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 6 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.

??

- Chapter 4 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 [2007a] 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 7 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 6: 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 4: 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 7: 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. Section 2.1 reviews the Hindley-Milner type system [Damas and Milner 1982; Hindley 1969; Milner 1978], a classical type system for the lambda calculus with parametric polymorphism. Section 2.2 presents the Odersky-Läufer type system [Odersky and Läufer 1996], which extends upon the Hindley-Milner type system by putting higher-rank type annotations to work. Finally in Section 2.3 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 Hindley [1969], later rediscovered by Milner [1978], and also closely formalized by Damas and Milner [1982]. In what follows, we first review its declarative specification, then discuss the property of principality, and finally talk briefly about its algorithmic system.

2.1.1 Declarative System

The declarative system of HM is given in Figure 2.1.

Syntax. 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.

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	$::=$	$\forall \overline{a_i}^{\,i}.\, \tau$
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)*

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

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

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

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

TYPING. 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 given for the binder, 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 $\overline{a_i}^{\,i}$ which are not bound in the type context Ψ. In rule HM-INST, we can instantiate the type variables with arbitrary *monotypes*.

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

2.1.2 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 instantiations 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. Here are some examples of the subtyping relation:

$$\begin{array}{ll} \vdash^{HM} & \mathsf{Int} \to \mathsf{Int} <: \mathsf{Int} \to \mathsf{Int} \\ \vdash^{HM} & \forall a.\, a \to a <: \mathsf{Int} \to \mathsf{Int} \end{array}$$

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.

2.1.3 Algorithmic Type System

The declarative specification of HM given in Figure 2.1 does not directly lead to an algorithm. In particular, the system is not *syntax-directed*, and there are still many guesses in the system, such as in rule HM-LAM.

Syntax-directed System. A type system is *syntax-directed*, if the typing rules are completely driven by the syntax of expressions; in other words, there is exactly one typing rule for each syntactic form of expressions. However, in Figure 2.1, the rule for generalization (rule HM-GEN) and instantiation (rule HM-INST) can be applied anywhere.

A syntax-directed presentation of HM can be easily derived. In particular, from the typing rules we observe that, except for fetching a variable from the context (rule HM-VAR), the only place where a polymorphic type can be generated is for the let expressions (rule HM-LET). Thus, a syntax-directed system of HM can be presented as the original system, with instantiation applied to only variables, and generalization applied to only let expressions. Specifically,

$$\textsc{hm-var-inst}\quad \frac{(x : \forall \overline{a_i}^{\,i}. \tau) \in \Psi}{\Psi \vdash^{HM} x : \tau[\overline{a_i \mapsto \tau_i}^{\,i}]}$$

$$\textsc{hm-let-gen}\quad \frac{\Psi \vdash^{HM} e_1 : \tau \qquad \overline{a_i}^{\,i} = \mathrm{FV}(\tau) - \mathrm{FV}(\Psi) \qquad \Psi, x : \forall \overline{a_i}^{\,i}. \tau \vdash^{HM} e_2 : \tau}{\Psi \vdash^{HM} \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2 : \tau}$$

Type Inference. The guessing part of the system can be deterministically solved by the technique of *type inference*. 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 [Odersky and Läufer 1996; Peyton Jones et al. 2007], 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. Yet, 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 [Wells 1999]. 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. 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`: `F` is supposed to be polymorphic as it is applied to two arguments of different types. 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)
```

In the rest of this section, we first give the definition of the rank of a type, and then present the declarative specification of OL, and show that OL is a conservative extension of HM.

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,

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

Below we give some examples:

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

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.

[1]For the purpose of 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)*

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

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

2.2.2 Declarative System

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 can be 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, with polymorphic types still subsuming 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, it provides a complete view of possible formalisms of contexts in a type system with generalization.

Now since the context tracks type variables, we define the notion of *well-formedness* of types, given in Figure 2.4. For a type to be well-formedness, it must have all its free variable bound in the context. All rules are straightforward.

Type System. The typing rules for OL are given in Figure 2.5.

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

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

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

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

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

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

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 and 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 fresh 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 ($\Psi \vdash^{OL} \sigma_1 <: \sigma_2$). Note that the instantiation rule HM-INST in HM is a special case of rule OL-SUB, as we have $\forall \overline{a_i}^{\,i}.\,\tau <: \tau[\,\overline{a_i \mapsto \tau_i}^{\,i}\,]$ 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.

2.2.3 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 to be given explicitly. The Dunfield-Krishnaswami type system [Dunfield and Krishnaswami 2013], hereafter refered to as DK, 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. In this section, we first review the idea of bidirectional type checking, and then present the declarative DK and discuss its algorithm.

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. The "local" in local type inference comes from the fact 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 [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 Peyton Jones et al. [2007]. Let's revisit the example in Section 2.2:

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

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

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 Declarative System

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

Type System. 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 $\Psi \vdash^{HM} e : \sigma$ 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)*

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

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

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

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

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

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

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

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 Dunfield and Krishnaswami [2013], 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)*

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

$$\frac{\Psi \vdash^{DK} e \Leftarrow \sigma_1}{\Psi \vdash^{DK} \sigma_1 \to \sigma_2 \cdot e \Rrightarrow \sigma_2}\ \textsc{dk-app-arr}$$

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

$$\frac{\Psi \vdash^{DK} e_1 \Rightarrow \sigma \qquad \Psi \vdash^{DK} \sigma \cdot e_2 \Rrightarrow \sigma_2}{\Psi \vdash^{DK} e_1\, e_2 \Rightarrow \sigma_2}\ \textsc{dk-inf-app2}$$

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 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, as matching is a simple and independent 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.

2.3.3 Algorithmic Type System

Dunfield and Krishnaswami [2013] 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 Part III and Part IV. We will discuss more about it there.

Part II

Bidirectional Type Checking With Application Mode

3 Higher-Rank Polymorphism with Application Mode

We have seen in Section 2.3 that bi-directional type checking is an useful and versatile tool for type checking and type inference. In traditional bi-directional type-checking, type information flows from functions to arguments. In this section, we present a novel variant of bi-directional type checking where the type information flows from arguments to functions. This variant retains the inference mode, but adds a so-called *application* mode. Such design can remove annotations that basic bi-directional type checking cannot, and is useful when type information from arguments is required to type-check the functions being applied.

We illustrate the novel variant of bi-directional type-checking using System AP, a lambda calculus with implicit higher-rank polymorphism. This section first presents the declarative, syntax-directed type system of System AP (Section 3.2). The interesting aspects about the new type system are: 1) the typing rules, which employ a combination of the inference mode and the *application* mode; 2) the novel subtyping relation under an application context. Later, we prove our type system is type-safe by a type directed translation to System F in Section 3.3. An algorithmic type system is discussed in Section 3.4.

3.1 Introduction and Motivation

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 Section 2.3, bidirectional type checking [Pierce and Turner 2000] 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:

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

Specifically, if we know that the type of e_1 is a function from $\sigma_1 \rightarrow \sigma_2$, 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 [Dunfield and Pfenning 2004] 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:

$$\frac{}{\Psi \vdash^{DK} n \Leftarrow \mathsf{Int}}\ \textsc{dk-chk-int}$$

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.

$$\frac{}{\Psi \vdash^{DK} n \Rightarrow \mathsf{Int}}\ \textsc{dk-inf-int}$$

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) : ` $\mathsf{Int} \rightarrow \mathsf{Int}$`) 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 $\mathsf{Int} \rightarrow \mathsf{Int}$, from the argument `1`.

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,

$$\textsc{App}\quad \frac{\Psi \vdash e_2 \Rightarrow \sigma_1 \qquad \Psi; \Sigma, \sigma_1 \vdash e_1 \Rightarrow \sigma \rightarrow \sigma_2}{\Psi; \Sigma \vdash e_1\, e_2 \Rightarrow \sigma_2}$$

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:

$$\textsc{Lam}\quad \frac{\Psi, x : \sigma; \Sigma \vdash e \Rightarrow \sigma_2}{\Psi; \Sigma, \sigma \vdash \lambda x.\, e \Rightarrow \sigma \rightarrow \sigma_2}$$

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,

[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.

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 [Dunfield and Pfenning 2004], 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 3.5.2 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.

[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.

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.

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, Peyton Jones et al. [2007] 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 [Xie and Oliveira 2018] includes another application to System F.

Consider again the motivation example used in Section 2.2:

```
(\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 (Figure 2.5).

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 8. Among those works, we believe DK and the work by Peyton Jones et al. [2007] 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.

3.2 Declarative System

This section first presents the declarative, *syntax-directed* specification of AP. The interesting aspects about the new type system are: 1) the typing rules, which employ a combination of inference and application modes; 2) the novel subtyping relation under an application context.

3.2.1 Syntax

The syntax of the language is given in Figure 3.1.

Expressions. The definition of expressions e include variables (x), integers (n), annotated lambda abstractions ($\lambda x : \sigma.\, e$), lambda abstractions ($\lambda x.\, e$), and applications ($e_1\, e_2$). No-

Expressions	e	$::=$	$x \mid n \mid \lambda x : \sigma.\, e \mid \lambda x.\, e \mid e_1\, e_2$
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	$::=$	$\bullet \mid \Psi, x : \sigma$
Application Contexts	Σ	$::=$	$\bullet \mid \Sigma, \sigma$

Figure 3.1: Syntax of System AP.

tably, the syntax does not include a **let** expression ($\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$). Let expressions can be regarded as the standard syntax sugar $(\lambda x.\, e_2)\, e_1$, as illustrated in more detail later.

Types. Types include the integer type Int, type variables (a), functions ($\sigma_1 \to \sigma_2$) and polymorphic types ($\forall a.\, \sigma$). Monotypes are types without universal quantifiers.

Contexts. Typing contexts Ψ are standard: they map a term variable x to its type σ. In this system, the context is modeled in a HM-style context (Section 2.1), which does not track type variables. Again, we implicitly assume that all the variables in Ψ are distinct.

The main novelty lies in the *application contexts* Σ, which are the main data structure needed to allow types to flow from arguments to functions. Application contexts are modeled as a stack. The stack collects the types of arguments in applications. The context is a stack because if a type is pushed last then it will be popped first. For example, inferring expression e under application context (a, Int), means e is now being applied to two arguments e_1, e_2, with $e_1 : \mathsf{Int}$, $e_2 : a$, so e should be of type $\mathsf{Int} \to a \to \sigma$ for some σ.

3.2.2 Type System

The top part of Figure 3.2 gives the typing rules for our language. The judgment $\Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma$ is read as: under typing context Ψ, and application context Σ, e has type σ. The standard inference mode $\Psi \vdash^{AP} e \Rightarrow \sigma$ can be regarded as a special case when the application context is empty. Note that the variable names are assumed to be fresh enough when new variables are added into the typing context, or when generating new type variables.

We discuss the rules when the application context is empty first. Those rules are unsurprising. Rule ap-inf-int shows that integer literals are only inferred to have type Int under an empty application context. This is obvious since an integer cannot accept any arguments. Rule ap-inf-lam deals with lambda abstractions when the application context is empty. In this situation, a monotype τ is *guessed* for the argument, just like previous calculi. Rule ap-inf-lamann also works as expected: a new variable x is put with its type σ into the typing context, and inference continues on the abstraction body.

$\boxed{\Psi \vdash^{AP} e \Rightarrow \sigma}$ *(Typing Inference)*

$$\textsc{ap-inf-int}\quad \frac{}{\Psi \vdash^{AP} n \Rightarrow \mathsf{Int}} \qquad \textsc{ap-inf-lam}\quad \frac{\Psi, x : \tau \vdash^{AP} e \Rightarrow \sigma}{\Psi \vdash^{AP} \lambda x.\, e \Rightarrow \tau \to \sigma} \qquad \textsc{ap-inf-lamann}\quad \frac{\Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_2}{\Psi \vdash^{AP} \lambda x : \sigma_1.\, e \Rightarrow \sigma_1 \to \sigma_2}$$

$\boxed{\Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma}$ *(Typing Application Mode)*

$$\textsc{ap-app-var}\quad \frac{(x : \sigma_1) \in \Psi \qquad \Sigma \vdash^{AP} \sigma_1 <: \sigma_2}{\Psi; \Sigma \vdash^{AP} x \Rightarrow \sigma_2} \qquad \textsc{ap-app-lam}\quad \frac{\Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_2}{\Psi; \Sigma, \sigma_1 \vdash^{AP} \lambda x.\, e \Rightarrow \sigma_1 \to \sigma_2}$$

$$\textsc{ap-app-lamann}\quad \frac{\vdash^{AP} \sigma_2 <: \sigma_1 \qquad \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_3}{\Psi; \Sigma, \sigma_2 \vdash^{AP} \lambda x : \sigma_1.\, e \Rightarrow \sigma_2 \to \sigma_3}$$

$$\textsc{ap-app-app}\quad \frac{\begin{array}{c}\Psi \vdash^{AP} e_2 \Rightarrow \sigma_1 \qquad \overline{a_i}^{\,i} = \textsc{fv}\,(\sigma_1) - \textsc{fv}\,(\Psi) \\ \sigma_2 = \forall \overline{a_i}^{\,i}.\, \sigma_1 \qquad \Psi; \Sigma, \sigma_2 \vdash^{AP} e_1 \Rightarrow \sigma_2 \to \sigma_3\end{array}}{\Psi; \Sigma \vdash^{AP} e_1\, e_2 \Rightarrow \sigma_3}$$

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

$$\textsc{ap-s-tvar}\quad \frac{}{\vdash^{AP} a <: a} \qquad \textsc{ap-s-int}\quad \frac{}{\vdash^{AP} \mathsf{Int} <: \mathsf{Int}} \qquad \textsc{ap-s-arrow}\quad \frac{\vdash^{AP} \sigma_3 <: \sigma_1 \qquad \vdash^{AP} \sigma_2 <: \sigma_4}{\vdash^{AP} \sigma_1 \to \sigma_2 <: \sigma_3 \to \sigma_4}$$

$$\textsc{ap-s-forall}\mathrm{L}\quad \frac{\vdash^{AP} \sigma[a \mapsto \tau] <: \sigma_2}{\vdash^{AP} \forall a.\, \sigma_1 <: \sigma_2} \qquad \textsc{ap-s-forall}\mathrm{R}\quad \frac{a \notin \textsc{fv}\,(\sigma_1) \qquad \vdash^{AP} \sigma_1 <: \sigma_2}{\vdash^{AP} \sigma_1 <: \forall a.\, \sigma_2}$$

$\boxed{\Sigma \vdash^{AP} \sigma_1 <: \sigma_2}$ *(Application Subtyping)*

$$\textsc{ap-as-empty}\quad \frac{}{\bullet \vdash^{AP} \sigma <: \sigma} \qquad \textsc{ap-as-forall}\quad \frac{\Sigma, \sigma_3 \vdash^{AP} \sigma_1[a \mapsto \tau] <: \sigma_2}{\Sigma, \sigma_3 \vdash^{AP} \forall a.\, \sigma_1 <: \sigma_2} \qquad \textsc{ap-as-arrow}\quad \frac{\vdash^{AP} \sigma_3 <: \sigma_1 \qquad \Sigma \vdash^{AP} \sigma_2 <: \sigma_4}{\Sigma, \sigma_3 \vdash^{AP} \sigma_1 \to \sigma_2 <: \sigma_3 \to \sigma_4}$$

Figure 3.2: Typing rules of System AP.

Now we turn to the cases when the application context is not empty. Rule AP-APP-VAR says that if $x : \sigma_1$ is in the typing context, and σ_1 is a subtype of σ_2 under application context Σ, then x has type σ_2. It depends on the subtyping rules that are explained in Section 3.2.3.

Rule AP-APP-LAM shows the strength of application contexts. It states that, without annotations, if the application context is non-empty, a type can be popped from the application context to serve as the type for x. Inference of the body then continues with the rest of the application context. This is possible, because the expression $\lambda x.\, e$ is being applied to an argument of type σ_1, which is the type at the top of the application context stack.

For lambda abstraction with annotations $\lambda x : \sigma_1.\, e$, if the application context has type σ_2, then rule AP-APP-LAMANN first checks that σ_2 is a subtype of σ_1 before putting $x : \sigma_1$ in the typing context. However, note that it is always possible to remove annotations in an abstraction if it has been applied to some arguments.

Rule AP-APP-APP pushes types into the application context. The application rule first infers the type of the argument e_2 with type σ_1. Then the type σ_1 is generalized in the same way as the HM type system. The resulting generalized type is σ_2. Thus the type of e_1 is now inferred under an application context extended with type σ_2. The generalization step is important to infer higher ranked types: since σ_2 is a possibly polymorphic type, which is the argument type of e_1, then e_1 is of possibly a higher rank type.

Let Expressions. The language does not have built-in let expressions, but instead supports **let** as syntactic sugar. Recall the syntactic-directed typing rule (rule HM-LET-GEN) for let expressions with generalization in the HM system. With slight reformat to match AP, we get (without the gray-shaded part):

$$\frac{\Psi \vdash e_1 \Rightarrow \sigma_1 \qquad \overline{a_i}^{\,i} = \textsc{fv}\,(\tau) - \textsc{fv}\,(\Psi) \qquad \sigma_2 = \forall \overline{a_i}^{\,i}.\, \sigma_1 \qquad \Psi, x : \sigma_2; \boxed{\Sigma} \vdash e_2 \Rightarrow \sigma_3}{\Psi; \boxed{\Sigma} \vdash \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2 \Rightarrow \sigma_3}$$

where we do generalization on the type of e_1, which is then assigned as the type of x while inferring e_2. Adapting this rule to our system with application contexts would result in the gray-shaded part, where the application context is only used for e_2, because e_2 is the expression being applied. If we desugar the let expression ($\mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2$) to $(\lambda x.\, e_2)\, e_1$, we have the following derivation:

$$\frac{\Psi \vdash e_1 \Rightarrow \sigma_1 \qquad \overline{a_i}^{\,i} = \textsc{fv}\,(\sigma_1) - \textsc{fv}\,(\Psi) \qquad \sigma_2 = \forall \overline{a_i}^{\,i}.\, \sigma_1 \qquad \dfrac{\Psi, x : \sigma_2; \Sigma \vdash e_2 \Rightarrow \sigma_3}{\Psi; \Sigma, \sigma_2 \vdash \lambda x.\, e_2 \Rightarrow \sigma_2 \to \sigma_3}}{\Psi; \Sigma \vdash (\lambda x.\, e_2)\, e_2 \Rightarrow \sigma_3}$$

The type σ_2 is now pushed into application context in rule AP-APP-APP, and then assigned to x in rule AP-APP-LAM. Comparing this with the typing derivations for let expressions, we now have the same preconditions. Thus we can see that the rules in Figure 3.2 are sufficient to express an HM-style polymorphic let construct.

METATHEORY. The type system enjoys several interesting properties, especially lemmas about application contexts. Before we present those lemmas, we need a helper definition of what it means to use arrows on application contexts.

Definition 2 ($\Sigma \to \sigma$). If $\Sigma = \sigma_1, \sigma_2, ...\sigma_n$, then $\Sigma \to \sigma$ means the function type $\sigma_n \to ... \to \sigma_2 \to \sigma_1 \to \sigma$.

Such definition is useful to reason about the typing result with application contexts. One specific property is that the application context determines the form of the typing result.

Lemma 3.1 (Σ Coincides with Typing Results). *If $\Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma$, then for some σ', we have $\sigma = \Sigma \to \sigma'$.*

Having this lemma, we can always use the judgment $\Psi; \Sigma \vdash^{AP} e \Rightarrow \Sigma \to \sigma'$ instead of $\Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma$.

In traditional bi-directional type checking, we often have one subsumption rule that transfers between inference and checked mode, which states that if an expression can be inferred to some type, then it can be checked with this type (e.g., rule DK-CHK-SUB in DK). In our system, we regard the normal inference mode $\Psi \vdash^{AP} e \Rightarrow \sigma$ as a special case, when the application context is empty. We can also turn from normal inference mode into application mode with an application context.

Lemma 3.2 (Subsumption). *If $\Psi \vdash^{AP} e \Rightarrow \Sigma \to \sigma$, then $\Psi; \Sigma \vdash^{AP} e \Rightarrow \Sigma \to \sigma$.*

This lemma is actually a special case for the following one:

Lemma 3.3 (Generalized Subsumption). *If $\Psi; \Sigma_1 \vdash^{AP} e \Rightarrow \Sigma_1 \to \Sigma_2 \to \sigma$, then $\Psi; \Sigma_2, \Sigma_1 \vdash^{AP} e \Rightarrow \Sigma_1 \to \Sigma_2 \to \sigma$.*

The relationship between our system and standard Hindley Milner type system (HM) can be established through the desugaring of let expressions. Namely, if e is typeable in HM, then the desugared expression e' is typeable in our system, with a more general typing result.

Lemma 3.4 (AP Conservative over HM). *If $\Psi \vdash^{HM} e : \sigma$, and desugaring let expression in e gives back e', then for some σ', we have $\Psi \vdash^{AP} e' \Rightarrow \sigma'$, and $A' <: A$.*

3.2.3 Subtyping

We present our subtyping rules at the bottom of Figure 3.2. Interestingly, our subtyping has two different forms.

Subtyping. The first subtyping judgment $\vdash^{AP} \sigma_1 <: \sigma_2$ follows OL, but in HM-style; that is, without tracking type variables. Rule AP-S-FORALLR states σ_1 is subtype of $\forall a.\, \sigma_2$ only if σ_1 is a subtype of σ_2, with the assumption a is a fresh variable. Rule AP-S-FORALLL says $\forall a.\, \sigma_1$ is a subtype of σ_2 if we can instantiate it with some τ and show the result is a subtype of σ_2.

Application Subtyping. The typing rule AP-APP-VAR uses the second subtyping judgment $\Sigma \vdash^{AP} \sigma_1 <: \sigma_2$. To motivate this new kind of judgment, consider the expression id 1 for example, whose derivation is stuck at rule AP-APP-VAR (here we assume $\mathsf{id} : \forall a.\, a \to a \in \Psi$):

$$\dfrac{\Psi \vdash^{AP} 1 \Rightarrow \mathsf{Int} \quad \emptyset = \textsc{fv}\,(\mathsf{Int}) - \textsc{fv}\,(\Psi) \quad \dfrac{\mathsf{id} : \forall a.\, a \to a \in \Psi \quad ???}{\Psi; \mathsf{Int} \vdash^{AP} \mathsf{id} \Rightarrow ?}\ \textsc{ap-app-var}}{\Psi \vdash^{AP} \mathsf{id}\ 1 \Rightarrow ?}\ \textsc{ap-app-app}$$

Here we know that $\mathsf{id} : \forall a.\, a \to a$ and also, from the application context, that id is applied to an argument of type Int. Thus we need a mechanism for solving the instantiation $a = \mathsf{Int}$ and return a supertype $\mathsf{Int} \to \mathsf{Int}$ as the type of id. This is precisely what the application subtyping achieves: resolve instantiation constraints according to the application context. Notice that unlike existing works (Peyton Jones et al. [2007] or DK),application subtyping provides a way to solve instantiation more *locally*, since it does not mutually depend on typing.

