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

Type inference, as implemented in various modern programming languages, reconstructs missing types in expressions and increases programmers' productivity. Modern functional languages such as Haskell come with powerful forms of type inference. The global type-inference algorithms employed in those languages are derived from the Hindley-Milner type system, with multiple extensions. As the languages evolve, researchers also formalize the key aspects of type inference for the new extensions.

This dissertation studies *predicative implicit higher-rank polymorphism*, where polymorphic types can be arbitrarily nested, and monomorphic types can be inferred automatically. Predicative implicit higher-rank polymorphism is a common extension that has been studied extensively in the literature, and has been used pervasively in modern statically typed programming languages.

The goal of this dissertation is to explore the design space of type inference for implicit predicative higher-rank polymorphism, as well as to study its integration with other advanced type system features. The first contribution of this dissertation is a new type inference algorithm for implicit higher-rank polymorphism which can accepts programs that many existing type inference algorithms cannot. The proposed *application* mode provides new insights for *bidirectional type checking*. The second contribution is the first combination of predicative implicit higher-rank polymorphism with *gradual typing*, which provides a step forward in gradualizing modern functional programming languages. The third contribution is an arguably simpler algorithmic implementation of *subtyping* for higher-rank polymorphism. The technique developed is then further applied to the *kind inference* problem for *datatypes*, which provides a first known formal model of datatype declarations in modern functional programming languages.

An abstract of 253 words

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

Part I

Prologue

1 Introduction

Modern functional languages such as Haskell, ML, and OCaml come with powerful forms of type inference. The global type-inference algorithms employed in those languages are derived from the Hindley-Milner type system (HM) [Damas and Milner 1982; Hindley 1969], with multiple extensions. As the languages evolve, researchers also formalize the key aspects of type inference for the new extensions. One common extension of HM, which is also the central theme of this dissertation, is *higher-rank polymorphism* [Dunfield and Krishnaswami 2013; Odersky and Läufer 1996; Peyton Jones et al. 2007]. In particular, we are interested in *predicative implicit higher-rank polymorphism*, which extends type inference for functional programming languages in the presence of polymorphic types.

1.1 Preliminaries

1.1.1 Type Inference

In real world, many programming languages are typed, including C, Java, and most functional programming languages like Haskell. In those languages, numbers like `1,2,3` are given type **Int**, while `True` and `False` are given type **Bool**. With such type information, if we know that

```
add : Int → Int → Int
```

we can accept expressions like

```
add 1 2
```

while correctly rejecting programs like

```
add 1 True
```

Typed programs are more reliable, as they offer strong static guarantees. For example, if the program is type-checked, then we know for sure that expressions like `add 1 True` will never occur during runtime. Moreover, typed programs often have better performance at runtime since a compiler can apply optimizations according to the type information.

However, writing type annotations can be tedious, especially when the type annotations can be *inferred* from the context. Consider the definition of `add`, which uses the built-in primitive $+ \; : \; \textbf{Int} \to \textbf{Int} \to \textbf{Int}$[1].

```
add = \x:Int. \y:Int. x + y
```

Here we have provided explicit type annotations for `x` and `y`. But we do not really have to: from the use of `+`, it is obvious that the type of these two variables are **Int**. What we really want to write is instead

```
add2 = \x. \y. x + y
```

We thus need *type inference*, which reconstructs missing types in expressions. In this case, with type inference, we would write `add2`, and type inference would automatically figure out the right type annotations, generating `add` for free. Such a facility eliminates a great deal of needless verbosity without losing the benefits of static guarantees. Moreover, it reduces the burden of programmers, as programs are now easier to read and write.

1.1.2 The Hindley-Milner Type System

Most type inference systems used in practice are based on the Hindley-Milner (HM) type system [Damas and Milner 1982; Hindley 1969]. The HM system comes with a simple yet effective algorithm that can infer the most general, or *principal*, types for expressions without any type annotations.

For example, consider the expression

```
id = \x. x
```

There are many possible types we can give for `id`, including $\textbf{Int} \to \textbf{Int}$, and $\textbf{Bool} \to \textbf{Bool}$, etc. In this case, HM will derive the principal type for `id`: $\forall \mathtt{a}.\ \mathtt{a} \to \mathtt{a}$. a *polymorphic* type with a universal quantifier over the type variable `a`. We call types without universal quantifiers, like $\textbf{Int} \to \textbf{Int}$ and $\textbf{Bool} \to \textbf{Bool}$, *monomorphic types* (i.e., *monotypes*), and types like $\forall \mathtt{a}.\ \mathtt{a} \to \mathtt{a}$ polymorphic types. For this example, from the principal type $\forall \mathtt{a}.\ \mathtt{a} \to \mathtt{a}$, other types like $\textbf{Int} \to \textbf{Int}$ and $\textbf{Bool} \to \textbf{Bool}$ can be derived by instantiating `a` to **Int** and **Bool** respectively. With the principal type, we can use `id` as in the following program:

```
let id = \x. x
in (id 1, id True)
```

[1] The syntax \ creates a *lambda* for defining functions. The definition is essentially equivalent to `add(Int x, Int y) {return x + y;}` in languages like Java.

1.1.3 Higher-rank Polymorphism

While elegant and expressive, the HM system comes with a restriction: universal quantifiers in types are restricted to the top-level. For example,

```
∀a. a → a
```

is a valid type, while

```
(∀a. a → a) → int
```

is not as ∀ appears inside the → constructor.

This is unfortunate, as modern programming often requires *higher-rank* polymorphism, i.e., universal quantifiers can appear anywhere inside a type. For example, it is well-known that *rank-2* polymorphic types (i.e., universal quantifier can appear one level *contravariantly* deeper in →) [Jones 1996; McCracken 1984] can be used for resource encapsulation. This is a well-understood technique used in Haskell's state monad [Gill et al. 1993], which has a function `runST` with the following type:

```
runST : ∀a. (∀s. ST s a) → a
```

The ∀ in the rank-2 type ensures by construction that the internal state `s` used by the `ST s a` computation is inaccessible to the rest of the program.

1.1.4 Implicit Polymorphism

System F [Girard 1986; Reynolds 1974] is the *polymorphic lambda calculus* with full power of higher-rank polymorphism, where functions like `runST` can be defined easily. System F has been used extensively in research on polymorphism, and has served as the basis for various programming language designs.

In System F, type arguments are passed explicitly. For example, consider

```
map :: ∀a b. (a → b) → [a] → [b]
fst :: ∀a b. (a, b) → a
```

where `map` takes a function, and a list, and applies the function to every element in the list; and `fst` takes out the first component from a tuple. We can use the functions as

```
map (Int, Char) Int (fst Int Char) [(1, 'a'), (2, 'b')]
-- [(1, 2)]
```

However, writing type arguments, much like writing type annotations, is quite tedious. In this case, the type arguments are almost as large as the program itself!

For systems with polymorphism, type inference enables *implicit polymorphism*, where missing type arguments are reconstructed automatically. In this case, as types can be inferred from the argument `([(1, 'a'), (2, 'b')])`, with type inference we could simply write

```
map fst [(1, 'a'), (2, 'b')]
```

There has been lots of work in extending the HM type system with implicit higher-rank polymorphism [Dunfield and Krishnaswami 2013; Le Botlan and Rémy 2003; Leijen 2009; Peyton Jones et al. 2007; Serrano et al. 2020, 2018].

1.1.5 Predicativity

In a system with polymorphism, 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 instantiated by a monotype; otherwise it is impredicative. For example, instantiating $\mathtt{a}$ with Int in $\forall\mathtt{a}.\ \mathtt{a} \to \mathtt{a}$, generating $\mathsf{Int} \to \mathsf{Int}$, is predicative; while instantiating $\mathtt{a}$ with $\forall\mathtt{a}.\ \mathtt{a} \to \mathsf{Int}$ in $\forall\mathtt{a}.\ \mathtt{a} \to \mathtt{a}$, generating $(\forall\mathtt{a}.\ \mathtt{a} \to \mathsf{Int}) \to (\forall\mathtt{a}.\ \mathtt{a} \to \mathsf{Int})$, is impredicative. HM is an example of predicative polymorphic system, with universal quantifiers restricted to the top-level, while System F is impredicative. It is well-known that general type inference for impredicativity is undecidable [Wells 1999]. The most recent line of work in impredicativity can be found in work by Serrano et al. [2020, 2018].

Type inference for a predicative type system is much easier, while still enables most of the expressiveness of higher-rank polymorphism. Thus just like Dunfield and Krishnaswami [2013]; Peyton Jones et al. [2007], in this work, we focus on *predicative implicit higher-rank polymorphism*. In the rest of this dissertation, whenever we refer to *higher-rank polymorphism*, unless otherwise specified, it denotes predicative implicit higher-rank polymorphism.

1.2 Contribution Overview

The goal of this dissertation is to explore the design space of type inference for implicit predicative higher-rank polymorphism, as well as to study the integration of techniques we have developed into other advanced type system features including *gradual typing* [Siek and Taha 2007] and *kind inference*.

1.2.1 Type Inference for Predicative Implicit Higher-rank Polymorphism

There has been much work on type inference for higher-rank polymorphism [Dunfield and Krishnaswami 2013; Odersky and Läufer 1996; Peyton Jones et al. 2007]. However, since general type inference for higher-rank polymorphism is undecidable [Wells 1999], all work involves difference design tradeoffs. In particular, given `id:∀a. a → a`, consider:

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

Systems including Dunfield and Krishnaswami [2013]; Odersky and Läufer [1996]; Peyton Jones et al. [2007] fail to type-check this program, as they fail to infer a polymorphic type for `f`. However, much like we do not need to write type annotations in expressions like `\x. \y. x + y`, we should not be required to provide an explicit type annotation for `f`, given that we can derive this type information from the context: `id` has type `∀a. a → a`, which can serve as the type of `f`.

Bidirectional type checking, popularized by local type inference [Pierce and Turner 2000], exploits the idea of recovering type information from adjacent nodes in the syntax tree. For example, using bidirectional type checking, type information can be propagated inwards in programs like `(\x. x + 1):` **Int** → **Int**. Several systems [Dunfield and Krishnaswami 2013; Peyton Jones et al. 2007] integrates bidirectional type checking into type inference for higher-rank polymorphism.

Unfortunately, traditional bidirectional typechecking is not working for this example. Specifically, traditional bidirectional checking does not make use of the type information from the *argument* (in this case, `id`) to infer the type of the function (in this case, `(\f. (f 1, f 'a'))`).

The first contribution of this dissertation is a design of a variant of bidirectional type checking algorithm that, when applied to higher-rank polymorphism, is able to accept the above example without any additional type annotations. Like other systems, the design of this system involves different tradeoffs, and those difference tradeoffs provide new insights for designing bidirectional type checking algorithms. Besides illustrating the key idea, we also compare our system in detail with other systems with (bidirectional) type inference for higher-rank polymorphism.

1.2.2 Gradually Typed Higher-rank Polymorphism

Static typing enjoys many benefits. For example, it is guaranteed that ill-typed programs will be rejected at compile-time. Also, types serve as good documentation for programs, as well as to accelerate program execution when combined with type-based compiler optimization. So far we have only considered programs with static typing.

On the other hand, *dynamic typing*, where majority of its type checking is performed at run-time, has its own merits. Languages with dynamic typing, like Python and Javascript, are generally considered to have less cognitive load, better expressiveness, as well as better support for fast prototyping.

Gradual typing [Siek and Taha 2006] is designed to enjoy the best of both worlds. Languages with gradual typing include Clojure [Bonnaire-Sergeant et al. 2016], Python [Lehtosalo et al. 2006; Vitousek et al. 2014], TypeScript [Bierman et al. 2014], etc. With gradual typing, programmers have fine-grained control over the static-to-dynamic spectrum: programs can be partially type-checked, where the type-checked part enjoys benefits from static typing, and the untype-checked part is dynamically type-checked. In particular, gradual typing also provides an explicit type annotation ?, which indicates unknown types that should be type-checked during runtime. As an example, in the following program:

```
\x:Int. \y:?. (x + 1, not y)
```

`x` is statically type-checked and `y` is dynamically type-checked, so that the following program is rejected at compile-time:

```
(\x:Int. \y:?. (x + 1, not y)) 'a' False
```

while the following is only rejected at runtime:

```
(\x:Int. \y:?. (x + 1, not y)) 1 'a'
```

However, while gradual typing is increasingly popular in the programming language research community [Tobin-Hochstadt 2019], the integration of gradual typing with advanced type features still largely remains unclear. This is not surprising though, as great care must be taken in the design of the interaction between static types features and the unknown type. Therefore, there has been more work in adding basic static typing support in dynamically typed languages, than gradualizing statically typed languages with advanced features.

The second contribution of this dissertation is the integration of gradual typing and higher-rank polymorphism. Higher-rank polymorphism, as we have shown, is pervasive in languages like Haskell. Therefore, our study provides a step forward in adding gradual types in modern static typing languages. In particular, with gradual typing, we are able to accept

```
(\f:?. (f 1, f 'a')) id
```

without providing explicitly the large type annotation for `f`.

Designing a gradually typed higher-rank polymorphic type system poses great challenges. First, it requires to integrate *subtyping* and *consistency*. Implicit polymorphism is often built on a *subtyping* relation, which implicitly converts a more general type (e.g., ∀`a. a` → `a`) to a more specific one (e.g., **Int** → **Int**) so that for example `id` can be used where an expression of type **Int** $\to$ **Int** is expected. On the other hand, gradual typing deals with the powerful unknown type, so that an expression with the unknown type can be used as an expression of any type. We show that existing design of such integration [Siek and Taha 2007] is inadequate, and we provide a generalized design that is able to deal with higher-rank polymorphism. Second, we must ensure that our system is well-designed, by showing that our system satisfies the *correctness criteria* [Siek et al. 2015]. We will show that the *dynamic gradual guarantee* is particular tricky to deal with.

1.2.3 Type Promotion and Kind Inference for Datatypes

An ideal type inference algorithm should enjoy various desired properties: *soundness*, *completeness* and *inference of principal types*. An algorithm is sound and complete, if it accepts and only accepts programs that are well-typed in the *declarative* type system.

However, design of type inference algorithms is challenging, as it often involves low-level details, including *constraint solving*, *unification*, etc. In systems with advanced type features, like higher-rank polymorphism, the inference algorithm further needs to deal with the scoping and dependency issues between different kinds of variables. For example, consider the type $\forall$`a.` $\forall$`b. a` $\to$ `b` and $\forall$`c. c` $\to$ `c`. Intuitively, we know that the first type is more general than the other, but how can show that algorithmically? We first need to *skolemize* `c` as a *type variable*, and then instantiate `a, b` with fresh *unification variables*, and finally show that we can *solve* those unification variables with `c`. Handling the scoping and dependency issues properly is tricky.

In the third part of the dissertation, we propose a novel *type promotion* process, which helps resolve the dependency between variables during type inference. We show that it leads to an arguably simpler type inference algorithm for higher-rank polymorphism, and can be easily applied to other advanced features like gradual typing.

Another advanced feature that involves more complicated scoping and dependency issues is *dependent types*. So far, we have only considered programs where expressions can depend on types, e.g., the term `2` has type **Int**. In dependently typed languages, types can depend on expressions, e.g., the type `Vec` **Int** `2` may express a vector of integer of length `2`. A vector with polymorphic length can then be expressed as $\forall$`n:`**Int**`. Vec` **Int** `n`. Note how the term `n` of type **Int** scopes over the body of the type.

In the second half of this part, as another application of promotion, we consider type inference for dependent types in a practical setting; that is, *kind inference* for *datatypes*. Datatype declarations offer a way to define new types along with their constructors. For example,

```
data Maybe a = Nothing | Just a
```

defines a type `Maybe a` with two constructors, `Nothing`, and `Just` which has one field of type `a`. This datatype is useful to express optional types. For example, we can express a division algorithm which, when the second argument is `0`, returns `Nothing`, or otherwise wraps the result inside `Just`.

```
div : Int → Int → Maybe Int
div 42 2  -- Just 21
div 42 0  -- Nothing
```

Note that `Maybe` takes a type (e.g., **Int** in this case), and returns another type (e.g., `Maybe` **Int**). In the same sense as expressions are classified using *types*, types are classified using *kinds*. We say that primitive types like **Int** have kind $*$, and therefore `Maybe` has *kind* $* \to *$. We call the process of inferring the kind of types *kind inference*.

In type systems with only simple types, kind inference for datatypes is straightforward. However, in recent years, languages like Haskell have seen a dramatic surge of new features, and kind inference for datatypes has become non-trivial. For example, consider inferring the kind of the following datatype declarations:

```
data App f a  = MkApp (f a)
data Fix f    = In (f (Fix f))
data T        =  MkT1 (App Maybe Int) |  MkT2 (App Fix Maybe)
```

which includes several complicated features: in the definition of `App`, the type of `f` and `a` can be polymorphic; in `T`, the type `Maybe` and `Fix` are both used in their unsaturated form (i.e., `Maybe` and `Fix` are not applied to any type arguments), and `App` is used polymorphically.

In the second half of this part, we study kind inference for datatypes in two systems: Haskell98, and a more advanced system we call PolyKinds, based on the extensions in modern Haskell, where the type and kind languages are *unified*, and *dependently typed*. We show that proper design of kind inference for datatypes is challenging, and *unification* between dependent types also poses a threat to termination. Both formulations are novel and without precedent, and thus this work can serve as a guide to language designers who wish to formalize their datatype declarations.

1.3 Contributions

In particular, I offer the following specific contributions:

Part II
- Chapter 3 presents an implicit higher-rank polymorphic type system AP, which infers higher-rank types, generalizes the HM type system, and has polymorphic

let as syntactic sugar. As far as we are aware, no previous work enables an HM-style let construct to be expressed as syntactic sugar.

The system is defined based on a variant of *bidirectional type (checking)* [Pierce and Turner 2000] with a new *application* mode. The new variant preserves the advantage of bidirectional type checking, namely many redundant type annotations are removed, while certain programs can type check with even fewer annotations than traditional bidirectional type checking algorithm. We believe that, similarly to standard bidirectional type checking, bidirectional type checking with an application mode can be applied to a wide range of type systems.

Part III

- Chapter 4 integrates implicit higher-rank polymorphism with *gradual types* [Siek and Taha 2006], which is, as far as we are aware, the first work on bridging the gap between implicit higher-rank polymorphism and gradual typing.

  We start by studying the gradually typed subtyping and *type consistency* [Siek and Taha 2006], the central concept for gradual typing, for implicit higher-rank polymorphism. To accomplish this, we first define a framework for *consistent subtyping* [Siek and Taha 2007] with

  - a new definition of consistent subtyping that subsumes and generalizes that of Siek and Taha, and can deal with polymorphism and top types. Our new definition of consistent subtyping preserves the orthogonality between consistency and subtyping. To slightly rephrase Siek and Taha [2007], the motto of this framework is that: *Gradual typing and polymorphism are orthogonal and can be combined in a principled fashion.*[2]
  - a syntax-directed version of consistent subtyping that is sound and complete with respect to our definition of consistent subtyping. The syntax-directed version of consistent subtyping is remarkably simple and well-behaved, and does not require the *restriction* operator of Siek and Taha [2007].

  Based on consistent subtyping, we then present the design of GPC, which stands for **G**radually **P**olymorphic **C**alculus: a (source-level) gradually typed calculus for predicative implicit higher-rank polymorphism that uses our new notion of consistent subtyping. 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 then give a sound and

[2] Note here that we borrow Siek and Taha's motto mostly to talk about the static semantics. As Ahmed et al. [2009] show there are several non-trivial interactions between polymorphism and casts at the level of the dynamic semantics.

complete bidirectional algorithm for implementing the declarative system based on the design principle of Garcia and Cimini [2015].

- Chapter 5 proposes an extension of GPC with type parameters [Garcia and Cimini 2015] as a step towards restoring the *dynamic gradual guarantee* [Siek et al. 2015]. The extension significantly changes the algorithmic system. The new algorithm features a novel use of existential variables with a different solution space, which is a natural extension of the approach by Dunfield and Krishnaswami [2013].

Part IV

- Chapter 6 proposes an arguably simpler algorithmic subtyping of the type inference algorithm for higher-rank implicit polymorphism, based on a new strategy called *promotion* in the *type inference in context* [Dunfield and Krishnaswami 2013; Gundry et al. 2010] framework. Promotion helps resolve the dependency between variables during solving, and can be naturally generalized to more complicated types.

  In this part, we first apply promotion to the unification algorithm for simply typed lambda calculus, and then its polymorphic extension to the subtyping algorithm for implicit predicative higher-rank polymorphism.

- Chapter 7 applies the design of promotion in the context of kind inference for datatypes, and presents two kind inference systems for Haskell. The first system, we believe, is the first formalization of this aspect of Haskell98, and the second one models the challenging features for kind inference in modern Haskell. Specifically,

  - 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* leads to incompleteness.

  - 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.

This thesis is largely based on the publications by the author [Xie et al. 2018, 2019a,c; Xie and Oliveira 2017, 2018], as indicated below. The metatheory of those works is mostly verified using the Coq proof assistant, including type safety, coherence, etc.

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

Chapter 4: Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. 2018. "Consistent Subtyping for All". In *European Symposium on Programming (ESOP)*[4].

Chapter 5: 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)*[5].

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)*[6].

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)*[7].

[3] Proofs in `https://bitbucket.org/ningningxie/let-arguments-go-first/src/master/`.
[4] Proofs in `https://github.com/xnning/Consistent-Subtyping-for-All`.
[5] Proofs in `https://github.com/xnning/Consistent-Subtyping-for-All`.
[6] Proofs in `https://xnning.github.io/papers/sanitized-type-inference-in-context.pdf`.
[7] Proofs in `https://arxiv.org/abs/1911.06153`.

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. Here we pay particular attention to the Dunfield-Krishnaswami system as it serves as a basis for extensions in later chapters; for example, Chapter 4 is a direct extension of Dunfield-Krishnaswami. There is plenty of other related work to higher-rank type system (e.g., Peyton Jones et al. [2007]), and we include a more substantive discussion of those work in Chapter 8.

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.

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

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

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

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

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

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.

