgment* instead of matching. The application judgment $\Psi \vdash^{DK} \sigma_1 \cdot e \Rrightarrow \sigma_2$, whose definition is given below, is interpreted as, when we apply an expression of type σ_1 to the expression e, we get a return type σ_2.

$\boxed{\Psi \vdash^{DK} \sigma_1 \cdot e \Rrightarrow \sigma_2}$ *(Application)*

$$\frac{\Psi \vdash^{DK} \tau \qquad \Psi \vdash^{DK} \sigma[a \mapsto \tau] \cdot e \Rrightarrow \sigma_1 \to \sigma_2}{\Psi \vdash^{DK} \forall a.\, \sigma \cdot e \Rrightarrow \sigma_1 \to \sigma_2}\ \text{DK-APP-FORALL} \qquad \frac{\Psi \vdash^{DK} e \Leftarrow \sigma_1}{\Psi \vdash^{DK} \sigma_1 \to \sigma_2 \cdot e \Rrightarrow \sigma_2}\ \text{DK-APP-ARR}$$

With the application judgment, rule DK-INF-APP is replaced by rule DK-INF-APP2.

$$\frac{\Psi \vdash^{DK} e_1 \Rightarrow \sigma \qquad \Psi \vdash^{DK} \sigma \cdot e_2 \Rrightarrow \sigma_2}{\Psi \vdash^{DK} e_1\, e_2 \Rightarrow \sigma_2}\ \text{DK-INF-APP2}$$

It can be easily shown that the presentation of rule DK-INF-APP with matching is equivalent to that of rule DK-INF-APP2 with the application judgment. Essentially, they both make sure that the expression being applied has arrow type $\sigma_1 \to \sigma_2$, and then check the argument against σ_1.

We prefer the presentation of rule DK-INF-APP with matching, as matching is a simple and standalone process whose purpose is clear. In contrast, it is relatively less comprehensible with rule DK-INF-APP2 and the application judgment, where all three forms of the judgment (inference, checking, application) are mutually dependent.

Subtyping. DK shares the same subtyping relation as of OL. We thus omit the definition and use $\Psi \vdash^{DK} \sigma_1 <: \sigma_2$ to denote the subtyping relation in DK.

2.3.4 Algorithmic Type System

? also presented a sound and complete bidirectional algorithmic type system. The key idea of the algorithm is using *ordered* algorithmic contexts for storing existential variables and their solutions. Comparing to the algorithm for HM, they argued that their algorithm is remarkably simple. The algorithm is later discussed and used in **??**, **??** and **??**. We will discuss more about it there.

Part II

Type Inference

3 Bidirectional Type Checking With The Application Mode

3.1 Introduction

3.1.1 Revisiting Bidirectional Type Checking

Traditional type checking rules can be heavyweight on annotations, in the sense that lambda-bound variables always need explicit annotations. As we have seen in ??, bidirectional type checking [?] provides an alternative, which allows types to propagate downward the syntax tree. For example, in the expression $(\lambda f : \mathsf{Int} \to \mathsf{Int}. f)\ (\lambda y. y)$, the type of y is provided by the type annotation on f. This is supported by the bidirectional typing rule DK-INF-APP for applications:

$$\frac{\Psi \vdash^{DK} e_1 \Rightarrow \sigma \qquad \Psi \vdash^{DK} \sigma \triangleright \sigma_1 \to \sigma_2 \qquad \Psi \vdash^{DK} e_2 \Leftarrow \sigma_1}{\Psi \vdash^{DK} e_1\ e_2 \Rightarrow \sigma_2}\ \text{DK-INF-APP}$$

Specifically, if we know that the type of e_1 is a function from $\sigma_1 \to \sigma_2$, we can check that e_2 has type σ_1. Notice that here the type information flows from functions to arguments.

One guideline for designing bidirectional type checking rules [?] is to distinguish introduction rules from elimination rules. Constructs which correspond to introduction forms are *checked* against a given type, while constructs corresponding to elimination forms *infer* (or synthesize) their types. For instance, under this design principle, the introduction rule for literals is supposed to be in checking mode, as in the rule rule DK-CHK-INT:

$$\frac{}{\Psi \vdash^{DK} n \Leftarrow \mathsf{Int}}\ \text{DK-CHK-INT}$$

Unfortunately, this means that the trivial program 1 cannot type-check, which in this case has to be rewritten to $1 : \mathsf{Int}$.