Back to the rules in Figure 3.2, one way to understand the judgment $\Sigma \vdash^{AP} \sigma_1 <: \sigma_2$ from a computational point-of-view is that the type σ_2 is a *computed* output, rather than an input. In other words σ_2 is determined from Σ and σ_1. This is unlike the judgment $\vdash^{AP} \sigma_1 <: \sigma_2$, where both σ_1 and σ_2 would be computationally interpreted as inputs. Therefore it is not possible to view $\vdash^{AP} \sigma_1 <: \sigma_2$ as a special case of $\Sigma \vdash^{AP} \sigma_1 <: \sigma_2$ where Σ is empty.

There are three rules dealing with application contexts. Rule AP-AS-EMPTY is for case when the application context is empty. Because it is empty, we have no constraints on the type, so we return it back unchanged. Note that this is where HM-style systems (also Peyton Jones et al. [2007]) would normally use an instantiation rule (e.g. rule HM-INST in HM) to remove top-level quantifiers. Our system does not need the instantiation rule, because in applications, type information flows from arguments to the function, instead of function to arguments.

In the latter case, the instantiation rule is needed because a function type is wanted instead of a polymorphic type. In our approach, instantiation of type variables is avoided unless necessary.

The two remaining rules apply when the application context is non-empty, for polymorphic and function types respectively. Note that we only need to deal with these two cases because Int or type variables a cannot have a non-empty application context. In rule AP-AS-FORALL, we instantiate the polymorphic type with some τ, and continue. This instantiation is forced by the application context. In rule AP-AS-ARROW, one function of type $\sigma_1 \to \sigma_2$ is now being applied to an argument of type σ_3. So we check $\vdash^{AP} \sigma_3 <: \sigma_1$. Then we continue with σ_2 and the rest application context, and return $\sigma_3 \to \sigma_4$ as the result type of the function.

Metatheory. Application subtyping is novel in our system, and it enjoys some interesting properties. For example, similarly to typing, the application context decides the form of the supertype.

Lemma 3.5 (Σ Coincides with Subtyping Results). *If* $\Sigma \vdash^{AP} \sigma_1 <: \sigma_2$, *then for some* σ_3, $\sigma_2 = \Sigma \to \sigma_3$.

Therefore we can always use the judgment $\Sigma \vdash^{AP} \sigma_1 <: \Sigma \to \sigma_2$, instead of $\Sigma \vdash^{AP} \sigma_1 <: \sigma_2$. Application subtyping is also reflexive and transitive. Interestingly, in those lemmas, if we remove all applications contexts, they are exactly the reflexivity and transitivity of traditional subtyping.

Lemma 3.6 (Reflexivity of Application Subtyping). $\Sigma \vdash^{AP} \Sigma \to \sigma <: \Sigma \to \sigma$.

Lemma 3.7 (Transitivity of Application Subtyping). *If* $\Sigma_1 \vdash^{AP} \sigma_1 <: \Sigma_1 \to \sigma_2$, *and* $\Sigma_2 \vdash^{AP} \sigma_2 <: \Sigma_2 \to \sigma_3$, *then* $\Sigma_2, \Sigma_1 \vdash^{AP} \sigma_1 <: \Sigma_1 \to \Sigma_2 \to \sigma_3$.

Finally, we can convert between subtyping and application subtyping. We can remove the application context and still get a subtyping relation:

Lemma 3.8 ($\Sigma \vdash^{AP}$ to $\vdash^{AP}$). *If* $\Sigma \vdash^{AP} \sigma_1 <: \sigma_2$, *then* $\vdash^{AP} \sigma_1 <: \sigma_2$.

Transferring from subtyping to application subtyping will result in a more general type.

Lemma 3.9 ($\vdash^{AP}$ to $\Sigma \vdash^{AP}$). *If* $\vdash^{AP} \sigma_1 <: \Sigma \to \sigma_2$, *then for some* σ_3, *we have* $\Sigma \vdash^{AP} \sigma_1 <: \Sigma \to \sigma_3$, *and* $\vdash^{AP} \sigma_3 <: \sigma_2$.

This lemma may not seem intuitive at first glance. Consider a concrete example. Consider the derivation for $\vdash^{AP} \mathsf{Int} \to \forall a.\, a <: \mathsf{Int} \to \mathsf{Int}$:

$$\dfrac{\dfrac{}{\vdash^{AP} \mathsf{Int} <: \mathsf{Int}}\ \textsc{ap-s-int} \qquad \dfrac{\dfrac{}{\vdash^{AP} \mathsf{Int} <: \mathsf{Int}}\ \textsc{ap-s-int}}{\vdash^{AP} \forall a.\, a <: \mathsf{Int}}\ \textsc{ap-s-forallL}}{\vdash^{AP} \mathsf{Int} \to \forall a.\, a <: \mathsf{Int} \to \mathsf{Int}}\ \textsc{ap-s-arrow}$$

and for $\mathsf{Int} \vdash^{AP} \mathsf{Int} \to \forall a.\, a <: \mathsf{Int} \to \forall a.\, a$:

$$\dfrac{\dfrac{}{\vdash^{AP} \mathsf{Int} <: \mathsf{Int}}\ \textsc{ap-s-int} \qquad \dfrac{}{\vdash^{AP} \forall a.\, a <: \forall a.\, a}\ \textsc{ap-as-empty}}{\mathsf{Int} \vdash^{AP} \mathsf{Int} \to \forall a.\, a <: \mathsf{Int} \to \forall a.\, a}\ \textsc{ap-as-arrow}$$

The former one, holds because we have $\vdash^{AP} \forall a.\, a <: \mathsf{Int}$ in the return type. But in the latter one, after Int is consumed from application context, we eventually reach rule AP-AS-EMPTY, which always returns the original type back.

3.3 Type-directed Translation

This section discusses the type-directed translation of System AP into a variant of System F that is also used in Peyton Jones et al. [2007]. The translation is shown to be coherent and type safe. The later result implies the type-safety of the source language. To prove coherency, we need to decide when two translated terms are the same using η-*id equality*, and show that the translation is unique up to this definition.

3.3.1 Target Language

The syntax and typing rules of our target language are given in Figure 3.3.

Expressions include variables x, integers n, annotated abstractions $\lambda x : \sigma.\, s$, type-level abstractions $\Lambda a.\, s$, and $s_1\, s_2$ for term application, and $s\, \sigma$ for type application. The types and the typing contexts are the same as our system, where typing contexts does not track type variables. In translation, we use f to refer to the coercion function produced by subtyping translation, and s to refer to the translated term in System F.

Most typing rules are straightforward. Rule F-TABS types a type abstraction with explicit generalization, while rule F-TAPP types a type application with explicit instantiation.

Expressions	s, f	$::=$	$x \mid n \mid \lambda x : \sigma.\, s \mid \Lambda a.\, s \mid s_1\, s_2 \mid s\, \sigma$
Types	σ	$::=$	$\mathsf{Int} \mid a \mid \sigma_1 \to \sigma_2 \mid \forall a.\, \sigma$
Contexts	Ψ	$::=$	$\bullet \mid \Psi, x : \sigma$

$\boxed{\Psi \vdash^F s : \sigma}$ *(Typing)*

$$\textsc{F-Var}\quad \frac{(x : \sigma) \in \Psi}{\Psi \vdash^F x : \sigma} \qquad \textsc{F-Int}\quad \frac{}{\Psi \vdash^F n : \mathsf{Int}} \qquad \textsc{F-LamAnn}\quad \frac{\Psi, x : \sigma_1 \vdash^F s : \sigma_2}{\Psi \vdash^F \lambda x : \sigma_1.\, s : \sigma_1 \to \sigma_2}$$

$$\textsc{F-App}\quad \frac{\Psi \vdash^F s_1 : \sigma_1 \to \sigma_2 \qquad \Psi \vdash^F s_2 : \sigma_1}{\Psi \vdash^F s_1\, s_2 : \sigma_2} \qquad \textsc{F-TAbs}\quad \frac{\Psi \vdash^F s : \sigma \qquad a \notin \textsc{fv}(\Psi)}{\Psi \vdash^F \Lambda a.\, s : \forall a.\, \sigma}$$

$$\textsc{F-TApp}\quad \frac{\Psi \vdash^F s : \forall a.\, \sigma_1}{\Psi \vdash^F s\, \sigma_2 : \sigma_1[a \mapsto \sigma_2]}$$

Figure 3.3: Syntax and typing rules of System F.

3.3.2 Subtyping Coercions

The type-directed translation of subtyping is shown in Figure 3.4. The translation follows the subtyping relations from Figure 3.2, but adds a new component. The judgment $\vdash^{AP} \sigma_1 <: \sigma_2 \rightsquigarrow f$ is read as: if $\vdash^{AP} \sigma_1 <: \sigma_2$ holds, it can be translated to a coercion function f in System F. The coercion function produced by subtyping is used to transform values from one type to another, so we should have $\bullet \vdash^F f : \sigma_1 \to \sigma_2$.

Rule AP-S-INT and rule AP-S-TVAR produce identity functions, since the source type and target type are the same. In rule AP-S-ARROW, the coercion function f_1 of type $\sigma_3 \to \sigma_1$ is applied to y to get a value of type σ_1. Then the resulting value becomes an argument to x, and a value of type σ_2 is obtained. Finally we apply f_2 to the value of type σ_2, so that a value of type σ_4 is computed. In rule A-PS-ForallL, the input argument is a polymorphic type, so we feed the type τ to it and apply the coercion function f from the precondition. Rule AP-S-FORALLR uses the coercion f and, in order to produce a polymorphic type, we add one type abstraction to turn it into a coercion of type $\sigma_1 \to \forall a.\, \sigma_2$.

The second part of the subtyping translation deals with coercions generated by application subtyping. The judgment $\Sigma \vdash^{AP} \sigma <: \sigma_2 \rightsquigarrow f$ is read as: if $\Sigma \vdash^{AP} \sigma <: \sigma_2$ holds, it can be translated to a coercion function f in System F. If we compare two parts, we can see application contexts play no role in the generation of the coercion. Notice the similarity

$\boxed{\vdash^{AP} \sigma_1 <: \sigma_2 \rightsquigarrow f}$ *(Subtyping Translation)*

$$\frac{}{\vdash^{AP} a <: a \rightsquigarrow \lambda x : a.\, x}\ \textsc{ap-s-tvar} \qquad \frac{}{\vdash^{AP} \mathsf{Int} <: \mathsf{Int} \rightsquigarrow \lambda x : \mathsf{Int}.\, x}\ \textsc{ap-s-int}$$

$$\frac{\vdash^{AP} \sigma_3 <: \sigma_1 \rightsquigarrow f_1 \qquad \vdash^{AP} \sigma_2 <: \sigma_4 \rightsquigarrow f_2}{\vdash^{AP} \sigma_1 \to \sigma_2 <: \sigma_3 \to \sigma_4 \rightsquigarrow \lambda x : \sigma_1 \to \sigma_2.\, \lambda y : \sigma_3.\, f_2\,(x\,(f_1\,y))}\ \textsc{ap-s-arrow}$$

$$\frac{\vdash^{AP} \sigma[a \mapsto \tau] <: \sigma_2 \rightsquigarrow f}{\vdash^{AP} \forall a.\, \sigma_1 <: \sigma_2 \rightsquigarrow \lambda x : \forall a.\, \sigma_1.\, f\,(x\,\tau)}\ \textsc{ap-s-forallL} \qquad \frac{a \notin \textsc{fv}\,(\sigma_1) \qquad \vdash^{AP} \sigma_1 <: \sigma_2 \rightsquigarrow f}{\vdash^{AP} \sigma_1 <: \forall a.\, \sigma_2 \rightsquigarrow \lambda x : \sigma_1.\, \Lambda a.\, f\, x}\ \textsc{ap-s-forallR}$$

$\boxed{\Sigma \vdash^{AP} \sigma_1 <: \sigma_2 \rightsquigarrow f}$ *(Application Subtyping)*

$$\frac{}{\bullet \vdash^{AP} \sigma <: \sigma \rightsquigarrow \lambda x : \sigma.\, x}\ \textsc{ap-as-empty} \qquad \frac{\Sigma, \sigma_3 \vdash^{AP} \sigma_1[a \mapsto \tau] <: \sigma_2 \rightsquigarrow f}{\Sigma, \sigma_3 \vdash^{AP} \forall a.\, \sigma_1 <: \sigma_2 \rightsquigarrow \lambda x : \forall a.\, \sigma_1.\, f\,(x\,\tau)}\ \textsc{ap-as-forall}$$

$$\frac{\vdash^{AP} \sigma_3 <: \sigma_1 \rightsquigarrow f_1 \qquad \Sigma \vdash^{AP} \sigma_2 <: \sigma_4 \rightsquigarrow f_2}{\Sigma, \sigma_3 \vdash^{AP} \sigma_1 \to \sigma_2 <: \sigma_3 \to \sigma_4 \rightsquigarrow \lambda x : \sigma_1 \to \sigma_2.\, \lambda y : \sigma_3.\, f_2\,(x\,(f_1\,y))}\ \textsc{ap-as-arrow}$$

Figure 3.4: Subtyping translation rules of System AP.

$$\boxed{\Psi \vdash^{AP} e \Rightarrow \sigma \rightsquigarrow s}$$ *(Typing Inference)*

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

$$\text{AP-INF-LAM}\quad \frac{\Psi, x : \tau \vdash^{AP} e \Rightarrow \sigma \rightsquigarrow s}{\Psi \vdash^{AP} \lambda x.\, e \Rightarrow \tau \to \sigma \rightsquigarrow \lambda x : \tau.\, s}$$

$$\text{AP-INF-LAMANN}\quad \frac{\Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_2 \rightsquigarrow s}{\Psi \vdash^{AP} \lambda x : \sigma_1.\, e \Rightarrow \sigma_1 \to \sigma_2 \rightsquigarrow \lambda x : \sigma_1.\, s}$$

$$\boxed{\Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma \rightsquigarrow s}$$ *(Typing Application Mode)*

$$\text{AP-APP-VAR}\quad \frac{(x : \sigma_1) \in \Psi \qquad \Sigma \vdash^{AP} \sigma_1 <: \sigma_2 \rightsquigarrow f}{\Psi; \Sigma \vdash^{AP} x \Rightarrow \sigma_2 \rightsquigarrow f\, x}$$

$$\text{AP-APP-LAM}\quad \frac{\Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_2 \rightsquigarrow s}{\Psi; \Sigma, \sigma_1 \vdash^{AP} \lambda x.\, e \Rightarrow \sigma_1 \to \sigma_2 \rightsquigarrow \lambda x : \sigma_1.\, s}$$

$$\text{AP-APP-LAMANN}\quad \frac{\vdash^{AP} \sigma_2 <: \sigma_1 \rightsquigarrow f \qquad \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_3 \rightsquigarrow s}{\Psi; \Sigma, \sigma_2 \vdash^{AP} \lambda x : \sigma_1.\, e \Rightarrow \sigma_2 \to \sigma_3 \rightsquigarrow \lambda y : \sigma_2.\, (\lambda x : \sigma_1.\, s)\, (f\, y)}$$

$$\text{AP-APP-APP}\quad \frac{\begin{array}{c}\Psi \vdash^{AP} e_2 \Rightarrow \sigma_1 \rightsquigarrow s_2 \\ \overline{a_i}^{\,i} = \textsc{fv}\,(\sigma_1) - \textsc{fv}\,(\Psi) \qquad \sigma_2 = \forall \overline{a_i}^{\,i}.\, \sigma_1 \\ \Psi; \Sigma, \sigma_2 \vdash^{AP} e_1 \Rightarrow \sigma_2 \to \sigma_3 \rightsquigarrow s_1\end{array}}{\Psi; \Sigma \vdash^{AP} e_1\, e_2 \Rightarrow \sigma_3 \rightsquigarrow s_1\, (\Lambda \overline{a_i}^{\,i}.\, s_2)}$$

Figure 3.5: Typing translation rules of System AP.

between rule AP-S-TVAR and rule AP-AS-EMPTY, between rule AP-S-FORALLR and rule AP-AS-FORALL, and between rule AP-S-ARROW and rule AP-AS-ARROW. We therefore omit more explanations.

3.3.3 Type-Directed Translation of Typing

The type directed translation of typing is shown in the Figure 3.5, which extends the rules in Figure 3.1. The judgment $\Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma \rightsquigarrow s$ is read as: if $\Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma$ holds, the expression can be translated to term s in System F. The judgment $\Psi \vdash^{AP} e \Rightarrow \sigma \rightsquigarrow s$ is the special case when the application context is empty.

Most translation rules are straightforward. In rule AP-APP-VAR, x is translated to $f\,x$, where f is the coercion function generated from subtyping. Rule AP-APP-LAMANN applies the coercion function f to y, then feeds y to the function generated from the abstraction. When generalizing over a type, rule AP-APP-APP generate type-level abstractions.

3.3.4 Type Safety

We prove that our system is type safe by proving that the translation produces well-typed terms.

Lemma 3.10 (Soundness of Typing Translation)**.** *If* $\Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma \rightsquigarrow s$, *then* $\Psi \vdash^{F} s : \sigma$.

The lemma relies on the translation of subtyping to produce type-correct coercions.

Lemma 3.11 (Soundness of Subtyping Translation)**.**

1. *If* $\vdash^{AP} \sigma <: \sigma_2 \rightsquigarrow f$, *then* $\bullet \vdash^{F} f : \sigma \to \sigma_2$.
2. *If* $\Sigma \vdash^{AP} \sigma <: \sigma_2 \rightsquigarrow f$, *then* $\bullet \vdash^{F} f : \sigma \to \sigma_2$.

3.3.5 Coherence

One problem with the translation is that there are multiple targets corresponding to one expression. This is because in original system there are multiple choices when instantiating a polymorphic type, or guessing the annotation for unannotated lambda abstraction (rule AP-INF-LAM). For each choice, the corresponding target will be different. For example, expression $\lambda x.\,x$ can be type checked with $\mathsf{Int} \to \mathsf{Int}$, or $a \to a$, and the corresponding targets are $\lambda x : \mathsf{Int}.\,x$, and $\lambda x : a.\,x$.

Therefore, in order to prove the translation is coherent, we turn to prove that all the translations have the same operational semantics. There are two steps towards the goal: type erasure, and considering η expansion and identity functions.

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
|x| & = & |x| & |\Lambda a.\, s| & = & |s| \\
|n| & = & |n| & |s_1\, s_2| & = & |s_1|\,|s_2| \\
|\lambda x : \sigma.\, s| & = & \lambda x.\, |s| & |s\,\sigma| & = & |s|
\end{array}
$$

$\boxed{f_1 =_{\eta id} f_2}$ *(Eta-id Equality)*

$$
\frac{x \notin \text{FV}\,(f)}{\lambda x.\, f\, x =_{\eta id} f}\ \textsc{eta-reduce}
\qquad
\frac{}{(\lambda x.\, x)\, f =_{\eta id} f}\ \textsc{eta-id}
\qquad
\frac{f_1 =_{\eta id} f_1' \qquad f_2 =_{\eta id} f_2'}{f_1\, f_2 =_{\eta id} f_1'\, f_2'}\ \textsc{eta-app}
$$

$$
\frac{f =_{\eta id} f'}{\lambda x.\, f =_{\eta id} \lambda x.\, f'}\ \textsc{eta-lam}
\qquad
\frac{}{f =_{\eta id} f}\ \textsc{eta-refl}
\qquad
\frac{f =_{\eta id} f'}{f' =_{\eta id} f}\ \textsc{eta-symm}
\qquad
\frac{f_1 =_{\eta id} f_2 \qquad f_2 =_{\eta id} f_3}{f_1 =_{\eta id} f_3}\ \textsc{eta-trans}
$$

Figure 3.6: Type erasure and eta-id equality of System F.

Type Erasure. Since type information is useless after type-checking, we erase the type information of the targets for comparison. The erasure process is given at the top of Figure 3.6.

The erasure process is standard, where we erase the type annotation in abstractions, and remove type abstractions and type applications. The calculus after erasure is the untyped lambda calculus.

Eta-id Equality. However, even if we have type erasure, multiple targets for one expression can still be syntactically different. The problem is that we can insert more coercion functions in one translation than another, since an expression can have a more polymorphic type in one derivation than another one. So we need a more refined definition of equality instead of syntactic equality.

We use a similar definition of eta-id equality as in Chen [2003], given in Figure 3.6. In $=_{\eta id}$ equality, two expressions are regarded as equivalent if they can turn into the same expression through η-reduction or removal of redundant identity functions. The $=_{\eta id}$ relation is reflexive, symmetrical, and transitive. As a small example, we can show that $\lambda x.\, (\lambda y.\, y)\, f\, x =_{\eta id} f$.

$$
\dfrac{\dfrac{\dfrac{}{f =_{\eta id} f}\ \textsc{eta-refl}}{(\lambda y.\, y)\, f =_{\eta id} f}\ \textsc{eta-id}}{\lambda x.\, (\lambda y.\, y)\, f\, x =_{\eta id} f}\ \textsc{eta-reduce}
$$

Now we first prove that the erasure of the translation result of subtyping is always $=_{\eta id}$ to an identity function.

Lemma 3.12 (Subtyping Coercions eta-id equal to id).

- *if* $\vdash^{AP} \sigma_1 <: \sigma_2 \rightsquigarrow f$, *then* $|f| =_{\eta id} \lambda x.\, x$.
- *if* $\Sigma \vdash^{AP} \sigma_1 <: \sigma_2 \rightsquigarrow f$, *then* $|f| =_{\eta id} \lambda x.\, x$.

We then prove that our translation actually generates a *unique* target:

Lemma 3.13 (Coherence). *If* $\Psi_1; \Sigma_1 \vdash^{AP} e \Rightarrow \sigma \rightsquigarrow s_1$, *and* $\Psi_2; \Sigma_2 \vdash^{AP} e \Rightarrow \sigma_2 \rightsquigarrow s_2$, *then* $|s_1| =_{\eta id} |s_2|$.

3.4 Type Inference Algorithm

Even though our specification is syntax-directed, it does not directly lead to an algorithm, because there are still many guesses in the system, such as in rule AP-INF-LAM. This subsection presents a brief introduction of the algorithm, which closely follows the approach by Peyton Jones et al. [2007].

