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:

Part II

- Chapter 3 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.
- Chapter 4 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.

Part III

- Chapter 5 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 Siek and Taha [2007] 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 [Siek et al. 2015], and is type-safe by a type-directed translation to λB [Ahmed et al. 2009].
  - We present a sound and complete bidirectional algorithm for implementing the declarative system based on the design principle of Garcia and Cimini [2015].
- Chapter 6 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] For convenience, whenever possible, definitions, lemmas and theorems have hyperlinks (click ☞) to their Coq counterparts.

1.2 Organization

This thesis is largely based on the publications by the author [Xie et al. 2018, 2019a,b; Xie and Oliveira 2017, 2018], as indicated below.

Chapter 3: Ningning Xie and Bruno C. d. S. Oliveira. 2018. "Let Arguments Go First". In *European Symposium on Programming (ESOP)*.

Chapter 4: 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)*.

Chapter 5: 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)*.

Chapter 6: 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

2.1 The Hindley-Milner Type System

The Hindley-Milner type system, hereafter referred to as HM, is a polymorphic type discipline first discovered in Hindley [1969], later rediscovered by Milner [1978], and also closely formalized by Damas and Milner [1982].

2.1.1 Syntax

The syntax of HM is given in Figure 2.1. The expressions e include variables x, literals n, lambda abstractions $\lambda x.\, e$, applications $e_1\ e_2$ and $\textbf{let}\, x = e_1\, \textbf{in}\, e_2$. 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 τ. A polymorphic type is a sequence of universal quantifications (which can be empty) followed by a monomorphic type τ, which can be integer Int, type variable a and function $\tau_1 \to \tau_2$. A context Ψ tracks the type information for variables.

2.1.2 Static Semantics

The 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 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 *monomorphic* type τ_1 as the type of x, and derives the type τ_2 as the body e, returning a function $\tau_1 \to \tau_2$. The function type is then eliminated by applications. In rule HM-APP, the type of the parameter must match the argument's type $t1$, and the 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 $e_1 : \sigma$ into the context to type-check the body e_2.

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

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

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

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

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

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

Rule HM-GEN and rule HM-INST correspond to type variable *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 *monomorphic* types.

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.

The judgment $\vdash^{HM} \sigma_1 <: \sigma_2$, given in Figure 2.2, 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.

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

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

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

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.

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 $\vdash^{HM} \sigma' <: \sigma$.*

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 with *rank 1*. That is, universal quantifiers only appear at the top level. Polymorphic types have *higher-rank*, if universal quantifiers can appear anywhere in a type.

Odersky and Läufer [1996] proposed a type system, hereafter referred to as OL, where lambda abstractions are allowed to have *higher-rank* types as type annotations, while unannotated lambda abstractions can still have only monotypes.

2.2.1 Higher-rank Types

We define the rank of types as follows.

Definition 1. The *rank* of a type is the depth at which universal quantifiers appear contravariantly [Kfoury and Tiuryn 1992]. Formally,

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

Expressions	e	$::=$	$x \mid n \mid \lambda x : \sigma.\, e \mid \lambda x.\, e \mid e_1\, e_2 \mid \textbf{let}\, x = e_1\, \textbf{in}\, e_2$
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.

From the definition, we can see that monotypes always have rank 0, and the polymorphic types σ in HM (Figure 2.1) has at most rank 1.

2.2.2 Syntax

The syntax of OL is given in Figure 2.3. Comparing to HM (Figure 2.1), we observe the following differences.

First, expressions e have both unannotated lambda abstractions $\lambda x.\, e$, and 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 can be polymorphic in OL.

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

Finally, in addition to variable types, the context Ψ now also keeps track of type variables. This representation is not necessary for the system, but it sets us well for future specification for algorithmic systems. We use the context to check the well-formedness of types, as we will see later.

2.2.3 Static Semantics

The static semantics of OL is given in Figure 2.4.

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 type annotation is required. Rule OL-APP, rule OL-LET and rule OL-GEN are similar as HM, except that polymorphic types may appear as the result in all rules. The subsumption rule OL-SUB is crucial for OL, which allows an expression of type σ_1 to have a subtype σ_2. Note that the instantiation rule HM-INST in HM is a special case of rule OL-SUB.