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

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

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

Figure 2.2: Declarative subtyping in the Hindley-Milner type system.

the type context Ψ. In rule HM-INST, we can instantiate the type variables with arbitrary *monotypes*.

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 declarative 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:

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

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 any 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,

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

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

A syntax-directed subtyping relation can also be derived, by restricting rule HM-S-FORALLL to allow only for a monotype on the right:

$$\textsc{hm-s-a-forall}L\quad \frac{\vdash^{HM} \sigma[a \mapsto \tau_1] <: \tau}{\vdash^{HM} \forall a.\, \sigma <: \tau}$$

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 implicit 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:

[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.

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

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.

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. The difference is more presentational rather than semantic. 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.

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

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

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

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

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

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.

2.2.4 Discussion: Variance and Eta-equality

As we have discussed before, the subtyping rule for functions in OL is contravariant in arguments, and covariant in return types (rule ol-s-arrow). This is a design choice rather than a requirement. In fact, in some systems, e.g., Serrano et al. [2020, 2018], all type constructs are invariant, including functions, which makes type inference for impredicativity much easier in their setting.

Contravariance and covariance are more powerful than invariance, in the sense that they can accept some more programs. But they also come with a cost. In particular, if we translate a higher-rank implicit polymorphic type system into a system with explicit polymorphism (e.g. System F [Girard 1986]), then contravariance and covariance often require the target language to support η-equality (i.e., an expression $\lambda x.\, e\ x$ is equivalent to e). We show such a translation in Section 3.3. Unfortunately, while η-equality is sound in System F, it may be unsound in some other languages, like Haskell. Specifically, in Haskell, an expression of type Int may reduce to a value of type Int, or it may actually be *bottom* (e.g., `undefined` in Haskell), which throws a runtime error when evaluated. On the other hand, `\x. undefined x`, the η-expanded form of `undefined`, is always a value, and so is semantically different from `undefined`. Thus η-equality is unsound in Haskell.

In systems presented in this thesis, as we do not model the bottom value (or any *side effects*), we support the contravariant/ covariant function subtyping rule as rule OL-S-ARROW, while we see no particular challenges in supporting invariant constructs instead.

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:

Expressions	e	$::=$	$x \mid n \mid \lambda x.\, e \mid e_1\ e_2 \mid e : \sigma$
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

> *"... 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 checking 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:

```
(\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 bidirectional 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 bidirectional type checking.

Dunfield and Pfenning [2004] establish a design principle of bidirectional type checking inspired by *mode correctness* from logical programming, where *introduction rules* are distinguished from *elimination rules*. Following the design principle, constructors corresponding to introduction rules (e.g., tuples) are checked against a given type, while destructors corresponding to elimination rules (e.g., tuple projections) infer a type. DK is designed following this principle.

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 $e : \sigma$ (instead of annotated lambda expressions $\lambda x : \sigma.\, e$), 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.

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. The *application judgment* (discussed shortly) then takes the type σ and the argument e_2, and returns the final result type σ_2. 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 against 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

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

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

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

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

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

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

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

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

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

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.

Application judgment. Notably, unlike HM or OL, DK does not feature an explicit instantiation rule. Instead, rule DK-INF-VAR directly returns a (possibly) polymorphic type, and thus when typing applications (rule DK-APP-INF), we need to explicitly discuss the possible shape of the function type.

The application judgment $\Psi \vdash^{DK} \sigma_1 \cdot e \Rrightarrow \sigma_2$ is interpreted as, when we apply an expression of type σ_1 to the expression e, we get a return type σ_2. For a polymorphic type (rule DK-APP-FORALL), we instantiate the universal quantifier with a monotype, until the type becomes a function type (rule DK-APP-ARR). In the function type case, since the function expects an argument of type σ_1, the rule proceeds by checking e_2 against σ_1.

In some other type systems [Garcia and Cimini 2015; Xie et al. 2018, 2019a], the application judgment is replaced by *matching*. Using matching, rule DK-INF-APP is replaced by rule DK-INF-APP2.

$$\text{DK-INF-APP2}\quad \frac{\begin{array}{c}\Psi \vdash^{DK} e_1 \Rightarrow \sigma \\ \Psi \vdash^{DK} \sigma \rhd \sigma_1 \to \sigma_2 \qquad \Psi \vdash^{DK} e_2 \Leftarrow \sigma_1\end{array}}{\Psi \vdash^{DK} e_1\ e_2 \Rightarrow \sigma_2}$$

In rule DK-INF-APP2, we first derive that e_1 has type σ. But e_1 must have a function type so that it can be applied to an argument. We thus use the *matching* judgment to instantiate σ into a function $\sigma_1 \to \sigma_2$, and proceed by checking e_2 against σ_1, and return the final result σ_2. The definition of matching is given below.

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

$$\text{DK-M-FORALL}\quad \frac{\Psi \vdash^{DK} \tau \qquad \Psi \vdash^{DK} \sigma[a \mapsto \tau] \rhd \sigma_1 \to \sigma_2}{\Psi \vdash^{DK} \forall a.\, \sigma \rhd \sigma_1 \to \sigma_2} \qquad \text{DK-M-ARR}\quad \frac{}{\Psi \vdash^{DK} \sigma_1 \to \sigma_2 \rhd \sigma_1 \to \sigma_2}$$

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.

It can be easily shown that the presentation of rule DK-INF-APP with the application judgment is equivalent to that of rule DK-INF-APP2 with matching. 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 sometimes use the presentation of rule DK-INF-APP2 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-APP 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.

2.3.4 Discussion: lazy and eager instantiation

We say that DK's style of instantiation is *lazy*, where top-level quantifiers are only instantiated when needed (e.g., when applied as a function to arguments); while HM's style of instantiation is *eager*, where in the syntax-directed rules, instantiation eagerly instantiates all top-level universal quantifiers when possible (as in rule HM-VAR-INST). Eager instantiation is also used in the higher-rank polymorphic type system in Peyton Jones et al. [2007].

The differences between lazy and eager instantiation have pervasive consequences. The first and direct consequence is that for the same expression lazy instantiation can derive a more polymorphic type. Consider that given `id: ∀a. a → a`, we want to infer the type of `id`. With lazy instantiation, the algorithm can directly return `∀a. a → a` unchanged, while eager instantiation can only return `a → a` (among others). In this case, we can make two types match by generalizing the final result from eager instantiation, which gives us `∀a. a → a`. However, when the system features higher-rank types, after generalization eager instantiation may still fail to derive the same type. For example, consider the expression

```
\x: Int. id
```

In this case, lazy instantion returns the type `Int → ∀a. a → a`, while eager instantiation returns the type `Int → a → a` which, after generalization, becomes `∀a. Int → a → a`, and is less polymorphic than `Int → ∀a. a → a`, according to the subtyping relation in OL. In this sense naive eager instantiation may reject programs that lazy instantiation can accept, e.g.,

```
let g = \x:Int. id in (\f: Int → ∀a. a → a. f) g
```

The problem can be avoided by, for example, featuring *deep skolemisation* as in Peyton Jones et al. [2007]. Deep skolemisation floats out all its universal quantifiers that appear to the right of a top level arrow, so that `∀a. Int → a → a` and `Int → ∀a. a → a` are actually isomorphic.

Secondly, with lazy instantiation, as instantiation is done only when necessary, we must be aware of *when* it is necessary. We have seen one particular case in the formalism, i.e., when applying a top-level polymorphic type as a function to arguments (rule DK-APP-FORALL). When we extend the system with more constructs, there may be more cases where instantiation is needed. For example, consider we extend the system with **if** expressions:

```
if flag then id else \x. x + 1
```

We expect the two branches in **if** to return the same type. Therefore when collecting the result type from branches, we must instantiate the type of `id` to `Int → Int`, so that it matches the type of another branch. Peyton Jones et al. [2007] discuss three possible typing formalizations of **if** expressions under eager instantiation. To adopt either formalization in lazy instantiation systems, instantiation of the return types of the branches must be performed.

Finally, the choice can have unexpected interaction with other rules involving generalization, for example, **let** expressions. Specifically, under let generalization and lazy instantiation, typability may not be preserved after let-binding inlining. To illustrate the issue, consider the expression:

```
let f = \x. x
in let g = \y. f
in e
```

If we generalize the type of `f` as in HM, then we get `f:∀a. a → a`. Now we continue type-checking `g`, which requires another step of generalization. With lazy instantiation, we get `g: ∀b. b → ∀a. a → a`, which is again, more polymorphic than the type `g: ∀b a. b → a → a` we get from eager instantiation. However, if we inline the definition of `f`, then the program becomes