In this particular case, bidirectional type checking goes against its original intention of removing burden from programmers, since a seemingly unnecessary annotation is needed.

Therefore, in practice, bidirectional type systems do not strictly follow the guideline, and usually have additional inference rules for the introduction form of constructs. For literals, the corresponding rule is rule DK-INF-INT.

$$\frac{}{\Psi \vdash^{DK} n \Rightarrow \mathsf{Int}}\ \text{DK-INF-INT}$$

Now we can type check 1, but the price to pay is that two typing rules for literals are needed. Worse still, the same criticism applies to other constructs (e.g., pairs). This shows one drawback of bidirectional type checking: often to minimize annotations, many rules are duplicated for having both inference and checking mode, which scales up with the typing rules in a type system.

3.1.2 Type Checking with The Application Mode

We propose a variant of bidirectional type checking with a new *application mode* (unrelated to the application judgment in DK). The application mode preserves the advantage of bidirectional type checking, namely many redundant annotations are removed, while certain programs can type check with even fewer annotations. Also, with our proposal, the inference mode is a special case of the application mode, so it does not produce duplications of rules in the type system. Additionally, the checking mode can still be *easily* combined into the system. The essential idea of the application mode is to enable the type information flow in applications to propagate from arguments to functions (instead of from functions to arguments as in traditional bidirectional type checking).

To motivate the design of bidirectional type checking with an application mode, consider the simple expression

```
(\x. x) 1
```

This expression cannot type check in traditional bidirectional type checking, because unannotated abstractions, as a construct which correspond to introduction forms, only have a checking mode, so annotations are required [1]. For example, `((\x. x) :` $\mathsf{Int} \to \mathsf{Int}$`) 1`.

In this example we can observe that if the type of the argument is accounted for in inferring the type of `\x. x`, then it is actually possible to deduce that the lambda expression has type $\mathsf{Int} \to \mathsf{Int}$, from the argument `1`.

[1] It type-checks in DK, because in DK rules for lambdas are duplicated for having both inference (integrated with type inference techniques) and checking mode.

The Application Mode. If types flow from the arguments to the function, an alternative idea is to push the type of the arguments into the typing of the function, as follows,

$$\textsc{App}\quad \frac{\Psi \vdash e_2 \Rightarrow \sigma_1 \qquad \Psi; \Sigma, \sigma_1 \vdash e_1 \Rightarrow \sigma \to B}{\Psi; \Sigma \vdash e_1\, e_2 \Rightarrow B}$$

In this rule, there are two kinds of judgments. The first judgment is just the usual inference mode, which is used to infer the type of the argument e_2. The second judgment, the application mode, is similar to the inference mode, but it has an additional context Σ. The context Σ is a stack that tracks the types of the arguments of outer applications. In the rule for application, the type of the argument e_2 synthesizes its type σ_1, which then is pushed into the application context Σ for inferring the type of e_1. Applications are themselves in the application mode, since they can be in the context of an outer application.

Lambda expressions can now make use of the application context, leading to the following rule:

$$\textsc{Lam}\quad \frac{\Psi, x : \sigma; \Sigma \vdash e \Rightarrow B}{\Psi; \Sigma, \sigma \vdash \lambda x.\, e \Rightarrow \sigma \to B}$$

The type σ that appears last in the application context serves as the type for x, and type checking continues with a smaller application context and $x : \sigma$ in the typing context. Therefore, using the rule rule App and rule Lam, the expression $(\lambda x.\, x)\ 1$ can type-check without annotations, since the type Int of the argument 1 is used as the type of the binding x.

Note that, since the examples so far are based on simple types, obviously they can be solved by integrating type inference and relying on techniques like unification or constraint solving (as in DK). However, here the point is that the application mode helps to reduce the number of annotations *without requiring such sophisticated techniques*. Also, the application mode helps with situations where those techniques cannot be easily applied, such as type systems with subtyping.

Interpretation of the Application Mode. As we have seen, the guideline for designing bi-directional type checking [?], based on introduction and elimination rules, is often not enough in practice. This leads to extra introduction rules in the inference mode. The application mode does not distinguish between introduction rules and elimination rules. Instead, to decide whether a rule should be in inference or application mode, we need to think whether the expression can be applied or not. Variables, lambda expressions and applications are all examples of expressions that can be applied, and they should have application mode rules. However literals or pairs cannot be applied and should have inference rules. For example, type checking pairs would simply have the inference mode. Nevertheless elimination rules of pairs could have non-empty application contexts (see Section **??** for details). In the application mode, arguments are always inferred first in applications and propagated through application contexts. An empty application context means that an expression is not being applied to anything, which allows us to model the inference mode as a particular case[2].