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

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

$$\textsc{ol-app}\ \frac{\Psi \vdash^{OL} e_1 : \sigma_1 \to \sigma_2 \qquad \Psi \vdash^{OL} e_2 : \sigma_1}{\Psi \vdash^{OL} e_1\, e_2 : \sigma_2} \qquad \textsc{ol-let}\ \frac{\Psi \vdash^{OL} e_1 : \sigma_1 \qquad \Psi, x:\sigma_1 \vdash^{OL} e_2 : \sigma_2}{\Psi \vdash^{OL} \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \sigma_2}$$

$$\textsc{ol-gen}\ \frac{\Psi, a \vdash^{OL} e : \sigma}{\Psi \vdash^{OL} e : \forall a.\, \sigma} \qquad \textsc{ol-sub}\ \frac{\Psi \vdash^{OL} e : \sigma_1 \qquad \Psi \vdash \sigma_1 <: \sigma_2}{\Psi \vdash^{OL} e : \sigma_2}$$

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

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

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

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

$$\textsc{ol-wf-int}\ \frac{}{\Psi \vdash^{OL} \mathsf{Int}} \qquad \textsc{ol-wf-tvar}\ \frac{a \in \Psi}{\Psi \vdash^{OL} a} \qquad \textsc{ol-wf-arrow}\ \frac{\Psi \vdash^{OL} \sigma_1 \qquad \Psi \vdash^{OL} \sigma_2}{\Psi \vdash^{OL} \sigma_1 \to \sigma_2} \qquad \textsc{ol-wf-forall}\ \frac{\Psi, a \vdash^{OL} \sigma}{\Psi \vdash^{OL} \forall a.\, \sigma}$$

Figure 2.4: Static semantics of the Odersky-Läufer type system.

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.

Predicativity

2.3 The Dunfield-Krishnaswami Type System

Part II

Type Inference

3 Type Inference With The Application Mode

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

Bibliography

[Citing pages are listed after each reference.]

Amal Ahmed, Robert Bruce Findler, Jacob Matthews, and Philip Wadler. 2009. Blame for All. In *Proceedings for the 1st Workshop on Script to Program Evolution (STOP '09)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/1570506.1570507 [cited on page 4]

Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*. Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176 [cited on page 7]

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 303–315. https://doi.org/10.1145/2676726.2676992 [cited on page 4]

J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60. [cited on page 7]

Assaf J Kfoury and Jerzy Tiuryn. 1992. Type reconstruction in finite rank fragments of the second-order λ-calculus. *Information and computation* 98, 2 (1992), 228–257. [cited on page 9]

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375. [cited on page 7]

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. Association for Computing Machinery, New York, NY, USA, 54–67. https://doi.org/10.1145/237721.237729 [cited on page 9]

Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*. Springer-Verlag, Berlin, Heidelberg, 2–27. [cited on page 4]

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. [cited on page 4]

Ningning Xie, Xuan Bi, and Bruno C d S Oliveira. 2018. Consistent Subtyping for All. In *European Symposium on Programming*. Springer, 3–30. [cited on page 5]

Ningning Xie, Xuan Bi, Bruno C. D. S. Oliveira, and Tom Schrijvers. 2019a. Consistent Subtyping for All. *ACM Transactions on Programming Languages and Systems* 42, 1, Article 2 (Nov. 2019), 79 pages. `https://doi.org/10.1145/3310339` [cited on page 5]

Ningning Xie, Richard A. Eisenberg, and Bruno C. d. S. Oliveira. 2019b. Kind Inference for Datatypes. *Proc. ACM Program. Lang.* 4, POPL, Article 53 (Dec. 2019), 28 pages. `https://doi.org/10.1145/3371121` [cited on page 5]

Ningning Xie and Bruno C d S Oliveira. 2017. Towards Unification for Dependent Types. In *Draft Proceedings of the 18th Symposium on Trends in Functional Programming (TFP '18)*. Extended abstract. [cited on page 5]

Ningning Xie and Bruno C d S Oliveira. 2018. Let Arguments Go First. In *European Symposium on Programming*. Springer, 272–299. [cited on page 5]

Part VI

Technical Appendix