Instead of guessing, the algorithm creates *meta* type variables $\widehat{\alpha}, \widehat{\beta}$ which are waiting to be solved. The judgment for the algorithmic type system is

$$(S_0, N_0); \Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma \hookrightarrow (S_1, N_1)$$

Here we use N as name supply, from which we can always extract new names. Also, every time a meta type variable is solved, we need to record its solution. We use S as a notation for the substitution that maps meta type variables to their solutions. For example, rule AP-INF-LAM becomes

$$\frac{(S_0, N_0); \Psi, x : \widehat{\beta} \vdash^{AP} \lambda x.\, e \Rightarrow \sigma \hookrightarrow (S_1, N_1)}{(S_0, N_0\widehat{\beta}); \Psi \vdash^{AP} \lambda x.\, e \Rightarrow \widehat{\beta} \rightarrow \sigma \hookrightarrow (S_1, N_1)} \ \text{AP-INF-LAM-ALGO}$$

Comparing it to rule rule AP-INF-LAM, τ is replaced by a new meta type variable $\widehat{\beta}$ from name supply $N_0\widehat{\beta}$. But despite of the name supply and substitution, the rule retains the structure of rule AP-INF-LAM.

Having the name supply and substitutions, the algorithmic system is a direct extension of the specification in Figure 3.2, with a process to do unifications that solve meta type variables. Such unification process is quite standard and similar to the one used in the Hindley-Milner system. We proved our algorithm is sound and complete with respect to the specification. Here *fmv* means free meta type variables.

Theorem 3.14 (Soundness). *If* $([], N_0); \Psi \vdash^{AP} e \Rightarrow \sigma \hookrightarrow (S_1, N_1)$*, then for any substitution* V *with* $dom(V) = fmv\ (S_1\Psi, S_1\sigma)$*, we have* $VS_1\Psi \vdash^{AP} e \Rightarrow VS_1\sigma$.

Theorem 3.15 (Completeness). *If* $\Psi \vdash^{AP} e \Rightarrow \sigma$*, then for a fresh* N_0*, we have* $([], N_0); \Psi \vdash^{AP} e \Rightarrow \sigma_2 \hookrightarrow (S_1, N_1)$*, and for some* S_2*, if* $\overline{a_i}^i = \textsc{fv}\,(\Psi) - \textsc{fv}(S_2S_1\sigma_2)$*, and* $\overline{b_i}^i = \textsc{fv}\,(\Psi) - \textsc{fv}\,(\sigma)$*, we have* $\vdash^{AP} \forall\overline{a_i}^i.S_2S_1\sigma_2 <: \forall\overline{b_i}^i.\sigma$.

3.5 Discussion

3.5.1 Combining Application and Checking Modes

Although the application mode provides us with alternative design choices in a bi-directional type system, a checking mode can still be *easily* added. One motivation for the checking mode would be annotated expressions $e : \sigma$, where the type of expressions is known and is therefore used to check expressions, as in DK.

Consider adding $e : \sigma$ for introducing the checking mode for the language. Notice that, since the checking mode is stronger than application mode, when entering checking mode the application context is no longer useful. Instead we use application subtyping to satisfy the application context requirements. A possible typing rule for annotation expressions is:

$$\textsc{ap-app-anno}\quad \frac{\Psi \vdash^{AP} e \Leftarrow \sigma_1 \qquad \Sigma \vdash^{AP} \sigma_1 <: \sigma_2}{\Psi; \Sigma \vdash^{AP} e : \sigma_1 \Rightarrow \sigma_2}$$

Here, e is checking using its annotation σ_1, and then we instantiate σ_1 to σ_2 using subtyping with application context Σ.

Now we can have a rule set of the checking mode for all expressions, much like those checking rules in DK. For example, one useful rule for abstractions in checking mode could be rule ap-chk-lam, where the parameter type σ_1 serves as the type of x, and typing checks the body with σ_2. Also, combined with the information flow, the checking rule for application checks the function with the full type.

$$\textsc{ap-chk-lam}\quad \frac{\Psi, x : \sigma_1 \vdash^{AP} e \Leftarrow \sigma_2}{\Psi \vdash^{AP} \lambda x.\, e \Leftarrow \sigma_1 \to \sigma_2}$$

Moreover, combined with the information flow, the checked rule for application checks the function with the full type.

$$\text{AP-CHK-APP}\quad \frac{\Psi \vdash^{AP} e_2 \Rightarrow \sigma_1 \qquad \Psi \vdash^{AP} e_1 \Leftarrow \sigma_1 \to \sigma_2}{\Psi \vdash^{AP} e_1\, e_2 \Leftarrow \sigma_2}$$

Note that adding expression annotations might bring convenience for programmers, since annotations can be more freely placed in a program. For example, $(\lambda x.\, x\, 1) : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$ becomes valid. However this does not add expressive power, since programs that are typeable under expression annotations, would remain typeable after moving the annotations to bindings. For example the previous program is equivalent to $(\lambda x : \mathsf{Int} \to \mathsf{Int}.\, x\, 1)$.

This discussion is a sketch. We have not defined the corresponding declarative system nor algorithm. However we believe that the addition of a checked mode will *not* bring surprises to the meta-theory.

3.5.2 Additional Constructs

In this section, we show that the application mode is compatible with other constructs, by discussing how to add support for pairs in the language. A similar methodology would apply to other constructs like sum types, data types, if-then-else expressions and so on.

The introduction rule for pairs must be in the inference mode with an empty application context. Also, the subtyping rule for pairs is as expected.

$$\text{AP-INF-PAIR}\quad \frac{\Psi \vdash^{AP} e_1 \Rightarrow \sigma_1 \qquad \Psi \vdash^{AP} e_2 \Rightarrow \sigma_2}{\Psi \vdash^{AP} (e_1, e_2) \Rightarrow (\sigma_1, \sigma_2)} \qquad \text{AP-S-PAIR}\quad \frac{\vdash^{AP} \sigma_1 <: \sigma_3 \qquad \vdash^{AP} \sigma_2 <: \sigma_4}{\vdash^{AP} (\sigma_1, \sigma_2) <: (\sigma_3, \sigma_4)}$$

The application mode can apply to the elimination constructs of pairs. If one component of the pair is a function, for example, $\mathbf{fst}\,(\lambda x.\, x, 1)\, 2$, then it is possible to have a judgment with a non-empty application context. Therefore, we can use the application subtyping to account for the application contexts:

$$\text{AP-APP-FST}\quad \frac{\Psi \vdash^{AP} e \Rightarrow (\sigma_1, \sigma_2) \qquad \Sigma \vdash^{AP} \sigma_1 <: \sigma_3}{\Psi; \Sigma \vdash^{AP} \mathbf{fst}\, e \Rightarrow \sigma_3}$$

$$\text{AP-APP-SND}\quad \frac{\Psi \vdash^{AP} e \Rightarrow (\sigma_1, \sigma_2) \qquad \Sigma \vdash^{AP} \sigma_2 <: \sigma_3}{\Psi; \Sigma \vdash^{AP} \mathbf{snd}\, e \Rightarrow \sigma_3}$$

However, in polymorphic type systems, we need to take the subsumption rule into consideration. For example, in the expression $(\lambda x : \forall a.\,(a, b).\,\mathbf{fst}\,x)$, **fst** is applied to a polymorphic type. Interestingly, instead of a non-deterministic subsumption rule, having polymorphic types actually leads to a simpler solution. According to the philosophy of the application mode, the types of the arguments always flow into the functions. Therefore, instead of regarding **fst** *e* as an expression form, where *e* is itself an argument, we could regard **fst** as a function on its own, whose type is $\forall a.\,\forall b.\,(a, b) \to a$. Then as in the variable case, we use the subtyping rule to deal with application contexts. Thus the typing rules for **fst** and **snd** can be modeled as:

$$\textsc{ap-app-fst-var}\quad \frac{\Sigma \vdash^{AP} \forall a.\,\forall b.\,(a, b) \to a <: \sigma}{\Psi; \Sigma \vdash^{AP} \mathbf{fst} \Rightarrow \sigma} \qquad \textsc{ap-app-snd-var}\quad \frac{\Sigma \vdash^{AP} \forall a.\,\forall b.\,(a, b) \to b <: \sigma}{\Psi; \Sigma \vdash^{AP} \mathbf{snd} \Rightarrow \sigma}$$

Note that another way to model those two rules would be to simply have an initial typing environment $\Psi_{init} \equiv \mathbf{fst} : \forall a.\,\forall b.\,(a, b) \to a, \mathbf{snd} : \forall a.\,\forall b.\,(a, b) \to b$. In this case the elimination of pairs be dealt directly by the rule for variables.

An extended version of the calculus extended with rules for pairs (rule AP-INF-PAIR, rule AP-S-PAIR, rule AP-APP-FST-VAR and rule AP-APP-SND-VAR), has been formally studied. All the theorems presented before hold with the extension of pairs.

3.5.3 More Expressive Type Applications

The design choice of propagating arguments to functions was subject to consideration in the original work on local type inference [Pierce and Turner 2000], but was rejected due to possible non-determinism introduced by explicit type applications:

> *"It is possible, of course, to come up with examples where it would be beneficial to synthesize the argument types first and then use the resulting information to avoid type annotations in the function part of an application expression.... Unfortunately this refinement does not help infer the type of polymorphic functions. For example, we cannot uniquely determine the type of x in the expression* $(\mathbf{fun}[X](x)\,e)\,[\mathsf{Int}]\,3$."

As a response to this challenge, we also present an application of the application mode to a variant of System F [Xie and Oliveira 2018]. The development of the calculus shows that the application mode can actually work well with calculi with explicit type applications. Here we explain the key ideas of the design of the system, but refer to Xie and Oliveira [2018] for more details.

To explain the new design, consider the expression:

$$\Lambda a.(\lambda x : a.\, x + 1)\ \mathsf{Int}$$

which is not typeable in the traditional type system for System F. In System F the lambda abstractions do not account for the context of possible function applications. Therefore when type checking the inner body of the lambda abstraction, the expression $x + 1$ is ill-typed, because all that is known is that x has the (abstract) type a.

If we are allowed to propagate type information from arguments to functions, then we can verify that $a = \mathsf{Int}$ and $x + 1$ is well-typed. The key insight in the new type system is to use contexts to track type equalities induced by type applications. This enables us to type check expressions such as the body of the lambda above ($x + 1$). The key rules for type abstractions and type applications are:

$$\frac{\Psi; \Sigma, [[\Psi]\sigma_1] \vdash^{AP} e \Rightarrow \sigma_2}{\Psi; \Sigma \vdash^{AP} e\, \sigma_1 \Rightarrow \sigma_2}\ \textsc{ap-app-tapp} \qquad \frac{\Psi, a = \sigma_1; \Sigma \vdash^{AP} e \Rightarrow \sigma_2}{\Psi; \Sigma, [\sigma_1] \vdash^{AP} \Lambda a.e \Rightarrow \sigma_2}\ \textsc{ap-app-tlam}$$

For type applications, rule AP-APP-TAPP stores the type argument σ_1 into the application context. Since Ψ tracks type equalities, we apply Ψ as a type substitution to σ_1 (i.e., $[\Psi]\sigma_1$) Moreover, to distinguish between type arguments and types of term arguments, we put type arguments in brackets (i.e., $[[\Psi]\sigma_1]$). For type abstractions (rule AP-APP-TLAM), if the application context is non-empty, we put a new type equality between the type variable a and the type argument σ_1 into the context.

Now, back to the problematic expression $(\mathbf{fun}[X](x)\ e)\ [\mathsf{Int}]\ 3$, the type of x can be inferred as either X or Int since they are actually equivalent.

Sugar for Type Synonyms. In the same way that we can regard **let** expressions as syntactic sugar, in the new type system we further *gain built-in type synonyms for free*. A *type synonym* is a new name for an existing type. Type synonyms are common in languages such as Haskell. In our calculus a simple form of type synonyms can be desugared as follows:

$$\mathbf{type}\ a = \sigma\ \mathbf{in}\ e \rightsquigarrow (\Lambda a.e)\sigma$$

One practical benefit of such syntactic sugar is that it enables a direct encoding of a System F-like language with declarations (including type-synonyms). Although declarations are often viewed as a routine extension to a calculus, and are not formally studied, they are highly relevant in practice. Therefore, a more realistic formalization of a programming language should directly account for declarations. By providing a way to encode declarations, our new calculus enables a simple way to formalize declarations.

Type Abstraction. The type equalities introduced by type applications may seem like we are breaking System F type abstraction. However, we argue that *type abstraction* is still supported by our System F variant. For example:

$$\textbf{let}\ inc = \Lambda a.\lambda x : a.\, x + 1\ \textbf{in}\ inc\ \mathsf{Int}\ 1$$

(after desugaring) does *not* type-check, as in a System-F like language. In our type system lambda abstractions that are immediatelly applied to an argument, and unapplied lambda abstractions behave differently. Unapplied lambda abstractions are just like System F abstractions and retain type abstraction. The example above illustrates this. In contrast the typeable example $(\Lambda a.\lambda x : a.\, x + 1)\ \mathsf{Int}$, which uses a lambda abstraction directly applied to an argument, can be regarded as the desugared expression for **type** $a = \mathsf{Int}$ **in** $\lambda x : a.\, x + 1$.

3.5.4 Dependent Type Systems

Ningning: Move to future work.

One remark about the application mode is that the same idea is possibly applicable to systems with advanced features, where type inference is sophisticated or even undecidable. One promising application is, for instance, dependent type systems [Xi and Pfenning 1999]. Type systems with dependent types usually unify the syntax for terms and types, with a single lambda abstraction generalizing both type and lambda abstractions. Unfortunately, this means that the **let** desugar is not valid in those systems. As a concrete example, consider desugaring the expression **let** $a = \mathsf{Int}$ **in** $\lambda x : a.\, x + 1$ into $(\lambda a.\lambda x : a.\, x + 1)\ \mathsf{Int}$, which is ill-typed because the type of x in the abstraction body is a and not Int.

Because **let** cannot be encoded, declarations cannot be encoded either. Modeling declarations in dependently typed languages is a subtle matter, and normally requires some additional complexity [Severi and Poll 1994].

We believe that the same technique presented in Section 3.5.3 can be adapted into a dependently typed language to enable a **let** encoding. In a dependent type system with unified syntax for terms and types, we can combine the two forms in the typing context, i.e., $x : \sigma$ and $a = \sigma$, into a unified form $x = e : \sigma$. Then we can combine two application rules rule AP-APP-APP and rule AP-APP-TAPP into rule AP-APP-DAPP, and also two abstraction rules rule AP-APP-LAM and rule AP-APP-TLAM into rule AP-APP-DLAM.

$$\frac{\Psi \vdash^{AP} e_2 \Rightarrow \sigma_1 \qquad \Psi; \Sigma, e_2 : \sigma_1 \vdash^{AP} e_1 \Rightarrow \sigma_2}{\Psi; \Sigma \vdash^{AP} e_1\, e_2 \Rightarrow \sigma_2} \text{ AP-APP-DAPP}$$

$$\frac{\Psi, x = e_1 : \sigma_1; \Sigma \vdash^{AP} e \Rightarrow \sigma_2}{\Psi; \Sigma, e_1 : \sigma_1 \vdash^{AP} \lambda x.\, e \Rightarrow \sigma_2} \text{ AP-APP-DLAM}$$

With such rules it would be possible to handle declarations easily in dependent type systems. Note this is still a rough idea and we have not fully worked out the typing rules for this type system yet.

Part III

Higher-Rank Polymorphism and Gradual Typing

4 Gradually Typed Higher-Rank Polymorphism

Consistent subtyping is employed in some gradual type systems to validate type conversions. The original definition by Siek and Taha [2007b] serves as a guideline for designing gradual type systems with subtyping. Polymorphic types à la System F also induce a subtyping relation that relates polymorphic types to their instantiations. However Siek and Taha's definition is not adequate for polymorphic subtyping.

This section first proposes a generalization of consistent subtyping (Section 4.2) that is adequate for polymorphic subtyping, and subsumes the original definition by Siek and Taha. The new definition of consistent subtyping provides novel insights with respect to previous polymorphic gradual type systems, which did not employ consistent subtyping. We then present GPC, a gradually typed calculus for implicit higher-rank polymorphism that uses our new notion of consistent subtyping. We develop both declarative (Section 4.3) and bidirectional algorithmic versions (Section 4.4) for the type system. The algorithmic version employs techniques developed by DK (Section 2.3) for higher-rank polymorphism to deal with instantiation.

4.1 Introduction and Motivation

4.1.1 Background: Gradual Typing

Siek and Taha [2007b] developed a gradual type system for object-oriented languages that they call $\mathbf{FOb}^{?}_{<:}$. Central to gradual typing is the concept of *consistency* (written $\sim$) between gradual types, which are types that may involve the unknown type $?$. The intuition is that consistency relaxes the structure of a type system to tolerate unknown positions in a gradual type. They also defined the subtyping relation in a way that static type safety is preserved. Their key insight is that the unknown type $?$ is neutral to subtyping, with only $? <: ?$. Both relations are defined in Figure 4.1.

A primary contribution of their work is to show that consistency and subtyping are orthogonal. However, the orthogonality of consistency and subtyping does not lead to a de-

$\boxed{\sigma_1 <: \sigma_2}$ *(Subtyping)*

$$\frac{}{\mathsf{Int} <: \mathsf{Int}} \qquad \frac{}{\mathsf{Bool} <: \mathsf{Bool}} \qquad \frac{}{\mathsf{Float} <: \mathsf{Float}} \qquad \frac{}{\mathsf{Int} <: \mathsf{Float}}$$

$$\frac{\sigma_3 <: \sigma_1 \qquad \sigma_2 <: \sigma_4}{\sigma_1 \to \sigma_2 <: \sigma_3 \to \sigma_4} \qquad \frac{\overline{\sigma_i <: \sigma_i'}^{\,i}}{[\,\overline{l_i : \sigma_i}^{\,i}\,] <: [\,\overline{l_i : \sigma_i'}^{\,i}\,]} \qquad \frac{}{? <: ?}$$

$\boxed{\sigma_1 \sim \sigma_2}$ *(Type Consistency)*

$$\frac{}{\sigma \sim \sigma} \qquad \frac{}{\sigma \sim ?} \qquad \frac{}{? \sim \sigma} \qquad \frac{\sigma_1 \sim \sigma_3 \qquad \sigma_2 \sim \sigma_4}{\sigma_1 \to \sigma_2 \sim \sigma_3 \to \sigma_4} \qquad \frac{\overline{\sigma_i \sim \sigma_i'}^{\,i}}{[\,\overline{l_i : \sigma_i}^{\,i}\,] \sim [\,\overline{l_i : \sigma_i'}^{\,i}\,]}$$

Figure 4.1: Subtyping and type consistency in $\mathbf{FOb}^{?}_{<:}$

terministic relation. Thus Siek and Taha defined *consistent subtyping* (written $\lesssim$) based on a *restriction operator*, written $\sigma_1|_{\sigma_2}$ that “masks off” the parts of type σ_1 that are unknown in type σ_2. For example,

$$\begin{array}{lcl} \mathsf{Int} \to \mathsf{Int}|_{\mathsf{Bool}\to\mathsf{Bool}} & = & \mathsf{Int} \to ? \\ \mathsf{Bool} \to ?|_{\mathsf{Int}\to\mathsf{Int}} & = & \mathsf{Bool} \to ? \end{array}$$

The definition of the restriction operator is given below:

$$\begin{array}{l} \sigma|_{\sigma'} = \textbf{case}\ (\sigma, \sigma')\ \textbf{of} \\ \mid (\_, ?) \Rightarrow ? \\ \mid (\sigma_1 \to \sigma_2, \sigma_1' \to \sigma_2') \Rightarrow \sigma_1|_{\sigma_1'} \to \sigma_2|_{\sigma_2'} \\ \mid ([l_1 : \sigma_1, \ldots, l_n : \sigma_n], [l_1 : \sigma_1', \ldots, l_m : \sigma_m'])\ \textbf{if}\ n \leq m \Rightarrow [l_1 : \sigma_1|_{\sigma_1'}, \ldots, l_n : \sigma_n|_{\sigma_n'}] \\ \mid ([l_1 : \sigma_1, \ldots, l_n : \sigma_n], [l_1 : \sigma_1', \ldots, l_m : \sigma_m'])\ \textbf{if}\ n > m \Rightarrow [l_1 : \sigma_1|_{\sigma_1'}, \ldots, l_m : \sigma_m|_{\sigma_m'}, \ldots, l_n : \sigma_n] \\ \mid (\_, \_) \Rightarrow \sigma \end{array}$$

With the restriction operator, consistent subtyping is simply defined as:

Definition 3 (Algorithmic Consistent Subtyping of Siek and Taha [2007b]). $\sigma_1 \lesssim \sigma_2 \equiv \sigma_1|_{\sigma_2} <: \sigma_2|_{\sigma_1}$.

Later they show a proposition that consistent subtyping is equivalent to two declarative definitions, which we refer to as the strawman for *declarative* consistent subtyping because it servers as a good guideline on superimposing consistency and subtyping. Both definitions are non-deterministic because of the intermediate type σ_3.

Definition 4 (Strawman for Declarative Consistent Subtyping)**.** The following two are equivalent:

1. $\sigma_1 \lesssim \sigma_2$ if and only if $\sigma_1 \sim \sigma_3$ and $\sigma_3 <: \sigma_2$ for some σ_3.

2. $\sigma_1 \lesssim \sigma_2$ if and only if $\sigma_1 <: \sigma_3$ and $\sigma_3 \sim \sigma_2$ for some σ_3.

In our later discussion, it will always be clear which definition we are referring to. For example, we focus more on Definition 4 in Section 4.2.2, and more on Definition 3 in Section 4.2.5.

4.1.2 Motivation: Gradually Typed Higher-Rank Polymorphism

Our work combines implicit (higher-rank) polymorphism with gradual typing. As is well known, a gradually typed language supports both fully static and fully dynamic checking of program properties, as well as the continuum between these two extremes. It also offers programmers fine-grained control over the static-to-dynamic spectrum, i.e., a program can be evolved by introducing more or less precise types as needed [Garcia et al. 2016].

Haskell is a language renowned for its advanced type system, but it does not feature gradual typing. Of particular interest to us is its support for implicit higher-rank polymorphism, which is supported via explicit type annotations. In Haskell some programs that are safe at run-time may be rejected due to the conservativity of the type system. For example, consider again the example from Section 2.2:

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

This program is rejected by Haskell's type checker because Haskell implements the HM rule that a lambda-bound argument (such as `f`) can only have a monotype, i.e., the type checker can only assign `f` the type **Int** $\rightarrow$ **Int**, or `Char` $\rightarrow$ `Char`, but not $\forall$`a. a` $\rightarrow$ `a`. Finding such manual polymorphic annotations can be non-trivial, especially when the program scales up and the annotation is long and complicated.

Instead of rejecting the program outright, due to missing type annotations, gradual typing provides a simple alternative by giving `f` the unknown type $?$. With this type the same program type-checks and produces `(1, 'a')`. By running the program, programmers can gain more insight about its run-time behaviour. Then, with this insight, they can also give `f` a more precise type ($\forall$`a. a` $\rightarrow$ `a`) a posteriori so that the program continues to type-check via implicit polymorphism and also grants more static safety. In this paper, we envision such a language that combines the benefits of both implicit higher-rank polymorphism and gradual typing.