Partial Type Checking. The inference mode synthesizes the type of an expression, and the checked mode checks an expression against some type. A natural question is how do these modes compare to application mode. An answer is that, in some sense: the application mode is stronger than inference mode, but weaker than checked mode. Specifically, the inference mode means that we know nothing about the type an expression before hand. The checked mode means that the whole type of the expression is already known before hand. With the application mode we know some partial type information about the type of an expression: we know some of its argument types (since it must be a function type when the application context is non-empty), but not the return type.

Instead of nothing or all, this partialness gives us a finer grain notion on how much we know about the type of an expression. For example, assume $e : \sigma_1 \to \sigma_2 \to \sigma_3$. In the inference mode, we only have e. In the checked mode, we have both e and $\sigma_1 \to \sigma_2 \to \sigma_3$. In the application mode, we have e, and maybe an empty context (which degenerates into inference mode), or an application context σ_1 (we know the type of first argument), or an application context σ_1, σ_2 (we know the types of both arguments).

Trade-offs. Note that the application mode is *not* conservative over traditional bidirectional type checking due to the different information flow. However, it provides a new design choice for type inference/checking algorithms, especially for those where the information about arguments is useful. Therefore we next discuss some benefits of the application mode for two interesting cases where functions are either variables; or lambda (or type) abstractions.

[2] Although the application mode generalizes the inference mode, we refer to them as two different modes. Thus the variant of bi-directional type checking in this paper is interpreted as a type system with both *inference* and *application* modes.

3.1.3 Benefits of Information Flowing from Arguments to Functions

Local Constraint Solver for Function Variables. Many type systems, including type systems with *implicit polymorphism* and/or *static overloading*, need information about the types of the arguments when type checking function variables. For example, in conventional functional languages with implicit polymorphism, function calls such as $(\mathsf{id}\ 1)$ where $\mathsf{id} : \forall a. (a \to a)$, are *pervasive*. In such a function call the type system must instantiate a to Int. Dealing with such implicit instantiation gets trickier in systems with *higher-rank types*. For example, **?** require additional syntactic forms and relations, whereas DK add a special purpose matching or the application judgment.

With the application mode, all the type information about the arguments being applied is available in application contexts and can be used to solve instantiation constraints. To exploit such information, the type system employs a special subtyping judgment called *application subtyping*, with the form $\Sigma \vdash \sigma_1 <: \sigma_2$. Unlike conventional subtyping, computationally Ψ and σ_1 are interpreted as inputs and σ_2 as output. In above example, we have that $\mathsf{Int} \vdash \forall a. a \to a <: \sigma$ and we can determine that $a = \mathsf{Int}$ and $\sigma = \mathsf{Int} \to \mathsf{Int}$. In this way, type system is able to solve the constraints *locally* according to the application contexts since we no longer need to propagate the instantiation constraints to the typing process.

Declaration Desugaring for Lambda Abstractions. An interesting consequence of the usage of an application mode is that it enables the following **let** sugar:

$$\mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2 \rightsquigarrow (\lambda x.\, e_2)\, e_1$$

Such syntactic sugar for **let** is, of course, standard. However, in the context of implementations of typed languages it normally requires extra type annotations or a more sophisticated type-directed translation. Type checking $(\lambda x.\, e_2)\, e_1$ would normally require annotations (for example a higher-rank type annotation for x as in OL and DK), or otherwise such annotation should be inferred first. Nevertheless, with the application mode no extra annotations/inference is required, since from the type of the argument e_1 it is possible to deduce the type of x. Generally speaking, with the application mode *annotations are never needed for applied lambdas*. Thus **let** can be the usual sugar from the untyped lambda calculus, including HM-style **let** expression and even type declarations.

3.1.4 Type Inference of Higher-rank Types

We believe the application mode can be integrated into many traditional bidirectional type systems. In this chapter, we focus on integrating the application mode into a bidirectional type system with higher-rank types. Our paper [?] includes another application to a variant of System F.