```
    in let g = \y. \x. x
    in e
```

Now both lazy and eager instantiation can only get the type $\mathtt{g} : \forall \mathtt{b}\ \mathtt{a}.\ \mathtt{b} \to \mathtt{a} \to \mathtt{a}$. Namely, with lazy instantiation, let-binding inlining gets us a less polymorphic type (according to the subtyping rule in OL). The problem has also been discussed in DK [Dunfield and Krishnaswami 2013], where they fix the issue by not treating let expressions specially – namely, there is no generalization for let expressions at all.

In this thesis, the choice does not matter so much for the key contributions of the work. As we do not feature deep skolemisation, we mostly follow the idea of lazy instantiation as in DK. Nevertheless, we believe that deep skolemisation is compatible with our subtyping relations used in later chapters.

Part II

Bidirectional Type Checking With the Application Mode

3 Higher-Rank Polymorphism with the Application Mode

We have seen in Section 2.3 that bidirectional type checking is a useful and versatile tool for type checking and type inference. In traditional bidirectional type-checking, type information flows from functions to arguments (e.g., rule DK-IN-APP in Section 2.3.2). In this section, we present a novel variant of bidirectional 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 bidirectional type checking cannot, and is useful when type information from arguments is required to type-check the functions being applied.

We illustrate our novel design of bidirectional 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 in 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 provides an alternative, which allows types to propagate downward the syntax tree. For example, in the expression $(\lambda f : \mathsf{Int} \to \mathsf{Int}.\, f)\ (\lambda y.\, y)$, the type of y is provided by the type annotation on f. This is supported by the bidirectional typing rule DK-INF-APP for applications:

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

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

One guideline for designing bidirectional type checking rules [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 the checking mode, as in the rule rule DK-CHK-INT:

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

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

In this particular case, bidirectional type checking goes against its original intention of removing burden from programmers, since a seemingly unnecessary annotation is needed. Therefore, in practice, bidirectional type systems do not strictly follow the guideline, and usually have additional inference rules for the introduction form of constructs. For literals, the corresponding rule is rule DK-INF-INT.

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

Now we can type check 1, but the price to pay is that two typing rules for literals are needed. Worse still, the same criticism applies to other constructs, for example, pairs. Below we give the type inference and checking rules for pairs.

$$\frac{\Psi \vdash e_1 \Rightarrow \sigma_1 \qquad \Psi \vdash e_2 \Rightarrow \sigma_2}{\Psi \vdash (e_1, e_2) \Rightarrow (\sigma_1, \sigma_2)}\ \text{INF-PAIR} \qquad \frac{\Psi \vdash e_1 \Leftarrow \sigma_1 \qquad \Psi \vdash e_2 \Leftarrow \sigma_2}{\Psi \vdash (e_1, e_2) \Leftarrow (\sigma_1, \sigma_2)}\ \text{CHECK-PAIR}$$

This shows one drawback of bidirectional type checking: often to minimize annotations, many rules are duplicated for having both the inference mode and the checking mode, which scales up with the typing rules in a type system. It is possible, though, to parameterize the rules over the direction, as is done in Peyton Jones et al. [2007], which is helpful to eliminate the notational duplication but perhaps not the cognitive one.

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 a traditional bidirectional type system if the system follows strictly the design guideline for bidirectional type checking [Dunfield and Pfenning 2004], 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) : Int → 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 **Int** $\rightarrow$ **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

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

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:

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

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

Instead of nothing or all, this partialness gives us a finer grain notion on how much we know about the type of an expression. For example, assume $e : \sigma_1 \to \sigma_2 \to \sigma_3$. In the inference mode, we only have e. In the checking 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 the 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 type of both arguments). Partial type checking has also been used in techniques like *colored local type inference* [Odersky et al. 2001] and *boxy types* [Vytiniotis et al. 2006].

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

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

$\mathsf{id} : \forall a. (a \to a)$, 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 adds a special-purpose application judgment.

With the application mode, all the type information about the arguments being applied is available in the application context 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 Σ and σ_1 are interpreted as inputs and σ_2 as output. In the example above, 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, the type system is able to solve the constraints *locally* according to the application context 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:

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

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:

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

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 bindings 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 systems. However, this is not true because the application mode is *not* conservative over traditional bidirectional 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 systems. This requires understanding how the applications in the checking 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 presents the declarative, *syntax-directed* specification of AP. As mentioned before, the interesting aspects about the new type system are: 1) the typing rules, which employ

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.

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 ($e_1\ e_2$). Notably, 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. Again, we implicitly assume that all variables in Ψ are distinct. In this system, the context is modeled as the HM-style context (Section 2.1), which does not contain type variables; so the type system needs to explicitly ensure freshness of type variables.

The key 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

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

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

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

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

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

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

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

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

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

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

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

Figure 3.2: Typing rules of System AP.

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 in 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.

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-rank 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 some slight reformating to match AP, we get (without the gray-shaded parts):

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

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 parts, where the application context is only used for e_2, because e_2 is the expression being applied. If we desugar the let expression (**let** $x = e_1$ **in** e_2) to $(\lambda x.\, e_2)\ e_1$, we have the following derivation:

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

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 bidirectional type checking, we often have one rule that transfers between the inference and the checking 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 the normal inference mode into the application mode with an application context.

Lemma 3.2 ($\Psi \vdash^{AP} \Rightarrow$ to $\Psi; \Sigma \vdash^{AP} \Rightarrow$). *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 $\Psi \vdash^{AP} \Rightarrow$ to $\Psi; \Sigma \vdash^{AP} \Rightarrow$). *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* $\sigma' <: \sigma$.

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 returning a supertype $\mathsf{Int} \to \mathsf{Int}$ as the type of id. This is precisely what the application subtyping achieves: resolving 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, As with 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* $\vdash^{AP} \sigma_1 <: \Sigma \rightarrow \sigma_2$*, then for some* σ_3*, we have* $\Sigma \vdash^{AP} \sigma_1 <: \Sigma \rightarrow \sigma_3$*, and* $\vdash^{AP} \sigma_3 <: \sigma_2$.

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

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

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

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

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	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)*

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

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

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

Figure 3.3: Syntax and typing rules of System F.

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 do not track type variables. In translation, we use f to refer to the coercion function produced by the 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.

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

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

AP-S-TVAR
$$\frac{}{\vdash^{AP} a <: a \rightsquigarrow \lambda x : a.\, x}$$

AP-S-INT
$$\frac{}{\vdash^{AP} \mathsf{Int} <: \mathsf{Int} \rightsquigarrow \lambda x : \mathsf{Int}.\, x}$$

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

AP-S-FORALLL
$$\frac{\vdash^{AP} \sigma_1[a \mapsto \tau] <: \sigma_2 \rightsquigarrow f}{\vdash^{AP} \forall a.\, \sigma_1 <: \sigma_2 \rightsquigarrow \lambda x : \forall a.\, \sigma_1.\, f\,(x\,\tau)}$$

AP-S-FORALLR
$$\frac{a \notin \textsc{fv}\,(\sigma_1) \qquad \vdash^{AP} \sigma_1 <: \sigma_2 \rightsquigarrow f}{\vdash^{AP} \sigma_1 <: \forall a.\, \sigma_2 \rightsquigarrow \lambda x : \sigma_1.\, \Lambda a.\, f\, x}$$

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

AP-AS-EMPTY
$$\frac{}{\bullet \vdash^{AP} \sigma <: \sigma \rightsquigarrow \lambda x : \sigma.\, x}$$

AP-AS-FORALL
$$\frac{\Sigma, \sigma_3 \vdash^{AP} \sigma_1[a \mapsto \tau] <: \sigma_2 \rightsquigarrow f}{\Sigma, \sigma_3 \vdash^{AP} \forall a.\, \sigma_1 <: \sigma_2 \rightsquigarrow \lambda x : \forall a.\, \sigma_1.\, f\,(x\,\tau)}$$

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

Figure 3.4: Subtyping translation rules of System AP.

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

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

$$\textsc{ap-inf-lamann}\quad \frac{\Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_2 \rightsquigarrow s}{\Psi \vdash^{AP} \lambda x : \sigma_1.\, e \Rightarrow \sigma_1 \to \sigma_2 \rightsquigarrow \lambda x : \sigma_1.\, s}$$

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

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

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

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

Figure 3.5: Typing translation rules of System AP.

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 of type $\sigma \to \sigma_2$ 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 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 in System F [Girard 1986], and it is known that well-typed terms in System F is type safe. In particular, type safety ensures that a well-typed program cannot go wrong [Wright and Felleisen 1994].

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 $\bullet \vdash^{F} f : \sigma \to \sigma_2$.*

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

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

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

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

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

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.

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]. The full rules of the algorithm can be found in Appendix A.

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

$$(S_1, N_1); \Psi \vdash^{AP} e \Rightarrow \sigma \hookrightarrow (S_2, N_2)$$

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

$$\textsc{ap-a-inf-lam}\quad \frac{(S_0, N_0); \Psi, x : \widehat{\beta} \vdash^{AP} e \Rightarrow \sigma \hookrightarrow (S_1, N_1)}{(S_0, N_0\,\widehat{\beta}); \Psi \vdash^{AP} \lambda x.\, e \Rightarrow \widehat{\beta} \to \sigma \hookrightarrow (S_1, N_1)}$$

Comparing it to 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.

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

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* $\overline{a_i}^{\,i} = \textsc{fv}\,(\Psi) - \textsc{fv}(S_2 S_1 \sigma_2)$, *and* $\overline{b_i}^{\,i} = \textsc{fv}\,(\Psi) - \textsc{fv}\,(\sigma)$, *we have* $\vdash^{AP} \forall \overline{a_i}^{\,i}. S_2 S_1 \sigma_2 <: \forall \overline{b_i}^{\,i}. \sigma$.

3.5 Discussion

3.5.1 Combining Application and Checking Modes

Although the application mode provides us with alternative design choices in a bidirectional 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 the expression is known and is therefore used to check the expression, as in DK.

Consider adding $e : \sigma$ for introducing the checking mode for the language. Notice that, since the checking mode is stronger than the application mode, when entering the 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 annotated 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 checked using its annotation σ_1, and then we instantiate σ_1 to σ_2 using application subtyping with the 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 the 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.

$$\text{AP-CHK-LAM}\quad \frac{\Psi, x : \sigma_1 \vdash^{AP} e \Leftarrow \sigma_2}{\Psi \vdash^{AP} \lambda x.\, e \Leftarrow \sigma_1 \to \sigma_2}$$

Moreover, combined with the information flow, the checking 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 annotated expressions 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 any expressive power, since annotated expressions that are typeable 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 the checking 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.

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

The application mode can apply to the elimination constructs of pairs. If one component of the pair is a function, for example, **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:

AP-APP-FST

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

AP-APP-SND

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

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 $\forall a. \forall b. (a, b) \rightarrow a$. 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:

AP-APP-FST-VAR

$$\frac{\Sigma \vdash^{AP} \forall a. \forall b. (a, b) \rightarrow a <: \sigma}{\Psi; \Sigma \vdash^{AP} \mathbf{fst} \Rightarrow \sigma}$$

AP-APP-SND-VAR

$$\frac{\Sigma \vdash^{AP} \forall a. \forall b. (a, b) \rightarrow b <: \sigma}{\Psi; \Sigma \vdash^{AP} \mathbf{snd} \Rightarrow \sigma}$$

Note that another way to model those two rules would be to simply have an initial typing environment $\Psi_{init} \equiv \mathbf{fst} : \forall a. \forall b. (a, b) \rightarrow a, \mathbf{snd} : \forall a. \forall b. (a, b) \rightarrow b$. 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* $(\textbf{fun}[X](x)\ e)\ [\mathsf{Int}]\ 3$."

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 $(\textbf{fun}[X](x)\ e)\ [\mathsf{Int}]\ 3$, 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:

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

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** $\lambda x : a.\, x + 1$.

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 [2007] 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 [Dunfield and Krishnaswami 2013] for higher-rank polymorphism to deal with instantiation.

4.1 Introduction and Motivation

4.1.1 Background: Gradual Typing

Gradual typing [Siek and Taha 2006] is an increasingly popular topic in both programming language practice and theory. On the practical side there is a growing number of programming languages adopting gradual typing. Those languages include Clojure [Bonnaire-Sergeant et al. 2016], Python [Lehtosalo et al. 2006; Vitousek et al. 2014], TypeScript [Bierman et al. 2014], Hack [Verlaguet 2013], and the addition of Dynamic to C# [Bierman et al. 2010], to name a few. On the theoretical side, recent years have seen a large body of research that defines the foundations of gradual typing [Cimini and Siek 2016, 2017; Garcia et al. 2016], explores their use for both functional and object-oriented programming [Siek

$\boxed{\sigma_1 <: \sigma_2}$ *(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 $\mathbf{FOb}^{?}_{<:}$

and Taha 2006, 2007], as well as its applications to many other areas [Bañados Schwerter et al. 2014; Castagna and Lanvin 2017; Jafery and Dunfield 2017].

Siek and Taha [2007] developed a gradual type system for object-oriented languages that they call $\mathbf{FOb}^{?}_{<:}$. A key concept in gradual type systems is the concept of *consistency* (written $\sim$) between gradual types. Consistency weakens type equality to allow for the presence of *unknown* types $?$. 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. As Siek and Taha [2007] put it, this shows that "gradual typing and subtyping are orthogonal and can be combined in a principled fashion".

However, the orthogonality of consistency and subtyping does not lead to a deterministic 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:

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

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

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

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** $\to$ **Int**, or `Char` $\to$ `Char`, but not $\forall$`a. a` $\to$ `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 $\star$. 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` $\to$ `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 λB 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 λB.

[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.

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 well-known 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 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 $?$:

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

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

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:

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

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 type for it in our system.

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

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 $\mathsf{zero}_? : \mathsf{Int}_?$ and $\mathsf{succ}_? : \mathsf{Int}_? \to \mathsf{Int}_?$ are the encodings of the constructors for natural numbers $\mathsf{Int}_?$. 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 $\mathsf{zero}_? : \mathsf{Int}_?$ 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 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 [2007] 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, function 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.

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

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 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.

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.

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 $\forall a.\, a \rightarrow a \prec \mathsf{Int} \rightarrow \mathsf{Bool}$, 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 $\forall$-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]:

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

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 [2007].

[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*.

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

(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.

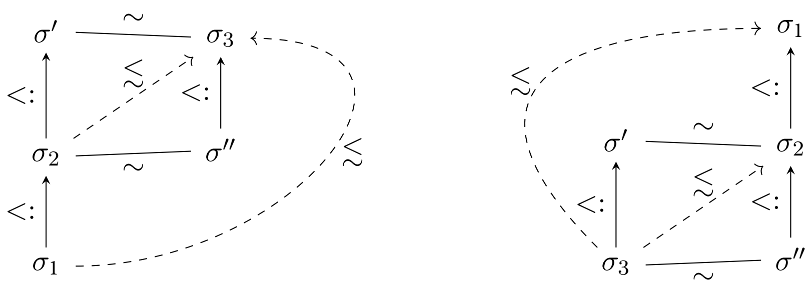
4.2.2 Towards Consistent Subtyping

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: $(\forall a.\, a \to \mathsf{Int}) \to \mathsf{Int}$, and $(? \to \mathsf{Int}) \to \mathsf{Int}$. 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-



(a) (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 $\sigma <: \sigma_2$, and $\sigma_2 \lesssim \sigma_3$, then $\sigma \lesssim \sigma_3$.

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 $\sigma_2 <: \sigma_1$, then $\sigma_3 \lesssim \sigma_1$.

From the above observations, we see what the problem is with the original definition. In Figure 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 Figure 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 subtyping step, one consistency step, and one more step of subtyping*. Specifically, $\sigma_1 <: \sigma_2 \sim \sigma'' <: \sigma_3$ (in Figure 4.4a) and $\sigma_3 <: \sigma' \sim \sigma_2 <: \sigma_1$ (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:

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

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

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

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 [2007]. However, more generally, Definition 5 is compatible with polymorphic types. Furthermore, as we shall see in Section 4.5.1, this definition is also compatible with top types (which are also problematic with the original definition).

Proposition 4.1 (Generalization of Declarative Consistent Subtyping).

- *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).

$$\begin{array}{lcl} \gamma(\mathsf{Int}) & = & \{\mathsf{Int}\} \\ \gamma(\sigma_1 \to \sigma_2) & = & \{\sigma_1' \to \sigma_2' \mid \sigma_1' \in \gamma(\sigma_1), \sigma_2' \in \gamma(\sigma_2)\} \\ \gamma(?) & = & \{\text{All static types}\} \end{array}$$

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], this 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".[5] In their setting, precision is first defined for type constructors and then lifted to gradual types, and subtyping is defined

[5] 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 this work. The full rules can be found in Figure 4.8.

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.

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 $\sigma_1 \sim \sigma_2$.*

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 $\mathsf{Int} \to ? \to \mathsf{Bool} \lesssim \mathsf{Int} \to \mathsf{Int} \to ?$.

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

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,

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

$$\textsc{gpc-cs-tvar}\ \frac{a \in \Psi}{\Psi \vdash^G a \lesssim a} \qquad \textsc{gpc-cs-int}\ \frac{}{\Psi \vdash^G \mathsf{Int} \lesssim \mathsf{Int}} \qquad \textsc{gpc-cs-arrow}\ \frac{\Psi \vdash^G \sigma_3 \lesssim \sigma_1 \qquad \Psi \vdash^G \sigma_2 \lesssim \sigma_4}{\Psi \vdash^G \sigma_1 \to \sigma_2 \lesssim \sigma_3 \to \sigma_4} \qquad \textsc{gpc-cs-forallR}\ \frac{\Psi, a \vdash^G \sigma_1 \lesssim \sigma_2}{\Psi \vdash^G \sigma_1 \lesssim \forall a.\, \sigma_2}$$

$$\textsc{gpc-cs-forallL}\ \frac{\Psi \vdash^G \tau \qquad \Psi \vdash^G \sigma_1[a \mapsto \tau] \lesssim \sigma_2}{\Psi \vdash^G \forall a.\, \sigma_1 \lesssim \sigma_2} \qquad \textsc{gpc-cs-unknownL}\ \frac{}{\Psi \vdash^G\, ? \lesssim \sigma} \qquad \textsc{gpc-cs-unknownR}\ \frac{}{\Psi \vdash^G \sigma \lesssim\, ?}$$

Figure 4.6: Consistent Subtyping for implicit polymorphism.

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 $\Psi \vdash^G \sigma \triangleright \sigma_1 \to \sigma_2$ 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 $\Psi \vdash^G \sigma \triangleright \sigma_1 \to \sigma_2$. Note that the matching rules generalize that of DK in Section 2.3.2 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 Figure 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.

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:

$$\lambda\text{B Terms} \quad s \quad ::= \quad x \mid n \mid \lambda x : \sigma.\, s \mid \Lambda a.\, s \mid s_1\ s_2 \mid \langle \sigma_1 \hookrightarrow \sigma_2 \rangle s$$

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 $\sigma_1 \prec \sigma_2$, 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 $\prec$). *If $\Psi \vdash^G \sigma \triangleright \sigma_1 \to \sigma_2$, then $\sigma \prec \sigma_1 \to \sigma_2$.*

Lemma 4.6 ($\lesssim$ to $\prec$). *If $\Psi \vdash^G \sigma_1 \lesssim \sigma_2$, then $\sigma_1 \prec \sigma_2$.*

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

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

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

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

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

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

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

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

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

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

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:

$$\frac{\Psi, a \vdash^G e : \sigma \rightsquigarrow v}{\Psi \vdash^G e : \forall a.\, \sigma \rightsquigarrow \Lambda a.\, v}\ \textsc{gpc-gen2}$$

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:

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

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 $\mathcal{C}$, written $\mathcal{C} : (\Psi \vdash^{B} \sigma) \rightsquigarrow (\Psi' \vdash^{B} \sigma')$. The definitions of contexts and context typing are standard and thus omitted. We first define contextual approximation in a conventional way. 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_1*,*
  - *if* $\Psi \vdash^{OL} e : \sigma_1$*, then there exists* σ_2*, such that* $\Psi \vdash^G e : \sigma_2$*, and* $\Psi \vdash^G \sigma_2 <: \sigma_1$.
  - *if* $\Psi \vdash^G e : \sigma$*, then* $\Psi \vdash^{OL} e : \sigma$

- ***Monotonicity w.r.t. precision:*** *for all* Ψ, e, e', σ_1, *if* $\Psi \vdash^G e : \sigma_1$, *and* $e' \sqsubseteq e$, *then* $\Psi \vdash^G e' : \sigma_2$, *and* $\sigma_2 \sqsubseteq \sigma_1$ *for some* σ_2.

- ***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 ?[6]. 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 $\lambda x : \mathsf{Int}.\, x + 1$ 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.

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 Figure 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* [Siek et al. 2015].

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

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

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

$$\frac{}{? \sqsubseteq \sigma}\ \text{GPC-L-UNKNOWN} \qquad \frac{}{\mathsf{Int} \sqsubseteq \mathsf{Int}}\ \text{GPC-L-INT} \qquad \frac{\sigma_1 \sqsubseteq \sigma_3 \qquad \sigma_2 \sqsubseteq \sigma_4}{\sigma_1 \to \sigma_2 \sqsubseteq \sigma_3 \to \sigma_4}\ \text{GPC-L-ARROW} \qquad \frac{}{a \sqsubseteq a}\ \text{GPC-L-TVAR}$$

$$\frac{\sigma_1 \sqsubseteq \sigma_2}{\forall a.\, \sigma_1 \sqsubseteq \forall a.\, \sigma_2}\ \text{GPC-L-FORALL}$$

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