4.1.3 Application: Efficient (Partly) Typed Encodings of ADTs

We illustrate two concrete applications of gradually typed higher-rank polymorphism related to algebraic datatypes. The first application shows how gradual typing helps in defining Scott encodings of algebraic datatypes [Curry et al. 1958; Parigot 1992], which are impossible to encode in plain System F. The second application shows how gradual typing makes it easy to model and use heterogeneous containers.

Our calculus does not provide built-in support for algebraic datatypes (ADTs). Nevertheless, the calculus is expressive enough to support efficient function-based encodings of (optionally polymorphic) ADTs[1]. This offers an immediate way to model algebraic datatypes in our calculus without requiring extensions to our calculus or, more importantly, to its target—the polymorphic blame calculus. While we believe that such extensions are possible, they would likely require non-trivial extensions to the polymorphic blame calculus. Thus the alternative of being able to model algebraic datatypes without extending $\lambda\mathsf{B}$ is appealing. The encoding also paves the way to provide built-in support for algebraic datatypes in the source language, while elaborating them via the encoding into $\lambda\mathsf{B}$.

Church and Scott Encodings. It is well-known that polymorphic calculi such as System F can encode datatypes via Church encodings. However these encodings have well-known drawbacks. In particular, some operations are hard to define, and they can have a time complexity that is greater than that of the corresponding functions for built-in algebraic datatypes. A good example is the definition of the predecessor function for Church numerals [Church 1941]. Its definition requires significant ingenuity (while it is trivial with built-in algebraic datatypes), and it has *linear* time complexity (versus the *constant* time complexity for a definition using built-in algebraic datatypes).

An alternative to Church encodings are the so-called Scott encodings [Curry et al. 1958]. They address the two drawbacks of Church encodings: they allow simple definitions that directly correspond to programs implemented with built-in algebraic datatypes, and those definitions have the same time complexity to programs using algebraic datatypes.

Unfortunately, Scott encodings, or more precisely, their typed variant [Parigot 1992], cannot be expressed in System F: in the general case they require recursive types, which System

[1] In a type system with impure features, such as non-termination or exceptions, the encoded types can have valid inhabitants with side-effects, which means we only get the *lazy* version of those datatypes.

F does not support. However, with gradual typing, we can remove the need for recursive types, thus enabling Scott encodings in our calculus.

A Scott Encoding of Parametric Lists Consider for instance the typed Scott encoding of parametric lists in a system with implicit polymorphism:

$$\begin{aligned}
\mathsf{List}\, a &\triangleq \mu L.\, \forall b.\, b \to (a \to L \to b) \to b \\
\mathsf{nil} &\triangleq \mathbf{fold}_{\mathsf{List}\, a}\, (\lambda m.\, \lambda c.\, m) : \forall a.\, \mathsf{List}\, a \\
\mathsf{cons} &\triangleq \lambda x.\, \lambda xs.\, \mathbf{fold}_{\mathsf{List}\, a}\, (\lambda m.\, \lambda c.\, c\, x\, xs) : \forall a.\, a \to \mathsf{List}\, a \to \mathsf{List}\, a
\end{aligned}$$

This encoding requires both polymorphic and recursive types[2]. Like System F, our calculus only supports the former, but not the latter. Nevertheless, gradual types still allow us to use the Scott encoding in a partially typed fashion. The trick is to omit the recursive type binder μL and replace the recursive occurrence of L by the unknown type *?*:

$$\mathsf{List}_?\, a \triangleq \forall b.\, b \to (a \to ? \to b) \to b$$

As a consequence, we need to replace the term-level witnesses of the iso-recursion by explicit type annotations to respectively forget or recover the type structure of the recursive occurrences:

$$\begin{aligned}
\mathbf{fold}_{\mathsf{List}_?\, a} &\triangleq \lambda x.\, x : (\forall b.\, b \to (a \to \mathsf{List}_?\, a \to b) \to b) \to \mathsf{List}_?\, a \\
\mathbf{unfold}_{\mathsf{List}_?\, a} &\triangleq \lambda x.\, x : \mathsf{List}_?\, a \to (\forall b.\, b \to (a \to \mathsf{List}_?\, a \to b) \to b)
\end{aligned}$$

With the reinterpretation of **fold** and **unfold** as functions instead of built-in primitives, we have exactly the same definitions of $\mathsf{nil}_?$ and $\mathsf{cons}_?$.

Note that when we elaborate our calculus into the polymorphic blame calculus, the above type annotations give rise to explicit casts. For instance, after elaboration $\mathbf{fold}_{\mathsf{List}_?\, a}\, e$ results in the cast $\langle(\forall b.\, b \to (a \to \mathsf{List}_?\, a \to b) \to b) \hookrightarrow \mathsf{List}_?\, a\rangle s$ where s is the elaboration of e.

In order to perform recursive traversals on lists, e.g., to compute their length, we need a fixpoint combinator like the Y combinator. Unfortunately, this combinator cannot be assigned a type in the simply typed lambda calculus or System F. Yet, we can still provide a gradual typing for it in our system.

$$\mathsf{fix} \triangleq \lambda f.\, (\lambda x : ?.\, f\,(x\, x))\, (\lambda x : ?.\, f\,(x\, x)) : \forall a.\, (a \to a) \to a$$

[2]Here we use iso-recursive types, but equi-recursive types can be used too.

This allows us for instance to compute the length of a list.

$$\mathsf{length} \triangleq \mathsf{fix}\,(\lambda len.\, \lambda l.\, \mathsf{zero}_?\,(\lambda xs.\, \mathsf{succ}_?\,(len\, xs)))$$

Here $\mathsf{zero}_? : \mathsf{Nat}_?$ and $\mathsf{succ}_? : \mathsf{Nat}_? \to \mathsf{Nat}_?$ are the encodings of the constructors for natural numbers $\mathsf{Nat}_?$. In practice, for performance reasons, we could extend our language with a **letrec** construct in a standard way to support general recursion, instead of defining a fixpoint combinator.

Observe that the gradual typing of lists still enforces that all elements in the list are of the same type. For instance, a heterogeneous list like $\mathsf{cons}_?\,\mathsf{zero}_?\,(\mathsf{cons}_?\,\mathsf{true}_?\,\mathsf{nil}_?)$, is rejected because $\mathsf{zero}_? : \mathsf{Nat}_?$ and $\mathsf{true}_? : \mathsf{Bool}_?$ have different types.

Heterogeneous Containers. Heterogeneous containers are datatypes that can store data of different types, which is very useful in various scenarios. One typical application is that an XML element is heterogeneously typed. Moreover, the result of a SQL query contains heterogeneous rows.

In statically typed languages, there are several ways to obtain heterogeneous lists. For example, in Haskell, one option is to use *dynamic types*. Haskell's library **Data.Dynamic** provides the special type **Dynamic** along with its injection **toDyn** and projection **fromDyn**. The drawback is that the code is littered with **toDyn** and **fromDyn**, which obscures the program logic. One can also use the HList library [Kiselyov et al. 2004], which provides strongly typed data structures for heterogeneous collections. The library requires several Haskell extensions, such as multi-parameter classes [Peyton Jones et al. 1997] and functional dependencies [Jones 2000]. With fake dependent types [McBride 2002], heterogeneous vectors are also possible with type-level constructors.

In our type system, with explicit type annotations that set the element types to the unknown type we can disable the homogeneous typing discipline for the elements and get gradually typed heterogeneous lists[3]. Such gradually typed heterogeneous lists are akin to Haskell's approach with Dynamic types, but much more convenient to use since no injections and projections are needed, and the $?$ type is built-in and natural to use.

An example of such gradually typed heterogeneous collections is:

$$l \triangleq \mathsf{cons}_?\,(\mathsf{zero}_? : ?)\,(\mathsf{cons}_?\,(\mathsf{true}_? : ?)\,\mathsf{nil}_?)$$

Here we annotate each element with type annotation $?$ and the type system is happy to type-check that $l : \mathsf{List}_?\,?$. Note that we are being meticulous about the syntax, but with proper

[3] This argument is based on the extended type system in Chapter 5.

implementation of the source language, we could write more succinct programs akin to Haskell's syntax, such as `[0, True]`.

4.2 Revisiting Consistent Subtyping

In this section we explore the design space of consistent subtyping. We start with the definitions of consistency and subtyping for polymorphic types, and compare with some relevant work. We then discuss the design decisions involved in our new definition of consistent subtyping, and justify the new definition by demonstrating its equivalence with that of Siek and Taha [2007b] and the AGT approach [Garcia et al. 2016] on simple types.

The syntax of types is given at the top of Figure 4.2. Types σ are either the integer type Int, type variables a, functions types $\sigma_1 \to \sigma_2$, universal quantification $\forall a.\, \sigma$, or the unknown type $?$. Note that monotypes τ contain all types other than the universal quantifier and the unknown type $?$. We will discuss this restriction when we present the subtyping rules. Contexts Ψ are *ordered* lists of type variable declarations and term variables.

4.2.1 Consistency and Subtyping

We start by giving the definitions of consistency and subtyping for polymorphic types, and comparing our definitions with the compatibility relation by Ahmed et al. [2009] and type consistency by Igarashi et al. [2017].

Consistency. The key observation here is that consistency is mostly a structural relation, except that the unknown type $?$ can be regarded as any type. In other words, consistency is an equivalence relation lifted from static types to gradual types [Garcia et al. 2016]. Following this observation, we naturally extend the definition from Figure 4.1 with polymorphic types, as shown in the middle of Figure 4.2. In particular a polymorphic type $\forall a.\, \sigma$ is consistent with another polymorphic type $\forall a.\, \sigma_2$ if σ is consistent with σ_2.

Subtyping. We express the fact that one type is a polymorphic generalization of another by means of the subtyping judgment $\Psi \vdash^G \sigma <: \sigma_2$. Compared with the subtyping rules of Odersky and Läufer [1996] in Figure 2.5, the only addition is the neutral subtyping of $?$. Notice that, in rule gpc-s-forallL, the universal quantifier is only allowed to be instantiated with a *monotype*. The judgment $\Psi \vdash^G \sigma$ checks whether all the type variables in σ are bound in the context Ψ. According to the syntax in Figure 4.2, monotypes do not include the unknown type $?$. This is because if we were to allow the unknown type to be used for instantiation, we could have $\forall a.\, a \to a <: ? \to ?$ by instantiating a with $?$. Since $? \to ?$ is

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

$\boxed{\sigma \sim \sigma_2}$ *(Type Consistency)*

$$\frac{}{\sigma \sim \sigma} \qquad \frac{}{\sigma \sim ?} \qquad \frac{}{? \sim \sigma} \qquad \frac{\sigma_1 \sim \sigma_3 \qquad \sigma_2 \sim \sigma_4}{\sigma_1 \to \sigma_2 \sim \sigma_3 \to \sigma_4} \qquad \frac{\sigma \sim \sigma_2}{\forall a.\, \sigma \sim \forall a.\, \sigma_2}$$

$\boxed{\Psi \vdash^G \sigma <: \sigma_2}$ *(Subtyping)*

$$\frac{a \in \Psi}{\Psi \vdash^G a <: a}\ \textsc{gpc-s-tvar} \qquad \frac{}{\Psi \vdash^G \mathsf{Int} <: \mathsf{Int}}\ \textsc{gpc-s-int}$$

$$\frac{\Psi \vdash^G \sigma_3 <: \sigma_1 \qquad \Psi \vdash^G \sigma_2 <: \sigma_4}{\Psi \vdash^G \sigma_1 \to \sigma_2 <: \sigma_3 \to \sigma_4}\ \textsc{gpc-s-arrow}$$

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

$$\frac{}{\Psi \vdash^G ? <: ?}\ \textsc{gpc-s-unknown}$$

$\boxed{\Psi \vdash^G \sigma}$ *(Well-formedness of types)*

$$\frac{}{\Psi \vdash^G \mathsf{Int}} \qquad \frac{}{\Psi \vdash^G ?} \qquad \frac{a \in \Psi}{\Psi \vdash^G a} \qquad \frac{\Psi \vdash^G \sigma \qquad \Psi \vdash^G \sigma_2}{\Psi \vdash^G \sigma \to \sigma_2} \qquad \frac{\Psi, a \vdash^G \sigma}{\Psi \vdash^G \forall a.\, \sigma}$$

Figure 4.2: Syntax of types, consistency, subtyping and well-formedness of types in declarative GPC.

consistent with any functions $\sigma_1 \to \sigma_2$, for instance, $\mathsf{Int} \to \mathsf{Bool}$, this means that we could provide an expression of type $\forall a.\, a \to a$ to a function where the input type is supposed to be $\mathsf{Int} \to \mathsf{Bool}$. However, as we know, $\forall a.\, a \to a$ is definitely not compatible with $\mathsf{Int} \to \mathsf{Bool}$. Indeed, this does not hold in any polymorphic type systems without gradual typing. So the gradual type system should not accept it either. (This is the *conservative extension* property that will be made precise in Section 4.3.3.)

Importantly there is a subtle distinction between a type variable and the unknown type, although they both represent a kind of "arbitrary" type. The unknown type stands for the absence of type information: it could be *any type* at *any instance*. Therefore, the unknown type is consistent with any type, and additional type-checks have to be performed at runtime. On the other hand, a type variable indicates *parametricity*. In other words, a type variable can only be instantiated to a single type. For example, in the type $\forall a.\, a \to a$, the two occurrences of a represent an arbitrary but single type (e.g., $\mathsf{Int} \to \mathsf{Int}$, $\mathsf{Bool} \to \mathsf{Bool}$), while $? \to ?$ could be an arbitrary function (e.g., $\mathsf{Int} \to \mathsf{Bool}$) at runtime.

Comparison with Other Relations. In other polymorphic gradual calculi, consistency and subtyping are often mixed up to some extent. In λB [Ahmed et al. 2009], the compatibility relation for polymorphic types is defined as follows:

$$\frac{\sigma_1 \prec \sigma_2}{\sigma_1 \prec \forall a.\, \sigma_2}\ \textsc{Comp-AllR} \qquad\qquad \frac{\sigma_1[a \mapsto ?] \prec \sigma_2}{\forall a.\, \sigma_1 \prec \sigma_2}\ \textsc{Comp-AllL}$$

Notice that, in rule Comp-AllL, the universal quantifier is *always* instantiated to ?. However, this way, λB allows $\forall a.\, a \to a \prec \mathsf{Int} \to \mathsf{Bool}$, which as we discussed before might not be what we expect. Indeed λB relies on sophisticated runtime checks to rule out such instances of the compatibility relation a posteriori.

Igarashi et al. [2017] introduced the so-called *quasi-polymorphic* types for types that may be used where a ∀-type is expected, which is important for their purpose of conservativity over System F. Their type consistency relation, involving polymorphism, is defined as follows[4]:

$$\frac{\sigma \sim \sigma_2}{\forall a.\, \sigma \sim \forall a.\, \sigma_2} \qquad\qquad \frac{\sigma \sim \sigma_2 \qquad \sigma_2 \neq \forall a.\, \sigma_2' \qquad ? \in \mathsf{Types}(\sigma_2)}{\forall a.\, \sigma \sim \sigma_2}$$

[4]This is a simplified version. These two rules are presented in Section 3.1 in their paper as one of the key ideas of the design of type consistency, which are later amended with *labels*.

Compared with our consistency definition in Figure 4.2, their first rule is the same as ours. The second rule says that a non $\forall$-type can be consistent with a $\forall$-type only if it contains *?*. In this way, their type system is able to reject $\forall a.\, a \rightarrow a \sim \mathsf{Int} \rightarrow \mathsf{Bool}$. However, in order to keep conservativity, they also reject $\forall a.\, a \rightarrow a \sim \mathsf{Int} \rightarrow \mathsf{Int}$, which is perfectly sensible in their setting of explicit polymorphism. However with implicit polymorphism, we would expect $\forall a.\, a \rightarrow a$ to be related with $\mathsf{Int} \rightarrow \mathsf{Int}$, since a can be instantiated to Int.

Nonetheless, when it comes to interactions between dynamically typed and polymorphically typed terms, both relations allow $\forall a.\, a \rightarrow \mathsf{Int}$ to be related with $? \rightarrow \mathsf{Int}$ for example, which in our view, is a kind of (implicit) polymorphic subtyping combined with type consistency, and that should be derivable by the more primitive notions in the type system (instead of inventing new relations). One of our design principles is that subtyping and consistency are *orthogonal*, and can be naturally superimposed, echoing the opinion of Siek and Taha [2007b].

4.2.2 Towards Consistent Subtyping

With the definitions of consistency and subtyping, the question now is how to compose the two relations so that two types can be compared in a way that takes both relations into account.

Unfortunately, the strawman version of consistent subtyping (Definition 4) does not work well with our definitions of consistency and subtyping for polymorphic types. Consider two types: $(\forall a.\, a \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int}$, and $(? \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int}$. The first type can only reach the second type in one way (first by applying consistency, then subtyping), but not the other way, as shown in Figure 4.3a. We use $\emptyset$ to mean that we cannot find such a type. Similarly, there are situations where the first type can only reach the second type by the other way (first applying subtyping, and then consistency), as shown in Figure 4.3b.

What is worse, if those two examples are composed in a way that those types all appear co-variantly, then the resulting types cannot reach each other in either way. For example, Figure 4.3c shows two such types by putting a Bool type in the middle, and neither definition of consistent subtyping works.

Observations on Consistent Subtyping Based on Information Propagation. In order to develop a correct definition of consistent subtyping for polymorphic types, we need to understand how consistent subtyping works. We first review two important properties of subtyping: (1) subtyping induces the subsumption rule: if $\sigma_1 <: \sigma_2$, then an expression of type σ_1 can be used where σ_2 is expected; (2) subtyping is transitive: if $\sigma_1 <: \sigma_2$, and $\sigma_2 <: \sigma_3$, then $\sigma_1 <: \sigma_3$. Though consistent subtyping takes the unknown type into consid-

$$\begin{tikzcd}
\emptyset \arrow[r, dash, "\sim"] & (? \to \mathsf{Int}) \to \mathsf{Int} \\
(\forall a.\, a \to \mathsf{Nat}) \to \mathsf{Nat} \arrow[u, "<:"] \arrow[r, dash, "\sim"'] & (\forall a.\, ? \to \mathsf{Nat}) \to \mathsf{Nat} \arrow[u, "<:"']
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
\mathsf{Int} \to \mathsf{Int} \arrow[r, dash, "\sim"] & \mathsf{Int} \to ? \\
\forall a.\, a \arrow[u, "<:"] \arrow[r, dash, "\sim"'] & \emptyset \arrow[u, "<:"']
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
\emptyset \arrow[r, dash, "\sim"] & (((? \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to ?) \\
(((\forall a.\, a \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\forall a.\, a) \arrow[u, "<:"] \arrow[r, dash, "\sim"'] & \emptyset \arrow[u, "<:"']
\end{tikzcd}$$

(c)

Figure 4.3: Examples that break the original definition of consistent subtyping.

$$\begin{tikzcd}
\sigma' \arrow[r, dash, "\sim"] & \sigma_3 \\
\sigma_2 \arrow[u, "<:"] \arrow[ur, dashed, "\lesssim"] \arrow[r, dash, "\sim"'] & \sigma'' \arrow[u, "<:"'] \\
\sigma_1 \arrow[u, "<:"] \arrow[uur, dashed, bend right, "\lesssim"']
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
 & \sigma_1 \\
\sigma' \arrow[r, dash, "\sim"] & \sigma_2 \arrow[u, "<:"'] \\
\sigma_3 \arrow[u, "<:"] \arrow[ur, dashed, "\lesssim"] \arrow[r, dash, "\sim"'] \arrow[uur, dashed, bend left, "\lesssim"] & \sigma'' \arrow[u, "<:"']
\end{tikzcd}$$

(b)

Figure 4.4: Observations of consistent subtyping

eration, the subsumption rule should also apply: if $\sigma_1 \lesssim \sigma_2$, then an expression of type σ_1 can also be used where σ_2 is expected, given that there might be some information lost by consistency. A crucial difference from subtyping is that consistent subtyping is *not* transitive because information can only be lost once (otherwise, any two types are a consistent subtype of each other). Now consider a situation where we have both $\sigma_1 <: \sigma_2$, and $\sigma_2 \lesssim \sigma_3$, this means that σ_1 can be used where σ_2 is expected, and σ_2 can be used where σ_3 is expected, with possibly some loss of information. In other words, we should expect that σ_1 can be used where σ_3 is expected, since there is at most one-time loss of information.

Observation 1. If $\sigma_3 \lesssim \sigma_2$, and $\sigma_2 <: \sigma_1$, then $\sigma_3 \lesssim \sigma_1$.

This is reflected in Figure 4.4a. A symmetrical observation is given in Figure 4.4b:

Observation 2. If $\sigma_3 \lesssim \sigma_2$, and $\sigma_2 <: \sigma$, then $\sigma_3 \lesssim \sigma$.

$$
\begin{tikzcd}
\sigma_2 \arrow[r, dash, "\sim"] & \sigma_3 \arrow[d, "<:"'] \\
\sigma_1 \arrow[u, "<:"] \arrow[r, dashed, bend right, "\lesssim"] & \sigma_4
\end{tikzcd}
$$

$$
\begin{aligned}
\sigma_1 &= (((\forall a.\, a \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\forall a.\, a) \\
\sigma_2 &= (((\forall a.\, a \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to \mathsf{Int}) \\
\sigma_3 &= (((\forall a.\, ? \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to ?) \\
\sigma_4 &= (((? \to \mathsf{Int}) \to \mathsf{Int}) \to \mathsf{Bool}) \to (\mathsf{Int} \to ?)
\end{aligned}
$$

Figure 4.5: Example that is fixed by the new definition of consistent subtyping.

From the above observations, we see what the problem is with the original definition. In Fig. 4.4a, if σ_2 can reach σ_3 by σ', then by subtyping transitivity, σ_1 can reach σ_3 by σ'. However, if σ_2 can only reach σ_3 by σ'', then σ cannot reach σ_3 through the original definition. A similar problem is shown in Fig. 4.4b.

It turns out that these two problems can be fixed using the same strategy: instead of taking one-step subtyping and one-step consistency, our definition of consistent subtyping allows types to take *one-step subtyping, one-step consistency, and one more step subtyping*. Specifically, $\sigma_1 <: \sigma_2 \sim \sigma'' <: \sigma_3$ (in Figure 4.4a) and $\sigma_3 <: \sigma' \sim \sigma_2 <: \sigma$ (in Figure 4.4b) have the same relation chain: subtyping, consistency, and subtyping.

Definition of Consistent subtyping. From the above discussion, we are ready to modify Definition 4, and adapt it to our notation[5]:

Definition 5 (Consistent Subtyping). $\Psi \vdash^G \sigma_1 \lesssim \sigma_2$ if and only if $\Psi \vdash^G \sigma_1 <: \sigma'$, $\sigma' \sim \sigma''$ and $\Psi \vdash^G \sigma'' <: \sigma_2$ for some σ' and σ''.