Consider again the motivation example used in **??**:

```
(\f. (f 1, f 'a')) (\x. x)
```

which is not typeable in HM, but can be rewritten to include type annotations in OL and DK. For example, both in OL and DK we can write:

```
(\f:(∀a. a → a). (f 1, f 'c')) (\x. x)
```

However, although some redundant annotations are removed by bidirectional type checking, the burden of inferring higher-rank types is still carried by programmers: they are forced to add polymorphic annotations to help with the type derivation of higher-rank types. For the above example, the type annotation is still *provided by programmers*, even though the necessary type information can be derived intuitively without any annotations: `f` is applied to `\x. x`, which is of type `∀a. a → a`.

Type Inference for Higher-rank Types with the Application Mode. Using our bidirectional type system with an application mode, the original expression can type check without annotations or rewrites: `(\f. (f 1, f 'c')) (\x. x)`.

This result comes naturally if we allow type information flow from arguments to functions. For inferring polymorphic types for arguments, we use *generalization*. In the above example, we first infer the type `∀a. a → a` for the argument, then pass the type to the function. A nice consequence of such an approach is that, as mentioned before, HM-style polymorphic **let** expressions are simply regarded as syntactic sugar to a combination of lambda/application:

$$\textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \rightsquigarrow (\lambda x.\, e_2)\, e_1$$

With this approach, nested lets can lead to types which are *more general* than HM. For example, consider the following expression:

```
let s = \x. x in let t = \y. s in e
```

The type of `s` is `∀a. a → a` after generalization. Because `t` returns `s` as a result, we might expect `t: ∀b. b → (∀a. a → a)`, which is what our system will return. However, HM will return type `t: ∀b. ∀a. b → (a → a)`, as it can only return rank 1 types, which is less general than the previous one according to the subtyping relation for polymorphic types in OL (**??**).

Conservativity over the Hindley-Milner Type System. Our type system is a conservative extension over HM, in the sense that every program that can type-check in HM is accepted in our type system. This result is not surprising: after desugaring **let** into a lambda and an application, programs remain typeable.

Comparing Predicative Higher-rank Type Inference Systems. We will give a full discussion and comparison of related work in Section **??**. Among those works, we believe DK and the work by **?** are the most closely related work to our system. Both their systems and ours are based on a *predicative* type system: universal quantifiers can only be instantiated by monotypes. So we would like to emphasize our system's properties in relation to those works. In particular, here we discuss two interesting differences, and also briefly (and informally) discuss how the works compare in terms of expressiveness.

1) Inference of higher-rank types. In both works, every polymorphic type inferred by the system must correspond to one annotation provided by the programmer. However, in our system, some higher-rank types can be inferred from the expression itself without any annotation. The motivating expression above provides an example of this.

2) Where are annotations needed? Since type annotations are useful for inferring higher rank types, a clear answer to the question where annotations are needed is necessary so that programmers know when they are required to write annotations. To this question, previous systems give a concrete answer: only on the binding of polymorphic types. Our answer is slightly different: only on the bindings of polymorphic types in abstractions *that are not applied to arguments*. Roughly speaking this means that our system ends up with fewer or smaller annotations.

3) Expressiveness. Based on these two answers, it may seem that our system should accept all expressions that are typeable in their system. However, this is not true because the application mode is *not* conservative over traditional bi-directional type checking. Consider the expression:

```
(\f : (∀a. a → a) → (nat, char). f) (\g. (g 1, g 'a'))
```

which is typeable in their system. In this case, even if g is a polymorphic binding without a type annotation the expression can still type-check. This is because the original application rule propagates the information from the outer binding into the inner expressions. Note that the fact that such expression type-checks does not contradict their guideline of providing type annotations for every polymorphic binder. Programmers that strictly follow their guideline can still add a polymorphic type annotation for g. However it does mean that it is a little harder to understand where annotations for polymorphic binders can be *omitted* in their system. This requires understanding how the applications in checked mode operate.

In our system the above expression is not typeable, as a consequence of the information flow in the application mode. However, following our guideline for annotations leads to a program that can be type-checked with a smaller annotation:

```
(\f. f) (\g : (∀a. a → a). (g 1, g 'a')).
```

This means that our work is not conservative over their work, which is due to the design choice of the application typing rule. Nevertheless, we can always rewrite programs using our guideline, which often leads to fewer/smaller annotations.

4 Unification with Promotion

Part III

Extensions

5 Higher Rank Gradual Types

6 Dependent Types

Part IV

Related and Future Work

7 Related Work

8 Future Work

Part V

Epilogue

9 Conclusion

Part VI

Technical Appendix