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 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 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 $\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$. 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 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.

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-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.

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

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

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 Figure 2.1 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 [Damas and Milner 1982], which has served as the basis for the type inference algorithm for many other systems [Jones et al. 2007; Odersky and Läufer 1996], including the system presented in Chapter 3. We will discuss more about it in Chapter 3.

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

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.

is, universal quantifiers only appear at the top level. Polymorphic types are of *higher-rank*, if universal quantifiers can appear anywhere in a type. Odersky and Läufer [1996] 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.

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 [Kfoury and Tiuryn 1992]. Formally,

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

Below we give some examples:

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

From the definition, we can see that monotypes always have rank 0, and the polymorphic types in HM (σ in 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, 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.

$\boxed{\Psi \vdash^{OL} \sigma}$ *(Type 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: Well-formedness of types in the Odersky-Läufer type system.

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 Odersky and Läufer [1996], 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 Figure 2.4. All rules are straightforward.

2.2.3 Static Semantics

The static semantics of OL is given in Figure 2.5.

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

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

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

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

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

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

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

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

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

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

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 [Wells 1999]. 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 as long as the polymorphic type annotations are given explicitly.

The Dunfield-Krishnaswami type system [Dunfield and Krishnaswami 2013], hereafter refered to as DM, give a bidirectional account of higher-rank polymorphism, where type

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

information can be propagated through the syntax tree. Therefore, it is possible for a variable bound in a lambda abstraction to get a polymorphic type without requiring explicit type annotations.

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 [Pierce and Turner 2000]. 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 bi- directional 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 [Davies and Pfenning 2000; Dunfield and Pfenning 2004], systems with dependent types [Asperti et al. 2012; Coquand 1996; Löh et al. 2010; Xi and Pfenning 1999], 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 Jones et al. [2007].

2.3.2 Syntax

The syntax of the DK is given in Figure 2.6. Comparing to OL, the definition of expressions slightly differs. First, the expressions e in DK have no let expressions. Dunfield and Krishnaswami [2013] 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 definitions as in OL (Figure 2.4). We thus omit the definitions, but use $\vdash^{DK}$ to denote the corresponding judgments in DK.

2.3.3 Static Semantics

Figure 2.7 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 $\Psi \vdash e : \sigma$ in previous systems; the new 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.

Type Inference. Rule DK-INF-VAR and rule DK-INF-INT are straightforward. To infer unannotated lambdas, rule DK-INF-LAM guesses a monotype. Rule DK-INF-APP first infers the type σ of the expression e_1. Then, because e_1 is applied as a function, 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 the 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

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

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

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

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

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

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

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

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

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

mode, we can only guess a monotype for x. With the checking mode, when $\lambda x.\, e$ is checked against $\sigma_1 \to \sigma_2$, instead of a guessed type, x gets directly the argument type σ_1, which may be a polymorphic type. Then the rule proceeds by checking e with σ_2, allowing the type information to be pushed further inside.

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 $e_1\, e_2$, 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 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 Dunfield and Krishnaswami [2013], they use an *application* judgment instead of matching. The 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)*

DK-APP-FORALL

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

DK-APP-ARR

$$\frac{\Psi \vdash^{DK} e \Leftarrow \sigma_1}{\Psi \vdash^{DK} \sigma_1 \to \sigma_2 \cdot e \Rrightarrow \sigma_2}$$

with the application judgment, rule DK-INF-APP is changed to rule DK-INF-APP2.

DK-INF-APP2

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

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 an 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, where matching is a simple and standalone process whose purpose is clear. In contrast, it is 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.

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]

Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 2012. A Bi-Directional Refinement Algorithm for the Calculus of (Co) Inductive Constructions. *Logical Methods in Computer Science* 8 (2012), 1–49. [cited on page 14]

Thierry Coquand. 1996. An algorithm for type-checking dependent types. *Science of Computer Programming* 26, 1-3 (1996), 167–177. [cited on page 14]

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 pages 7 and 9]

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. *SIGPLAN Not.* 35, 9 (Sept. 2000), 198–208. https://doi.org/10.1145/357766.351259 [cited on page 14]

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 429–442. https://doi.org/10.1145/2500365.2500582 [cited on pages 13, 15, and 17]

Joshua Dunfield and Frank Pfenning. 2004. Tridirectional Typechecking. *SIGPLAN Not.* 39, 1 (Jan. 2004), 281–292. https://doi.org/10.1145/982962.964025 [cited on page 14]

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]

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of functional programming* 17, 1 (2007), 1–82. [cited on pages 9 and 14]

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 10]

Andres Löh, Conor McBride, and Wouter Swierstra. 2010. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta informaticae* 102, 2 (2010), 177–207. [cited on page 14]

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 pages 9, 10, and 11]

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. https://doi.org/10.1145/345099.345100 [cited on page 14]

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]

Joe B Wells. 1999. Typability and Type Checking in System F are Equivalent and Undecidable. *Annals of Pure and Applied Logic* 98, 1-3 (1999), 111–156. [cited on page 13]

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560 [cited on page 14]

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