With Definition 5, Figure 4.5 illustrates the correct relation chain for the broken example shown in Figure 4.3c.

At first sight, Definition 5 seems worse than the original: we need to guess *two* types! It turns out that Definition 5 is a generalization of Definition 4, and they are equivalent in the system of Siek and Taha [2007b]. However, more generally, Definition 5 is compatible with polymorphic types.

Proposition 4.1 (Generalization of Declarative Consistent Subtyping).

[5]For readers who are familiar with category theory, this defines consistent subtyping as the least subtyping bimodule extending consistency.

- *Definition 5 subsumes Definition 4.*
  In Definition 5, by choosing $\sigma'' = \sigma_2$, we have $\sigma_1 <: \sigma'$ and $\sigma' \sim \sigma_2$; by choosing $\sigma' = \sigma_1$, we have $\sigma_1 \sim \sigma''$, and $\sigma'' <: \sigma_2$.
- *Definition 4 is equivalent to Definition 5 in the system of Siek and Taha.*
  If $\sigma_1 <: \sigma'$, $\sigma' \sim \sigma''$, and $\sigma'' <: \sigma_2$, by Definition 4, $\sigma_1 \sim \sigma_3$, $\sigma_3 <: \sigma''$ for some σ_3. By subtyping transitivity, $\sigma_3 <: \sigma_2$. So $\sigma_1 \lesssim \sigma_2$ by $\sigma_1 \sim \sigma_3$ and $\sigma_3 <: \sigma_2$.

4.2.3 Abstracting Gradual Typing

Garcia et al. [2016] presented a new foundation for gradual typing that they call the *Abstracting Gradual Typing* (AGT) approach. In the AGT approach, gradual types are interpreted as sets of static types, where static types refer to types containing no unknown types. In this interpretation, predicates and functions on static types can then be lifted to apply to gradual types. Central to their approach is the so-called *concretization* function. For simple types, a concretization γ from gradual types to a set of static types is defined as follows:

Definition 6 (Concretization).

$$\gamma(\mathsf{Int}) = \{\mathsf{Int}\}$$

$$\gamma(\sigma_1 \to \sigma_2) = \{\sigma_1' \to \sigma_2' \mid \sigma_1' \in \gamma(\sigma_1), \sigma_2' \in \gamma(\sigma_2)\}$$

$$\gamma(?) = \{\text{All static types}\}$$

Based on the concretization function, subtyping between static types can be lifted to gradual types, resulting in the consistent subtyping relation:

Definition 7 (Consistent Subtyping in AGT). $\sigma_1 \mathrel{\widetilde{<:}} \sigma_2$ if and only if $\sigma_1' <: \sigma_2'$ for some *static types* σ_1' and σ_2' such that $\sigma_1' \in \gamma(\sigma_1)$ and $\sigma_2' \in \gamma(\sigma_2)$.

Later they proved that this definition of consistent subtyping coincides with that of Definition 4. By Proposition 4.1, we can directly conclude that our definition coincides with AGT:

Corollary 4.2 (Equivalence to AGT on Simple Types). *$\sigma_1 \lesssim \sigma_2$ if and only if $\sigma_1 \mathrel{\widetilde{<:}} \sigma_2$.*

However, AGT does not show how to deal with polymorphism (e.g. the interpretation of type variables) yet. Still, as noted by Garcia et al. [2016], it is a promising line of future work for AGT, and the question remains whether our definition would coincide with it.

Another note related to AGT is that the definition is later adopted by Castagna and Lanvin [2017] in a gradual type system with union and intersection types, where the static types

σ_1', σ_2' in Definition 7 can be algorithmically computed by also accounting for top and bottom types.

4.2.4 Directed Consistency

Directed consistency [Jafery and Dunfield 2017] is defined in terms of precision and subtyping:

$$\frac{\sigma_1' \sqsubseteq \sigma_1 \qquad \sigma_1 <: \sigma_2 \qquad \sigma_2' \sqsubseteq \sigma_2}{\sigma_1' \lesssim \sigma_2'}$$

The judgment $\sigma_1 \sqsubseteq \sigma_2$ is read "σ_1 is less precise than σ_2".[6] In their setting, precision is first defined for type constructors and then lifted to gradual types, and subtyping is defined for gradual types. If we interpret this definition from the AGT point of view, finding a more precise static type has the same effect as concretization. Namely, $\sigma_1' \sqsubseteq \sigma_1$ implies $\sigma_1 \in \gamma(\sigma_1')$ and $\sigma_2' \sqsubseteq \sigma_2$ implies $\sigma_2 \in \gamma(\sigma_2')$ if σ_1 and σ_2 are static types. Therefore we consider this definition as AGT-style. From this perspective, this definition naturally coincides with Definition 7, and by Corollary 4.2, it coincides with Definition 5.

The value of their definition is that consistent subtyping is derived compositionally from *gradual subtyping* and *precision*. Arguably, gradual types play a role in both definitions, which is different from Definition 5 where subtyping is neutral to unknown types. Still, the definition is interesting as it takes precision into consideration, rather than consistency. Then a question arises as to *how are consistency and precision related*.

Consistency and Precision. Precision is a partial order (anti-symmetric and transitive), while consistency is symmetric but not transitive. Recall that consistency is in fact an equivalence relation lifted from static types to gradual types [Garcia et al. 2016], which embodies the key role of gradual types in typing. Therefore defining consistency independently is straightforward, and it is theoretically viable to validate the definition of consistency directly. On the other hand, precision is usually connected with the gradual criteria [Siek et al. 2015], and finding a correct partial order that adheres to the criteria is not always an easy task. For example, Igarashi et al. [2017] argued that term precision for gradual System F is actually nontrivial, leaving the gradual guarantee of the semantics as a conjecture. Thus precision can be difficult to extend to more sophisticated type systems, e.g. dependent types.

[6] Jafery and Dunfield actually read $\sigma_1 \sqsubseteq \sigma_2$ as "σ_1 is *more precise* than σ_2". We, however, use the "less precise" notation (which is also adopted by Cimini and Siek [2016]) throughout the paper. The full rules can be found in Figure 4.8.

Nonetheless, in our system, precision and consistency can be related by the following lemma:

Lemma 4.3 (Consistency and Precision).

- *If $\sigma_1 \sim \sigma_2$, then there exists (static) σ_3, such that $\sigma_1 \sqsubseteq \sigma_3$, and $\sigma_2 \sqsubseteq \sigma_3$.*
- *If for some (static) σ_3, we have $\sigma_1 \sqsubseteq \sigma_3$, and $\sigma_2 \sqsubseteq \sigma_3$, then we have σ_1 σ_2.*

4.2.5 Consistent Subtyping Without Existentials

Definition 5 serves as a fine specification of how consistent subtyping should behave in general. But it is inherently non-deterministic because of the two intermediate types σ' and σ''. As Definition 3, we need a combined relation to directly compare two types. A natural attempt is to try to extend the restriction operator for polymorphic types. Unfortunately, as we show below, this does not work. However it is possible to devise an equivalent inductive definition instead.

Attempt to Extend the Restriction Operator. Suppose that we try to extend Definition 3 to account for polymorphic types. The original restriction operator is structural, meaning that it works for types of similar structures. But for polymorphic types, two input types could have different structures due to universal quantifiers, e.g, $\forall a.\, a \to \mathsf{Int}$ and $(\mathsf{Int} \to ?) \to \mathsf{Int}$. If we try to mask the first type using the second, it seems hard to maintain the information that a should be instantiated to a function while ensuring that the return type is masked. There seems to be no satisfactory way to extend the restriction operator in order to support this kind of non-structural masking.

Interpretation of the Restriction Operator and Consistent Subtyping. If the restriction operator cannot be extended naturally, it is useful to take a step back and revisit what the restriction operator actually does. For consistent subtyping, two input types could have unknown types in different positions, but we only care about the known parts. What the restriction operator does is (1) erase the type information in one type if the corresponding position in the other type is the unknown type; and (2) compare the resulting types using the normal subtyping relation. The example below shows the masking-off procedure for the types $\mathsf{Int} \to ? \to \mathsf{Bool}$ and $\mathsf{Int} \to \mathsf{Int} \to ?$. Since the known parts have the relation that $\mathsf{Int} \to ? \to ? <: \mathsf{Int} \to ? \to ?$, we conclude that $\mathsf{Int} \to ? \to \mathsf{Bool} \lesssim \mathsf{Int} \to \mathsf{Int} \to ?$.

$$\left.\begin{array}{l} \mathsf{Int} \to \boxed{?} \to \boxed{\mathsf{Bool}}\ |_{\mathsf{Int} \to \mathsf{Int} \to ?} = \mathsf{Int} \to ? \to ? \\ \mathsf{Int} \to \boxed{\mathsf{Int}} \to \boxed{?}\ |_{\mathsf{Int} \to ? \to \mathsf{Bool}} = \mathsf{Int} \to ? \to ? \end{array}\right\} <:$$

$\boxed{\Psi \vdash^G \sigma_1 \lesssim \sigma_2}$ *(Consistent Subtyping)*

$$\frac{a \in \Psi}{\Psi \vdash^G a \lesssim a}\ \text{GPC-CS-TVAR} \qquad \frac{}{\Psi \vdash^G \mathsf{Int} \lesssim \mathsf{Int}}\ \text{GPC-CS-INT} \qquad \frac{\Psi \vdash^G \sigma_3 \lesssim \sigma_1 \qquad \Psi \vdash^G \sigma_2 \lesssim \sigma_4}{\Psi \vdash^G \sigma_1 \to \sigma_2 \lesssim \sigma_3 \to \sigma_4}\ \text{GPC-CS-ARROW} \qquad \frac{\Psi, a \vdash^G \sigma_1 \lesssim \sigma_2}{\Psi \vdash^G \sigma_1 \lesssim \forall a.\, \sigma_2}\ \text{GPC-CS-FORALLR}$$

$$\frac{\Psi \vdash^G \tau \qquad \Psi \vdash^G \sigma_1[a \mapsto \tau] \lesssim \sigma_2}{\Psi \vdash^G \forall a.\, \sigma_1 \lesssim \sigma_2}\ \text{GPC-CS-FORALLL} \qquad \frac{}{\Psi \vdash^G ? \lesssim \sigma}\ \text{GPC-CS-UNKNOWNL} \qquad \frac{}{\Psi \vdash^G \sigma \lesssim ?}\ \text{GPC-CS-UNKNOWNR}$$

Figure 4.6: Consistent Subtyping for implicit polymorphism.

Here differences of the types in boxes are erased because of the restriction operator. Now if we compare the types in boxes directly instead of through the lens of the restriction operator, we can observe that the *consistent subtyping relation always holds between the unknown type and an arbitrary type.* We can interpret this observation directly from Definition 5: the unknown type is neutral to subtyping ($? <: ?$), the unknown type is consistent with any type ($? \sim \sigma$), and subtyping is reflexive ($\sigma <: \sigma$). Therefore, *the unknown type is a consistent subtype of any type ($? \lesssim \sigma$), and vice versa ($\sigma \lesssim ?$).* Note that this interpretation provides a general recipe for lifting a (static) subtyping relation to a (gradual) consistent subtyping relation, as discussed below.

Defining Consistent Subtyping Directly. From the above discussion, we can define the consistent subtyping relation directly, *without* resorting to subtyping or consistency at all. The key idea is that we replace $<:$ with $\lesssim$ in Figure 4.2, get rid of rule GPC-S-UNKNOWN and add two extra rules concerning $?$, resulting in the rules of consistent subtyping in Figure 4.6. Of particular interest are the rules GPC-CS-UNKNOWNL and GPC-CS-UNKNOWNR, both of which correspond to what we just said: the unknown type is a consistent subtype of any type, and vice versa.

From now on, we use the symbol $\lesssim$ to refer to the consistent subtyping relation in Figure 4.6. What is more, we can prove that the two definitions are equivalent.

Theorem 4.4. $\Psi \vdash^G \sigma_1 \lesssim \sigma_2 \Leftrightarrow \Psi \vdash^G \sigma_1 <: \sigma', \sigma' \sim \sigma'', \Psi \vdash^G \sigma'' <: \sigma_2$ *for some* σ', σ''.

4.3 Gradually Typed Implicit Polymorphism

In Section 4.2 we introduced our consistent subtyping relation that accommodates polymorphic types. In this section we continue with the development by giving a declarative type system for predicative implicit polymorphism, GPC, that employs the consistent subtyping relation. The declarative system itself is already quite interesting as it is equipped with both higher-rank polymorphism and the unknown type.

The syntax of expressions in the declarative system is given at the top of Figure 4.7. The definition of expressions are the same as of OL in Figure 2.3. Meta-variable e ranges over expressions. Expressions include variables x, integers n, annotated lambda abstractions $\lambda x : \sigma.\, e$, un-annotated lambda abstractions $\lambda x.\, e$, applications $e_1\, e_2$, and let expressions $\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$.

4.3.1 Typing in Detail

Figure 4.7 gives the typing rules for our declarative system (the reader is advised to ignore the gray-shaded parts for now). Rule GPC-VAR extracts the type of the variable from the typing context. Rule GPC-INT always infers integer types. Rule GPC-LAMANN puts x with type annotation σ into the context, and continues type checking the body e. Rule GPC-LAM assigns a monotype τ to x, and continues type checking the body e. Gradual types and polymorphic types are introduced via explicit annotations. Rule GPC-GEN puts a fresh type variable a into the type context and generalizes the typing result σ to $\forall a.\, \sigma$. Rule GPC-LET infers the type σ of e_1, then puts $x : \sigma$ in the context to infer the type of e_2. Rule GPC-APP first infers the type of e_1, then the matching judgment $\Psi \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2$ extracts the domain type σ_1 and the codomain type σ_2 from type σ. The type σ_3 of the argument e_2 is then compared with σ_1 using the consistent subtyping judgment.

Matching. The matching judgment of Siek et al. [2015] is extended to polymorphic types naturally, resulting in $\Psi \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2$. Note that the matching rules generalize that of DK in Figure 2.7 with the unknown type. In rule GPC-M-FORALL, a monotype τ is guessed to instantiate the universal quantifier a. If σ is a polymorphic type, the judgment works by guessing instantiations until it reaches an arrow type. Rule GPC-M-ARR returns the domain type σ_1 and range type σ_2 as expected. If the input is *?*, then rule GPC-M-UNKNOWN returns *?* as both the type for the domain and the range.

Note that in GPC, matching saves us from having a subsumption rule (rule OL-SUB in Fig. 2.5). The subsumption rule is incompatible with consistent subtyping, since the latter is not transitive. A discussion of a subsumption rule based on normal subtyping can be found in Section 4.5.2.

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

$\boxed{\Psi \vdash^G e : \sigma \rightsquigarrow s}$ *(Typing)*

$$\frac{(x : \sigma) \in \Psi}{\Psi \vdash^G x : \sigma \rightsquigarrow x}\ \textsc{gpc-var} \qquad \frac{}{\Psi \vdash^G n : \mathsf{Int} \rightsquigarrow n}\ \textsc{gpc-int} \qquad \frac{\Psi, a \vdash^G e : \sigma \rightsquigarrow s}{\Psi \vdash^G e : \forall a .\, \sigma \rightsquigarrow \Lambda a .\, s}\ \textsc{gpc-gen}$$

$$\frac{\Psi, x : \sigma \vdash^G e : \sigma_2 \rightsquigarrow s}{\Psi \vdash^G \lambda x : \sigma .\, e : \sigma \to \sigma_2 \rightsquigarrow \lambda x : \sigma .\, s}\ \textsc{gpc-lamann} \qquad \frac{\Psi, x : \tau \vdash^G e : \sigma_2 \rightsquigarrow s}{\Psi \vdash^G \lambda x .\, e : \tau \to \sigma_2 \rightsquigarrow \lambda x : \tau .\, s}\ \textsc{gpc-lam}$$

$$\frac{\Psi \vdash^G e_1 : \sigma \rightsquigarrow s_1 \qquad \Psi, x : \sigma \vdash^G e_2 : \sigma_2 \rightsquigarrow s_2}{\Psi \vdash^G \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \sigma_2 \rightsquigarrow (\lambda x : \sigma .\, s_2)\, s_1}\ \textsc{gpc-let}$$

$$\frac{\Psi \vdash^G e_1 : \sigma \rightsquigarrow s_1 \qquad \Psi \vdash^G \sigma \rhd \sigma_1 \to \sigma_2 \qquad \Psi \vdash^G e_2 : \sigma_3 \rightsquigarrow s_2 \qquad \Psi \vdash^G \sigma_3 \lesssim \sigma_1}{\Psi \vdash^G e_1\, e_2 : \sigma_2 \rightsquigarrow (\langle \sigma \hookrightarrow \sigma_1 \to \sigma_2 \rangle s_1)\, (\langle \sigma_3 \hookrightarrow \sigma_1 \rangle s_2)}\ \textsc{gpc-app}$$

$\boxed{\Psi \vdash^G \sigma \rhd \sigma_1 \to \sigma_2}$ *(Matching)*

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

$$\frac{}{\Psi \vdash^G ? \rhd ? \to ?}\ \textsc{gpc-m-unknown}$$

Figure 4.7: Syntax of expressions and declarative typing of declarative GPC

4.3.2 Type-directed Translation

We give the dynamic semantics of our language by translating it to λB [Ahmed et al. 2009]. Below we show a subset of the terms in λB that are used in the translation:

$$\lambda\text{B Terms} \quad s \quad ::= \quad x \mid n \mid \lambda x : \sigma.\, s \mid \Lambda a.\, s \mid s_1\, s_2 \mid \langle \sigma \hookrightarrow \sigma_2 \rangle s$$

A cast $\langle \sigma_1 \hookrightarrow \sigma_2 \rangle s$ converts the value of term s from type σ_1 to type σ_2. A cast from σ_1 to σ_2 is permitted only if the types are *compatible*, written $\sigma_1 \hookrightarrow \sigma_2$, as briefly mentioned in Section 4.2.1. The syntax of types in λB is the same as ours.

The translation is given in the gray-shaded parts in Figure 4.7. The only interesting case here is to insert explicit casts in the application rule. Note that there is no need to translate matching or consistent subtyping. Instead we insert the source and target types of a cast directly in the translated expressions, thanks to the following two lemmas:

Lemma 4.5 ($\triangleright$ to $\hookrightarrow$). *If* $\Psi \vdash^G \sigma \triangleright \sigma_1 \to \sigma_2$, *then* $\sigma \hookrightarrow \sigma_1 \to \sigma_2$.

Lemma 4.6 ($\lesssim$ to $\hookrightarrow$). *If* $\Psi \vdash^G \sigma \lesssim \sigma_2$, *then* $\sigma \hookrightarrow \sigma_2$.

In order to show the correctness of the translation, we prove that our translation always produces well-typed expressions in λB. By Lemmas 4.5 and 4.6, we have the following theorem:

Theorem 4.7 (Type Safety). *If* $\Psi \vdash^G e : \sigma \rightsquigarrow s$, *then* $\Psi \vdash^B s : \sigma$.

Parametricity. An important semantic property of polymorphic types is *relational parametricity* [Reynolds 1983]. The parametricity property says that all instances of a polymorphic function should behave *uniformly*. A classic example is a function with the type $\forall a.\, a \to a$. The parametricity property guarantees that a value of this type must be either the identity function (i.e., $\lambda x.\, x$) or the undefined function (one which never returns a value). However, with the addition of the unknown type ?, careful measures are to be taken to ensure parametricity. Our translation target λB is taken from Ahmed et al. [2009], where relational parametricity is enforced by dynamic sealing [Matthews and Ahmed 2008; Neis et al. 2009], but there is no rigorous proof. Later, Ahmed et al. [2009] imposed a syntactic restriction on terms of λB, where all type abstractions must have *values* as their body. With this invariant, they proved that the restricted λB satisfies relational parametricity. It remains to see if our translation process can be adjusted to target restricted λB. One possibility is to impose similar restriction to the rule gpc-gen:

$$\frac{\Psi, a \vdash^G e : \sigma \rightsquigarrow v}{\Psi \vdash^G e : \forall a.\, \sigma \rightsquigarrow \Lambda a.\, v}\ \text{GPC-GEN2}$$

where we only generate type abstractions if the inner body is a value. However, the type system with this rule is a weaker calculus, which is not a conservative extension of the OL type system.

Ambiguity from Casts. The translation does not always produce a unique target expression. This is because when guessing some monotype τ in rule GPC-M-FORALL and rule GPC-CS-FORALLL, we could have many choices, which inevitably leads to different types. This is usually not a problem for (non-gradual) System F-like systems [Dunfield and Krishnaswami 2013; Peyton Jones et al. 2007] because they adopt a type-erasure semantics [Pierce 2002]. However, in our case, the choice of monotypes may affect the runtime behaviour of translated programs, since they could appear inside the explicit casts. For instance, the following example shows two possible translations for the same source expression $(\lambda x : ?. f\,x) : ? \to \mathsf{Int}$, where the type of f is instantiated to $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Int}$, respectively:

$$
\begin{aligned}
f : \forall a.\, a \to \mathsf{Int} \vdash^G (\lambda x : ?. f\,x) &: ? \to \mathsf{Int} \\
&\rightsquigarrow (\lambda x : ?. (\langle \forall a.\, a \to \mathsf{Int} \hookrightarrow \mathsf{Int} \to \mathsf{Int}\rangle f)\ (\boxed{\langle ? \hookrightarrow \mathsf{Int}\rangle}\ x)) \\
f : \forall a.\, a \to \mathsf{Int} \vdash^G (\lambda x : ?. f\,x) &: ? \to \mathsf{Int} \\
&\rightsquigarrow (\lambda x : ?. (\langle \forall a.\, a \to \mathsf{Int} \hookrightarrow \mathsf{Bool} \to \mathsf{Int}\rangle f))\ (\boxed{\langle ? \hookrightarrow \mathsf{Bool}\rangle}\ x))
\end{aligned}
$$

If we apply $\lambda x : ?. f\,x$ to 3, which is fine since the function can take any input, the first translation runs smoothly in λB, while the second one will raise a cast error (Int cannot be cast to Bool). Similarly, if we apply it to true, then the second succeeds while the first fails. The culprit lies in the highlighted parts where the instantiation of a appears in the explicit cast. More generally, any choice introduces an explicit cast to that type in the translation, which causes a runtime cast error if the function is applied to a value whose type does not match the guessed type. Note that this does not compromise the type safety of the translated expressions, since cast errors are part of the type safety guarantees.

The semantic discrepancy is due to the guessing nature of the *declarative* system. As far as the static semantics is concerned, both $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Int}$ are equally acceptable. But this is not the case at runtime. The astute reader may have found that the *only* appropriate choice is to instantiate the type of f to $? \to \mathsf{Int}$ in the matching judgment. However, as specified by rule GPC-M-FORALL in Figure 4.7, we can only instantiate type variables to monotypes, but $?$ is *not* a monotype! We will get back to this issue in Chapter 5.