$$\frac{}{e \sqsubseteq e}\ \text{GPC-LE-REFL} \qquad \frac{\sigma_1 \sqsubseteq \sigma_2 \qquad e_1 \sqsubseteq e_2}{\lambda x : \sigma_1.\, e_1 \sqsubseteq \lambda x : \sigma_2.\, e_2}\ \text{GPC-LE-LAMANN} \qquad \frac{e_1 \sqsubseteq e_3 \qquad e_2 \sqsubseteq e_4}{e_1\ e_2 \sqsubseteq e_3\ e_4}\ \text{GPC-LE-APP}$$

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

$$\frac{}{x \sqsubseteq x}\ \text{B-LE-VAR} \qquad \frac{}{n \sqsubseteq n}\ \text{B-LE-NAT} \qquad \frac{\sigma_1 \sqsubseteq \sigma_2 \qquad s_1 \sqsubseteq s_2}{\lambda x : \sigma_1.\, s_1 \sqsubseteq \lambda x : \sigma_2.\, s_2}\ \text{B-LE-LAMANN} \qquad \frac{s_1 \sqsubseteq s_2}{\Lambda a.\, s_1 \sqsubseteq \Lambda a.\, s_2}\ \text{B-LE-TABS}$$

$$\frac{s_1 \sqsubseteq s_3 \qquad s_2 \sqsubseteq s_4}{s_1\ s_2 \sqsubseteq s_3\ s_4}\ \text{B-LE-APP} \qquad \frac{\sigma_1 \sqsubseteq \sigma_3 \qquad \sigma_2 \sqsubseteq \sigma_4 \qquad s_1 \sqsubseteq s_2}{\langle \sigma_1 \hookrightarrow \sigma_2 \rangle s_1 \sqsubseteq \langle \sigma_3 \hookrightarrow \sigma_4 \rangle s_2}\ \text{B-LE-CAST}$$

Figure 4.8: Less Precision

- 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 $s \Downarrow$ blame. If $s' \Uparrow$ then $s \Uparrow$ or $s \Downarrow$ 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* [Garcia and Cimini 2015]. We then prove that our algorithm is both sound and complete with respect to the declarative type system. We also provide an implementation.

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* $\blacktriangleright_{\widehat{\alpha}}$ (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.

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	$::=$	$\mathsf{Int} \mid a \mid \widehat{\alpha} \mid \tau_1 \to \tau_2$
Algorithmic Contexts	Γ, Δ, Θ	$::=$	$\bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \widehat{\alpha} \mid \Gamma, \widehat{\alpha} = \tau \mid \Gamma, \blacktriangleright_{\widehat{\alpha}}$
Complete Contexts	Ω	$::=$	$\bullet \mid \Omega, x : \sigma \mid \Omega, a \mid \Omega, \widehat{\alpha} = \tau \mid \Omega, \blacktriangleright_{\widehat{\alpha}}$

$\boxed{\Gamma \vdash^G \sigma}$ *(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)*

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

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

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

Notational convenience. Following DK's 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 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

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

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

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

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

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

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

Figure 4.10: Algorithmic consistent subtyping

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 $\blacktriangleright_{\widehat{\alpha}}$. The output context $(\Delta, \blacktriangleright_{\widehat{\alpha}}, \Theta)$ allows additional existential variables to appear after $\blacktriangleright_{\widehat{\alpha}}$ 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 applies. 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 Figure 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.

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[7] 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

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

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

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

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

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

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

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

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

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

Figure 4.11: Algorithmic instantiation

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 $\widehat{\alpha}_1 \to \widehat{\alpha}_2$. 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.

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 $\bullet \vdash^G \lambda x : \mathsf{Int}.\, \lambda y.\, y \Rightarrow \mathsf{Int} \to \widehat{\alpha} \to \widehat{\alpha} \dashv \bullet$ 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.

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

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

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

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

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

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

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

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

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

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

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

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

Figure 4.12: Algorithmic typing

Algorithmic Matching. Rule gpc-inf-app deserves attention. It relies on the algorithmic matching judgment $\Gamma \vdash^G \sigma \rhd \sigma_1 \to \sigma_2 \dashv \Delta$. 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 $\widehat{\alpha}_1 \to \widehat{\alpha}_2$ 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 $\sigma_1 \to \sigma_2$ 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 $\sigma_1 \to \sigma_2$. 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 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. For (M4), we use *contextual size*—the size of well-formed types under certain contexts, which penalizes solved variables $(*)$.

Definition 10 (Contextual Size).

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

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 $\Gamma \vdash^G \sigma \rhd \sigma_1 \rightarrow \sigma_2 \dashv \Delta$.*

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 $\Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta$.*

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 $\bullet \vdash^G$

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

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

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

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

Figure 4.13: Context extension

$\lambda x.\, x \Rightarrow \widehat{\alpha} \rightarrow \widehat{\alpha} \dashv \widehat{\alpha}$, 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 $\lambda x : (\forall a.\, a \rightarrow a).\, x$, which is typeable in the declarative system with the same type. To relate $\lambda x.\, x$ and $\lambda x : (\forall a.\, a \rightarrow a).\, x$, we erase all annotations on both expressions.

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

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

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* $\widehat{\alpha} \in$ *unsolved*(Γ) *and* $\widehat{\alpha} \notin$ *fv*(σ)*:*

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 $[\Omega]\Gamma \vdash^G [\Omega]\sigma \rhd \sigma_1 \to \sigma_2$ then there exist Δ, Ω', σ_1' and σ_2' such that $\Gamma \vdash^G [\Gamma]\sigma \rhd \sigma_1' \to \sigma_2' \dashv \Delta$ 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 of 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 [2007]. We have shown its applicability to polymorphic types, for which Siek and Taha [2007] 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 [2007] 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, $\mathsf{Int} \to ? \lesssim \top$.

$$\frac{\sigma \text{ static}}{\Psi \vdash^G \sigma <: \top}\ \textsc{gpc-s-top} \qquad \frac{}{\top \sim \top} \qquad \frac{}{\Psi \vdash^G \sigma \lesssim \top}\ \textsc{gpc-cs-top}$$

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 (Equivalence 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, $\mathsf{Int} \to ? \lesssim \top$ only holds when we first apply consistency, then subtyping. However we cannot find a type σ such that $\mathsf{Int} \to ? <: \sigma$ and $\sigma \sim \top$. The following diagram depicts the situation:

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

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 $\Psi \vdash^G \mathsf{plus} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$, then we can apply it to 1 to get

$$\frac{\Psi \vdash^G \mathsf{plus} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \qquad \Psi \vdash^G 1 : \mathsf{Int} \qquad \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \sim \mathsf{Int} \to \mathord{?} \to \mathsf{Int}}{\Psi \vdash \mathsf{plus}\ 1 : \mathord{?} \to \mathsf{Int}}\ \textsc{gpc-v-app1}$$

Further applying it to true we get

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

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:

$$\textsc{gpc-v-sub}\quad \frac{\Psi \vdash^{G} e : \sigma \qquad \sigma <: \sigma_2}{\Psi \vdash^{G} e : \sigma_2}$$

$$\textsc{gpc-v-app2}\quad \frac{\Psi \vdash^{G} e : \sigma \qquad \Psi \vdash^{G} \sigma \rhd \sigma_1 \to \sigma_2 \qquad \Psi \vdash^{G} e_2 : \sigma_3 \qquad \sigma_1 \sim \sigma_3}{\Psi \vdash^{G} e_1\ e_2 : \sigma_2}$$

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

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 [2007].

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 $\vdash^{G}_{m}$ to indicate the more declarative version):

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

Lemma 4.25 (Soundness of $\vdash^{G}_{m}$). *If* $\Psi \vdash^{G}_{m} e : \sigma_1$*, then there exists some* σ_2*, such that* $\Psi \vdash^{G} e : \sigma_2$ *and* $\Psi \vdash^{G} \sigma_2 <: \sigma_1$.

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 λB.

5.1 Declarative Type System

The new syntax of types is given at the top of Figure 5.1, with the differences highlighted. In addition to the types of Figure 4.2, we add *static type parameters* $\mathcal{S}$, and *gradual type parameters* $\mathcal{G}$. Both kinds of type parameters are monotypes. The addition of type parameters, however, leads to two new syntactic categories of types. *Castable types* $\mathbb{C}$ represent types that can be cast from or to $?$. It includes all types, except those that contain static type parameters. *Castable monotypes* t are those castable types that are also monotypes.

Consistent Subtyping. The new definition of consistent subtyping is given at the bottom of Figure 5.1, again with the differences highlighted. Now the unknown type is only a con-

Types	σ	$::=$	$\mathsf{Int} \mid a \mid \sigma_1 \to \sigma_2 \mid \forall a.\, \sigma \mid ? \mid \mathcal{S} \mid \mathcal{G}$
Monotypes	τ	$::=$	$\mathsf{Int} \mid a \mid \tau_1 \to \tau_2 \mid \mathcal{S} \mid \mathcal{G}$
Castable Types	$\mathbb{C}$	$::=$	$\mathsf{Int} \mid a \mid \mathbb{C}_1 \to \mathbb{C}_2 \mid \forall a.\, \mathbb{C} \mid ? \mid \mathcal{G}$
Castable Monotypes	t	$::=$	$\mathsf{Int} \mid a \mid t_1 \to t_2 \mid \mathcal{G}$

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

$$\textsc{gpc-cs-tvar}\quad \frac{a \in \Psi}{\Psi \vdash^G a \lesssim a} \qquad \textsc{gpc-cs-int}\quad \frac{}{\Psi \vdash^G \mathsf{Int} \lesssim \mathsf{Int}} \qquad \textsc{gpc-cs-arrow}\quad \frac{\Psi \vdash^G \sigma_3 \lesssim \sigma_1 \qquad \Psi \vdash^G \sigma_2 \lesssim \sigma_4}{\Psi \vdash^G \sigma_1 \to \sigma_2 \lesssim \sigma_3 \to \sigma_4}$$

$$\textsc{gpc-cs-forallR}\quad \frac{\Psi, a \vdash^G \sigma_1 \lesssim \sigma_2}{\Psi \vdash^G \sigma_1 \lesssim \forall a.\, \sigma_2} \qquad \textsc{gpc-cs-forallL}\quad \frac{\Psi \vdash^G \tau \qquad \Psi \vdash^G \sigma_1[a \mapsto \tau] \lesssim \sigma_2}{\Psi \vdash^G \forall a.\, \sigma_1 \lesssim \sigma_2} \qquad \textsc{gpc-cs-unknownLL}\quad \frac{}{\Psi \vdash^G ? \lesssim \mathbb{C}}$$

$$\textsc{gpc-cs-unknownRR}\quad \frac{}{\Psi \vdash^G \mathbb{C} \lesssim ?} \qquad \textsc{gpc-cs-spar}\quad \frac{}{\Psi \vdash^G \mathcal{S} \lesssim \mathcal{S}} \qquad \textsc{gpc-cs-gpar}\quad \frac{}{\Psi \vdash^G \mathcal{G} \lesssim \mathcal{G}}$$

Figure 5.1: Syntax of types, and consistent subtyping in the extended declarative system.

sistent subtype of all castable types, rather than of all types (rule GPC-CS-UNKNOWNLL), and vice versa (rule GPC-CS-UNKNOWNRR). Moreover, the static type parameter $\mathcal{S}$ is a consistent subtype of itself (rule GPC-CS-SPAR), and similarly for the gradual type parameter (rule GPC-CS-GPAR). From this definition it follows immediately that $?$ is incomparable with types that contain static type parameters $\mathcal{S}$, such as $\mathcal{S} \to \mathsf{Int}$.

TYPING AND TRANSLATION. Given these extensions to types and consistent subtyping, the typing process remains the same as in Figure 4.7. To account for the changes in the translation, if we extend λB with type parameters as in Garcia and Cimini [2015], then the translation remains the same as well.

5.2 Substitutions and Representative Translations

As we mentioned, type parameters serve as placeholders for monotypes. As a consequence, wherever a type parameter is used, any *suitable* monotype could appear just as well. To formalize this observation, we define substitutions for type parameters as follows:

Definition 12 (Substitution). Substitutions for type parameters are defined as:

1. Let $S^{\mathcal{S}} : \mathcal{S} \to \tau$ be a total function mapping static type parameters to monotypes.

2. Let $S^{\mathcal{G}} : \mathcal{G} \to t$ be a total function mapping gradual type parameters to castable monotypes.

3. Let $S^{\mathcal{P}} = S^{\mathcal{G}} \cup S^{\mathcal{S}}$ be a union of $S^{\mathcal{S}}$ and $S^{\mathcal{G}}$ mapping static and gradual type parameters accordingly.

Note that since $\mathcal{G}$ might be compared with $?$, only castable monotypes are suitable substitutes, whereas $\mathcal{S}$ can be replaced by any monotypes. Therefore, we can substitute $\mathcal{G}$ for $\mathcal{S}$, but not the other way around.

Let us go back to our example and its two translations in Section 4.3.2. The problem with those translations is that neither $\mathsf{Int} \to \mathsf{Int}$ nor $\mathsf{Bool} \to \mathsf{Int}$ is general enough. With type parameters, however, we can state a more *general* translation that covers both through substitution:

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

The advantage of type parameters is that they help reasoning about the dynamic semantics. Now we are not limited to a particular choice, such as $\mathsf{Int} \to \mathsf{Int}$ or $\mathsf{Bool} \to \mathsf{Int}$, which might or might not emit a cast error at runtime. Instead we have a general choice $\mathcal{G} \to \mathsf{Int}$.

What does the more general choice with type parameters tell us? First, we know that in this case, there is no concrete constraint on a, so we can instantiate it with a type parameter. Second, the fact that the general choice uses $\mathcal{G}$ rather than $\mathcal{S}$ indicates that any chosen instantiation needs to be a castable type. It follows that any concrete instantiation will have an impact on the runtime behavior; therefore it is best to instantiate a with $?$. However, type inference cannot instantiate a with $?$, and substitution cannot replace $\mathcal{G}$ with $?$ either. This means that we need a syntactic refinement process of the translated programs in order to replace type parameters with allowed gradual types.

Syntactic Refinement. We define syntactic refinement of the translated expressions as follows. As $\mathcal{S}$ denotes no constraints at all, substituting it with any monotype would work. Here we arbitrarily use Int. We interpret $\mathcal{G}$ as $?$ since any monotype could possibly lead to a cast error.

Definition 13 (Syntactic Refinement). The syntactic refinement of a translated expression s is denoted by $\lceil s \rceil$, and defined as follows:

$$
\begin{array}{lcl}
\lceil \mathsf{Int} \rceil & = & \mathsf{Int} \\
\lceil a \rceil & = & a \\
\lceil \sigma_1 \to \sigma_2 \rceil & = & \lceil \sigma_1 \rceil \to \lceil \sigma_2 \rceil \\
\lceil \forall a.\, \sigma \rceil & = & \forall a.\, \lceil \sigma \rceil \\
\lceil ? \rceil & = & ? \\
\lceil \mathcal{S} \rceil & = & \mathsf{Int} \\
\lceil \mathcal{G} \rceil & = & ?
\end{array}
$$

Applying the syntactic refinement to the translated expression, we get

$$(\lambda x : ?.(\langle \forall a.\, a \to \mathsf{Int} \hookrightarrow \boxed{?} \to \mathsf{Int} \rangle\, f)(\langle ? \hookrightarrow \boxed{?} \rangle\, x))$$

where two $\mathcal{G}$ are refined by ? as highlighted. It is easy to verify that both applying this expression to 3 and to *true* now results in a translation that evaluates to a value.

Representative Translations. To decide whether one translation is more general than the other, we define a preorder between translations.

Definition 14 (Translation Pre-order). Suppose $\Psi \vdash^G e : \sigma \rightsquigarrow s_1$ and $\Psi \vdash^G e : \sigma \rightsquigarrow s_2$, we define $s_1 \leqslant s_2$ to mean $s_2 \equiv_\alpha S^{\mathcal{P}}\,(s_1)$ for some $S^{\mathcal{P}}$.

Proposition 5.1. *If $s_1 \leqslant s_2$ and $s_2 \leqslant s_1$, then s_1 and s_2 are α-equivalent (i.e., equivalent up to renaming of type parameters).*

The preorder between translations gives rise to a notion of what we call *representative translations*:

Definition 15 (Representative Translation). A translation s is said to be a representative translation of a typing derivation $\Psi \vdash^G e : \sigma \rightsquigarrow s$ if and only if for any other translation $\Psi \vdash^G e : \sigma \rightsquigarrow s'$ such that $s' \leqslant s$, we have $s \leqslant s'$. From now on we use r to denote a representative translation.

An important property of representative translations, which we conjecture for the lack of rigorous proof, is that if there exists any translation of an expression that (after syntactic refinement) can reduce to a value, so can a representative translation of that expression. Conversely, if a representative translation runs into a blame, then no translation of that expression can reduce to a value.

Conjecture 5.2 (Property of Representative Translations). *For any expression e such that $\Psi \vdash^G e : \sigma \rightsquigarrow s$ and $\Psi \vdash^G e : \sigma \rightsquigarrow r$ and $\forall \mathcal{C}.\, \mathcal{C} : (\Psi \vdash^B \sigma) \rightsquigarrow (\bullet \vdash^B \mathsf{Int})$, we have*

- *If* $\mathcal{C}\{\lceil s \rceil\} \Downarrow n$, *then* $\mathcal{C}\{\lceil r \rceil\} \Downarrow n$.
- *If* $\mathcal{C}\{\lceil r \rceil\} \Downarrow \mathsf{blame}$, *then* $\mathcal{C}\{\lceil s \rceil\} \Downarrow \mathsf{blame}$.

Given this conjecture, we can state a stricter coherence property (without the "up to casts" part) between any two representative translations. We first strengthen Definition 8 following Ahmed et al. [2009]:

Definition 16 (Contextual Approximation à la Ahmed et al. [2009]).

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

The only difference is that now when a program containing s_1 reduces to a value, so does one containing s_2.

From Conjecture 5.2, it follows that coherence holds between two representative translations of the same expression.

Corollary 5.3 (Coherence for Representative Translations). *For any expression e such that $\Psi \vdash^G e : \sigma \rightsquigarrow r_1$ and $\Psi \vdash^G e : \sigma \rightsquigarrow r_2$, we have $\Psi \vdash r_1 \simeq_{ctx} r_2 : \sigma$.*

We have proved that for every typing derivation, at least one representative translation exists.

Lemma 5.4 (Representative Translation for Typing). *For any typing derivation $\Psi \vdash^G e : \sigma$ there exists at least one representative translation r such that $\Psi \vdash^G e : \sigma \rightsquigarrow r$.*

For our example, $(\lambda x : ?.\, (\langle \forall a.\, a \rightarrow \mathsf{Int} \hookrightarrow \mathcal{G} \rightarrow \mathsf{Int} \rangle f)\ (\langle ? \hookrightarrow \mathcal{G} \rangle x))$ is a representative translation, while the other two are not.

5.3 Dynamic Gradual Guarantee, Reloaded

Given the above propositions, we are ready to revisit the dynamic gradual guarantee. The nice thing about representative translations is that the dynamic gradual guarantee of our source language is essentially that of λB, our target language. However, the dynamic gradual guarantee for λB is still an open question. According to Igarashi et al. [2017], the difficulty lies in the definition of term precision that preserves the semantics. We leave it here as a conjecture as well. From a declarative point of view, we cannot prevent the system from picking undesirable instantiations, but we know that some choices are better than the others, so we can restrict the discussion of dynamic gradual guarantee to representative translations.

Types	σ	$::=$	$\mathsf{Int} \mid a \mid \widehat{\alpha} \mid \sigma_1 \to \sigma_2 \mid \forall a.\, \sigma \mid ? \mid \mathcal{S} \mid \mathcal{G}$
Monotypes	τ	$::=$	$\mathsf{Int} \mid a \mid \widehat{\alpha} \mid \tau_1 \to \tau_2 \mid \mathcal{S} \mid \mathcal{G}$
Existential variables	$\widehat{\alpha}$	$::=$	$\widehat{\alpha}_S \mid \widehat{\alpha}_G$
Castable Types	$\mathbb{C}$	$::=$	$\mathsf{Int} \mid a \mid \widehat{\alpha} \mid \mathbb{C}_1 \to \mathbb{C}_2 \mid \forall a.\, \mathbb{C} \mid ? \mid \mathcal{G}$
Castable Monotypes	t	$::=$	$\mathsf{Int} \mid a \mid \widehat{\alpha} \mid t_1 \to t_2 \mid \mathcal{G}$
Algorithmic Contexts	Γ, Δ, Θ	$::=$	$\bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \widehat{\alpha} \mid \Gamma, \widehat{\alpha}_S = \tau \mid \Gamma, \widehat{\alpha}_G = t \mid \Gamma, \blacktriangleright_{\widehat{\alpha}}$
Complete Contexts	Ω	$::=$	$\bullet \mid \Omega, x : \sigma \mid \Omega, a \mid \Omega, \widehat{\alpha}_S = \tau \mid \Omega, \widehat{\alpha}_G = t \mid \Omega, \blacktriangleright_{\widehat{\alpha}}$

Figure 5.2: Syntax of types, contexts and consistent subtyping in the extended algorithmic system.

Conjecture 5.5 (Dynamic Gradual Guarantee in terms of Representative Translations)**.** *Suppose $e' \sqsubseteq e$,*

1. *If $\bullet \vdash^G e : \sigma \rightsquigarrow r$, $\lceil r \rceil \Downarrow v$, then for some σ_2 and r', we have $\bullet \vdash^G e' : \sigma_2 \rightsquigarrow r'$, and $\sigma_2 \sqsubseteq \sigma$, and $\lceil r' \rceil \Downarrow v'$, and $v' \sqsubseteq v$.*
2. *If $\bullet \vdash^G e' : \sigma_2 \rightsquigarrow r'$, $\lceil r' \rceil \Downarrow v'$, then for some σ and r, we have $\bullet \vdash^G e : \sigma \rightsquigarrow r$, and $\sigma_2 \sqsubseteq \sigma$. Moreover, $\lceil r \rceil \Downarrow v$ and $v' \sqsubseteq v$, or $\lceil r \rceil \Downarrow \mathsf{blame}$.*

For the example in Section 4.3.3, now we know that the representative translation of the right one will evaluate to 1 as well.

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

More importantly, in what follows, we show that our extended algorithm is able to find those representative translations.

5.4 Extended Algorithmic Type System

To understand the design choices involved in the new algorithmic system, we consider the following algorithmic typing example:

$$f : \forall a.\, a \to \mathsf{Int}, x : ? \vdash^G f\, x : \mathsf{Int} \dashv f : \forall a.\, a \to \mathsf{Int}, x : ?, \widehat{\alpha}$$

Compared with declarative typing, where we have many choices (e.g., $\mathsf{Int} \to \mathsf{Int}$, $\mathsf{Bool} \to \mathsf{Int}$, and so on) to instantiate $\forall a.\, a \to \mathsf{Int}$, the algorithm computes the instantiation $\widehat{\alpha} \to \mathsf{Int}$ with $\widehat{\alpha}$ unsolved in the output context. What can we know from the algorithmic typing? First we know that, here $\widehat{\alpha}$ is *not constrained* by the typing problem. Second, and more importantly,

$\widehat{\alpha}$ has been compared with an unknown type (when typing $(f\,x)$). Therefore, it is possible to make a more refined distinction between different kinds of existential variables:

- the first kind of existential variables are those that indeed have no constraints at all, as they do not affect the dynamic semantics;
- the second kind (as in this example) are those where the only constraint is that *the variable was once compared with an unknown type* [Garcia and Cimini 2015].

The syntax of types is shown in Figure 5.2. A notable difference, apart from the addition of static and gradual parameters, is that we further split existential variables $\widehat{\alpha}$ into static existential variables $\widehat{\alpha}_S$ and gradual existential variables $\widehat{\alpha}_G$. Depending on whether an existential variable has been compared with *?* or not, its solution space changes. More specifically, static existential variables can be solved to a monotype τ, whereas gradual existential variables can only be solved to a castable monotype t, as can be seen in the changes of algorithmic contexts and complete contexts. As a result, the typing result for the above example now becomes

$$f : \forall a.\, a \to \mathsf{Int}, x : ? \vdash^{G} f\,x : \mathsf{Int} \dashv f : \forall a.\, a \to \mathsf{Int}, x : ?, \boxed{\widehat{\alpha}_G}$$

since we can solve any unconstrained $\widehat{\alpha}_G$ to $\mathcal{G}$, it is easy to verify that the resulting translation is indeed a representative translation.

Our extended algorithm is novel in the following aspects. We naturally extend the concept of existential variables [Dunfield and Krishnaswami 2013] to deal with comparisons between existential variables and unknown types. Unlike Garcia and Cimini [2015], where they use an extra set to store types that have been compared with unknown types, our two kinds of existential variables emphasize the type distinction better, and correspond more closely to the two kinds of type parameters, as we can solve $\widehat{\alpha}_S$ to $\mathcal{S}$ and $\widehat{\alpha}_G$ to $\mathcal{G}$.

5.4.1 Extended Algorithmic Consistent Subtyping

While the changes in the syntax seem negligible, the addition of static and gradual type parameters changes the algorithmic judgments in a significant way. We first discuss the algorithmic consistent subtyping, which is shown in Figure 5.3. For notational convenience, when static and gradual existential variables have the same rule form, we compress them into one rule. For example, rule GPC-AS-EVAR is really two rules $\Gamma[\widehat{\alpha}_S] \vdash^{G} \widehat{\alpha}_S \lesssim \widehat{\alpha}_S \dashv \Gamma[\widehat{\alpha}_S]$ and $\Gamma[\widehat{\alpha}_G] \vdash^{G} \widehat{\alpha}_G \lesssim \widehat{\alpha}_G \dashv \Gamma[\widehat{\alpha}_G]$; same for rules GPC-AS-INSTL and GPC-AS-INSTR.

$\boxed{\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta}$ *(Algorithmic Consistent Subtyping)*

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

$$\frac{}{\Gamma \vdash^G \mathcal{S} \lesssim \mathcal{S} \dashv \Gamma}\ \textsc{gpc-as-spar} \qquad \frac{}{\Gamma \vdash^G \mathcal{G} \lesssim \mathcal{G} \dashv \Gamma}\ \textsc{gpc-as-gpar}$$

$$\frac{}{\Gamma \vdash^G ? \lesssim \mathbb{C} \dashv \mathsf{contaminate}(\Gamma, \mathbb{C})}\ \textsc{gpc-as-unknown}\mathrm{LL} \qquad \frac{}{\Gamma \vdash^G \mathbb{C} \lesssim ? \dashv \mathsf{contaminate}(\Gamma, \mathbb{C})}\ \textsc{gpc-as-unknown}\mathrm{RR}$$

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

$$\frac{\Gamma, \blacktriangleright_{\widehat{a}_S}, \widehat{\alpha}_S \vdash^G \sigma_1[a \mapsto \widehat{\alpha}_S] \lesssim \sigma_2 \dashv \Delta, \blacktriangleright_{\widehat{a}_S}, \Theta}{\Gamma \vdash^G \forall a.\, \sigma_1 \lesssim \sigma_2 \dashv \Delta}\ \textsc{gpc-as-forall}\mathrm{LL}$$

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

Figure 5.3: Extended algorithmic consistent subtyping

