

# 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

DECLARATION	I
ACKNOWLEDGMENTS	III
LIST OF FIGURES	VII
LIST OF TABLES	IX
I PROLOGUE	1
1 INTRODUCTION	3
1.1 Contributions . . . . .	3
1.2 Organization . . . . .	5
2 BACKGROUND	7
2.1 The Hindley-Milner Type System . . . . .	7
2.1.1 Syntax . . . . .	7
2.1.2 Static Semantics . . . . .	7
2.1.3 Principal Type Scheme . . . . .	8
2.2 The Odersky-Läufer Type System . . . . .	9
2.2.1 Higher-rank Types . . . . .	9
2.2.2 Syntax . . . . .	10
2.2.3 Static Semantics . . . . .	10
2.2.4 Predicativity . . . . .	11
2.2.5 Relating to HM . . . . .	13
2.3 The Dunfield-Krishnaswami Type System . . . . .	13
2.3.1 Bi-directional Type Checking . . . . .	13
2.3.2 Syntax . . . . .	13
	v

## *Contents*

II	TYPE INFERENCE	15
3	TYPE INFERENCE WITH THE APPLICATION MODE	17
4	UNIFICATION WITH PROMOTION	19
III	EXTENSIONS	21
5	HIGHER RANK GRADUAL TYPES	23
6	DEPENDENT TYPES	25
IV	RELATED AND FUTURE WORK	27
7	RELATED WORK	29
8	FUTURE WORK	31
V	EPILOGUE	33
9	CONCLUSION	35
	BIBLIOGRAPHY	37
VI	TECHNICAL APPENDIX	41

## LIST OF FIGURES

2.1	Syntax and static semantics of the Hindley-Milner type system. . . . .	8
2.2	Subtyping in the Hindley-Milner type system. . . . .	9
2.3	Syntax of the Odersky-Läufer type system. . . . .	10
2.4	Well-formedness of types and contexts in the Odersky-Läufer type system. .	11
2.5	Static semantics of the Odersky-Läufer type system. . . . .	12
2.6	Syntax of the algorithmic Dunfield-Krishnaswami Type System . . . . .	14



## 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 the 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  $\lambda B$  [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.<sup>1</sup>

---

<sup>1</sup>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 **let**  $x = e_1$  **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  $\sigma$  and monomorphic types  $\tau$ . A polymorphic type is a sequence of universal quantifications (which can be empty) followed by a monomorphic type  $\tau$ , which can be integer  $\text{Int}$ , type variable  $a$  and function  $\tau_1 \rightarrow \tau_2$ .

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

#### 2.1.2 STATIC SEMANTICS

The typing judgment  $\Psi \vdash^{HM} e : \sigma$  derives the type  $\sigma$  of the expression  $e$  under the context  $\Psi$ . 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  $\tau_1$  as the type of  $x$ , and derives the type  $\tau_2$  as the body  $e$ , returning a function  $\tau_1 \rightarrow \tau_2$ . The function type is then eliminated by applications. In rule **HM-APP**, the type of the parameter must match the argument's type  $\tau_1$ , and the application returns type  $\tau_2$ .

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  $\sigma$ , and the rule adds  $e_1 : \sigma$  into the context to type-check the body  $e_2$ .

## 2 Background

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	$\sigma ::= \forall \bar{a}^i. \tau$
Monotypes	$\tau ::= \mathbf{Int} \mid a \mid \tau_1 \rightarrow \tau_2$
Contexts	$\Psi ::= \bullet \mid \Psi, x : \sigma$

  

$\Psi \vdash^{HM} e : \sigma$

(Typing)

  

$\frac{\text{HM-VAR} \quad (x : \sigma) \in \Psi}{\Psi \vdash^{HM} x : \sigma}$	$\frac{\text{HM-INT}}{\Psi \vdash^{HM} n : \mathbf{Int}}$	$\frac{\text{HM-LAM} \quad \Psi, x : \tau_1 \vdash^{HM} e : \tau_2}{\Psi \vdash^{HM} \lambda x. e : \tau_1 \rightarrow \tau_2}$
$\frac{\text{HM-APP} \quad \Psi \vdash^{HM} e_1 : \tau_1 \rightarrow \tau_2 \quad \Psi \vdash^{HM} e_2 : \tau_1}{\Psi \vdash^{HM} e_1 e_2 : \tau_2}$	$\frac{\text{HM-LET} \quad \Psi \vdash^{HM} e_1 : \sigma \quad \Psi, x : \sigma \vdash^{HM} e_2 : \tau}{\Psi \vdash^{HM} \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau}$	
$\frac{\text{HM-GEN} \quad \bar{a}^i \notin \text{FV}(\Psi) \quad \Psi \vdash^{HM} e : \tau}{\Psi \vdash^{HM} e : \forall \bar{a}^i. \tau}$	$\frac{\text{HM-INST} \quad \Psi \vdash^{HM} e : \forall \bar{a}^i. \tau}{\Psi \vdash^{HM} e : \tau[\bar{a}_i \mapsto \bar{\tau}_i^i]}$	

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  $\bar{a}^i$  which are not bound in the type context  $\Psi$ . 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  $\sigma_1$  is a subtype of  $\sigma_2$ . The subtyping relation indicates that  $\sigma_1$  is more *general* than  $\sigma_2$ : for any instantiation of  $\sigma_2$ , we can find an instantiation of  $\sigma_1$  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  $\sigma_1$  (up to  $\alpha$ -renaming). In this case, if  $\sigma_1$  is still a subtype of  $\sigma_2$ , we are sure then whatever  $a$  can be instantiated to,  $\sigma_1$  can be instantiated to match  $\sigma_2$ .