Coherence. The ambiguity of translation seems to imply that the declarative system is *incoherent*. A semantics is coherent if distinct typing derivations of the same typing judgment possess the same meaning [Reynolds 1991]. We argue that the declarative system is *coherent up to cast errors* in the sense that a well-typed program produces a unique value, or results in a cast error. In the above example, suppose f is defined as $(\lambda x.\, 1)$, then whatever the translation might be, applying $(\lambda x : ?.\, f\, x)$ to 3 either results in a cast error, or produces 1, nothing else.

We defined contextual equivalence [Morris Jr 1969] to formally characterize that two open expressions have the same behavior. The definition of contextual equivalence requires a notion of well-typed expression contexts σ_3, written $\mathcal{C} : (\Psi \vdash^B \sigma) \rightsquigarrow (\Psi' \vdash^B \sigma')$. The definitions of contexts and context typing are standard and thus omitted. As is common, we first define contextual approximation. In our setting, we need to relax the notion of contextual approximation of λB [Ahmed et al. 2009] to also take into consideration of cast errors. We write $\Psi \vdash s_1 \preceq_{ctx} s_2 : \sigma$ to say that s_2 mimics the behaviour of s_1 at type σ in the sense that whenever a program containing s_1 reduces to an integer, replacing it with s_2 either reduces to the same integer, or emits a cast error. We restrict the program results to integers to eliminate the role of types in values. If it is not an integer, it is always possible to embed it into another context that reduces to an integer. Then we write $\Psi \vdash s_1 \simeq_{ctx} s_2 : \sigma$ to say s_1 and s_2 are contextually equivalent, that is, they approximate each other.

Definition 8 (Contextual Approximation and Equivalence up to Cast Errors).

$$\begin{array}{lcl}
\Psi \vdash s_1 \preceq_{ctx} s_2 : \sigma & \triangleq & \Psi \vdash^B s_1 : \sigma \land \Psi \vdash^B s_2 : \sigma \land \\
& & \text{for all } \mathcal{C}.\, \mathcal{C} : (\Psi \vdash^B \sigma) \rightsquigarrow (\bullet \vdash^B \mathsf{Int}) \Longrightarrow \\
& & \quad \mathcal{C}\{s_1\} \Downarrow n \Longrightarrow (\mathcal{C}\{s_2\} \Downarrow n \lor \mathcal{C}\{s_2\} \Downarrow \mathsf{blame}) \\
\Psi \vdash s_1 \simeq_{ctx} s_2 : \sigma & \triangleq & \Psi \vdash s_1 \preceq_{ctx} s_2 : \sigma \land \Psi \vdash s_2 \preceq_{ctx} s_1 : \sigma
\end{array}$$

Before presenting the formal definition of coherence, first we observe that after erasing types and casts, all translations of the same expression are exactly the same. This is easy to see by examining each elaboration rule. We use $\lfloor s \rfloor$ to denote an expression in λB after erasure.

Lemma 4.8. *If* $\Psi \vdash^G e : \sigma \rightsquigarrow s_1$, *and* $\Psi \vdash^G e : \sigma \rightsquigarrow s_2$, *then* $\lfloor s_1 \rfloor \equiv_\alpha \lfloor s_2 \rfloor$.

Second, at runtime, the only role of types and casts is to emit cast errors caused by type mismatch. Therefore, By Lemma 4.8 coherence follows as a corollary:

Lemma 4.9 (Coherence up to cast errors). *For any expression e such that* $\Psi \vdash^G e : \sigma \rightsquigarrow s_1$ *and* $\Psi \vdash^G e : \sigma \rightsquigarrow s_2$, *we have* $\Psi \vdash s_1 \simeq_{ctx} s_2 : \sigma$.

4.3.3 Correctness Criteria

Siek et al. [2015] present a set of properties, the *refined criteria*, that a well-designed gradual typing calculus must have. Among all the criteria, those related to the static aspects of gradual typing are well summarized by Cimini and Siek [2016]. Here we review those criteria and adapt them to our notation. We have proved in Coq that our type system satisfies all these criteria.

Lemma 4.10 (Correctness Criteria)**.**

- ***Conservative extension:*** *for all static* Ψ*, e, and* σ*,*
  - *if* $\Psi \vdash^{OL} e : \sigma$*, then there exists* σ_2*, such that* $\Psi \vdash^{G} e : \sigma_2$*, and* $\Psi \vdash^{G} \sigma_2 <: \sigma$*.*
  - *if* $\Psi \vdash^{G} e : \sigma$*, then* $\Psi \vdash^{OL} e : \sigma$
- ***Monotonicity w.r.t. precision:*** *for all* Ψ, e, e', σ*, if* $\Psi \vdash^{G} e : \sigma$*, and* $e' \sqsubseteq e$*, then* $\Psi \vdash^{G} e' : \sigma_2$*, and* $\sigma_2 \sqsubseteq \sigma$ *for some B.*
- ***Type Preservation of cast insertion:*** *for all* Ψ, e, σ*, if* $\Psi \vdash^{G} e : \sigma$*, then* $\Psi \vdash^{G} e : \sigma \rightsquigarrow s$*, and* $\Psi \vdash^{B} s : \sigma$ *for some* s*.*
- ***Monotonicity of cast insertion:*** *for all* $\Psi, e_1, e_2, s_1, s_2, \sigma$*, if* $\Psi \vdash^{G} e_1 : \sigma \rightsquigarrow s_1$*, and* $\Psi \vdash^{G} e_2 : \sigma \rightsquigarrow s_2$*, and* $e_1 \sqsubseteq e_2$*, then* $\Psi \mid \Psi \vdash s_1 \sqsubseteq^{B} s_2$*.*

The first criterion states that the gradual type system should be a conservative extension of the original system. In other words, a *static* program is typeable in the OL type system if and only if it is typeable in the gradual type system. A static program is one that does not contain any type *?*[7]. However since our gradual type system does not have the subsumption rule, it produces more general types.

The second criterion states that if a typeable expression loses some type information, it remains typeable. This criterion depends on the definition of the precision relation, written $\sigma_1 \sqsubseteq \sigma_2$, which is given in Figure 4.8. The relation intuitively captures a notion of types containing more or less unknown types (*?*). The precision relation over types lifts to programs, i.e., $e_1 \sqsubseteq e_2$ means that e_1 and e_2 are the same program except that e_1 has more unknown types.

The first two criteria are fundamental to gradual typing. They explain for example why these two programs $\lambda x : \mathsf{Nat}.\, x + 1$ and $\lambda x : ?.\, x + 1$ are typeable, as the former is typeable in the OL type system and the latter is a less-precise version of it.

[7] Note that the term *static* has appeared several times with different meanings.

$\boxed{\sigma_1 \sqsubseteq \sigma_2}$ *(Type Precision)*

$$\frac{}{? \sqsubseteq \sigma}\ \textsc{gpc-l-unknown} \qquad \frac{}{\mathsf{Int} \sqsubseteq \mathsf{Int}}\ \textsc{gpc-l-int} \qquad \frac{\sigma_1 \sqsubseteq \sigma_3 \qquad \sigma_2 \sqsubseteq \sigma_4}{\sigma_1 \to \sigma_2 \sqsubseteq \sigma_3 \to \sigma_4}\ \textsc{gpc-l-arrow} \qquad \frac{}{a \sqsubseteq a}\ \textsc{gpc-l-tvar}$$

$$\frac{\sigma_1 \sqsubseteq \sigma_2}{\forall a.\, \sigma_1 \sqsubseteq \forall a.\, \sigma_2}\ \textsc{gpc-l-forall}$$

$\boxed{e_1 \sqsubseteq e_2}$ *(Term Precision)*

$$\frac{}{e \sqsubseteq e}\ \textsc{gpc-le-refl} \qquad \frac{\sigma_1 \sqsubseteq \sigma_2 \qquad e_1 \sqsubseteq e_2}{\lambda x : \sigma_1.\, e_1 \sqsubseteq \lambda x : \sigma_2.\, e_2}\ \textsc{gpc-le-lamann} \qquad \frac{e_1 \sqsubseteq e_3 \qquad e_2 \sqsubseteq e_4}{e_1\, e_2 \sqsubseteq e_3\, e_4}\ \textsc{gpc-le-app}$$

$\boxed{s_1 \sqsubseteq s_2}$ *(Term Precision in* $\lambda\mathsf{B}$*)*

$$\frac{}{x \sqsubseteq x}\ \textsc{b-le-var} \qquad \frac{}{n \sqsubseteq n}\ \textsc{b-le-nat} \qquad \frac{\sigma_1 \sqsubseteq \sigma_2 \qquad s_1 \sqsubseteq s_2}{\lambda x : \sigma_1.\, s_1 \sqsubseteq \lambda x : \sigma_2.\, s_2}\ \textsc{b-le-lamann} \qquad \frac{s_1 \sqsubseteq s_2}{\Lambda a.\, s_1 \sqsubseteq \Lambda a.\, s_2}\ \textsc{b-le-tabs}$$

$$\frac{s_1 \sqsubseteq s_3 \qquad s_2 \sqsubseteq s_4}{s_1\, s_2 \sqsubseteq s_3\, s_4}\ \textsc{b-le-app} \qquad \frac{\sigma_1 \sqsubseteq \sigma_3 \qquad \sigma_2 \sqsubseteq \sigma_4 \qquad s_1 \sqsubseteq s_2}{\langle \sigma_1 \hookrightarrow \sigma_2 \rangle s_1 \sqsubseteq \langle \sigma_3 \hookrightarrow \sigma_4 \rangle s_2}\ \textsc{b-le-cast}$$

Figure 4.8: Less Precision

The last two criteria relate the compilation to the cast calculus. The third criterion is essentially the same as Theorem 4.7, given that a target expression should always exist, which can be easily seen from Fig. 4.7. The last criterion ensures that the translation must be monotonic over the precision relation $\sqsubseteq$. Ahmed et al. [2009] does not include a formal definition of precision, but an *approximation* definition and a *simulation relation*. Here we adapt the simulation relation as the precision, and a subset of it that is used in our system is given at the bottom of Figure 4.8.

The Dynamic Gradual Guarantee. Besides the static criteria, there is also a criterion concerning the dynamic semantics, known as *the dynamic gradual guarantee* [?].

Definition 9 (Dynamic Gradual Guarantee). Suppose $e' \sqsubseteq e$, and $\bullet \vdash^G e : \sigma \rightsquigarrow s$ and $\bullet \vdash^G e' : \sigma' \rightsquigarrow s'$,

- if $s \Downarrow v$, then $s' \Downarrow v'$ and $v' \sqsubseteq v$. If $s \Uparrow$ then $s' \Uparrow$.
- if $s' \Downarrow v'$, then $s \Downarrow v$ where $v' \sqsubseteq v$, or $s \Downarrow \mathsf{blame}$. If $s' \Uparrow$ then $s \Uparrow$ or $s \Downarrow \mathsf{blame}$.

The first part of the dynamic gradual guarantee says that if a gradually typed program evaluates to a value, then making type annotations less precise always produces a program that evaluates to an less precise value. Unfortunately, coherence up to cast errors in the declarative system breaks the dynamic gradual guarantee. For instance:

$$(\lambda f : \forall a.\, a \to \mathsf{Int}.\, \lambda x : \mathsf{Int}.\, f\, x)\, (\lambda x.\, 1)\, 3 \qquad (\lambda f : \forall a.\, a \to \mathsf{Int}.\, \lambda x : ?.\, f\, x)\, (\lambda x.\, 1)\, 3$$

The left one evaluates to 1, whereas its less precise version (right) will give a cast error if a is instantiated to Bool for example. In Chapter 5, we will present an extension of the declarative system that will alleviate the issue.

4.4 Algorithmic Type System

In this section we give a bidirectional account of the algorithmic type system that implements the declarative specification. The algorithm is largely inspired by the algorithmic bidirectional system of DK Dunfield and Krishnaswami [2013]. However our algorithmic system differs from theirs in three aspects: (1) the addition of the unknown type $?$; (2) the use of the matching judgment; and 3) the approach of *gradual inference only producing static types* [?]. We then prove that our algorithm is both sound and complete with respect to the declarative type system. We also provide an implementation.

Expressions	e	$::=$	$x \mid n \mid \lambda x : \sigma.\, e \mid \lambda x.\, e \mid e_1\, e_2 \mid e : \sigma \mid \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$
Types	σ	$::=$	$\mathsf{Int} \mid a \mid \widehat{\alpha} \mid \sigma_1 \to \sigma_2 \mid \forall a.\, \sigma \mid ?$
Monotypes	τ, σ	$::=$	$\mathsf{Int} \mid a \mid \widehat{\alpha} \mid \tau \to \sigma$
Algorithmic Contexts	Γ, Δ, Θ	$::=$	$\bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \widehat{\alpha} \mid \Gamma, \widehat{\alpha} = \tau \mid \Gamma, \blacktriangleright_{\widehat{a}}$
Complete Contexts	Ω	$::=$	$\bullet \mid \Omega, x : \sigma \mid \Omega, a \mid \Omega, \widehat{\alpha} = \tau \mid \Omega, \blacktriangleright_{\widehat{a}}$

$\boxed{\Gamma \vdash^G \sigma}$ *(Well-formedness of types)*

$$\frac{}{\Gamma \vdash^G \mathsf{Int}}\ \textsc{gpc-ad-int} \qquad \frac{}{\Gamma \vdash^G ?}\ \textsc{gpc-ad-unknown} \qquad \frac{}{\Gamma[a] \vdash^G a}\ \textsc{gpc-ad-tvar} \qquad \frac{}{\Gamma[\widehat{\alpha}] \vdash^G \widehat{\alpha}}\ \textsc{gpc-ad-evar}$$

$$\frac{}{\Gamma[\widehat{\alpha} = \tau] \vdash^G \widehat{\alpha}}\ \textsc{gpc-ad-solved} \qquad \frac{\Gamma \vdash^G \sigma_1 \qquad \Gamma \vdash^G \sigma_2}{\Gamma \vdash^G \sigma_1 \to \sigma_2}\ \textsc{gpc-ad-arrow} \qquad \frac{\Gamma, a \vdash^G \sigma}{\Gamma \vdash^G \forall a.\, \sigma}\ \textsc{gpc-ad-forall}$$

$\boxed{\vdash^G \Gamma}$ *(Well-formedness of algorithmic contexts)*

$$\frac{}{\vdash^G \bullet}\ \textsc{gpc-wf-empty} \qquad \frac{\vdash^G \Gamma \qquad x \notin \textsc{fv}\,(\Gamma) \qquad \Gamma \vdash^G \sigma}{\vdash^G \Gamma, x : \sigma}\ \textsc{gpc-wf-var} \qquad \frac{\vdash^G \Gamma \qquad a \notin \textsc{fv}\,(\Gamma)}{\vdash^G \Gamma, a}\ \textsc{gpc-wf-tvar}$$

$$\frac{\vdash^G \Gamma \qquad \widehat{\alpha} \notin \textsc{fv}(\Gamma)}{\vdash^G \Gamma, \widehat{\alpha}}\ \textsc{gpc-wf-evar} \qquad \frac{\vdash^G \Gamma \qquad \widehat{\alpha} \notin \textsc{fv}(\Gamma) \qquad \Gamma \vdash^G \tau}{\vdash^G \Gamma, \widehat{\alpha} = \tau}\ \textsc{gpc-wf-solved} \qquad \frac{\vdash^G \Gamma \qquad \blacktriangleright_{\widehat{a}} \notin \textsc{fv}(\Gamma)}{\vdash^G \Gamma, \blacktriangleright_{\widehat{a}}}\ \textsc{gpc-wf-marker}$$

Figure 4.9: Syntax and well-formedness of the algorithmic GPC

Algorithmic Contexts. The top of Figure 4.9 shows the syntax of the algorithmic system. A noticeable difference are the algorithmic contexts Γ, which are represented as an *ordered* list containing declarations of type variables a and term variables $x : \sigma$. Unlike declarative contexts, algorithmic contexts also contain declarations of existential type variables $\widehat{\alpha}$, which can be either unsolved (written $\widehat{\alpha}$) or solved to some monotype (written $\widehat{\alpha} = \tau$). Finally, algorithmic contexts include a *marker* $\blacktriangleright_{\widehat{a}}$ (read "marker $\widehat{\alpha}$"), which is used to delineate existential variables created by the algorithm. We will have more to say about markers when we examine the rules. Complete contexts Ω are the same as contexts, except that they contain no unsolved variables.

Apart from expressions in the declarative system, we add annotated expressions $e : \sigma$. The well-formedness judgments for types and contexts are shown in Figure 4.9.

Notational convenience. Following DK system, we use contexts as substitutions on types. We write $[\Gamma]\sigma$ to mean Γ applied as a substitution to type σ. We also use a hole notation, which is useful when manipulating contexts by inserting and replacing declarations in the middle. The hole notation is used extensively in proving soundness and completeness. For example, $\Gamma[\Theta]$ means Γ has the form $\Gamma_L, \Theta, \Gamma_R$; if we have $\Gamma[\widehat{\alpha}] = (\Gamma_L, \widehat{\alpha}, \Gamma_R)$, then $\Gamma[\widehat{\alpha} = \tau] = (\Gamma_L, \widehat{\alpha} = \tau, \Gamma_R)$. Occasionally, we will see a context with two *ordered* holes, e.g., $\Gamma = \Gamma_0[\Theta_1][\Theta_2]$ means Γ has the form $\Gamma_L, \Theta_1, \Gamma_M, \Theta_2, \Gamma_R$.

Input and output contexts. The algorithmic system, compared with the declarative system, includes similar judgment forms, except that we replace the declarative context Ψ with an algorithmic context Γ (the *input context*), and add an *output context* Δ after a backward turnstile, e.g., $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$ is the judgment form for the algorithmic consistent subtyping. All algorithmic rules manipulate input and output contexts in a way that is consistent with the notion of *context extension*, which will be described in Section 4.4.5.

We start with the explanation of the algorithmic consistent subtyping as it involves manipulating existential type variables explicitly (and solving them if possible).

4.4.1 Algorithmic Consistent Subtyping

Figure 4.10 presents the rules of algorithmic consistent subtyping $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$, which says that under input context Γ, σ_1 is a consistent subtype of σ_2, with output context Δ. The first five rules do not manipulate contexts, but illustrate how contexts are propagated.

Rule gpc-as-tvar and rule gpc-as-int do not involve existential variables, so the output contexts remain unchanged. Rule gpc-as-evar says that any unsolved existential variable is a consistent subtype of itself. The output is still the same as the input context as the rule gives no

$\boxed{\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta}$ *(Under input context Γ, σ_1 is a consistent subtype of σ_2, with output context Δ)*

$$\frac{}{\Gamma[a] \vdash^G a \lesssim a \dashv \Gamma[a]}\ \textsc{gpc-as-tvar} \qquad \frac{}{\Gamma \vdash^G \mathsf{Int} \lesssim \mathsf{Int} \dashv \Gamma}\ \textsc{gpc-as-int} \qquad \frac{}{\Gamma[\widehat{\alpha}] \vdash^G \widehat{\alpha} \lesssim \widehat{\alpha} \dashv \Gamma[\widehat{\alpha}]}\ \textsc{gpc-as-evar}$$

$$\frac{}{\Gamma \vdash^G ? \lesssim \sigma \dashv \Gamma}\ \textsc{gpc-as-unknown}\text{L} \qquad \frac{}{\Gamma \vdash^G \sigma \lesssim ? \dashv \Gamma}\ \textsc{gpc-as-unknown}\text{R}$$

$$\frac{\Gamma \vdash^G \sigma_3 \lesssim \sigma_1 \dashv \Theta \qquad \Theta \vdash^G [\Theta]\sigma_2 \lesssim [\Theta]\sigma_4 \dashv \Delta}{\Gamma \vdash^G \sigma_1 \to \sigma_2 \lesssim \sigma_3 \to \sigma_4 \dashv \Delta}\ \textsc{gpc-as-arrow} \qquad \frac{\Gamma, a \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta, a, \Theta}{\Gamma \vdash^G \sigma_1 \lesssim \forall a.\, \sigma_2 \dashv \Delta}\ \textsc{gpc-as-forall}\text{R}$$

$$\frac{\Gamma, \blacktriangleright_{\widehat{a}}, \widehat{\alpha} \vdash^G \sigma_1[a \mapsto \widehat{\alpha}] \lesssim \sigma_2 \dashv \Delta, \blacktriangleright_{\widehat{a}}, \Theta}{\Gamma \vdash^G \forall a.\, \sigma_1 \lesssim \sigma_2 \dashv \Delta}\ \textsc{gpc-as-forall}\text{L} \qquad \frac{\widehat{\alpha} \notin \textsc{fv}(\sigma) \qquad \Gamma[\widehat{\alpha}] \vdash^G \widehat{\alpha} \lessapprox \sigma \dashv \Delta}{\Gamma[\widehat{\alpha}] \vdash^G \widehat{\alpha} \lesssim \sigma \dashv \Delta}\ \textsc{gpc-as-inst}\text{L}$$

$$\frac{\widehat{\alpha} \notin \textsc{fv}(\sigma) \qquad \Gamma[\widehat{\alpha}] \vdash^G \sigma \lessapprox \widehat{\alpha} \dashv \Delta}{\Gamma[\widehat{\alpha}] \vdash^G \sigma \lesssim \widehat{\alpha} \dashv \Delta}\ \textsc{gpc-as-inst}\text{R}$$

Figure 4.10: Algorithmic consistent subtyping

clue as to what is the solution of that existential variable. Rules GPC-AS-UNKNOWNL and AS-UNKNOWNR are the counterparts of rule GPC-CS-UNKNOWNL and rule GPC-CS-UNKNOWNR.

Rule GPC-AS-ARROW is a natural extension of its declarative counterpart. The output context of the first premise is used by the second premise, and the output context of the second premise is the output context of the conclusion. Note that we do not simply check $\sigma_2 \lesssim \sigma_4$, but apply Θ (the input context of the second premise) to both types (e.g., $[\Theta]\sigma_2$). This is to maintain an important invariant: whenever $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$ holds, the types σ_1 and σ_2 are fully applied under input context Γ (they contain no existential variables already solved in Γ). The same invariant applies to every algorithmic judgment.