Rules GPC-AS-SPAR and GPC-AS-GPAR are direct analogies of rules GPC-CS-SPAR and GPC-CS-GPAR. Though looking simple, rules GPC-AS-UNKNOWNLL and GPC-AS-UNKNOWNRR deserve much explanation. To understand what the output context contaminate$(\Gamma, \mathbb{C})$ is for, let us first see why this seemingly intuitive rule $\Gamma \vdash^G ? \lesssim \mathbb{C} \dashv \Gamma$ (like rule GPC-AS-UNKNOWNL in the original algorithmic system) is wrong. Consider the judgment $\widehat{\alpha}_S \vdash^G ? \lesssim \widehat{\alpha}_S \to \widehat{\alpha}_S \dashv \widehat{\alpha}_S$, which seems fine. If this holds, then – since $\widehat{\alpha}_S$ is unsolved in the output context – we can solve it to $\mathcal{S}$ for example (recall that $\widehat{\alpha}_S$ can be solved to some monotype), resulting in $? \lesssim \mathcal{S} \to \mathcal{S}$. However, this is in direct conflict with rule GPC-CS-UNKNOWNLL in the declarative system precisely because $\mathcal{S} \to \mathcal{S}$ is not a castable type! A possible solution would be to transform all static existential variables to gradual existential variables within $\mathbb{C}$ whenever it is being compared to ?: while $\widehat{\alpha}_S \vdash^G ? \lesssim \widehat{\alpha}_S \to \widehat{\alpha}_S \dashv \widehat{\alpha}_S$ does not hold, $\widehat{\alpha}_G \vdash^G ? \lesssim \widehat{\alpha}_G \to \widehat{\alpha}_G \dashv \widehat{\alpha}_G$ does. While substituting static existential variables with gradual existential variables seems to be intuitively correct, it is rather hard to formulate—not only do we need to perform substitution in $\mathbb{C}$, we also need to substitute accordingly in both the input and output contexts in order to ensure that no existential variables become unbound. However, making such changes is at odds with the interpretation of input contexts: they are "input", which evolve into output contexts with more variables solved. Therefore, in line with the use of input contexts, a simple solution is to generate a new gradual existential variable and solve the static existential variable to it in the output context, without touching $\mathbb{C}$ at all. So we have $\widehat{\alpha}_S \vdash^G ? \lesssim \widehat{\alpha}_S \to \widehat{\alpha}_S \dashv \widehat{\alpha}_G, \widehat{\alpha}_S = \widehat{\alpha}_G$.

Based on the above discussion, the following defines contaminate(Γ, σ):

Definition 17. contaminate(Γ, σ) is defined inductively as follows

$$
\begin{array}{lcl}
\mathsf{contaminate}(\bullet, \sigma) & = & \bullet \\
\mathsf{contaminate}((\Gamma, x : \sigma), \sigma) & = & \mathsf{contaminate}(\Gamma, \sigma), x : \sigma \\
\mathsf{contaminate}((\Gamma, a), \sigma) & = & \mathsf{contaminate}(\Gamma, \sigma), a \\
\mathsf{contaminate}((\Gamma, \widehat{\alpha}_S), \sigma) & = & \mathsf{contaminate}(\Gamma, \widehat{\alpha}_G, \sigma), \widehat{\alpha}_S = \widehat{\alpha}_G \\
 & & \text{if } \widehat{\alpha}_S \text{ occurs in } \sigma \\
\mathsf{contaminate}((\Gamma, \widehat{\alpha}_S), \sigma) & = & \mathsf{contaminate}(\Gamma, \sigma), \widehat{\alpha}_S \\
 & & \text{if } \widehat{\alpha}_S \text{ does not occur in } \sigma \\
\mathsf{contaminate}((\Gamma, \widehat{\alpha}_G), \sigma) & = & \mathsf{contaminate}(\Gamma, \sigma), \widehat{\alpha}_G \\
\mathsf{contaminate}((\Gamma, \widehat{\alpha} = \tau), \sigma) & = & \mathsf{contaminate}(\Gamma, \sigma), \widehat{\alpha} = \tau \\
\mathsf{contaminate}((\Gamma, \blacktriangleright_{\widehat{\alpha}}), \sigma) & = & \mathsf{contaminate}(\Gamma, \sigma), \blacktriangleright_{\widehat{\alpha}}
\end{array}
$$

contaminate(Γ, σ) solves all static existential variables found within σ to fresh gradual existential variables in Γ. Notice the case for contaminate$((\Gamma, \widehat{\alpha}_S), \sigma)$ is exactly what we have just described.

Rule gpc-as-forallLL is slightly different from rule gpc-as-forallL in the original algorithmic system in that we replace a with a new static existential variable $\widehat{\alpha}_S$. Note that $\widehat{\alpha}_S$ might be solved to a gradual existential variable later. The rest of the rules are the same as those in the original system.

5.4.2 Extended Instantiation

The instantiation judgments shown in Figure 5.4 also change significantly. The complication comes from the fact that now we have two different kinds of existential variables, and the relative order that they appear in the context affects their solutions.

Rules gpc-instl-solveS and gpc-instl-solveG are the refinement to rule gpc-instl-solve in the original system. The next two rules deal with situations where one side is an existential variable and the other side is an unknown type. Rule gpc-instl-solveUS is a special case of rule gpc-as-unknownRR where we create a new gradual existential variable $\widehat{\alpha}_G$ and set the solution of $\widehat{\alpha}_S$ to be $\widehat{\alpha}_G$ in the output context. Rule gpc-instl-solveUG is the same as rule gpc-instl-solveU in the original system and simply propagates the input context. The next two rules gpc-instl-reachSG1 and gpc-instl-reachSG2 are a bit involved, but they both answer to the same question: how to solve a gradual existential variable when it is declared after some static existential variable. More concretely, in rule gpc-instl-reachSG1, we feel that we need to solve $\widehat{\beta}_G$ to another existential variable. However, simply setting $\widehat{\beta}_G = \widehat{\alpha}_S$ and leaving $\widehat{\alpha}_S$ untouched in the output context is wrong. The reason is that $\widehat{\beta}_G$ could be a gradual existential variable created by rule gpc-as-unknownLL/gpc-as-unknownRR and solving $\widehat{\beta}_G$ to a static existential variable would result in the same problem as we have discussed. Instead, we create another new gradual existential variable $\widehat{\alpha}_G$ and set the solutions of both $\widehat{\alpha}_S$ and $\widehat{\beta}_G$ to it; similarly in rule gpc-instl-reachSG2. Rule gpc-instl-reachOther deals with the other cases (e.g., $\widehat{\alpha}_S \lessapprox \widehat{\beta}_S$, $\widehat{\alpha}_G \lessapprox \widehat{\beta}_G$ and so on). In those cases, we employ the same strategy as in the original system.

As for the other instantiation judgment, most of the rules are symmetric and thus omitted. The only interesting rule is gpc-instr-forallLL, which is similar to what we did for rule gpc-as-forallLL.

5.4.3 Algorithmic Typing and Metatheory

Fortunately, the changes in the algorithmic bidirectional system are minimal: we replace every existential variable with a static existential variable. Furthermore, we proved that the extended algorithmic system is sound and complete with respect to the extended declarative system. The full extended algorithmic system can be found in Appendix B.

$\boxed{\Gamma \vdash^G \widehat{\alpha} \lessapprox \sigma \dashv \Delta}$ *(Instantiation I)*

$$\text{GPC-INSTL-SOLVES}\quad \frac{\Gamma \vdash^G \tau}{\Gamma, \widehat{\alpha}_S, \Gamma' \vdash^G \widehat{\alpha}_S \lessapprox \tau \dashv \Gamma, \widehat{\alpha}_S = \tau, \Gamma'}$$

$$\text{GPC-INSTL-SOLVEG}\quad \frac{\Gamma \vdash^G t}{\Gamma, \widehat{\alpha}_G, \Gamma' \vdash^G \widehat{\alpha}_G \lessapprox t \dashv \Gamma, \widehat{\alpha}_G = t, \Gamma'}$$

$$\text{GPC-INSTL-SOLVEUS}\quad \frac{}{\Gamma[\widehat{\alpha}_S] \vdash^G \widehat{\alpha}_S \lessapprox \,?\, \dashv \Gamma[\widehat{\alpha}_G, \widehat{\alpha}_S = \widehat{\alpha}_G]}$$

$$\text{GPC-INSTL-SOLVEUG}\quad \frac{}{\Gamma[\widehat{\alpha}_G] \vdash^G \widehat{\alpha}_G \lessapprox \,?\, \dashv \Gamma[\widehat{\alpha}_G]}$$

$$\text{GPC-INSTL-REACHSG1}\quad \frac{}{\Gamma[\widehat{\alpha}_S][\widehat{\beta}_G] \vdash^G \widehat{\alpha}_S \lessapprox \widehat{\beta}_G \dashv \Gamma[\widehat{\alpha}_G, \widehat{\alpha}_S = \widehat{\alpha}_G][\widehat{\beta}_G = \widehat{\alpha}_G]}$$

$$\text{GPC-INSTL-REACHSG2}\quad \frac{}{\Gamma[\widehat{\beta}_S][\widehat{\alpha}_G] \vdash^G \widehat{\alpha}_G \lessapprox \widehat{\beta}_S \dashv \Gamma[\widehat{\beta}_G, \widehat{\beta}_S = \widehat{\beta}_G][\widehat{\alpha}_G = \widehat{\beta}_G]}$$

$$\text{GPC-INSTL-REACHOTHER}\quad \frac{}{\Gamma[\widehat{\alpha}][\widehat{\beta}] \vdash^G \widehat{\alpha} \lessapprox \widehat{\beta} \dashv \Gamma[\widehat{\alpha}][\widehat{\beta} = \widehat{\alpha}]}$$

$$\text{GPC-INSTL-FORALLR}\quad \frac{\Gamma[\widehat{\alpha}], b \vdash^G \widehat{\alpha} \lessapprox \sigma \dashv \Delta, b, \Theta}{\Gamma[\widehat{\alpha}] \vdash^G \widehat{\alpha} \lessapprox \forall b.\, \sigma \dashv \Delta}$$

$$\text{GPC-INSTL-ARR}\quad \frac{\Gamma[\widehat{\alpha}_2, \widehat{\alpha}_1, \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2] \vdash^G \sigma_1 \lessapprox \widehat{\alpha}_1 \dashv \Theta \qquad \Theta \vdash^G \widehat{\alpha}_2 \lessapprox [\Theta]\sigma_2 \dashv \Delta}{\Gamma[\widehat{\alpha}] \vdash^G \widehat{\alpha} \lessapprox \sigma_1 \to \sigma_2 \dashv \Delta}$$

$\boxed{\Gamma \vdash^G \sigma \lessapprox \widehat{\alpha} \dashv \Delta}$ *(Instantiation II, excerpt)*

$$\text{GPC-INSTR-FORALLLL}\quad \frac{\Gamma[\widehat{\alpha}], \blacktriangleright_{\widehat{b}_S}, \widehat{\beta}_S \vdash^G \sigma[b \mapsto \widehat{\beta}_S] \lessapprox \widehat{\alpha} \dashv \Delta, \blacktriangleright_{\widehat{b}_S}, \Theta}{\Gamma[\widehat{\alpha}] \vdash^G \forall b.\, \sigma \lessapprox \widehat{\alpha} \dashv \Delta}$$

Figure 5.4: Instantiation in the extended algorithmic system

5.4.4 Discussion

Do We Really Need Type Parameters in the Algorithmic System? As we mentioned earlier, type parameters in the declarative system are merely an analysis tool, and in practice, type parameters are inaccessible to programmers. For the sake of proving soundness and completeness, we have to endow the algorithmic system with type parameters. However, the algorithmic system already has static and gradual existential variables, which can serve the same purpose. In that regard, we could directly solve every *unsolved* static and gradual existential variable in the output context to Int and $?$, respectively.

5.5 Restricted Generalization

In Section 4.3.2, we discussed the issue that the translation produces multiple target expressions due to the different choices for instantiations, and those translations have different dynamic semantics. Besides that, there is another cause for multiple translations: redundant generalization during translation by rule GEN. Consider the simple expression $(\lambda x : \mathsf{Int}.\, x)\ 1$, the following shows two possible translations:

$$\bullet \vdash (\lambda x : \mathsf{Int}.\, x)\ 1 : \mathsf{Int} \rightsquigarrow (\lambda x : \mathsf{Int}.\, x)\ (\langle \mathsf{Int} \hookrightarrow \mathsf{Int} \rangle 1)$$
$$\bullet \vdash (\lambda x : \mathsf{Int}.\, x)\ 1 : \mathsf{Int} \rightsquigarrow (\lambda x : \mathsf{Int}.\, x)\ (\langle \forall a.\, \mathsf{Int} \hookrightarrow \mathsf{Int} \rangle (\Lambda a.\, 1))$$

The difference comes from the fact that in the second translation, we apply rule GEN while typing 1 to get $\bullet \vdash 1 : \forall a.\, \mathsf{Int}$. As a consequence, the translation of 1 is accompanied by a cast from $\forall a.\, \mathsf{Int}$ to Int since the former is a consistent subtype of the latter. This difference is harmless, because obviously these two expressions will reduce to the same value in $\lambda \mathsf{B}$, thus preserving coherence (up to cast error). While it is not going to break coherence, it does result in multiple representative translations for one expression (e.g., the above two translations are both the representative translations).

There are several ways to make the translation process more deterministic. For example, we can restrict generalization to happen only in let expressions and require let expressions to include annotations, as $\textbf{let}\, x : \sigma = e_1 \,\textbf{in}\, e_2$. Another feasible option would be to give a declarative, bidirectional system as the specification (instead of the type assignment one), in the same spirit of DK [Dunfield and Krishnaswami 2013]. Then we can restrict generalization to be performed through annotations in checking mode.

With restricted generalization, we hypothesize that now each expression has exactly one representative translation (up to renaming of fresh type parameters). Instead of calling it a *representative* translation, we can say it is a *principal* translation. Of course the above is only a sketch; we have not defined the corresponding rules, nor studied metatheory.

Part IV

Type Inference with Promotion

6 Higher-Rank Type Inference with Type Promotion

Gundry et al. [2010] proposed type inference in context as a general foundation for unification/type inference algorithms. The key idea is based on the notion of information increase. However, their semantic definition of information increase makes it hard to prove metatheory formally. Following their work, a more syntactic foundation for information increase is presented by DK (Dunfield and Krishnaswami [2013]) to deal with higher-rank polymorphism. However, the DK approach produces duplication and cannot be easily generalized to handle more complicated types.

In this chapter, we propose a strategy called *promotion* that helps resolve the dependency of existential variables in the framework of type inference in context. As an illustration, Section 6.2 applies promotion to the unification algorithm for the simply typed lambda calculus. Section 6.3 further proposes *polymorphic promotion* to deal with subtyping for higher-rank polymorphism. Finally, we briefly discuss how to promote dependent types and gradual types in Section 6.4. This chapter also sets up the stage for Chapter 7, where promotion is used in a more complex setting.

6.1 Introduction and Motivation

6.1.1 Background:Type Inference in Context

Gundry et al. [2010] model unification and type inference from a general perspective of information increase. Specifically, they studied the unification and type inference problem for a ML-style polymorphic system, based on the requirement that types may depend only on earlier bindings in the type context. Besides contexts being ordered, a key insight of the approach lies in how to solve existential variables with other types. In particular, it requires to resolve the dependency between existential variables. Consider unifying $\widehat{\alpha}$ with $\widehat{\beta} \to \mathsf{Int}$ under the context $\widehat{\alpha}, \widehat{\beta}$. Here we cannot simply set $\widehat{\alpha}$ to $\widehat{\beta} \to \mathsf{Int}$, as $\widehat{\beta}$ is out of the scope of $\widehat{\alpha}$. The way Gundry et al. [2010] solve this problem is to examine variables in the context from the tail to the head, *moving segments of the context to the left if necessary*. The process finishes when the existential variable being unified is found. In this case, Gundry et al. [2010] would return a solution context $\widehat{\beta}, \widehat{\alpha} = \widehat{\beta} \to \mathsf{Int}$, where $\widehat{\beta}$ is moved to the front of $\widehat{\alpha}$. However, while moving type variables around is a feasible way to resolve the dependency between existential variables, the unpredictable context movements make the information increase hard to formalize and reason about. In their system, the information increase of contexts is defined in a semantic way: a context Γ_1 is more informative than another context Γ_2, if there exists a substitution such that every item in Γ_2 is, after context substitution, well-formed under Γ_1. This semantic definition makes it hard to prove metatheory formally, especially when advanced features are involved.

The Dunfield-Krishnaswami type system (DK) [Dunfield and Krishnaswami 2013] also uses ordered contexts as the input and output of the type inference algorithm for a higher-rank polymorphic type system (Section 2.3). Unlike Gundry et al. [2010], DK does it in a more syntactic way. In their type system, instantiation rules decompose type constructs so that unification between existential variables can only happen between single variables, which can then be solved by setting the one that appears later to the one that appears earlier. This way, the information increase of contexts is modeled as the intuitive and syntactic definition of *context extension* ($\Gamma \longrightarrow \Delta$), which allows for inductive reasoning and proofs. This approach is adopted in GPC (Chapter 4), so let us consider how DK works in terms of the instantiation rules in GPC. Specifically, consider the derivation of $\widehat{\alpha}, \widehat{\beta} \vdash^G \widehat{\alpha} \lessapprox \widehat{\beta} \to \mathsf{Int}$ in GPC:

$$\Delta = \widehat{\alpha}_1, \widehat{\alpha}_2, \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2, \widehat{\beta}$$

$$\dfrac{\dfrac{\textsc{gpc-instl-reach}}{\Delta \vdash^G \widehat{\beta} \lessapprox \widehat{\alpha}_1 \dashv \Delta[\widehat{\beta} = \widehat{\alpha}_1]} \qquad \dfrac{\textsc{gpc-instr-solve}}{\Delta[\widehat{\beta} = \widehat{\alpha}_1] \vdash^G \widehat{\alpha}_2 \lessapprox \mathsf{Int} \dashv \Delta[\widehat{\alpha}_2 = \mathsf{Int}][\widehat{\beta} = \widehat{\alpha}_1]}}{\widehat{\alpha}, \widehat{\beta} \vdash^G \widehat{\alpha} \lessapprox \widehat{\beta} \to \mathsf{Int} \dashv \Delta[\widehat{\alpha}_2 = \mathsf{Int}][\widehat{\beta} = \widehat{\alpha}_1]} \;\textsc{gpc-instl-arr}$$

By rule GPC-INSTL-ARR, the variable $\widehat{\alpha}$ is solved by an arrow type $\widehat{\alpha}_1 \to \widehat{\alpha}_2$ consisting of two fresh existential variables. The two variables $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ are then instantiated with $\widehat{\beta}$ and Int, respectively. By rule GPC-INSTL-REACH, the variable $\widehat{\beta}$ is solved by $\widehat{\alpha}_1$, as $\widehat{\alpha}_1$ appears in the context before $\widehat{\beta}$, or otherwise we need to apply rule GPC-INSTR-REACH to set $\widehat{\alpha}_1$ to $\widehat{\beta}$ instead. The final solution context is $\Delta[\widehat{\alpha}_2 = \mathsf{Int}][\widehat{\beta} = \widehat{\alpha}_1] = \widehat{\alpha}_1, \widehat{\alpha}_2 = \mathsf{Int}, \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2, \widehat{\beta} = \widehat{\alpha}_1$.

CHALLENGES. However, while the approach of decomposing type constructs works perfectly for DK and GPC, it has two drawbacks. First, it produces duplication: in order to deal with both cases, the instantiation rules are duplicated for when the existential variable appears on the left ($\Gamma \vdash^G \widehat{\alpha} \lessapprox \sigma \dashv \Delta$ in Figure 4.11), and when it appears on the right

($\Gamma \vdash^G \sigma \lessapprox \widehat{\alpha} \dashv \Delta$ in Figure 4.11). For example, rule GPC-INSTL-ARR has its symmetric counterpart rule GPC-INSTR-ARR:

GPC-INSTL-ARR
$$\frac{\Gamma[\widehat{\alpha}_2, \widehat{\alpha}_1, \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2] \vdash^G \sigma_1 \lessapprox \widehat{\alpha}_1 \dashv \Theta \qquad \Theta \vdash^G \widehat{\alpha}_2 \lessapprox [\Theta]\sigma_2 \dashv \Delta}{\Gamma[\widehat{\alpha}] \vdash^G \widehat{\alpha} \lessapprox \sigma_1 \to \sigma_2 \dashv \Delta}$$

GPC-INSTR-ARR
$$\frac{\Gamma[\widehat{\alpha}_2, \widehat{\alpha}_1, \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2] \vdash^G \widehat{\alpha}_1 \lessapprox \sigma_1 \dashv \Theta \qquad \Theta \vdash^G [\Theta]\sigma_2 \lessapprox \widehat{\alpha}_2 \dashv \Delta}{\Gamma[\widehat{\alpha}] \vdash^G \sigma_1 \to \sigma_2 \lessapprox \widehat{\alpha} \dashv \Delta}$$

Worse, this kind of "duplication" would scale up with the number of type constructs in the system.

Second, while decomposition works for function types, it may not work easily for more complicated types, e.g., dependent types. For example, consider that under the context $\widehat{\alpha}, \widehat{\beta}$, we want to instantiate $\widehat{\alpha}$ with a dependent type $\Pi a : \widehat{\beta}.a$. Here because $\widehat{\beta}$ appears after $\widehat{\alpha}$, we cannot directly set $\widehat{\alpha} = \Pi a : \widehat{\beta}.a$, which is ill-typed. However, if we try to decompose the type $\Pi a : \widehat{\beta}.a$ like in rule GPC-INSTL-ARR, in which case we have $\widehat{\alpha} = \Pi a : \widehat{\alpha}_1.\widehat{\alpha}_2$, it is obvious that $\widehat{\alpha}_2$ should be solved by a. Then, in order to make the solution well typed, we need to put a in the front of $\widehat{\alpha}_2$ in the context. However, this means that a would remain in the context, and it would be available for any later existential variables that should not have access to a.

6.1.2 Our Approach: Type Promotion

We propose the *promotion* process, which helps resolve the dependency between existential variables. Promotion combines the advantages of Gundry et al. [2010] and DK: it is a simple and predictable process, so that information increase can still be modeled as the syntactic context extension; moreover, it does not cause any duplication.

To understand how promotion works, let us consider again the example $\widehat{\alpha}, \widehat{\beta} \vdash^G \widehat{\alpha} \lessapprox \widehat{\beta} \to \mathsf{Int}$. The problem here is that $\widehat{\beta}$ is out of the scope of $\widehat{\alpha}$ so we cannot directly set $\widehat{\alpha} = \widehat{\beta} \to \mathsf{Int}$. Therefore, we first *promote* the type $\widehat{\beta} \to \mathsf{Int}$. At a high level, the promotion process looks for free existential variables in the type, and solves those out-of-scope existential variables with fresh ones added to the front of $\widehat{\alpha}$, such that existential variables in the promoted type are all in the scope of $\widehat{\alpha}$. In this case, we will solve $\widehat{\beta}$ with a fresh variable $\widehat{\alpha}_1$, producing the context $\widehat{\alpha}_1, \widehat{\alpha}, \widehat{\beta} = \widehat{\alpha}_1$. Notice that $\widehat{\alpha}_1$ is inserted right before $\widehat{\alpha}$. Now the instantiation example becomes $\widehat{\alpha}_1, \widehat{\alpha}, \widehat{\beta} = \widehat{\alpha}_1 \vdash^G \widehat{\alpha} \lessapprox \widehat{\alpha}_1 \to \mathsf{Int}$, and $\widehat{\alpha}_1 \to \mathsf{Int}$ is a valid solution for

$\widehat{\alpha}$. Therefore, we get a final solution context $\widehat{\alpha}_1, \widehat{\alpha} = \widehat{\alpha}_1 \to \mathsf{Int}, \widehat{\beta} = \widehat{\alpha}_1$. Comparing the result with the solution context we get from DK ($\widehat{\alpha}_1, \widehat{\alpha}_2 = \mathsf{Int}, \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2, \widehat{\beta} = \widehat{\alpha}_1$), it is obvious that these two solutions are equivalent up to substitution.

Interpretation of Promotion. The approach taken by Gundry et al. [2010] and the approach used by DK are based on the same observation: *the relative order between existential variables does not matter for solving a constraint*. The promotion process captures precisely this observation. Its task is to "move" existential variables to suitable positions *indirectly*, by solving those out-of-scope existential variables with fresh in-scope ones.

This seems to go against the design principle that the contexts are ordered. However, ordering is still important for variables whose order matters. For instance, for polymorphic types, the order between existential variables $\widehat{\alpha}$ and type variables a is important, so we cannot set $\widehat{\alpha}$ to a under the context $(\widehat{\alpha}, a)$ as a is not in the scope of $\widehat{\alpha}$. Moreover, ordering prevents invalid cyclic contexts, e.g., $\widehat{\alpha} = \widehat{\beta} \to \mathsf{Int}, \widehat{\beta} = \widehat{\alpha} \to \mathsf{Int}$.

Unification for the Simply Typed Lambda Calculus. As a first illustration of the promotion process, Section 6.2 recasts the unification process for the simply typed lambda calculus (STLC) using the promotion process. This system illustrates the key idea of promotion.

6.1.3 Polymorphic Promotion

Instead of unification, the instantiation relation in DK actually deals with the polymorphic subtyping relation between existential variables and other types. The promotion process we described above only works for unification. In this section, we discuss promotion for polymorphic subtyping.

The difficulty of subtyping is that it needs to take unification into account at the same time. For example, given that $\widehat{\alpha}$ is a subtype of Int, the only possible solution is $\widehat{\alpha} = \mathsf{Int}$. Now consider $\widehat{\alpha} \vdash \forall a.\, a \to a <: \widehat{\alpha}$. How can we promote the polymorphic type $\forall a.\, a \to a$ into a monotype which can serve as a valid solution for $\widehat{\alpha}$? One possible answer is to set $\widehat{\alpha} = \mathsf{Int} \to \mathsf{Int}$, or $\widehat{\alpha} = \mathsf{Bool} \to \mathsf{Bool}$. In fact, the most general solution for this subtyping problem is $\widehat{\alpha} = \widehat{\beta} \to \widehat{\beta}$ with fresh $\widehat{\beta}$. Namely, we remove the universal quantifier in $\forall a.\, a \to a$ and replace the variable a with a fresh existential variable $\widehat{\beta}$ added to the front of $\widehat{\alpha}$, resulting in the solution context $\widehat{\beta}, \widehat{\alpha} = \widehat{\beta} \to \widehat{\beta}$.

On the other hand, how can we promote the type $\forall a.\, a \to a$ in $\widehat{\alpha} \vdash \widehat{\alpha} <: \forall a.\, a \to a$? It turns out that this subtyping is actually unsolvable, as there is no monotype that can be a subtype of $\forall a.\, a \to a$. Therefore, in this case, promoting $\forall a.\, a \to a$ will directly add the type variable a to the tail of the context to promote $a \to a$. Since a is added to the tail, it means that a is out of the scope of $\widehat{\alpha}$ and promoting $a \to a$ would fail, which is exactly what we want. In fact, the promotion would succeed only if the universally quantified variable is not used in the body of the polymorphic type. For example, $\forall a.\, \mathsf{Int} \to \mathsf{Int}$ can be promoted to $\mathsf{Int} \to \mathsf{Int}$, which is a valid solution for $\widehat{\alpha}$ in $\widehat{\alpha} \vdash \widehat{\alpha} <: \forall a.\, \mathsf{Int} \to \mathsf{Int}$.

From these observations, we extend promotion to *polymorphic promotion*, which is able to resolve the polymorphic subtyping relation for existential variables. Depending on whether the existential variable appears on the right or left, polymorphic promotion has two modes, which we call the *contravariant mode* and the *covariant mode* respectively.

The contravariant mode promotes types as $\forall a.\, a \to a$ in the case of $\widehat{\alpha} \vdash \forall a.\, a \to a <: \widehat{\alpha}$, where the universal quantifier is removed and the type variable a is replaced by a fresh existential variable added to front of the existential variable being solved. This corresponds to rule GPC-InstR-forallL, except that with promotion, the new existential variable $\widehat{\beta}$ (in rule GPC-InstR-forallL) will be added directly before $\widehat{\alpha}$ and there is no need to create a marker or to discard the context after $\widehat{\beta}$ anymore.

The covariant mode promotes types as $\forall a.\, a \to a$ in the case of $\widehat{\alpha} \vdash \widehat{\alpha} <: \forall a.\, a \to a$. In this case, promoting $\forall a.\, a \to a$ will directly add the type variable a to the tail of the context, which corresponds to rule GPC-InstL-forallR. Since the type variable is out of the scope of the existential variable being solved, and promotion will succeed only if the variable is not used in the body of the polymorphic type.

While promoting polymorphic types behaves differently according to the mode, the mode does not matter for monotypes, as in both $\widehat{\alpha} <: \mathsf{Int}$ and $\mathsf{Int} <: \widehat{\alpha}$, $\widehat{\alpha} = \mathsf{Int}$ would be the only solution. Since function types are contravariant in codomains and covariant in domains, promoting a function type under a certain mode proceeds to promote its codomain under the other mode and promote its domain under the original mode. For example, $\widehat{\alpha} = (\widehat{\beta} \to \widehat{\beta}) \to (\mathsf{Int} \to \mathsf{Int})$ is a solution for $\widehat{\alpha} \vdash \widehat{\alpha} <: (\forall a.\, a \to a) \to (\forall a.\, \mathsf{Int} \to \mathsf{Int})$, where $(\forall a.\, a \to a) \to (\forall a.\, \mathsf{Int} \to \mathsf{Int})$ is promoted under the covariant mode, which means $\forall a.\, a \to a$ is promoted under the *contravariant* mode and $\forall a.\, \mathsf{Int} \to \mathsf{Int}$ is promoted under the original covariant mode.

Polymorphic Promotion for Subtyping. We illustrate polymorphic promotion by showing that the original instantiation relationship in DK can be replaced by our polymorphic promotion process. Furthermore, we show that subtyping, which was built upon instantiation but now uses polymorphic promotion, remains sound and complete.