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

$$\begin{array}{c}
 \text{HM-S-REFL} \\
 \hline
 \vdash^{HM} \tau <: \tau
 \end{array}
 \quad
 \begin{array}{c}
 \text{HM-S-FORALLR} \\
 a \notin \text{FV}(\sigma_1) \quad \vdash^{HM} \sigma_1 <: \sigma_2 \\
 \hline
 \vdash^{HM} \sigma_1 <: \forall a. \sigma_2
 \end{array}
 \quad
 \begin{array}{c}
 \text{HM-S-FORALLL} \\
 \vdash^{HM} \sigma_1[a \mapsto \tau] <: \sigma_2 \\
 \hline
 \vdash^{HM} \forall a. \sigma_1 <: \sigma_2
 \end{array}$$

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  $\sigma'$  such that  $\Psi \vdash^{HM} e : \sigma'$ , and for all  $\sigma$  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 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, 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,

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

## 2 Background

Expressions	$e$	$::=$	$x \mid n \mid \lambda x : \sigma. e \mid \lambda x. e \mid e_1 e_2 \mid \text{let } x = e_1 \text{ in } e_2$
Types	$\sigma$	$::=$	$\text{Int} \mid a \mid \sigma_1 \rightarrow \sigma_2 \mid \forall a. \sigma$
Monotypes	$\tau$	$::=$	$\text{Int} \mid a \mid \tau_1 \rightarrow \tau_2$
Contexts	$\Psi$	$::=$	$\bullet \mid \Psi, x : \sigma \mid \Psi, a$

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  $\sigma$  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  $\sigma$  is a polymorphic type. Thus unlike HM, the argument type for a function can be polymorphic in OL.

Second, the polymorphic types  $\sigma$  now include integers  $\text{Int}$ , type variables  $a$ , functions  $\sigma_1 \rightarrow \sigma_2$  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 remains the same. Note that polymorphic types still subsume monotypes.

Finally, in addition to variable types, the contexts  $\Psi$  now also keep track of type variables. Moreover, contexts are *ordered*, in the sense that in a well-formed context, the type of a variable cannot refer to type variables that appear after the variable in the context. This form of contexts can also be adopted in an alternative representation for HM. In HM formalized in the previous section, contexts do not track type variables; instead, the system 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. Also, 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.

The well-formedness rules for OL are given in Figure 2.4. A type is well-formed if all its free type variables are bound in the context (rule [OL-WF-TVAR](#)), and a context is well-formed if all its types are well-formed in their corresponding prefix context (rule [OL-CWF-VAR](#)).

### 2.2.3 STATIC SEMANTICS

The static semantics of OL is given in Figure 2.5.



$\boxed{\Psi \vdash^{OL} \sigma}$		(Type Well-formedness)	
OL-WF-INT	OL-WF-TVAR	OL-WF-ARROW	OL-WF-FORALL
$\frac{}{\Psi \vdash^{OL} \text{Int}}$	$\frac{a \in \Psi}{\Psi \vdash^{OL} a}$	$\frac{\Psi \vdash^{OL} \sigma_1 \quad \Psi \vdash^{OL} \sigma_2}{\Psi \vdash^{OL} \sigma_1 \rightarrow \sigma_2}$	$\frac{\Psi, a \vdash^{OL} \sigma}{\Psi \vdash^{OL} \forall a. \sigma}$
$\boxed{\Psi \text{ ok}}$		(Context Well-formedness)	
OL-CWF-EMPTY	OL-CWF-VAR	OL-CWF-TVAR	
$\frac{}{\bullet \text{ ok}}$	$\frac{\Psi \text{ ok} \quad \Psi \vdash \sigma}{\Psi, x : \sigma \text{ ok}}$	$\frac{\Psi \text{ ok}}{\Psi, a \text{ ok}}$	

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

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** are similar as HM, except that polymorphic types may appear in all return types. Rule **OL-GEN** shows the generalization rule, we put a new type variable  $a$  into the context, and the return type  $\sigma$  is then generalized over  $a$ , returning  $\forall a. \sigma$ .

The subsumption rule **OL-SUB** is crucial for OL, which allows an expression of type  $\sigma_1$  to have type  $\sigma_2$  with  $\Psi \vdash \sigma_1 <: \sigma_2$ . Note that the instantiation rule **HM-INST** in HM is a special case of rule **OL-SUB**, as we have  $\forall \bar{a}^i. \tau <: \tau[\bar{a}_i \mapsto \bar{\tau}_i^i]$  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.