Rule GPC-AS-FORALLR, similar to the declarative rule GPC-CS-FORALLR, adds a to the input context. Note that the output context of the premise allows additional existential variables to appear after the type variable a, in a trailing context Θ. These existential variables could depend on a; since a goes out of scope in the conclusion, we need to drop them from the concluding output, resulting in Δ. The next rule is essential to eliminating the guessing work. Instead of guessing a monotype τ out of thin air, rule GPC-AS-FORALLL generates a fresh existential variable $\widehat{\alpha}$, and replaces a with $\widehat{\alpha}$ in the body σ. The new existential variable $\widehat{\alpha}$ is then added to the input context, just before the marker $\blacktriangleright_{\widehat{a}}$. The output context $(\Delta, \blacktriangleright_{\widehat{a}}, \Theta)$ allows additional existential variables to appear after $\blacktriangleright_{\widehat{a}}$ in Θ. For the same reasons as in rule GPC-AS-FORALLR, we drop them from the output context. A central idea behind these two rules is that we defer the decision of picking a monotype for a type variable, and hope that it could be solved later when we have more information at hand. As a side note, when both types are universal quantifiers, then either rule GPC-AS-FORALLR or rule GPC-AS-FORALLL could be tried. In practice, one can apply rule GPC-AS-FORALLR eagerly as it is invertible.

The last two rules (rule GPC-AS-INSTL and rule GPC-AS-INSTR) are specific to the algorithm, thus having no counterparts in the declarative version. They both check consistent subtyping with an unsolved existential variable on one side and an arbitrary type on the other side. Apart from checking that the existential variable does not occur in the type σ, both rules do not directly solve the existential variables, but leave the real work to the instantiation judgment.

4.4.2 Instantiation

Two symmetric judgments $\Gamma \vdash^G \widehat{\alpha} \lessapprox \sigma \dashv \Delta$ and $\Gamma \vdash^G \sigma \lessapprox \widehat{\alpha} \dashv \Delta$ defined in Fig. 4.11 instantiate unsolved existential variables. They read "under input context Γ, instantiate $\widehat{\alpha}$ to a consistent subtype (or supertype) of σ, with output context Δ". The judgments are extended naturally from DK system, whose original inspiration comes from Cardelli [1993]. Since these two judgments are mutually defined, we discuss them together.

$\boxed{\Gamma \vdash^G \widehat{\alpha} \lessapprox \sigma \dashv \Delta}$ *(Under input context Γ, instantiate $\widehat{\alpha}$ such that $\widehat{\alpha} \lesssim \sigma$, with output context Δ)*

$$\frac{\Gamma \vdash^G \tau}{\Gamma, \widehat{\alpha}, \Gamma' \vdash^G \widehat{\alpha} \lessapprox \tau \dashv \Gamma, \widehat{\alpha} = \tau, \Gamma'}\ \textsc{gpc-instl-solve}$$

$$\frac{}{\Gamma[\widehat{\alpha}] \vdash^G \widehat{\alpha} \lessapprox ? \dashv \Gamma[\widehat{\alpha}]}\ \textsc{gpc-instl-solveU}$$

$$\frac{}{\Gamma[\widehat{\alpha}][\widehat{\beta}] \vdash^G \widehat{\alpha} \lessapprox \widehat{\beta} \dashv \Gamma[\widehat{\alpha}][\widehat{\beta} = \widehat{\alpha}]}\ \textsc{gpc-instl-reach}$$

$$\frac{\Gamma[\widehat{\alpha}], b \vdash^G \widehat{\alpha} \lessapprox \sigma \dashv \Delta, b, \Theta}{\Gamma[\widehat{\alpha}] \vdash^G \widehat{\alpha} \lessapprox \forall b.\, \sigma \dashv \Delta}\ \textsc{gpc-instl-forallR}$$

$$\frac{\Gamma[\widehat{\alpha}_2, \widehat{\alpha}_1, \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2] \vdash^G \sigma_1 \lessapprox \widehat{\alpha}_1 \dashv \Theta \qquad \Theta \vdash^G \widehat{\alpha}_2 \lessapprox [\Theta]\sigma_2 \dashv \Delta}{\Gamma[\widehat{\alpha}] \vdash^G \widehat{\alpha} \lessapprox \sigma_1 \to \sigma_2 \dashv \Delta}\ \textsc{gpc-instl-arr}$$

$\boxed{\Gamma \vdash^G \sigma \lessapprox \widehat{\alpha} \dashv \Delta}$ *(Under input context Γ, instantiate $\widehat{\alpha}$ such that $\sigma \lesssim \widehat{\alpha}$, with output context Δ)*

$$\frac{\Gamma \vdash^G \tau}{\Gamma, \widehat{\alpha}, \Gamma' \vdash^G \tau \lessapprox \widehat{\alpha} \dashv \Gamma, \widehat{\alpha} = \tau, \Gamma'}\ \textsc{gpc-instr-solve}$$

$$\frac{}{\Gamma[\widehat{\alpha}] \vdash^G ? \lessapprox \widehat{\alpha} \dashv \Gamma[\widehat{\alpha}]}\ \textsc{gpc-instr-solveU}$$

$$\frac{}{\Gamma[\widehat{\alpha}][\widehat{\beta}] \vdash^G \widehat{\beta} \lessapprox \widehat{\alpha} \dashv \Gamma[\widehat{\alpha}][\widehat{\beta} = \widehat{\alpha}]}\ \textsc{gpc-instr-reach}$$

$$\frac{\Gamma[\widehat{\alpha}], \blacktriangleright_{\widehat{b}}, \widehat{\beta} \vdash^G \sigma[b \mapsto \widehat{\beta}] \lessapprox \widehat{\alpha} \dashv \Delta, \blacktriangleright_{\widehat{b}}, \Theta}{\Gamma[\widehat{\alpha}] \vdash^G \forall b.\, \sigma \lessapprox \widehat{\alpha} \dashv \Delta}\ \textsc{gpc-instr-forallL}$$

$$\frac{\Gamma[\widehat{\alpha}_2, \widehat{\alpha}_1, \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2] \vdash^G \widehat{\alpha}_1 \lessapprox \sigma_1 \dashv \Theta \qquad \Theta \vdash^G [\Theta]\sigma_2 \lessapprox \widehat{\alpha}_2 \dashv \Delta}{\Gamma[\widehat{\alpha}] \vdash^G \sigma_1 \to \sigma_2 \lessapprox \widehat{\alpha} \dashv \Delta}\ \textsc{gpc-instr-arr}$$

Figure 4.11: Algorithmic instantiation

Rule GPC-INSTL-SOLVE is the simplest one – when an existential variable meets a monotype – where we simply set the solution of $\widehat{\alpha}$ to the monotype τ in the output context. We also need to check that the monotype τ is well-formed under the prefix context Γ.

Rule GPC-INSTL-SOLVEU is similar to rule GPC-AS-UNKNOWNR in that we put no constraint[8] on $\widehat{\alpha}$ when it meets the unknown type $?$. This design decision reflects the point that type inference only produces static types [Garcia and Cimini 2015].

Rule GPC-INSTL-REACH deals with the situation where two existential variables meet. Recall that $\Gamma[\widehat{\alpha}][\widehat{\beta}]$ denotes a context where some unsolved existential variable $\widehat{\alpha}$ is declared before $\widehat{\beta}$. In this situation, the only logical thing we can do is to set the solution of one existential variable to the other one, depending on which one is declared before. For example, in the output context of rule GPC-INSTL-REACH, we have $\widehat{\beta} = \widehat{\alpha}$ because in the input context, $\widehat{\alpha}$ is declared before $\widehat{\beta}$.

Rule GPC-INSTL-FORALLR is the instantiation version of rule GPC-AS-FORALLR. Since our system is predicative, $\widehat{\alpha}$ cannot be instantiated to $\forall b.\, \sigma$, but we can decompose $\forall b.\, \sigma$ in the same way as in rule GPC-AS-FORALLR. Rule GPC-INSTR-FORALLL is the instantiation version of rule GPC-AS-FORALLL.

Rule GPC-INSTL-ARR applies when $\widehat{\alpha}$ meets an arrow type. It follows that the solution must also be an arrow type. This is why, in the first premise, we generate two fresh existential variables $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$, and insert them just before $\widehat{\alpha}$ in the input context, so that we can solve $\widehat{\alpha}$ to $\widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2$. Note that the first premise $\sigma_1 \lessapprox \widehat{\alpha}_1$ switches to the other instantiation judgment.

4.4.3 Algorithmic Typing

We now turn to the algorithmic typing rules in Figure 4.12. Because general type inference for System F is undecidable [Wells 1999], our algorithmic system uses bidirectional type checking to accommodate (first-class) polymorphism. Traditionally, two modes are employed in bidirectional systems: the checking mode $\Gamma \vdash^G e \Leftarrow \sigma \dashv \Theta$, which takes a term e and a type σ as input, and ensures that the term e checks against σ; the inference mode $\Gamma \vdash^G e \Rightarrow \sigma \dashv \Theta$, which takes a term e and produces a type σ. We first discuss rules in the inference mode.

Rule GPC-INF-VAR and rule GPC-INF-INT do not generate any new information and simply propagate the input context. Rule GPC-INF-ANNO is standard, switching to the checking mode in the premise.

[8] As we will see in Chapter 5 where we present a more refined system, the “no constraint” statement is not entirely true.

$\boxed{\Gamma \vdash^G e \Rightarrow \sigma \dashv \Delta}$ *(Under input context Γ, e infers output type σ, with output context Δ)*

$$\textsc{gpc-inf-var}\ \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash^G x \Rightarrow \sigma \dashv \Gamma} \qquad \textsc{gpc-inf-int}\ \frac{}{\Gamma \vdash^G n \Rightarrow \mathsf{Int} \dashv \Gamma} \qquad \textsc{gpc-inf-anno}\ \frac{\Gamma \vdash^G \sigma \qquad \Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta}{\Gamma \vdash^G e : \sigma \Rightarrow \sigma \dashv \Delta}$$

$$\textsc{gpc-inf-lamann}\ \frac{\Gamma \vdash^G \sigma \qquad \Gamma, \widehat{\beta}, x : \sigma \vdash^G e \Leftarrow \widehat{\beta} \dashv \Delta, x : \sigma, \Theta}{\Gamma \vdash^G \lambda x : \sigma.\, e \Rightarrow \sigma \rightarrow \widehat{\beta} \dashv \Delta}$$

$$\textsc{gpc-inf-lam}\ \frac{\Gamma, \widehat{\alpha}, \widehat{\beta}, x : \widehat{\alpha} \vdash^G e \Leftarrow \widehat{\beta} \dashv \Delta, x : \widehat{\alpha}, \Theta}{\Gamma \vdash^G \lambda x.\, e \Rightarrow \widehat{\alpha} \rightarrow \widehat{\beta} \dashv \Delta}$$

$$\textsc{gpc-inf-let}\ \frac{\Gamma \vdash^G e_1 \Rightarrow \sigma \dashv \Theta_1 \qquad \Theta_1, \widehat{\alpha}, x : \sigma \vdash^G e_2 \Leftarrow \widehat{\alpha} \dashv \Delta, x : \sigma, \Theta_2}{\Gamma \vdash^G \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \Rightarrow \widehat{\alpha} \dashv \Delta}$$

$$\textsc{gpc-inf-app}\ \frac{\Gamma \vdash^G e_1 \Rightarrow \sigma \dashv \Theta_1 \qquad \Theta_1 \vdash^G [\Theta_1]\sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Theta_2 \qquad \Theta_2 \vdash^G e_2 \Leftarrow [\Theta_2]\sigma_1 \dashv \Delta}{\Gamma \vdash^G e_1\, e_2 \Rightarrow \sigma_2 \dashv \Delta}$$

$\boxed{\Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta}$ *(Under input context Γ, e checks against input type σ, with output context Δ)*

$$\textsc{gpc-chk-lam}\ \frac{\Gamma, x : \sigma_1 \vdash^G e \Leftarrow \sigma_2 \dashv \Delta, x : \sigma_1, \Theta}{\Gamma \vdash^G \lambda x.\, e \Leftarrow \sigma_1 \rightarrow \sigma_2 \dashv \Delta} \qquad \textsc{gpc-chk-gen}\ \frac{\Gamma, a \vdash^G e \Leftarrow \sigma \dashv \Delta, a, \Theta}{\Gamma \vdash^G e \Leftarrow \forall a.\, \sigma \dashv \Delta}$$

$$\textsc{gpc-chk-sub}\ \frac{\Gamma \vdash^G e \Rightarrow \sigma_1 \dashv \Theta \qquad \Theta \vdash^G [\Theta]\sigma_1 \lesssim [\Theta]\sigma_2 \dashv \Delta}{\Gamma \vdash^G e \Leftarrow \sigma_2 \dashv \Delta}$$

$\boxed{\Gamma \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta}$ *(Under input context Γ, σ matches output type $\sigma_1 \rightarrow \sigma_2$, with output context Δ)*

$$\textsc{gpc-am-forall}\ \frac{\Gamma, \widehat{\alpha} \vdash^G \sigma[a \mapsto \widehat{\alpha}] \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta}{\Gamma \vdash^G \forall a.\, \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta} \qquad \textsc{gpc-am-arr}\ \frac{}{\Gamma \vdash^G \sigma_1 \rightarrow \sigma_2 \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Gamma}$$

$$\textsc{gpc-am-unknown}\ \frac{}{\Gamma \vdash^G\, ? \triangleright\, ? \rightarrow\, ? \dashv \Gamma} \qquad \textsc{gpc-am-var}\ \frac{}{\Gamma[\widehat{\alpha}] \vdash^G \widehat{\alpha} \triangleright \widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2 \dashv \Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2, \widehat{\alpha} = \widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2]}$$

Figure 4.12: Algorithmic typing

In rule GPC-INF-LAMANN, we generate a fresh existential variable $\widehat{\beta}$ for the function codomain, and check the function body against $\widehat{\beta}$. Note that it is tempting to write $\Gamma, x : \sigma \vdash^G e \Rightarrow \sigma_2 \dashv \Delta, x : \sigma, \Theta$ as the premise (in the hope of better matching its declarative counterpart rule GPC-LAMANN), which has a subtle consequence. Consider the expression $\lambda x : \mathsf{Int}.\, \lambda y.\, y$. Under the new premise, this is untypable because of $\bullet \vdash^G \lambda x : \mathsf{Int}.\, \lambda y.\, y \Rightarrow \mathsf{Int} \rightarrow \widehat{\alpha} \rightarrow \widehat{\alpha} \dashv \bullet$ where $\widehat{\alpha}$ is not found in the output context. This explains why we put $\widehat{\beta}$ *before* $x : \sigma$ so that it remains in the output context Δ. Rule GPC-INF-LAM, which corresponds to rule GPC-LAM, one of the guessing rules, is similar to rule GPC-INF-LAMANN. As with the other algorithmic rules that eliminate guessing, we create new existential variables $\widehat{\alpha}$ (for function domain) and $\widehat{\beta}$ (for function codomain) and check the function body against $\widehat{\beta}$. Rule GPC-INF-LET is similar to rule GPC-INF-LAMANN.

ALGORITHMIC MATCHING. Rule GPC-INF-APP deserves attention. It relies on the algorithmic matching judgment $\Gamma \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta$. The matching judgment algorithmically synthesizes an arrow type from an arbitrary type. Rule GPC-AM-FORALL replaces a with a fresh existential variable $\widehat{\alpha}$, thus eliminating guessing. Rule GPC-AM-ARR and rule GPC-AM-UNKNOWN correspond directly to the declarative rules. Rule GPC-AM-VAR, which has no corresponding declarative version, is similar to rule GPC-INSTL-ARR/GPC-INSTR-ARR: we create $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ and solve $\widehat{\alpha}$ to $\widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2$ in the output context.

Back to the rule GPC-INF-APP. This rule first infers the type of e_1, producing an output context Θ_1. Then it applies Θ_1 to A and goes into the matching judgment, which delivers an arrow type $\sigma_1 \rightarrow \sigma_2$ and another output context Θ_2. Θ_2 is used as the input context when checking e_2 against $[\Theta_2]\sigma_1$, where we go into the checking mode.

Rules in the checking mode are quite standard. Rule GPC-CHK-LAM checks against $\sigma_1 \rightarrow \sigma_2$. Rule GPC-CHK-GEN, like the declarative rule GPC-GEN, adds a type variable a to the input context. Rule GPC-CHK-SUB uses the algorithmic consistent subtyping judgment.

4.4.4 Decidability

Our algorithmic system is decidable. It is not at all obvious to see why this is the case, as many rules are not strictly structural (e.g., many rules have $[\Gamma]\sigma$ in the premises). This implies that we need a more sophisticated measure to support the argument. Since the typing rules (Figure 4.12) depend on the consistent subtyping rules (Figure 4.10), which in turn depends on the instantiation rules (Figure 4.11), to show the decidability of the typing judgment, we need to show that the instantiation and consistent subtyping judgments are decidable. The proof strategy mostly follows that of the DK system. Here only highlights of the proofs are given.

Decidability of Instantiation The basic idea is that we need to show σ in the instantiation judgments $\Gamma \vdash^G \widehat{\alpha} \lessapprox \sigma \dashv \Delta$ and $\Gamma \vdash^G \sigma \lessapprox \widehat{\alpha} \dashv \Delta$ always gets smaller. Most of the rules are structural and thus easy to verify (e.g., rule instl-forallR); the non-trivial cases are rule instl-arr and rule instr-arr where context applications appear in the premises. The key observation there is that the instantiation rules preserve the size of (substituted) types. The formal statement of decidability of instantiation needs a few pre-conditions: assuming $\widehat{\alpha}$ is unsolved in the input context Γ, that σ is well-formed under the context Γ, that σ is fully applied under the input context Γ ($[\Gamma]\sigma = \sigma$), and that $\widehat{\alpha}$ does not occur in σ. Those conditions are actually met when instantiation is invoked: rule chk-sub applies the input context, and the subtyping rules apply input context when needed.

Theorem 4.11 (Decidability of Instantiation). *If $\Gamma = \Gamma_0[\widehat{\alpha}]$ and $\Gamma \vdash^G \sigma$ such that $[\Gamma]\sigma = \sigma$ and $\widehat{\alpha} \notin \textsc{fv}(\sigma)$ then:*

1. *Either there exists Δ such that $\Gamma \vdash^G \widehat{\alpha} \lessapprox \sigma \dashv \Delta$, or not.*

2. *Either there exists Δ such that $\Gamma \vdash^G \sigma \lessapprox \widehat{\alpha} \dashv \Delta$, or not.*

Decidability of Algorithmic Consistent Subtyping Proving decidability of the algorithmic consistent subtyping is a bit more involved, as the induction measure consists of several parts. We measure the judgment $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$ lexicographically by

(M1) the number of $\forall$-quantifiers in σ_1 and σ_2;

(M2) the number of unknown types in σ_1 and σ_2;

(M3) $|\textsc{unsolved}(\Gamma)|$: the number of unsolved existential variables in Γ;

(M4) $|\Gamma \vdash^G \sigma_1| + |\Gamma \vdash^G \sigma_2|$.

Notice that because of our gradual setting, we also need to measure the number of unknown types (M2). This is a key difference from the DK system. We refer the reader to **??** for more details. For (M4), we use *contextual size*—the size of well-formed types under certain contexts, which penalizes solved variables (*).

Definition 10 (Contextual Size).

$$
\begin{array}{lcll}
|\Gamma \vdash^G \mathsf{Int}| & = & 1 & \\
|\Gamma \vdash^G \mathsf{?}| & = & 1 & \\
|\Gamma \vdash^G a| & = & 1 & \\
|\Gamma \vdash^G \widehat{\alpha}| & = & 1 & \\
|\Gamma[\widehat{\alpha} = \tau] \vdash^G \widehat{\alpha}| & = & 1 + |\Gamma[\widehat{\alpha} = \tau] \vdash^G \tau| & (*) \\
|\Gamma \vdash^G \forall a.\, \sigma| & = & 1 + |\Gamma, a \vdash^G \sigma| & \\
|\Gamma \vdash^G \sigma_1 \to \sigma_2| & = & 1 + |\Gamma \vdash^G \sigma_1| + |\Gamma \vdash^G \sigma_2| &
\end{array}
$$

Theorem 4.12 (Decidability of Algorithmic Consistent Subtyping). *Given a context Γ and types σ_1, σ_2 such that $\Gamma \vdash^G \sigma_1$ and $\Gamma \vdash^G \sigma_2$ and $[\Gamma]\sigma_1 = \sigma_1$ and $[\Gamma]\sigma_2 = \sigma_2$, it is decidable whether there exists Δ such that $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$.*

Decidability of Algorithmic Typing Similar to proving decidability of algorithmic consistent subtyping, the key is to come up with a correct measure. Since the typing rules depend on the matching judgment, we first show decidability of the algorithmic matching.

Lemma 4.13 (Decidability of Algorithmic Matching). *Given a context Γ and a type σ it is decidable whether there exist types σ_1, σ_2 and a context Δ such that $\Gamma \vdash^G \sigma \triangleright \sigma_1 \to \sigma_2 \dashv \Delta$.*

Now we are ready to show decidability of typing. The proof is obtained by induction on the lexicographically ordered triple: size of e, typing judgment (where the inference mode $\Rightarrow$ is considered smaller than the checking mode $\Leftarrow$) and contextual size.

$$
\left\langle \quad e, \quad \begin{array}{c} \Rightarrow \\ \Leftarrow, \end{array} \quad |\Gamma \vdash^G \sigma| \quad \right\rangle
$$

The above measure is much simpler than the corresponding one in the DK system, where they also need to consider the application judgment together with the inference and checking judgments. This shows another benefit (besides the independence from typing) of adopting the matching judgment.

Theorem 4.14 (Decidability of Algorithmic Typing).

1. *Inference: Given a context Γ and a term e, it is decidable whether there exist a type σ and a context Δ such that $\Gamma \vdash^G e \Rightarrow \sigma \dashv \Delta$.*

2. *Checking: Given a context Γ, a term e and a type σ such that $\Gamma \vdash^G \sigma$, it is decidable whether there exists a context Δ such that $\Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta$.*

$\boxed{\Gamma \longrightarrow \Delta}$ *(Context extension)*

$$\frac{}{\bullet \longrightarrow \bullet}\ \textsc{gpc-ext-id} \qquad \frac{\Gamma \longrightarrow \Delta \qquad [\Delta]\sigma = [\Delta]\sigma'}{\Gamma, x : \sigma \longrightarrow \Delta, x : \sigma'}\ \textsc{gpc-ext-var} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma, a \longrightarrow \Delta, a}\ \textsc{gpc-ext-tvar} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma, \widehat{\alpha} \longrightarrow \Delta, \widehat{\alpha}}\ \textsc{gpc-ext-evar}$$

$$\frac{\Gamma \longrightarrow \Delta \qquad [\Delta]\tau = [\Delta]\tau'}{\Gamma, \widehat{\alpha} = \tau \longrightarrow \Delta, \widehat{\alpha} = \tau'}\ \textsc{gpc-ext-solved} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma, \widehat{\alpha} \longrightarrow \Delta, \widehat{\alpha} = \tau}\ \textsc{gpc-ext-solve} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \widehat{\alpha}}\ \textsc{gpc-ext-add}$$

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \widehat{\alpha} = \tau}\ \textsc{gpc-ext-addSolve} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma, \blacktriangleright_{\widehat{a}} \longrightarrow \Delta, \blacktriangleright_{\widehat{a}}}\ \textsc{gpc-ext-marker}$$

Figure 4.13: Context extension

4.4.5 Context Extension

To be confident that our algorithmic type system and the declarative type system agree with each other, we need to prove that the algorithmic rules are sound and complete with respect to the declarative specification. Before we give the formal statements of the soundness and completeness theorems, we need a meta-theoretical device, called *context extension* [Dunfield and Krishnaswami 2013], to capture a notion of information increase from input contexts to output contexts.