6.2 Unification for the Simply Typed Lambda Calculus

This section first introduces the simply typed lambda calculus, and then presents a unification algorithm that uses the novel promotion mechanism.

6.2.1 Declarative System

The definition of declarative types in STLC is given below. We have only monotypes τ, which includes the integer type Int and function types $\tau_1 \to \tau_2$. In this section, we focus on the unification process. Hence, we do not elaborate the details of expressions' syntax or typing rules.

$$\text{Monotypes} \quad \tau \quad ::= \quad \mathsf{Int} \mid \tau_1 \to \tau_2$$

6.2.2 Algorithmic System

The syntax of the algorithmic system is given in Figure 6.1. Following DK [Dunfield and Krishnaswami 2013] and GPC, algorithmic monotypes include existential type variables $\widehat{\alpha}$. Algorithmic contexts also contain declarations of existential type variables, either unsolved ($\widehat{\alpha}$) or solved ($\widehat{\alpha} = \tau$). Complete contexts Ω contain only solved variables. We use the judgment $\Gamma \vdash^{\mathsf{wf}} \tau$ to indicate that all existential variables in τ are well-scoped. Its definition is standard and thus omitted. We also use $\Gamma \longrightarrow \Delta$ for context extension, whose definition is essentially a simplified version of the one in GPC (Section 4.4.5).

Unification. Figure 6.1 defines the unification process. The judgment $\Gamma \vdash^{\mathsf{u}} \tau_1 \approx \tau_2 \dashv \Delta$ reads that under the input context Γ, unifying τ_1 with τ_2 results in the output context Δ. Rule u-refl is our base case, and rule u-arrow unifies the components of the arrow types. When unifying $\widehat{\alpha} \approx \tau_1$ (rule u-evarL), we cannot simply set $\widehat{\alpha}$ to τ_1, as τ_1 might include variables bound to the right of $\widehat{\alpha}$. Instead, we need to *promote* ($\Vdash^{\mathsf{pr}}$) τ_1. After promoting τ_1 to τ_2, we can directly set $\widehat{\alpha} = \tau_2$. Rule u-evarR is symmetric to rule u-evarL. Note that when unifying $\widehat{\alpha} \approx \widehat{\beta}$, either rule u-evarL and rule u-evarR could be tried; an implementation can arbitrarily choose between them.

Promotion. The promotion relation $\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta$ given at the bottom of Figure 6.1 reads that under the input context Δ, promoting type τ_1 yields type τ_2, so that τ_2 is well-formed in the prefix context of $\widehat{\alpha}$, while retaining $[\Delta]\tau_1 = [\Delta]\tau_2$. At a high-level, $\Vdash^{\mathsf{pr}}$ looks for free variables in τ_1. Integers are always well-formed (rule pr-int). Promoting a function recursively promotes its components (rule pr-arrow). Variables bound to the left of $\widehat{\alpha}$ in Γ

Monotypes	τ	$::=$	$\mathsf{Int} \mid \tau_1 \to \tau_2 \mid \widehat{\alpha}$
Algorithmic Contexts	Γ, Δ, Θ	$::=$	$\bullet \mid \Gamma, \widehat{\alpha} \mid \Gamma, \widehat{\alpha} = \tau$
Complete Contexts	Ω	$::=$	$\bullet \mid \Omega, \widehat{\alpha} = \tau$

$\boxed{\Gamma \vdash^{\mathsf{u}} \tau_1 \approx \tau_2 \dashv \Delta}$ *(Unification)*

$$\textsc{u-refl}\ \frac{}{\Gamma \vdash^{\mathsf{u}} \tau \approx \tau \dashv \Gamma} \qquad \textsc{u-arrow}\ \frac{\Gamma \vdash^{\mathsf{u}} \tau_1 \approx \tau_3 \dashv \Theta \qquad \Theta \vdash^{\mathsf{u}} [\Theta]\tau_2 \approx [\Theta]\tau_4 \dashv \Delta}{\Gamma \vdash^{\mathsf{u}} \tau_1 \to \tau_2 \approx \tau_3 \to \tau_4 \dashv \Delta}$$

$$\textsc{u-evarL}\ \frac{\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta[\widehat{\alpha}]}{\Gamma \vdash^{\mathsf{u}} \widehat{\alpha} \approx \tau_1 \dashv \Delta[\widehat{\alpha} = \tau_2]} \qquad \textsc{u-evarR}\ \frac{\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta[\widehat{\alpha}]}{\Gamma \vdash^{\mathsf{u}} \tau_1 \approx \widehat{\alpha} \dashv \Delta[\widehat{\alpha} = \tau_2]}$$

$\boxed{\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta}$ *(Promotion)*

$$\textsc{pr-int}\ \frac{}{\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \mathsf{Int} \rightsquigarrow \mathsf{Int} \dashv \Gamma} \qquad \textsc{pr-arrow}\ \frac{\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \tau_1 \rightsquigarrow \tau_3 \dashv \Theta \qquad \Theta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} [\Theta]\tau_2 \rightsquigarrow \tau_4 \dashv \Delta}{\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \tau_1 \to \tau_2 \rightsquigarrow \tau_3 \to \tau_4 \dashv \Delta}$$

$$\textsc{pr-evarL}\ \frac{}{\Gamma[\widehat{\beta}][\widehat{\alpha}] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\beta} \rightsquigarrow \widehat{\beta} \dashv \Gamma[\widehat{\beta}][\widehat{\alpha}]} \qquad \textsc{pr-evarR}\ \frac{}{\Gamma[\widehat{\alpha}][\widehat{\beta}] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\beta} \rightsquigarrow \widehat{\beta}_1 \dashv \Gamma[\widehat{\beta}_1, \widehat{\alpha}][\widehat{\beta} = \widehat{\beta}_1]}$$

Figure 6.1: Types, contexts, unification and promotion of algorithmic STLC

are unaffected (rule PR-EVARL), as they are already well-formed. In rule PR-EVARR, a unification variable $\widehat{\beta}$ bound to the right of $\widehat{\alpha}$ in Γ is replaced by a fresh variable introduced to $\widehat{\alpha}$'s left. Promotion is a partial operation, as it requires $\widehat{\beta}$ either to be to the right or to the left of $\widehat{\alpha}$. There is yet another possibility: if $\widehat{\beta} = \widehat{\alpha}$, then no rule applies. This is a desired property, as the $\widehat{\beta} = \widehat{\alpha}$ case exactly corresponds to the "occurs-check" in a more typical presentation of unification. By preventing promoting $\widehat{\alpha}$ to the left of $\widehat{\alpha}$, we prevent the possibility of an infinite substitution when applying an algorithmic context. Note that rule U-REFL solves the unification case $\widehat{\alpha} \approx \widehat{\alpha}$.

Example. Below we give the derivation of $\widehat{\alpha}, \widehat{\beta} \vdash^{\mathsf{u}} \widehat{\alpha} \approx \widehat{\beta} \to \mathsf{Int}$ discussed in Section 6.1.1.

$$
\dfrac{
  \dfrac{
    \dfrac{\textsc{pr-evarR}}{\widehat{\alpha}, \widehat{\beta} \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\beta} \rightsquigarrow \widehat{\alpha}_1 \dashv \widehat{\alpha}_1, \widehat{\alpha}, \widehat{\beta} = \widehat{\alpha}_1}
    \qquad
    \dfrac{\textsc{pr-int}}{\widehat{\alpha}_1, \widehat{\alpha}, \widehat{\beta} = \widehat{\alpha}_1 \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \mathsf{Int} \rightsquigarrow \mathsf{Int} \dashv \widehat{\alpha}_1, \widehat{\alpha}, \widehat{\beta} = \widehat{\alpha}_1}
  }{
    \widehat{\alpha}, \widehat{\beta} \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\beta} \to \mathsf{Int} \rightsquigarrow \widehat{\alpha}_1 \to \mathsf{Int} \dashv \widehat{\alpha}_1, \widehat{\alpha}, \widehat{\beta} = \widehat{\alpha}_1
  }\ \textsc{pr-arrow}
}{
  \widehat{\alpha}, \widehat{\beta} \vdash^{\mathsf{u}} \widehat{\alpha} \approx \widehat{\beta} \to \mathsf{Int} \dashv \widehat{\alpha}_1, \widehat{\alpha} = \widehat{\alpha}_1 \to \mathsf{Int}, \widehat{\beta} = \widehat{\alpha}_1
}\ \textsc{u-eval-r}
$$

6.2.3 Soundness and Completeness

We prove that our type promotion strategy and the unification algorithm are sound. First, we show that except for resolving the order problem, promotion will not change the type. Namely, the input type and the output type are equivalent after substitution by the output context. Moreover, the promoted type is well-formed under the prefix context of $\widehat{\alpha}$.

Theorem 6.1 (Soundness of Promotion). *If* $\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta$, *then* $[\Delta]\tau_1 = [\Delta]\tau_2$. *Moreover, given* $\Delta = \Delta_1, \widehat{\alpha}, \Delta_2$, *we have* $\Delta_1 \vdash^{\mathsf{wf}} \tau_2$,

With soundness of promotion, we can prove that the unification algorithm is also sound:

Theorem 6.2 (Soundness of Unification). *If* $\Gamma \vdash^{\mathsf{u}} \tau_1 \approx \tau_2 \dashv \Delta$, *then* $[\Delta]\tau_1 = [\Delta]\tau_2$.

We can further prove that promotion is complete using the notion of context extension. Note that in the completeness statement we require $\widehat{\alpha} \notin \textsc{fv}(\tau_1)$, or otherwise promotion would fail.

Theorem 6.3 (Completeness of Promotion). *Given* $\Gamma \longrightarrow \Omega$, *and* $\Gamma \vdash^{\mathsf{wf}} \widehat{\alpha}$, *and* $\Gamma \vdash^{\mathsf{wf}} \tau_1$, *and* $[\Gamma]\widehat{\alpha} = \widehat{\alpha}$, *and* $[\Gamma]\tau_1 = \tau_1$, *if* $\widehat{\alpha} \notin \textsc{fv}(\tau_1)$, *there exist* τ_2, Δ *and* Ω' *such that* $\Gamma \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta$.

The completeness of unification is then built upon the completeness of promotion.

Theorem 6.4 (Completeness of Unification). *Given* $\Gamma \longrightarrow \Omega$, *and* $\Gamma \vdash^{\mathsf{wf}} \tau_1$, *and* $\Gamma \vdash^{\mathsf{wf}} \tau_2$, *and* $[\Gamma]\tau_1 = \tau_1$, *and* $[\Gamma]\tau_2 = \tau_2$, *if* $[\Omega]\tau_1 = [\Omega]\tau_2$, *there exist* Δ *and* Ω' *such that* $\Gamma \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash^{\mathsf{u}} \tau_1 \approx \tau_2 \dashv \Delta$.

6.3 Subtyping for Higher-Rank Polymorphism

In this section, we adopt the type promotion strategy to a higher-rank polymorphic type system from DK [Dunfield and Krishnaswami 2013]. We show that promotion can be further extended to polymorphic promotion to deal with subtyping, which can be used to replace the instantiation relation in the original DK system while preserving soundness and completeness.

6.3.1 Declarative System

The definition of types in DK (Figure 2.6 in Section 2.3.2) is repeated below. Comparing to STLC, we have polymorphic types $\forall a.\, \sigma$ and type variables a. Again, we omit the details about expressions since we focus on types in this section. Recall that DK shares the same subtyping relation as of OL (Figure 2.5), and we use the judgment $\Psi \vdash^{DK} \sigma_1 <: \sigma_2$ to denote the subtyping relation in DK.

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	$::=$	$\bullet \mid \Psi, x : \sigma \mid \Psi, a$

6.3.2 Algorithmic System

The syntax of the algorithmic system is given in Figure 6.2. The promotion mode $\otimes$ is either covariant (+) or contravariant (−). We can use $-\otimes$ to flip the promotion mode. Specifically,

$$\begin{array}{lcl} -(+) & = & - \\ -(-) & = & + \end{array}$$

Subtyping. Figure 6.2 also includes the subtyping judgment $\Gamma \vdash^{\mathsf{sub}} \sigma_1 <: \sigma_2 \dashv \Delta$, which reads that, under the input context Γ, type σ_1 is a subtype of σ_2, with the output context Δ. The rules except the last two are the same as the algorithmic subtyping rules in DK.

Rule s-instL and rule instR are specific to this system. Recall that in GPC (which follows DK), the consistent subtyping between $\widehat{\alpha}$ and σ replies on the instantiation rules, which are duplicated for the case when $\widehat{\alpha}$ is on the left and the case when $\widehat{\alpha}$ is on the right. Here, instead

Types	σ	$::=$	$\mathsf{Int} \mid a \mid \sigma_1 \to \sigma_2 \mid \forall a.\, \sigma \mid \widehat{\alpha}$
Monotypes	τ	$::=$	$\mathsf{Int} \mid a \mid \widehat{\alpha} \mid \tau_1 \to \tau_2$
Algorithmic Contexts	Γ, Δ, Θ	$::=$	$\bullet \mid \Gamma, a \mid \Gamma, \widehat{\alpha} \mid \Gamma, \widehat{\alpha} = \tau$
Complete Contexts	Ω	$::=$	$\bullet \mid \Omega, a \mid \Omega, \widehat{\alpha} = \tau$
Promotion Modes	$\otimes$	$::=$	$+ \mid -$

$\boxed{\Gamma \vdash^{\mathsf{sub}} \sigma_1 <: \sigma_2 \dashv \Delta}$ *(Subtyping)*

S-TVAR
$$\frac{}{\Gamma[a] \vdash^{\mathsf{sub}} a <: a \dashv \Gamma[a]}$$

S-INT
$$\frac{}{\Gamma \vdash^{\mathsf{sub}} \mathsf{Int} <: \mathsf{Int} \dashv \Gamma}$$

S-EVAR
$$\frac{}{\Gamma[\widehat{\alpha}] \vdash^{\mathsf{sub}} \widehat{\alpha} <: \widehat{\alpha} \dashv \Gamma[\widehat{\alpha}]}$$

S-ARROW
$$\frac{\Gamma \vdash^{\mathsf{sub}} \sigma_3 <: \sigma_1 \dashv \Theta \qquad \Theta \vdash^{\mathsf{sub}} [\Theta]\sigma_2 <: [\Theta]\sigma_4 \dashv \Delta}{\Gamma \vdash^{\mathsf{sub}} \sigma_1 \to \sigma_2 <: \sigma_3 \to \sigma_4 \dashv \Delta}$$

S-FORALLR
$$\frac{\Gamma, a \vdash^{\mathsf{sub}} \sigma_1 <: \sigma_2 \dashv \Delta, a, \Theta}{\Gamma \vdash^{\mathsf{sub}} \sigma_1 <: \forall a.\, \sigma_2 \dashv \Delta}$$

S-FORALLL
$$\frac{\Gamma, \blacktriangleright_{\widehat{\alpha}}, \widehat{\alpha} \vdash^{\mathsf{sub}} \sigma_1[a \mapsto \widehat{\alpha}] <: \sigma_2 \dashv \Delta, \blacktriangleright_{\widehat{\alpha}}, \Theta}{\Gamma \vdash^{\mathsf{sub}} \forall a.\, \sigma_1 <: \sigma_2 \dashv \Delta}$$

S-INSTL
$$\frac{\Gamma[\widehat{\alpha}] \vdash^{+}_{\widehat{\alpha}} \sigma \rightsquigarrow \tau \dashv \Delta[\widehat{\alpha}]}{\Gamma[\widehat{\alpha}] \vdash^{\mathsf{sub}} \widehat{\alpha} <: \sigma \dashv \Delta[\widehat{\alpha} = \tau]}$$

S-INSTR
$$\frac{\Gamma[\widehat{\alpha}] \vdash^{-}_{\widehat{\alpha}} \sigma \rightsquigarrow \tau \dashv \Delta[\widehat{\alpha}]}{\Gamma[\widehat{\alpha}] \vdash^{\mathsf{sub}} \sigma <: \widehat{\alpha} \dashv \Delta[\widehat{\alpha} = \tau]}$$

$\boxed{\Gamma \vdash^{\otimes}_{\widehat{\alpha}} \sigma \rightsquigarrow \tau \dashv \Delta}$ *(Polymorphic Promotion)*

P-PR-FORALLL
$$\frac{\Gamma[\widehat{\beta}, \widehat{\alpha}] \vdash^{-}_{\widehat{\alpha}} \sigma[a \mapsto \widehat{\beta}] \rightsquigarrow \tau \dashv \Delta}{\Gamma[\widehat{\alpha}] \vdash^{-}_{\widehat{\alpha}} \forall a.\, \sigma \rightsquigarrow \tau \dashv \Delta}$$

P-PR-FORALLR
$$\frac{\Gamma, a \vdash^{+}_{\widehat{\alpha}} \sigma \rightsquigarrow \tau \dashv \Delta, a}{\Gamma \vdash^{+}_{\widehat{\alpha}} \forall a.\, \sigma \rightsquigarrow \tau \dashv \Delta}$$

P-PR-ARROW
$$\frac{\Gamma \vdash^{-\otimes}_{\widehat{\alpha}} \sigma_1 \rightsquigarrow \tau_1 \dashv \Theta \qquad \Theta \vdash^{\otimes}_{\widehat{\alpha}} [\Theta]\sigma_2 \rightsquigarrow \tau_2 \dashv \Delta}{\Gamma \vdash^{\otimes}_{\widehat{\alpha}} \sigma_1 \to \sigma_2 \rightsquigarrow \tau_1 \to \tau_2 \dashv \Delta}$$

P-PR-MONO
$$\frac{\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta}{\Gamma \vdash^{\otimes}_{\widehat{\alpha}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta}$$

$\boxed{\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta}$ *(Promotion)*

PR-TVAR
$$\frac{}{\Gamma[a][\widehat{\alpha}] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} a \rightsquigarrow a \dashv \Gamma[a][\widehat{\alpha}]}$$

Figure 6.2: Types, contexts, subtyping and (polymorphic) promotion of the algorithmic system

of instantiation, we directly use polymorphic promotion to promote the possibly polymorphic type σ into a monotype τ. Specifically, rule s-instL uses polymorphic promotion under the covariant mode (+) and rule s-instR uses polymorphic promotion under the contravariant mode (−). If promotion succeeds, we can directly set $\widehat{\alpha}$ to τ.

Polymorphic promotion. The judgment $\Gamma \vdash^{\otimes}_{\widehat{\alpha}} \sigma \rightsquigarrow \tau \dashv \Delta$ reads that under the input context Γ, promoting σ under promotion mode $\otimes$ yields type τ, so that τ is well-formed in the prefix context of $\widehat{\alpha}$.

The only difference between these two promotion modes is how to promote polymorphic types. Under the contravariant mode (rule p-pr-forallL), a monotype would make the final type more polymorphic. Therefore, we replace the universal binder a with a fresh existential variable $\widehat{\alpha}$ and put it before $\widehat{\alpha}$. Otherwise, in rule p-pr-forallR, we put a in the context and promote σ. Notice that since a is added to the tail of the context, it is not in the scope of $\widehat{\alpha}$ and can actually never be used in σ or otherwise promotion would fails. This makes sense, as for a subtyping relation $\Gamma \vdash^{\mathsf{sub}} \widehat{\alpha} <: \forall a.\, \sigma$ to hold, a must not be used in σ. That means $\forall a.\, \sigma$ can only be types like $\forall a.\, \mathsf{Int}$ or $\forall a.\, \mathsf{Int} \to \mathsf{Int}$, in which case $\widehat{\alpha}$ can be promoted to Int or $\mathsf{Int} \to \mathsf{Int}$ respectively. In the conclusion of the rule, we discard a in the return context. Note that we can simplify the rule by directly requiring $a \notin \textsc{fv}\,(\sigma)$, as in rule p-pr-forallRR given below. This way we would not need to add a into the context and the rule would remain sound.

p-pr-forallRR

$$\frac{a \notin \textsc{fv}\,(\sigma) \qquad \Gamma \vdash^{+}_{\widehat{\alpha}} \sigma \rightsquigarrow \tau \dashv \Delta}{\Gamma \vdash^{+}_{\widehat{\alpha}} \forall a.\, \sigma \rightsquigarrow \tau \dashv \Delta}$$

Rule p-pr-arrow flips the mode for codomains, and uses the same mode for domains. When the type to be promoted is a monotype, rule p-pr-mono uses the promotion judgment ($\Vdash^{\mathsf{pr}}$) directly. Note that for a monotype the mode does not matter, so rule p-pr-mono applies in both modes.

Promotion. The promotion judgment is the same as before, and still only works for monotypes, except that now we have rule pr-tvar for type variables a. Note again that promotion is a partial operation, as it requires a to be the left of $\widehat{\alpha}$, since the order of variable matters.

6.3.3 Soundness and Completeness

The statement of soundness of promotion remains the same as before.

Theorem 6.5 (Soundness of Promotion). *If* $\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta$*, and* $\Delta = \Delta_1, \widehat{\alpha}, \Delta_2$*, then* $\Delta_1 \vdash^{\mathsf{wf}} \tau_2$*, and* $[\Delta]\tau_1 = [\Delta]\tau_2$.

Based on soundness of promotion, we prove that after polymorphic promotion, the promoted type is also well-formed under the prefix context of $\widehat{\alpha}$. Moreover, polymorphic promotion builds a subtyping relation according to the promotion mode: under the contravariant mode (-), the original type is a subtype of the promoted type; under the covariant mode (+), the promoted type is a subtype of the original type.

Theorem 6.6 (Soundness of Polymorphic Promotion). *If* $\Gamma \vdash^{\otimes}_{\widehat{\alpha}} \sigma \rightsquigarrow \tau \dashv \Delta$*, and* $\Delta = \Delta_1, \widehat{\alpha}, \Delta_2$*, then* $\Delta_1 \vdash^{\mathsf{wf}} \tau_2$*. Moreoever, given* $\Delta \longrightarrow \Omega$*,*

- *if* $\otimes = -$*, then* $[\Omega]\Gamma \vdash^{DK} [\Omega]\sigma <: [\Omega]\tau$*; and*

- *if* $\otimes = +$*, then* $[\Omega]\Gamma \vdash^{DK} [\Omega]\tau <: [\Omega]\sigma$.

With soundness of polymorphic promotion, next we show that the new subtyping judgment using polymorphic promotion instead of instantiation remains sound.

Theorem 6.7 (Soundness of Subtyping). *If* $\Gamma \vdash^{\mathsf{sub}} \sigma_1 <: \sigma_2 \dashv \Delta$*, and* $\Delta \longrightarrow \Omega$*, then* $[\Omega]\Gamma \vdash^{DK} [\Omega]\sigma_1 <: [\Omega]\sigma_2$.

Now we turn to completeness. The completeness of promotion is the same as before.

Theorem 6.8 (Completeness of Promotion). *Given* $\Gamma \longrightarrow \Omega$*, and* $\Gamma \vdash^{\mathsf{wf}} \widehat{\alpha}$*, and* $\Gamma \vdash^{\mathsf{wf}} \tau$*, and* $[\Gamma]\widehat{\alpha} = \widehat{\alpha}$*, and* $[\Gamma]\tau = \tau$*, if* $\widehat{\alpha} \notin \textsc{fv}(\tau)$*, there exist* τ_2*,* Δ *and* Ω' *such that* $\Gamma \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \tau \rightsquigarrow \tau_2 \dashv \Delta$.

Completeness of polymorphic promotion has two parts. If the existential variable appears on the left, then we promote the type under the covariant mode (+), or otherwise the contravariant mode (-). Moreover, it also requires $\widehat{\alpha} \notin \textsc{fv}(\sigma)$.

Theorem 6.9 (Completeness of Polymorphic Promotion). *Given* $\Gamma \longrightarrow \Omega$*, and* $\Gamma \vdash^{\mathsf{wf}} \widehat{\alpha}$*, and* $\Gamma \vdash^{\mathsf{wf}} \sigma$*, and* $[\Gamma]\widehat{\alpha} = \widehat{\alpha}$*, and* $[\Gamma]\tau = \sigma$*, and* $\widehat{\alpha} \notin \textsc{fv}(\sigma)$*,*

- *if* $[\Omega]\Gamma \vdash^{DK} [\Omega]\widehat{\alpha} <: [\Omega]\sigma$*, then there exist* τ*,* Δ *and* Ω' *such that* $\Gamma \vdash^{+}_{\widehat{\alpha}} \sigma \rightsquigarrow \tau \dashv \Delta$*; and*

- *if* $[\Omega]\Gamma \vdash^{DK} [\Omega]\sigma <: [\Omega]\widehat{\alpha}$*, then there exist* τ*,* Δ *and* Ω' *such that* $\Gamma \vdash^{-}_{\widehat{\alpha}} \sigma \rightsquigarrow \tau \dashv \Delta$.

Finally, we prove that our subtyping is complete. With this, we have proved our claim that the original instantiation relation in DK can be replaced by the polymorphic promotion process, as the subtyping algorithm using polymorphic promotion remains sound and complete.

Theorem 6.10 (Completeness of Subtyping). *Given* $\Gamma \longrightarrow \Omega$, *and* $\Gamma \vdash^{\mathsf{wf}} \sigma_1$, *and* $\Gamma \vdash^{\mathsf{wf}} \sigma_2$, *if* $[\Omega]\Gamma \vdash^{DK} [\Omega]\tau_1 <: [\Omega]\tau_2$, *there exist* Δ *and* Ω' *such that* $\Delta \longrightarrow \Omega'$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash^{\mathsf{sub}} [\Gamma]\sigma_1 <: [\Gamma]\sigma_2 \dashv \Delta$.

6.4 Discussion

This section discusses two extensions of promotion. The first extension explores dependent types, while the second extension considers gradual types.

6.4.1 Promoting Dependent Types

In Section 6.1.1 we mentioned the drawback of decomposing type constructs that it cannot be easily applied to more advanced types like dependent types. In this section, we discuss how we can apply promotion to dependent types.

Consider rule PR-PI given below that promotes a dependent type $\Pi\, a : \tau_1 .\, \tau_2$.

$$\frac{\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \tau_1 \rightsquigarrow \tau_3 \dashv \Theta \qquad \Theta, a \vdash^{\mathsf{pr}}_{\widehat{\alpha}} [\Theta]\tau_2 \rightsquigarrow \tau_4 \dashv \Delta, a}{\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \Pi\, a : \tau_1 .\, \tau_2 \to \mathsf{Int} \rightsquigarrow \Pi\, a : \tau_3 .\, \tau_4 \dashv \Delta}\ \textsc{pr-pi}$$

Here we first promote τ_1, returning τ_3. Then we add a into the context to promote τ_2. Finally, we return $\Pi\, a : \tau_3 .\, \tau_4$ and discard a in the output context.

Unfortunately, this design does not work. In particular, consider promoting $\Pi\, a : \widehat{\beta}.\, a$.

$$\frac{\dfrac{}{\widehat{\beta}, \widehat{\alpha} \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\beta} \rightsquigarrow \widehat{\beta} \dashv \widehat{\beta}, \widehat{\alpha}}\ \textsc{pr-evarL} \qquad \dfrac{}{\widehat{\beta}, \widehat{\alpha}, a \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} a \rightsquigarrow ???}\ \textsc{pr-evarL}}{\widehat{\beta}, \widehat{\alpha} \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \Pi\, a : \widehat{\beta}.\, a \to \mathsf{Int} \rightsquigarrow}\ \textsc{pr-pi}$$

We expect that the promotion would return $\Pi\, a : \widehat{\beta}.\, a$. However, after we add a into the context to promote a, rule PR-TVAR does not apply, as a is out of the scope of $\widehat{\alpha}$!

The issue can be fixed by changing rule PR-TVAR to rule PR-TVARR to not consider the order of type variables.

$$\frac{}{\Gamma \vdash^{\mathsf{pr}}_{\widehat{\alpha}} a \rightsquigarrow a \dashv \Gamma}\ \textsc{pr-tvarr}$$

Then, while promotion resolves the ordering of existential variables, since there is no constraint for type variables, it is not guaranteed anymore that the promoted type is well-formed in the prefix context of $\widehat{\alpha}$. Therefore, we need to adjust the rule of subtyping to check explicitly that the result is well-formed, i.e.,

$$\textsc{s-instLL}\quad \frac{\Gamma[\widehat{\alpha}] \vdash^{+}_{\widehat{\alpha}} \sigma \rightsquigarrow \tau \dashv \Delta_1, \widehat{\alpha}, \Delta_2 \qquad \Delta_1 \vdash^{\mathsf{wf}} \tau}{\Gamma[\widehat{\alpha}] \vdash^{\mathsf{sub}} \widehat{\alpha} <: \sigma \dashv \Delta_1, \widehat{\alpha} = \tau, \Delta_2}$$

$$\textsc{s-instRR}\quad \frac{\Gamma[\widehat{\alpha}] \vdash^{-}_{\widehat{\alpha}} \sigma \rightsquigarrow \tau \dashv \Delta_1, \widehat{\alpha}, \Delta_2 \qquad \Delta_1 \vdash^{\mathsf{wf}} \tau}{\Gamma[\widehat{\alpha}] \vdash^{\mathsf{sub}} \sigma <: \widehat{\alpha} \dashv \Delta_1, \widehat{\alpha} = \tau, \Delta_2}$$

Xie and Oliveira [2017] include a more detailed discussion and formalization of applying promotion to a dependently typed lambda calculus.

6.4.2 Promoting Gradual Types

We have shown that polymorphic promotion works for DK. A natural extension is to also apply polymorphic promotion to GPC (Chapter 4). Then the key is to show how to promote the unknown type. Since comparing with the unknown type does not impose any constraints, we can simply replace it with a fresh existential variable:

$$\textsc{p-pr-unknown}\quad \frac{}{\Gamma[\widehat{\alpha}] \vdash^{\otimes}_{\widehat{\alpha}} \star \rightsquigarrow \widehat{\beta} \dashv \Gamma[\widehat{\beta}, \widehat{\alpha}]}$$

For example, we have $\widehat{\alpha} \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \mathsf{Int} \to \star \rightsquigarrow \mathsf{Int} \to \widehat{\beta} \dashv \widehat{\beta}, \widehat{\alpha}$.

For the extended GPC which restores the dynamic guarantee (Chapter 5), we can replace the unknown type with a fresh gradual existential variables instead.

$$\textsc{p-pr-unknownG}\quad \frac{}{\Gamma[\widehat{\alpha}] \vdash^{\otimes}_{\widehat{\alpha}} \star \rightsquigarrow \widehat{\beta}_G \dashv \Gamma[\widehat{\beta}_G, \widehat{\alpha}]}$$

With these rules it would be possible to apply polymorphic promotion to GPC. Note this discussion is a sketch and we have not fully worked out the full algorithm yet.

7 Kind Inference for Datatypes

In recent years, languages like Haskell have seen a dramatic surge of new features that significantly extends the expressive power of their type systems. With these features, the challenge of *kind inference* for datatype declarations has presented itself and become a worthy research problem on its own.

In this chapter, we apply promotion to kind inference for datatypes. Inspired by previous research on type-inference, we offer declarative specifications for what datatype declarations should be accepted, both for Haskell98 and for a more advanced system we call PolyKinds, based on the extensions in modern Haskell, including a limited form of dependent types. We believe these formulations to be novel and without precedent, even for Haskell98. These specifications are complemented with implementable algorithmic versions. We study *soundness*, *completeness* and the existence of *principal kinds* in these systems, proving the properties where they hold. This work can serve as a guide both to language designers who wish to formalize their datatype declarations and also to implementors keen to have principled inference of principal types.

7.1 Introduction and Motivation

The global type-inference algorithms employed in modern functional languages such as Haskell, ML, and OCaml are derived from the Hindley-Milner type system (HM) [Damas and Milner 1982; Hindley 1969], with multiple extensions. Common extensions of HM include *higher-ranked polymorphism* [Odersky and Läufer 1996; Peyton Jones et al. 2007] and *type-inference for GADTs* [Peyton Jones et al. 2006], which have both been formally studied thoroughly.

Most research work for extensions of HM so far (including OL, DK, AP and GPC) has focused on forms of polymorphism, where type variables all have the same kind. In these systems, the type variables introduced by universal quantifiers and/or type declarations all stand for proper types (i.e., they have kind $\star$). In such a simplified setting, datatype declarations such as

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

pose no problem at all for type inference: with only one possible kind for a, there is nothing to infer.

However, real-world implementations for languages like Haskell support a non-trivial kind language, including kinds other than $\star$. Haskell98 accepts *higher-kinded polymorphism* [Jones 1995], enabling datatype declarations such as