#### 2.2.4 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

## 2 Background

$\Psi \vdash^{OL} e : \sigma$

*(Typing)*

$$\frac{\text{OL-VAR} \quad (x : \sigma) \in \Psi}{\Psi \vdash^{OL} x : \sigma}$$

$$\frac{\text{OL-INT}}{\Psi \vdash^{OL} n : \text{Int}}$$

$$\frac{\text{OL-LAMANN} \quad \Psi, x : \sigma_1 \vdash^{OL} e : \sigma_2}{\Psi \vdash^{OL} \lambda x : \sigma_1. e : \sigma_1 \rightarrow \sigma_2}$$

$$\frac{\text{OL-LAM} \quad \Psi \vdash^{OL} \tau \quad \Psi, x : \tau \vdash^{OL} e : \sigma}{\Psi \vdash^{OL} \lambda x. e : \tau \rightarrow \sigma}$$

$$\frac{\text{OL-APP} \quad \Psi \vdash^{OL} e_1 : \sigma_1 \rightarrow \sigma_2 \quad \Psi \vdash^{OL} e_2 : \sigma_1}{\Psi \vdash^{OL} e_1 e_2 : \sigma_2}$$

$$\frac{\text{OL-LET} \quad \Psi \vdash^{OL} e_1 : \sigma_1 \quad \Psi, x : \sigma_1 \vdash^{OL} e_2 : \sigma_2}{\Psi \vdash^{OL} \text{let } x = e_1 \text{ in } e_2 : \sigma_2}$$

$$\frac{\text{OL-GEN} \quad \Psi, a \vdash^{OL} e : \sigma}{\Psi \vdash^{OL} e : \forall a. \sigma}$$

$$\frac{\text{OL-SUB} \quad \Psi \vdash^{OL} e : \sigma_1 \quad \Psi \vdash \sigma_1 <: \sigma_2}{\Psi \vdash^{OL} e : \sigma_2}$$

$\Psi \vdash^{OL} \sigma_1 <: \sigma_2$

*(Subtyping)*

$$\frac{\text{OL-S-TVAR} \quad a \in \Psi}{\Psi \vdash^{OL} a <: a}$$

$$\frac{\text{OL-S-INT}}{\Psi \vdash^{OL} \text{Int} <: \text{Int}}$$

$$\frac{\text{OL-S-ARROW} \quad \Psi \vdash^{OL} \sigma_3 <: \sigma_1 \quad \Psi \vdash^{OL} \sigma_2 <: \sigma_4}{\Psi \vdash^{OL} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4}$$

$$\frac{\text{OL-S-FORALLL} \quad \Psi \vdash^{OL} \tau \quad \Psi \vdash^{OL} \sigma[a \mapsto \tau] <: \sigma_2}{\Psi \vdash^{OL} \forall a. \sigma_1 <: \sigma_2}$$

$$\frac{\text{OL-S-FORALLR} \quad \Psi, a \vdash^{OL} \sigma_1 <: \sigma_2}{\Psi \vdash^{OL} \sigma_1 <: \forall a. \sigma_2}$$

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

1999]. OL is predicative, which can be seen from rule [OL-S-FORALL](#). We focus only on predicative type systems throughout the thesis.

#### 2.2.5 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.

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

#### 2.3.1 BI-DIRECTIONAL TYPE CHECKING

Bi-directional 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]. The key idea in local type inference is simple. In particular, type inference is *local* in the sense that missing annotations are reconstructed using information from adjacent nodes in the syntax tree. Local type inference was introduced as an alternative to HM type systems, which could easily deal with polymorphic languages with subtyping.

Bi-directional 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 bi-directional 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, bi-directional 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 DK is given in Figure [2.6](#).

## 2 Background

Expressions	$e ::= x \mid n \mid \lambda x. e \mid \lambda x : \sigma. e \mid e_1 e_2 \mid e : \sigma$
Types	$\tau ::= \text{Int} \mid a \mid \hat{a} \mid \sigma_1 \rightarrow \sigma_2 \mid \forall a. \sigma$
Monotypes	$\tau ::= \text{Int} \mid a \mid \hat{a} \mid \tau_1 \rightarrow \tau_2$
Algorithmic Contexts	$\Delta, \Theta ::= \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \hat{a} \mid \Gamma, \hat{a} = \tau$
Complete Contexts	$\Omega ::= \bullet \mid \Omega, x : \sigma \mid \Omega, a \mid \Omega, \hat{a} = \tau$

Figure 2.6: Syntax of the algorithmic 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]

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 13]

Thierry Coquand. 1996. An algorithm for type-checking dependent types. *Science of Computer Programming* 26, 1-3 (1996), 167–177. [cited on page 13]

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]

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 13]

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 13]

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 page 13]
- Assaf J Kfoury and Jerzy Tiuryn. 1992. Type reconstruction in finite rank fragments of the second-order  $\lambda$ -calculus. *Information and computation* 98, 2 (1992), 228–257. [cited on page 9]
- 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 13]
- 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]
- 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 13]
- 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 11]
- 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 13]

- 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