A context extension judgment $\Gamma \longrightarrow \Delta$ reads "Γ is extended by Δ". Intuitively, this judgment says that Δ has at least as much information as Γ: some unsolved existential variables in Γ may be solved in Δ. The full inductive definition can be found Figure 4.13.

4.4.6 Soundness

Roughly speaking, soundness of the algorithmic system says that given a derivation of an algorithmic judgment with input context Γ, output context Δ, and a complete context Ω that extends Δ, applying Ω throughout the given algorithmic judgment should yield a derivable declarative judgment. For example, let us consider an algorithmic typing judgment $\bullet \vdash^G \lambda x.\, x \Rightarrow \widehat{\alpha} \rightarrow \widehat{\alpha} \dashv \widehat{\alpha}$, and any complete context, say, $\Omega = (\widehat{\alpha} = \mathsf{Int})$, then applying Ω to the above judgment yields $\bullet \vdash^G \lambda x.\, x : \mathsf{Int} \rightarrow \mathsf{Int}$, which is derivable in the declarative system.

However there is one complication: applying Ω to the algorithmic expression does not necessarily yield a typable declarative expression. For example, by rule GPC-CHK-LAM we have $\lambda x.\, x \Leftarrow (\forall a.\, a \rightarrow a) \rightarrow (\forall a.\, a \rightarrow a)$, but $\lambda x.\, x$ itself cannot have type $(\forall a.\, a \rightarrow a) \rightarrow (\forall a.\, a \rightarrow a)$ in the declarative system. To circumvent that, we add an annotation to the lambda abstraction, resulting in $\lambda x : (\forall a.\, a \to a).\, x$, which is typeable in the declarative system with the same type. To relate $\lambda x.\, x$ and $\lambda x : (\forall a.\, a \to a).\, x$, we erase all annotations on both expressions.

Definition 11 (Type annotation erasure)**.** The erasure function is denoted as $|\cdot|$, and defined as follows:

$$
\begin{array}{ll}
|x| = x & |n| = n \\
|\lambda x : \sigma.\, e| = \lambda x.\, |e| & |\lambda x.\, e| = \lambda x.\, |e| \\
|e_1\, e_2| = |e_1|\, |e_2| & |e : \sigma| = |e|
\end{array}
$$

Theorem 4.15 (Instantiation Soundness)**.** *Given $\Delta \longrightarrow \Omega$ and $[\Gamma]\sigma = \sigma$ and $\widehat{\alpha} \notin \textsc{fv}(\sigma)$:*

1. *If $\Gamma \vdash^G \widehat{\alpha} \lessapprox \sigma \dashv \Delta$ then $[\Omega]\Delta \vdash^G [\Omega]\widehat{\alpha} \lesssim [\Omega]\sigma$.*
2. *If $\Gamma \vdash^G \sigma \lessapprox \widehat{\alpha} \dashv \Delta$ then $[\Omega]\Delta \vdash^G [\Omega]\sigma \lesssim [\Omega]\widehat{\alpha}$.*

Notice that the declarative judgment uses $[\Omega]\Delta$, an operation that applies a complete context Ω to the algorithmic context Δ, essentially plugging in all known solutions and removing all declarations of existential variables (both solved and unsolved), resulting in a declarative context.

With instantiation soundness, next we show that the algorithmic consistent subtyping is sound:

Theorem 4.16 (Soundness of Algorithmic Consistent Subtyping)**.** *If $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$ where $[\Gamma]\sigma_1 = \sigma_1$ and $[\Gamma]\sigma_2 = \sigma_2$ and $\Delta \longrightarrow \Omega$ then $[\Omega]\Delta \vdash^G [\Omega]\sigma_1 \lesssim [\Omega]\sigma_2$.*

Finally the soundness theorem of algorithmic typing is:

Theorem 4.17 (Soundness of Algorithmic Typing)**.** *Given $\Delta \longrightarrow \Omega$:*

1. *If $\Gamma \vdash^G e \Rightarrow \sigma \dashv \Delta$ then $\exists e'$ such that $[\Omega]\Delta \vdash^G e' : [\Omega]\sigma$ and $|e| = |e'|$.*
2. *If $\Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta$ then $\exists e'$ such that $[\Omega]\Delta \vdash^G e' : [\Omega]\sigma$ and $|e| = |e'|$.*

4.4.7 Completeness

Completeness of the algorithmic system is the reverse of soundness: given a declarative judgment of the form $[\Omega]\Gamma \vdash^G [\Omega] \ldots$, we want to get an algorithmic derivation of $\Gamma \vdash^G \cdots \dashv \Delta$. It turns out that completeness is a bit trickier to state in that the algorithmic rules generate existential variables on the fly, so Δ could contain unsolved existential variables that are not found in Γ, nor in Ω. Therefore the completeness proof must produce another complete context Ω' that extends both the output context Δ, and the given complete context Ω. As with soundness, we need erasure to relate both expressions.

Theorem 4.18 (Instantiation Completeness). *Given* $\Gamma \longrightarrow \Omega$ *and* $\sigma = [\Gamma]\sigma$ *and* $\widehat{\alpha} \notin$ $\textsc{unsolved}(\Gamma)$ *and* $\widehat{\alpha} \notin \textsc{fv}(\sigma)$*:*

1. *If* $[\Omega]\Gamma \vdash^G [\Omega]\widehat{\alpha} \lesssim [\Omega]\sigma$ *then there are* Δ, Ω' *such that* $\Omega \longrightarrow \Omega'$ *and* $\Delta \longrightarrow \Omega'$ *and* $\Gamma \vdash^G \widehat{\alpha} \lessapprox \sigma \dashv \Delta$.
2. *If* $[\Omega]\Gamma \vdash^G [\Omega]\sigma \lesssim [\Omega]\widehat{\alpha}$ *then there are* Δ, Ω' *such that* $\Omega \longrightarrow \Omega'$ *and* $\Delta \longrightarrow \Omega'$ *and* $\Gamma \vdash^G \sigma \lessapprox \widehat{\alpha} \dashv \Delta$.

Next is the completeness of consistent subtyping:

Theorem 4.19 (Generalized Completeness of Consistent Subtyping). *If* $\Gamma \longrightarrow \Omega$ *and* $\Gamma \vdash^G \sigma_1$ *and* $\Gamma \vdash^G \sigma_2$ *and* $[\Omega]\Gamma \vdash^G [\Omega]\sigma_1 \lesssim [\Omega]\sigma_2$ *then there exist* Δ *and* Ω' *such that* $\Delta \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash^G [\Gamma]\sigma_1 \lesssim [\Gamma]\sigma_2 \dashv \Delta$.

We prove that the algorithmic matching is complete with respect to the declarative matching:

Theorem 4.20 (Matching Completeness). *Given* $\Gamma \longrightarrow \Omega$ *and* $\Gamma \vdash^G \sigma$*, if* $[\Omega]\Gamma \vdash^G [\Omega]\sigma \triangleright \sigma_1 \rightarrow \sigma_2$ *then there exist* Δ, Ω', σ_1' *and* σ_2' *such that* $\Gamma \vdash^G [\Gamma]\sigma \triangleright \sigma_1' \rightarrow \sigma_2' \dashv \Delta$ *and* $\Delta \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$ *and* $\sigma_1 = [\Omega']\sigma_1'$ *and* $\sigma_2 = [\Omega']\sigma_2'$.

Finally here is the completeness theorem of the algorithmic typing:

Theorem 4.21 (Completeness of Algorithmic Typing). *Given* $\Gamma \longrightarrow \Omega$ *and* $\Gamma \vdash^G \sigma$*, if* $[\Omega]\Gamma \vdash^G e : \sigma$ *then there exist* Δ, Ω', σ' *and* e' *such that* $\Delta \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash^G e' \Rightarrow \sigma' \dashv \Delta$ *and* $\sigma = [\Omega']\sigma'$ *and* $|e| = |e'|$.

4.5 Simple Extensions and Variants

This section considers two simple variations on the presented system. The first variation extends the system with a top type, while the second variation considers a more declarative formulation using a subsumption rule.

4.5.1 Top Types

We argued that our definition of consistent subtyping (Definition 5) generalizes the original definition by Siek and Taha [2007b]. We have shown its applicability to polymorphic types, for which Siek and Taha [2007b] approach cannot be extended naturally. To strengthen our argument, we show how to extend our approach to $\top$ types, which is also not supported by Siek and Taha [2007b] approach.

Consistent Subtyping with $\top$. In order to preserve the orthogonality between subtyping and consistency, we require $\top$ to be a common supertype of all static types, as shown in rule GPC-S-TOP. This rule might seem strange at first glance, since even if we remove the requirement σ static, the rule still seems reasonable. However, an important point is that, because of the orthogonality between subtyping and consistency, subtyping itself should not contain a potential information loss! Therefore, subtyping instances such as $? <: \top$ are not allowed. For consistency, we add the rule that $\top$ is consistent with $\top$, which is actually included in the original reflexive rule $\sigma \sim \sigma$. For consistent subtyping, every type is a consistent subtype of $\top$, for example, $\mathsf{Nat} \to ? \lesssim \top$.

$$\frac{\sigma \ \mathsf{static}}{\Psi \vdash^G \sigma <: \top}\ \text{GPC-S-TOP} \qquad \frac{}{\top \sim \top} \qquad \frac{}{\Psi \vdash^G \sigma \lesssim \top}\ \text{GPC-CS-TOP}$$

It is easy to verify that Definition 5 is still equivalent to that in Figure 4.6 extended with rule GPC-CS-TOP. That is, Theorem 4.4 holds:

Proposition 4.22 (Extension with $\top$). *$\Psi \vdash^G \sigma_1 \lesssim \sigma_2 \Leftrightarrow \Psi \vdash^G \sigma_1 <: \sigma'$, $\sigma' \sim \sigma''$, $\Psi \vdash^G \sigma'' <: \sigma_2$ for some σ', σ''.*

We extend the definition of concretization (Definition 6) with $\top$ by adding another equation $\gamma(\top) = \{\top\}$. Note that Castagna and Lanvin [2017] also have this equation in their calculus. It is easy to verify that Corollary 4.2 still holds:

Proposition 4.23 (Equivalent to AGT on $\top$). *$\sigma_1 \lesssim \sigma_2$ if and only if $\sigma_1 \mathrel{\widetilde{<:}} \sigma_2$.*

Siek and Taha's Definition of Consistent Subtyping Does Not Work for $\top$. As with the analysis in Section 4.2.2, $\mathsf{Nat} \to ? \lesssim \top$ only holds when we first apply consistency, then subtyping. However we cannot find a type σ such that $\mathsf{Nat} \to ? <: \sigma$ and $\sigma \sim \top$. The following diagram depicts the situation:

$$\begin{tikzcd}
\emptyset \arrow[r, dash, "\sim"] & \top \\
\mathsf{Nat} \to ? \arrow[u, "<:"] \arrow[r, dash, "\sim"'] & \mathsf{Nat} \to \mathsf{Nat} \arrow[u, "<:"]
\end{tikzcd}$$

Additionally we have a similar problem in extending the restriction operator: *non-structural* masking between $\mathsf{Int} \to ?$ and $\top$ cannot be easily achieved.

Note that both the top and universally quantified types can be seen as special cases of intersection types. Indeed, top is the intersection of the empty set, while a universally quantified type is the intersection of the infinite set of its instantiations [Davies and Pfenning 2000]. Recall from Section 4.2.3 that Castagna and Lanvin [2017] shows that consistent subtyping from AGT works well for intersection types, and our definition coincides with AGT (Corollary 4.2 and Proposition 4.23). We thus believe that our definition is compatible with conventional binary intersection types as well. Yet, a rigorous formalization would be needed to substantiate this belief.

4.5.2 A More Declarative Type System

In Section 4.3 we present our declarative system in terms of the matching and consistent subtyping judgments. The rationale behind this design choice is that the resulting declarative system combines subtyping and type consistency in the application rule, thus making it easier to design an algorithmic system accordingly. Still, one may wonder if it is possible to design a more declarative specification. For example, even though we mentioned that the subsumption rule is incompatible with consistent subtyping, it might be possible to accommodate a subsumption rule for normal subtyping (instead of consistent subtyping). In this section, we discuss an alternative for the design of the declarative system.

Wrong Design A naive design that does not work is to replace rule GPC-APP in Figure 4.7 with the following two rules:

$$\textsc{gpc-v-sub}\quad \frac{\Psi \vdash^G e : \sigma \qquad \sigma <: \sigma_2}{\Psi \vdash^G e : \sigma_2} \qquad\qquad \textsc{gpc-v-app1}\quad \frac{\Psi \vdash^G e_1 : \sigma \qquad \Psi \vdash^G e_2 : \sigma_1 \qquad \sigma \sim \sigma_1 \to \sigma_2}{\Psi \vdash^G e_1\, e_2 : \sigma_2}$$

Rule GPC-V-SUB is the standard subsumption rule: if an expression e has type σ, then it can be assigned some type σ_2 that is a supertype of σ. Rule GPC-V-APP1 first infers that e_1 has type σ, and e_2 has type σ_1, then it finds some σ_2 so that σ is consistent with $\sigma_1 \to \sigma_2$.

There would be two obvious benefits of this variant if it did work: firstly this approach closely resembles the traditional declarative type systems for calculi with subtyping; secondly it saves us from discussing various forms of σ in rule GPC-V-APP1, leaving the job to the consistency judgment.

The design is wrong because of the information loss caused by the choice of σ_2 in rule GPC-V-APP1. Suppose we have $\Psi \vdash^G \mathsf{plus} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$, then we can apply it to 1 to get

$$\dfrac{\Psi \vdash^G 1 : \mathsf{Nat} \qquad \begin{array}{c}\Psi \vdash^G \mathsf{plus} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \sim \mathsf{Nat} \to ? \to \mathsf{Nat}\end{array}}{\Psi \vdash \mathsf{plus}\ 1 : ? \to \mathsf{Nat}} \ \textsc{gpc-v-app1}$$

Further applying it to true we get

$$\dfrac{\Psi \vdash^G \mathsf{plus}\ 1 : ? \to \mathsf{Nat} \qquad \Psi \vdash^G \mathsf{true} : \mathsf{Bool} \qquad ? \to \mathsf{Nat} \sim \mathsf{Bool} \to \mathsf{Nat}}{\Psi \vdash \mathsf{plus}\ 1\ \mathsf{true} : \mathsf{Nat}} \ \textsc{gpc-v-app1}$$

which is obviously wrong! The type consistency in rule GPC-V-APP1 causes information loss for both the argument type σ_1 and the return type σ_2. The problem is that information of σ_2 can get lost again if it appears in further applications. The moral of the story is that we should be very careful when using type consistency. We hypothesize that it is inevitable to do case analysis for the type of the function in an application (i.e., σ in rule GPC-V-APP1).

Proper Declarative Design The proper design refines the first variant by using a matching judgment to carefully distinguish two cases for the typing result of e_1 in rule GPC-V-APP1: (1) when it is an arrow type, and (2) when it is an unknown type. This variant replaces rule GPC-APP in Figure 4.7 with the following rules:

GPC-V-SUB
$$\dfrac{\Psi \vdash^G e : \sigma \qquad \sigma <: \sigma_2}{\Psi \vdash^G e : \sigma_2}$$

GPC-V-APP2
$$\dfrac{\Psi \vdash^G e : \sigma \qquad \Psi \vdash^G \sigma \rhd \sigma_1 \to \sigma_2 \qquad \Psi \vdash^G e_2 : \sigma_3 \qquad \sigma_1 \sim \sigma_3}{\Psi \vdash^G e_1\ e_2 : \sigma_2}$$

$$\dfrac{}{\Psi \vdash^G \sigma_1 \to \sigma_2 \rhd \sigma_1 \to \sigma_2} \qquad \dfrac{}{\Psi \vdash^G ? \rhd ? \to ?}$$

Rule GPC-V-SUB is the same as in the first variant. In rule GPC-V-APP2, we infer that e_1 has type σ, and use the matching judgment to get an arrow type $\sigma_1 \to \sigma_2$. Then we need to ensure that the argument type σ_3 is *consistent with* (rather than a consistent subtype of) σ_1, and use σ_2 as the result type of the application. The matching judgment only deals with two cases, as polymorphic types are handled by rule GPC-V-SUB. These rules are closely related to the ones in Siek and Taha [2006] and Siek and Taha [2007b].

The more declarative nature of this system also implies that it is highly non-syntax-directed, and it does not offer any insight into combining subtyping and consistency. We have proved in Coq the following lemmas to establish soundness and completeness of this system with respect to our original system (to avoid ambiguity, we use the notation $\vdash_m^G$ to indicate the more declarative version):

Lemma 4.24 (Completeness of $\vdash_m^G$)**.** *If $\Psi \vdash^G e : \sigma$, then $\Psi \vdash_m^G e : \sigma$.*

Lemma 4.25 (Soundness of $\vdash_m^G$)**.** *If $\Psi \vdash_m^G e : \sigma$, then there exists some B, such that $\Psi \vdash^G e : \sigma_2$ and $\Psi \vdash^G \sigma_2 <: \sigma$.*

5 Restoring the Dynamic Gradual Guarantee with Type Parameters

In Section 4.3.2 we have seen an example where a single source expression could produce two different target expressions with different runtime behaviors. As we explained, this is due to the guessing nature of the declarative system, and, from the (source) typing point of view, no guessed type is particularly better than any other. As a consequence, this breaks the dynamic gradual guarantee as discussed in Section 4.3.3.

To alleviate this situation, we introduce *static type parameters*, which are placeholders for monotypes, and *gradual type parameters*, which are placeholders for monotypes that are consistent with the unknown type. The concept of static type parameters and gradual type parameters in the context of gradual typing was first introduced by Garcia and Cimini [2015], and later played a central role in the work of Igarashi et al. [2017]. In our type system, type parameters mainly help capture the notion of *representative translations*, and should not appear in a source program. With them we are able to recast the dynamic gradual guarantee in terms of representative translations, and to prove that every well-typed source expression possesses at least one representative translation. With a coherence conjecture regarding representative translations, the dynamic gradual guarantee of our extended source language now can be reduced to that of $\lambda \mathsf{B}$, which, at the time of writing, is still an open question. TODO: not open anymore?

Part IV

Unification and Type-inference for Dependent Types

6 Unification with Promotion

7 Dependent Types

Part V

Related and Future Work

8 Related Work

9 Future Work

Part VI

Epilogue

10 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 pages 4, 59, 61, 71, 73, and 76]

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 15]

Luca Cardelli. 1993. *An implementation of FSub*. Technical Report. Research Report 97, Digital Equipment Corporation Systems Research Center. [cited on page 80]

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 41 (Aug. 2017), 28 pages. [cited on pages 65, 90, and 91]

Gang Chen. 2003. Coercive Subtyping for the Calculus of Constructions *(POPL '03)*. 10. [cited on page 42]

Alonzo Church. 1941. *The calculi of lambda-conversion*. Number 6. Princeton University Press. [cited on page 56]

Matteo Cimini and Jeremy G. Siek. 2016. The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems. In *Proceedings of the 43rd Symposium on Principles of Programming Languages*. [cited on pages 66 and 74]

Thierry Coquand. 1996. An algorithm for type-checking dependent types. *Science of Computer Programming* 26, 1-3 (1996), 167–177. [cited on page 15]

Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. 1958. *Combinatory logic*. Vol. 1. North-Holland Amsterdam. [cited on page 56]

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 10]

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 198–208. `https://doi.org/10.1145/351240.351259` [cited on pages 15 and 91]

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 15, 16, 18, 19, 72, 76, and 87]

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 pages 15, 24, and 26]

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 pages 4, 82, and 95]

Ronald Garcia, Alison M Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Symposium on Principles of Programming Languages*. [cited on pages 55, 59, 65, and 66]

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]

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing. In *Proceedings of the 22nd International Conference on Functional Programming*. [cited on pages 59, 61, 66, and 95]

Khurram A. Jafery and Joshua Dunfield. 2017. Sums of Uncertainty: Refinements Go Gradual. In *Proceedings of the 44th Symposium on Principles of Programming Languages*. 14. [cited on page 66]

Mark P Jones. 2000. Type classes with functional dependencies. In *European Symposium on Programming*. Springer, 230–244. [cited on page 58]

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 11]

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. ACM, 96–107. [cited on page 58]

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 15]

Jacob Matthews and Amal Ahmed. 2008. Parametric polymorphism through run-time sealing or, theorems for low, low prices!. In *European Symposium on Programming*. Springer, 16–31. [cited on page 71]

Conor McBride. 2002. Faking it Simulating dependent types in Haskell. *Journal of functional programming* 12, 4-5 (2002), 375–392. [cited on page 58]

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]

James Hiram Morris Jr. 1969. *Lambda-calculus models of programming languages.* Ph.D. Dissertation. Massachusetts Institute of Technology. [cited on page 73]

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2009. Non-parametric Parametricity. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 135–148. `https://doi.org/10.1145/1596550.1596572` [cited on page 71]

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 7, 10, 12, and 59]

Michel Parigot. 1992. Recursive programming with proofs. *Theoretical Computer Science* 94, 2 (1992), 335–356. [cited on page 56]

Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: exploring the design space. In *Haskell workshop*, Vol. 1997. [cited on page 58]

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 10, 15, 27, 29, 35, 37, 43, and 72]

Benjamin C Pierce. 2002. *Types and programming languages*. [cited on page 72]

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 pages 15, 23, and 46]

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Proceedings of the IFIP 9th World Computer Congress*. [cited on page 71]

John C. Reynolds. 1991. The Coherence of Languages with Intersection Types. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*. [cited on page 73]

Paula Severi and Erik Poll. 1994. Pure Type Systems with Definitions. *Logical Foundations of Computer Science* (1994), 316–328. [cited on page 48]

Jeremy Siek and Walid Taha. 2007a. 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 and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings of the 2006 Scheme and Functional Programming Workshop*. [cited on page 93]

Jeremy G. Siek and Walid Taha. 2007b. Gradual Typing for Objects. In *European Conference on Object-Oriented Programming*. [cited on pages 53, 54, 59, 62, 64, 65, 89, 90, and 93]

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 pages 4, 66, 69, and 74]

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 pages 10, 14, and 82]

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 pages 15 and 48]

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 pages 5, 28, and 46]

Part VII

Technical Appendix