```
data AppInt f = Mk (f Int)
```

The type of constructor Mk applies the type variable f to an argument Int. Accordingly, $AppInt\ Bool$ would not work, as the type $Bool\ Int$ (in the instantiated type of Mk) is invalid. Instead, we must write something like $AppInt\ \ Maybe$: the argument to $AppInt$ must be suitable for applying to Int. In Haskell98, $AppInt$ has kind $(\star \to \star) \to \star$. For Haskell98-style higher-kinded polymorphism, Jones [1995] presents one of the few extensions of HM that deals with a non-trivial language of kinds. His work addresses the related problem of inference for *constructor type classes*, although he does not show directly how to do inference for datatype declarations.

Modern Haskell[1] has a much richer type and kind language compared to Haskell98. In recent years, Haskell has seen a dramatic surge of new features that extend the expressive power of algebraic datatypes. Such features include *GADTs*, *kind polymorphism* [Yorgey et al. 2012] with *implicit kind arguments*, and *dependent kinds* [Weirich et al. 2013], among others. With great power comes great responsibility: now we must be able to infer these kinds, too. For instance, consider these datatype declarations:

```
data App f a = MkApp (f a)
data Fix f   = In (f (Fix f))

data T = MkT1 (App Maybe Int)
       | MkT2 (App Fix Maybe)    -- accept or reject?
```

Should the declaration for T be accepted or rejected? In a Haskell98 setting, the kind of App is $(\star \to \star) \to \star \to \star$. Therefore T should be rejected, because in $MkT2$ the datatype App is applied to $Fix :: (\star \to \star) \to \star$ and $Maybe :: \star \to \star$, which do not match the expected kinds of App. However, with kind polymorphism, T is accepted, because App has the more general kind $\forall k.\, (k \to \star) \to k \to \star$. With this kind, both uses of App in T are valid.

The questions we ask in this section are these: *Which datatype declarations should be accepted? What kinds do accepted datatypes have?* Surprisingly, the literature is essentially silent

[1] We consider the Glasgow Haskell Compiler's implementation of Haskell, in version 8.8.

on these questions—we are unaware of any formal treatment of kind inference for datatype declarations.

Inspired by previous research on type inference, we offer declarative specifications for two languages: Haskell98, as standardized [Peyton Jones 2003] (Section 7.3); and PolyKinds, a significant fragment of modern Haskell (Section 7.6). These specifications are complemented with algorithmic versions that can guide implementations (Sections 7.4 and 7.7). To relate the declarative and algorithmic formulations we study various properties, including *soundness*, *completeness*, and the existence of *principal kinds* (Sections 7.4.7, 7.5, and 7.7.6).

7.2 Overview

This section gives an overview of this work. We start by contrasting kind inference with type inference, and then summarize the key aspects of the two systems of datatypes that we develop.

7.2.1 Kind Inference in Haskell98

Haskell98's kind language contains a constant (the kind $\star$) and kinds built from arrows ($k_1 \to k_2$). Kind inference for Haskell98 datatypes is thus closely related to type inference for the simply typed λ-calculus (STLC). For example, consider a term $+ :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ and a type constructor $(:+:) :: \star \to \star \to \star$. At the term level, we infer that $\mathit{add}\ a\ b = a + b$ yields $\mathit{add} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}$. Similarly, we can create a datatype

$$\textbf{data}\ \mathit{Add}\ a\ b = \mathit{Add}\ (a :+: b)$$

and infer $\mathit{Add} :: \star \to \star \to \star$.

No principal types. Consider now the function definition $k\ a = 1$. In the STLC, there are infinitely many (incomparable) types that can be assigned to k, including $k :: \mathit{Int} \to \mathit{Int}$ and $k :: (\mathit{Int} \to \mathit{Int}) \to \mathit{Int}$. Assuming that there are no type variables, the STLC accordingly has no *principal types*. An analogous datatype declaration is

$$\textbf{data}\ K\ a = K\ \mathit{Int}$$

As with k, there are infinitely many (incomparable) kinds that can be assigned to K, including $K :: \star \to \star$ and $K :: (\star \to \star) \to \star$.

Defaulting. Definitions like k (in STLC) or K (in Haskell98) do not have a principal type/kind, which raises the immediate question of what type/kind to infer. Haskell98 solves this problem by using a *defaulting* strategy: *if the kind of a type variable cannot be inferred, then it is defaulted to* $\star$. Therefore the kind of K in Haskell98 is $\star \to \star$. From the perspective of type inference, such defaulting strategy may seem somewhat ad-hoc, but due to the role that $\star$ plays at the type level it seems a defensible design for kind inference. Defaulting brings complications in writing a declarative specification. We discuss this point further in Section 7.4.3.

7.2.2 Kind Inference in Modern GHC Haskell

The type and kind languages for modern GHC are *unified* (i.e., types and kinds are indistinguishable), *dependently typed*, and the kind system includes the $\star :: \star$ axiom Cardelli [1986]; Weirich et al. [2013]. We informally use the word *type* or *kind* where we find it appropriate. Unlike Haskell98's datatypes, whose inference problem is quite closely related to the well-studied inference problem for STLC, type inference for various features in modern Haskell is not well-studied. While we are motivated concretely by Haskell, many of the challenges we face would be present in any dependently typed language seeking principled type inference. We use the term PolyKinds to refer to the fragment of modern Haskell that we model.[2] We enumerate the key features of this fragment below.

Kind polymorphism and dependent types Global type inference, in the style of Damas and Milner [1982], allows polymorphic kinds to be assigned to datatype definitions. For instance, reconsider

$$\textbf{data}\ K\ a = K\ Int$$

In PolyKinds, K can be given the kind $K :: \forall\{k\}.\, k \to \star$. This example shows one of the interesting new features of PolyKinds over Haskell98: *kind polymorphism* [Yorgey et al. 2012]. The polymorphic kind is obtained via *generalization*, which is a standard feature in Damas-Milner algorithms. Polymorphic types are helpful for recovering principal types, since they generalize many otherwise incomparable monomorphic types.

System-F-based languages do not have dependent types. In contrast, PolyKinds supports dependent kinds such as

$$\textbf{data}\ D :: \forall (k :: \star)\ (a :: k).\, K\ a \to \star$$

[2] Some of the features we model are slightly different in our presentation than they exist in GHC. Xie et al. [2019b] outlines the differences. These minor differences do not affect the applicability of our work to improving the GHC implementations, but they may affect the ability to test our examples in GHC.

There are two noteworthy aspects about the kind of D. Firstly, kind and type variables are *typed*: different type variables may have different kinds. Secondly, the kinds of later variables can *depend* on earlier ones. In D, the kind of a depends on k. Both typed variables and dependent kinds bring technical complications that do not exist in many previous studies of type inference (e.g., Peyton Jones et al. [2007]; Vytiniotis et al. [2011]).

First-order unification with dependent kinds and typed variables. Although PolyKinds is dependently typed, its unification problem is remarkably *first-order*. This is in contrast to many other dependently typed languages, where unification is usually *higher-order* [Andrews 1971; Huet 1973]. Since unification plays a central role in inference algorithms this is a crucial difference. Higher-order unification is well-known to be undecidable in the general case [Goldfarb 1981]. As a consequence, type-inference algorithms for most dependently typed languages make various trade-offs.

A key reason why unification can be kept as a first-order problem in PolyKinds is because the type language *does not include lambdas*. Type-level lambdas have been avoided since the start in Haskell, since they bring major challenges for (term-level) type inference [Jones 1995].

The unification problem for PolyKinds is still challenging, compared to unification for System-F-like languages: unification must be *kind-directed*, as first observed at the term level by Jones [1995]. Consider the following (contrived) example:

$$\begin{array}{ll} \textbf{data}\ X :: \forall a\ (b :: \star \to \star).\ a\ b \to \star & \text{-- accepted} \\ \textbf{data}\ Y :: \forall (c :: \mathit{Maybe}\ \mathit{Bool}).\ X\ c \to \star & \text{-- rejected} \end{array}$$

In X's kind, we discover $a :: (\star \to \star) \to \star$. When checking Y's kind, we must infer how to instantiate X: that is, we must choose a and b so that $a\ b$ unifies with $\mathit{Maybe}\ \mathit{Bool}$, which is c's kind. It is tempting to solve this with $a \mapsto \mathit{Maybe}$ and $b \mapsto \mathit{Bool}$, but doing so would be ill-kinded, as a and Maybe have different kinds. Our unification thus features *heterogeneous constraints* Gundry [2013]. When solving a unification variable, we need to first unify the kinds on both sides.

Because unification recurs into kinds, and because types are undifferentiated from kinds, it might seem that unification might not terminate. In Section 7.7.4 we show that the first-order unification with heterogeneous constraints employed in PolyKinds is guaranteed to terminate.

Mutual and polymorphic recursion Recursion and mutual recursion are omnipresent in datatype declarations. In PolyKinds, mutually recursive definitions will be kinded together and then get generalized together. For example, both P and Q get kind $\forall (k :: \star).\ k \to \star$.

$$\textbf{data}\ P\ a = MkP\ (Q\ a)$$
$$\textbf{data}\ Q\ a = MkQ\ (P\ a)$$

The recursion is simple here: all recursive occurrences are at the same type. In existing type-inference algorithms, such recursive definitions are well understood and do not bring considerable complexity to type inference. However, we must also consider *polymorphic recursion* as in $Poly$:

$$\textbf{data}\ Poly :: \forall k.\ k \rightarrow \star$$
$$\textbf{data}\ Poly\ k = C1\ (Poly\ Int) \mid C2\ (Poly\ Maybe)$$

This example includes a *kind signature*, meaning that we must *check* the kind of the datatype, not *infer* it. In the definition of $Poly$, the type $Poly\ Int$ requires an instantiation $k \mapsto \star$, while the type $Poly\ Maybe$ requires an instantiation of $k \mapsto (\star \rightarrow \star)$. These differing instantiations mean that the declaration employs polymorphic recursion.

PolyKinds deals with such cases of polymorphic recursion, which also appear at the term level—for example, when writing recursive functions over GADTs or nested datatypes [Bird and Meertens 1998]. Polymorphic recursion is known to render type-inference undecidable [Henglein 1993]. Furthermore, most existing formalizations of type inference avoid the question entirely, either by not modeling recursion at all or not allowing polymorphic recursion. Our PolyKinds system has full support for polymorphic recursion, implemented directly without the use of a fix operator. Polymorphic recursion is allowed only on datatypes with a kind signature; other datatypes are treated as monomorphic during inference.

Visible kind application PolyKinds lifts *visible type application (VTA)* [Eisenberg et al. 2016], whereby we can explicitly instantiate a function call, as in $id\ @Bool\ True$, to kinds, giving us *visible kind application (VKA)*. Following the design of VTA, we distinguish *specified variables* from *inferred variables*. As described by Eisenberg et al. [2016, Section 3.1], only specified variables can be instantiated via VKA. Instantiation of variables is inferred when no explicit kind application is given. To illustrate, consider

$$\textbf{data}\ T :: \forall a\ b.\ a\ b \rightarrow \star$$

Here, a and b are specified variables. Because their order is given, explicit instantiation of a must happen before b. For example, $T\ @Maybe$ instantiates a to $Maybe$. On the other hand, the kind of a and b can be generalized to $a :: k \rightarrow \star$ and $b :: k$. Elaborating the kind of T, we write $T :: \forall \{k :: \star\}\ (a :: k \rightarrow \star)\ (b :: k).\ a\ b \rightarrow \star$. The variable k is *inferred* and is not available for instantiation with VKA. This split between specified and inferred variables supports predictable type inference: if the variables generated by the compiler (e.g., k) were available for instantiation, then we have no way of knowing what order to instantiate them.

Open kind signatures and generalization order Echoing the design of Haskell, PolyKinds supports *open kind signatures*. We say a signature is *closed* if it contains no free variables, e.g.,

$$\mathbf{data}\ T :: \forall a.\ a \to \star$$

Otherwise, it is *open*, e.g.,

$$\mathbf{data}\ Q :: \forall (a :: (f\ b))\ (c :: k).\ f\ c \to \star$$

Free variables (in this case, f, b, k) will be generalized over. We have a decision to make: in which order do we generalize the free variables? This question is non-trivial, as there can be dependency between the variables. We infer $k :: \star$, $f :: k \to \star$, $b :: k$. Even though f and b appear before k, their kinds end up depending on k and we must quantify k before f and b. Inferring this order is a challenge: we cannot know the correct order before completing inference. We thus introduce *local scopes*, which are sets of variables that may be reordered. Since the ordering is not fixed by the programmer, these variables are considered *inferred*, not *specified*, with respect to VKA.

Existential quantification. PolyKinds supports existentially quantified variables on datatype constructors. This is useful, for example, to model GADTs. Given

$$\mathbf{data}\ T1 = \forall a.\ MkT1\ a$$

we get $MkT1 :: \forall (a :: \star).\ a \to T1$. The type of the data constructor declaration can also be generalized. Given

$$\mathbf{data}\ P1 :: \forall (a :: \star).\ \star$$

from $\mathbf{data}\ T2 = MkT2\ P1$, we infer $MkT2 :: \forall \{a :: \star\}.\ P1\ @a \to T2$, where $P1$ is elaborated to $P1\ @a$ with a generalized as an inferred variable.

7.2.3 Desirable Properties for Kind Inference

The goal of this work is to provide concrete, principled guidance to implementors of dependently typed languages, such as GHC/Haskell. It is thus important to be able to describe our inference algorithm as sound and complete against a *declarative specification*. This declarative specification is what we might imagine a programmer to have in her head as she programs. This system should be designed with a minimum of low-level detail and a minimum of surprises. It is then up to an algorithm to live up to the expectations set by the specification.

The algorithm is sound when all programs it accepts are also accepted by the specification; it is complete when all programs accepted by the specification are accepted by the algorithm.

Why choose the particular set of features described here? Because they lead to interesting kind inference challenges. We have found that the features above are sufficient in exploring kind inference in modern Haskell. We consider unformalized extensions in Section 7.8.

7.3 Datatypes in Haskell98

We begin our formal presentation with Haskell98. The fragment of the syntax of Haskell98 that concerns us appears at the top of Figure 7.1, including datatype declarations, types, kinds, and contexts. The metavariable e refers to expressions, but we do not elaborate the details of expressions' syntax or typing rules here. A program pgm is a sequence of groups (defined below) of datatype declarations $\mathcal{T}$, followed by an expression e. We write $\tau_1 \rightarrow \tau_2$ as an abbreviation for $(\rightarrow)\tau_1\ \tau_2$.

7.3.1 Groups and Dependency Analysis

Users are free to write declarations in any order: earlier declarations can depend on later ones in the same compilation unit. However, any kind-checking algorithm must process the declarations in dependency order. Complicating this is that type declarations may be mutually recursive. A formal analysis of this dependency analysis is not enlightening, so we consider it to be a preprocessing step that produces the grammar in Figure 7.1. This dependency analysis breaks up the (unordered) raw input into mutually recursive groups (potentially containing just one declaration), and puts these in dependency order. We use the term *group* to describe a set of mutually recursive declarations.

7.3.2 Declarative Typing Rules

The declarative typing rules are in Figure 7.1. There are no surprises here; we review these rules briefly. The top judgment is $\Sigma; \Psi \vdash^{\mathsf{pgm}} pgm : \sigma$. Its rule PGM-DT extends the input type context Σ with kinds for the datatype declarations to form Σ', which is used to check both the datatype declarations and the rest of the program. In rule PGM-DT, we implicitly extract the names $\overline{T}^{\,i}$ from the declarations $\overline{\mathcal{T}}^{\,i}$ (and use this abuse of notation throughout our work, relating T to $\mathcal{T}$ and D to $\mathcal{D}$). The kinds are *guessed* for an entire group all at once: they are added to the context *before* looking at the declarations. This is needed because the declarations in the group refer to one another. Guessing the right answer is typical of declarative type systems. The algorithmic system presented in Section 7.4 provides a mechanism

program	pgm	$::=$	$\mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i}\,; pgm \mid e$
datatype decl.	$\mathcal{T}$	$::=$	$\mathbf{data}\,T\,\overline{a_i}^{\,i} = \overline{\mathcal{D}_j}^{\,j}$
data c'tor decl.	$\mathcal{D}$	$::=$	$D\,\overline{\tau_i}^{\,i}$
expression	e	$::=$	$\ldots$
polytype	σ	$::=$	$\forall \overline{a_i : \kappa_i}^{\,i}.\,\tau$
monotype	τ	$::=$	$\mathsf{Int} \mid a \mid T \mid \tau_1\,\tau_2 \mid \rightarrow$
kind	κ	$::=$	$\star \mid \kappa_1 \rightarrow \kappa_2$
term context	Ψ	$::=$	$\bullet \mid \Psi, D : \sigma$
type context	Σ	$::=$	$\bullet \mid \Sigma, a : \kappa \mid \Sigma, T : \kappa$

$\boxed{\Sigma; \Psi \vdash^{\mathsf{pgm}} pgm : \sigma}$ *(Typing Program)*

$$\textsc{pgm-expr}\quad \frac{\Sigma; \Psi \vdash e : \sigma}{\Sigma; \Psi \vdash^{\mathsf{pgm}} e : \sigma}$$

$$\textsc{pgm-dt}\quad \frac{\Sigma' = \Sigma,\ \overline{T_i : \kappa_i}^{\,i} \qquad \overline{\Sigma' \vdash^{\mathsf{dt}} \mathcal{T}_i \rightsquigarrow \Psi_i}^{\,i} \qquad \Sigma'; \Psi,\ \overline{\Psi_i}^{\,i} \vdash^{\mathsf{pgm}} pgm : \sigma}{\Sigma; \Psi \vdash^{\mathsf{pgm}} \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i}\,; pgm : \sigma}$$

$\boxed{\Sigma \vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Psi}$ *(Typing Datatype Decl.)*

$$\textsc{dt-decl}\quad \frac{(T : \overline{\kappa_i}^{\,i} \rightarrow \star) \in \Sigma \qquad \overline{\Sigma,\ \overline{a_i : \kappa_i}^{\,i} \vdash^{\mathsf{dc}}_{T\,\overline{a_i}^{\,i}} \mathcal{D}_j \rightsquigarrow \tau_j}^{\,j}}{\Sigma \vdash^{\mathsf{dt}} \mathbf{data}\,T\,\overline{a_i}^{\,i} = \overline{\mathcal{D}_j}^{\,j} \rightsquigarrow \overline{D_j : \forall \overline{a_i : \kappa_i}^{\,i}.\,\tau_j}^{\,j}}$$

$\boxed{\Sigma \vdash^{\mathsf{dc}}_{\tau} \mathcal{D} \rightsquigarrow \tau'}$ *(Typing Data Constructor Decl.)*

$$\textsc{dc-decl}\quad \frac{\Sigma \vdash^{\mathsf{k}} \overline{\tau_i}^{\,i} \rightarrow \tau : \star}{\Sigma \vdash^{\mathsf{dc}}_{\tau} D\,\overline{\tau_i}^{\,i} \rightsquigarrow \overline{\tau_i}^{\,i} \rightarrow \tau}$$

$\boxed{\Sigma \vdash^{\mathsf{k}} \tau : \kappa}$ *(Kinding)*

$$\textsc{k-var}\quad \frac{(a : \kappa) \in \Sigma}{\Sigma \vdash^{\mathsf{k}} a : \kappa} \qquad \textsc{k-tcon}\quad \frac{(T : \kappa) \in \Sigma}{\Sigma \vdash^{\mathsf{k}} T : \kappa} \qquad \textsc{k-nat}\quad \frac{}{\Sigma \vdash^{\mathsf{k}} \mathsf{Int} : \star} \qquad \textsc{k-arrow}\quad \frac{}{\Sigma \vdash \rightarrow : \star \rightarrow \star \rightarrow \star}$$

$$\textsc{k-app}\quad \frac{\Sigma \vdash^{\mathsf{k}} \tau_2 : \kappa_1 \qquad \Sigma \vdash^{\mathsf{k}} \tau_1 : \kappa_1 \rightarrow \kappa_2}{\Sigma \vdash^{\mathsf{k}} \tau_1\,\tau_2 : \kappa_2}$$

Figure 7.1: Declarative specification of Haskell98 datatype declarations

for an implementation. Although there is no special judgment for typing a group of mutually recursive datatypes, we use $\Sigma \vdash^{\mathsf{grp}} \textbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\kappa_i}^{\,i}\,;\overline{\Psi_i}^{\,i}$ to denote that the kinding results of datatype declarations are $\overline{\kappa_i}^{\,i}$, and the output term contexts are $\overline{\Psi_i}^{\,i}$.

Declarations are checked with $\Sigma \vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Psi$. This uses the guessed kinds to process the data constructors of a declaration, producing a term context Ψ with the data constructors and their types. The rule DT-DECL ensures that the datatype has an appropriate kind in the context and then checks data constructors using the $\vdash^{\mathsf{dc}}$ judgment. These checks are done in a type context extended with bindings for the type variables $\overline{a_i}^{\,i}$, where each a_i has a kind extracted from the guessed kind of the datatype T. The subscript on the $\vdash^{\mathsf{dc}}$ judgment is the return type of the constructors, whose types are easily checked by rule DC-DECL. The kinding judgment $\Sigma \vdash^{\mathsf{k}} \tau : \kappa$ is standard.

7.4 Kind Inference for Haskell98

We now present the algorithmic system for Haskell98. Of particular interest is the defaulting rule (Section 7.4.3), which means that these rules are not complete with respect to the declarative system.

7.4.1 Syntax

The top of Figure 7.2 describes the syntax of kinds and contexts in the algorithmic system for Haskell98. The differences from the declarative system are highlighted in gray. Following Dunfield and Krishnaswami [2013], kinds are extended with unification kind variables $\widehat{\alpha}$. Algorithmic contexts are also extended with unification kind variables, either unsolved ($\widehat{\alpha}$) or solved ($\widehat{\alpha} = \kappa$). Although the grammar for algorithmic term contexts Γ appears identical to that of declarative contexts, note that the grammar for κ has been extended; accordingly, algorithmic contexts Γ might include kinds with unification variables, while declarative contexts Ψ do not.

7.4.2 Algorithmic Typing Rules

Figure 7.2 presents the typing rules for programs, datatype declarations and data constructor declarations. As this work focuses on the problem of kind inference of datatypes, we reduce the expression typing to the declarative system (rule A-PGM-EXPR); note the contexts used there are declarative. For type-checking a group of mutually recursive datatypes (rule A-PGM-DT), we first assign each type constructor a unification variable $\widehat{\alpha}$, and then type-check ($\Vdash^{\mathsf{dt}}$) each datatype definition (Section 7.4.4), producing the context Θ_{n+1}. Then we default (Sec-

kind	κ	$::=$	$\star \mid \kappa_1 \to \kappa_2 \mid \widehat{\alpha}$
term context	Γ	$::=$	$\bullet \mid \Gamma, D : \sigma$
type context	Δ, Θ	$::=$	$\bullet \mid \Delta, a : \kappa \mid \Delta, T : \kappa \mid \Delta, \widehat{\alpha} \mid \Delta, \widehat{\alpha} = \kappa$
complete type context	Ω	$::=$	$\bullet \mid \Omega, a : \kappa \mid \Omega, T : \kappa \mid \Omega, \widehat{\alpha} = \kappa$

$\boxed{\Omega; \Gamma \Vdash^{\mathsf{pgm}} pgm : \sigma}$ *(Typing Program)*

$$\text{A-PGM-EXPR}\quad \frac{[\Omega]\Omega; [\Omega]\Gamma \vdash e : \sigma}{\Omega; \Gamma \Vdash^{\mathsf{pgm}} e : \sigma}$$

$$\text{A-PGM-DT}\quad \frac{\begin{array}{c}\Theta_1 = \Omega, \overline{\widehat{\alpha}_i}^{\,i}, \overline{T_i : \widehat{\alpha}_i}^{\,i} \\ \overline{\Theta_i \Vdash^{\mathsf{dt}} \mathcal{T}_i \rightsquigarrow \Gamma_i \dashv \Theta_{i+1}}^{\,i} \qquad \Theta_{n+1} \longrightarrow\!\!\!\!\rightarrow \Omega' \qquad \Omega'; \Gamma, \overline{\Gamma_i}^{\,i} \Vdash^{\mathsf{pgm}} pgm : \sigma\end{array}}{\Omega; \Gamma \Vdash^{\mathsf{pgm}} \mathbf{rec}\, \overline{\mathcal{T}_i}^{\,i \in 1..n}; pgm : \sigma}$$

$\boxed{\Delta \Vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Gamma \dashv \Theta}$ *(Typing Datatype Decl.)*

$$\text{A-DT-DECL}\quad \frac{\begin{array}{c}(T : \kappa) \in \Delta \qquad \Delta, \overline{\widehat{\alpha}_i}^{\,i} \Vdash^{\mathsf{u}} [\Delta]\kappa \approx (\overline{\widehat{\alpha}_i}^{\,i} \to \star) \dashv \Theta_1, \overline{\widehat{\alpha}_i = \kappa_i}^{\,i} \\ \overline{\Theta_j, \overline{a_i : \kappa_i}^{\,i} \Vdash^{\mathsf{dc}}_{T\, \overline{a_i}^{\,i}} \mathcal{D}_j \rightsquigarrow \tau_j \dashv \Theta_{j+1}, \overline{a_i : \kappa_i}^{\,i}}^{\,j}\end{array}}{\Delta \Vdash^{\mathsf{dt}} \mathbf{data}\, T\, \overline{a_i}^{\,i} = \overline{\mathcal{D}_j}^{\,j \in 1..n} \rightsquigarrow \overline{D_j : \forall \overline{a_i : \kappa_i}^{\,i}. \tau_j}^{\,j} \dashv \Theta_{n+1}}$$

$\boxed{\Delta \Vdash^{\mathsf{dc}}_{\tau} \mathcal{D} \rightsquigarrow \tau' \dashv \Theta}$ *(Typing Data Constructor Decl.)*

$$\text{A-DC-DECL}\quad \frac{\Delta \Vdash^{\mathsf{k}} \overline{\tau_i}^{\,i} \to \tau : \star \dashv \Theta}{\Delta \Vdash^{\mathsf{dc}}_{\tau} D\, \overline{\tau_i}^{\,i} \rightsquigarrow \overline{\tau_i}^{\,i} \to \tau \dashv \Theta}$$

Figure 7.2: Algorithmic program typing in Haskell98

tion 7.4.3) all unsolved unification variables with $\star$ using $\Theta_{n+1} \longrightarrow\!\!\!\!\rightarrow \Omega$, and continue with the rest of the program. Defaulting here means that the constraints of one group do not propagate to the rest of the program; accordingly, the input context of $\Vdash^{\mathsf{pgm}}$ is always a complete context. Echoing the notation for the declarative system, we write $\Omega \Vdash^{\mathsf{grp}} \mathbf{rec}\ \overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\kappa_i}^{\,i} ; \overline{\Gamma_i}^{\,i} \dashv \Theta$ to denote that the results of type-checking a group of datatype declarations are the kinds $\overline{\kappa_i}^{\,i}$, the output term contexts $\overline{\Gamma_i}^{\,i}$, and the final output type context Θ.

7.4.3 Defaulting

One of the key properties of datatypes in Haskell98 is the *defaulting* rule. In a datatype definition, if a type parameter is not fully determined by the definitions in its mutually recursive group, it is defaulted to have kind $\star$.

Definition 18 (Defaulting, $\longrightarrow\!\!\!\!\rightarrow$)**.** An algorithmic context Δ is defaulted to a complete context Ω, written $\Delta \longrightarrow\!\!\!\!\rightarrow \Omega$ by replacing all unsolved unification variables $\widehat{\alpha}$ in Δ with $\widehat{\alpha} = \star$.

To understand how this rule affects code in practice, consider the following definitions:

```
data Q1 a = MkQ1          -- Q1 :: (⋆ → ⋆)
data Q2   = MkQ2 (Q1 Maybe)   -- rejected

data P1 a = MkP1 P2       -- P1 :: (⋆ → ⋆) → ⋆
data P2   = MkP2 (P1 Maybe)   -- accepted
```

One might think that the result of checking *Q1* and *Q2* would be the same as checking *P1* and *P2*. However, this is not true. *Q1* and *Q2* are not mutually recursive: they will not be in the same group and are checked separately. In contrast, *P1* and *P2* are mutually recursive and are checked together. This difference leads to the rejection of *Q2*: after kinding *Q1*, the parameter *a* is defaulted to $\star$, and then *Q1 Maybe* fails to kind check. Our algorithm is a faithful model of datatypes in Haskell98, and this rejection is exactly what the step $\Theta_{n+1} \longrightarrow\!\!\!\!\rightarrow \Omega$ (in rule A-PGM-DT) brings.

Other design alternatives. One alternative design is to default in rule A-PGM-EXPR instead of rule A-PGM-DT, as shown in rule A-PGM-EXPR-ALT. This means constraints in one group propagate to other groups, but not to expressions. Then *Q2* above is accepted.

A-PGM-EXPR-ALT

$$\frac{\Delta \longrightarrow\!\!\!\!\rightarrow \Omega \qquad [\Omega]\Omega; [\Omega]\Gamma \vdash e : \sigma}{\Delta; \Gamma \Vdash^{\mathsf{pgm}} e : \sigma}$$

A second alternative is that defaulting happens at the very end of type-checking a compilation unit. In this scenario, we wait to commit to the kind of a datatype until checking expressions. Now we can accept the following program, which would otherwise be rejected. However, this strategy does not play along well with modular design, as it takes an extra action at a module boundary.

```
data Q1 a = MkQ1
mkQ1      = MkQ1 :: Q1 Maybe
```

In the rest of this section, we stay with the standard, doing defaulting as portrayed in Figure 7.2.

7.4.4 Checking Datatype Declarations

The judgment $\Delta \Vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Gamma \dashv \Theta$ checks the datatype declaration $\mathcal{T}$ under the input context Δ, returning a term context Γ and an output context Θ. Its rule A-DT-DECL first gets the kind κ of the the type constructor from the context. It then assigns a fresh unification variable $\widehat{\alpha}$ to each type parameter. The expected kind of the type constructor is $\overline{\widehat{\alpha}_i}^{\,i} \to \star$. The rule then unifies κ with $\overline{\widehat{\alpha}_i}^{\,i} \to \star$. Before unification, we apply the context to κ; unification (Section 7.4.6) requires its inputs to be inert with respect to the context substitution. Our implementation of unification guarantees that all the $\widehat{\alpha}_i$ will be solved, as reflected in the rule A-DT-DECL. The type parameters are added to the context to type check each data constructor. Checking the data constructor $\mathcal{D}_j$ returns its type τ_j and the context $\Theta_{j+1}, \overline{a_i : \widehat{\alpha}_i}^{\,i}$. Note that each output context must be of this form as no new entries are added to the end of the context during checking individual data constructors. We can then generalize the type τ_j over type parameters, returning Θ_{n+1} as the result context.

The data constructor declaration judgment $\Delta \Vdash^{\mathsf{dc}}_{\tau} \mathcal{D} \rightsquigarrow \tau' \dashv \Theta$ type-checks a data constructor, by simply checking that the expected type $\overline{\tau_i}^{\,i} \to \tau$ is well-kinded.

7.4.5 Kinding

The algorithmic kinding $\Delta \Vdash^{\mathsf{k}} \tau : \kappa \dashv \Theta$ is given in Figure 7.3. Most rules are self-explanatory. For applications (rule A-K-APP), we synthesize the type for an application $\tau_1\ \tau_2$, where τ_1 and τ_2 have kinds κ_1 and κ_2, respectively. The hard work is delegated to the *application kinding* judgment.

Application kinding $\Delta \Vdash^{\mathsf{kapp}} \kappa_1 \bullet \kappa_2 : \kappa \dashv \Theta$ says that, under the context Δ, applying an expression of kind κ_1 to an argument of kind κ_2 returns the result kind κ and an output context Θ. We require the invariants that $[\Delta]\kappa_1 = \kappa_1$ and $[\Delta]\kappa_2 = \kappa_2$. Therefore, if the kind

$\boxed{\Delta \Vdash^{k} \tau : \kappa \dashv \Theta}$ *(Kinding)*

$$\textsc{a-k-arrow}\quad \frac{}{\Delta \Vdash^{k} \rightarrow : \star \rightarrow \star \rightarrow \star \dashv \Delta}$$

$$\textsc{a-k-tcon}\quad \frac{(T : \kappa) \in \Delta}{\Delta \Vdash^{k} T : \kappa \dashv \Delta}$$

$$\textsc{a-k-nat}\quad \frac{}{\Delta \Vdash^{k} \mathsf{Int} : \star \dashv \Delta}$$

$$\textsc{a-k-var}\quad \frac{(a : \kappa) \in \Delta}{\Delta \Vdash^{k} a : \kappa \dashv \Delta}$$

$$\textsc{a-k-app}\quad \frac{\Delta \Vdash^{k} \tau_1 : \kappa_1 \dashv \Theta_1 \qquad \Theta_1 \Vdash^{k} \tau_2 : \kappa_2 \dashv \Theta_2 \qquad \Theta_2 \Vdash^{\mathsf{kapp}} [\Theta_2]\kappa_1 \bullet [\Theta_2]\kappa_2 : \kappa_3 \dashv \Theta}{\Delta \Vdash^{k} \tau_1\,\tau_2 : \kappa_3 \dashv \Theta}$$

$\boxed{\Delta \Vdash^{\mathsf{kapp}} \kappa_1 \bullet \kappa_2 : \kappa \dashv \Theta}$ *(Application Kinding)*

$$\textsc{a-kapp-kuvar}\quad \frac{\Delta[\widehat{\alpha}_1, \widehat{\alpha}_2, \widehat{\alpha} = \widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2] \Vdash^{\mathsf{u}} \widehat{\alpha}_1 \approx \kappa \dashv \Theta}{\Delta[\widehat{\alpha}] \Vdash^{\mathsf{kapp}} \widehat{\alpha} \bullet \kappa : \widehat{\alpha}_2 \dashv \Theta}$$

$$\textsc{a-kapp-arrow}\quad \frac{\Delta \Vdash^{\mathsf{u}} \kappa_1 \approx \kappa \dashv \Theta}{\Delta \Vdash^{\mathsf{kapp}} \kappa_1 \rightarrow \kappa_2 \bullet \kappa : \kappa_2 \dashv \Theta}$$

$\boxed{\Delta \Vdash^{\mathsf{u}} \kappa_1 \approx \kappa_2 \dashv \Theta}$ *(Kind Unification)*

$$\textsc{a-u-refl}\quad \frac{}{\Delta \Vdash^{\mathsf{u}} \kappa \approx \kappa \dashv \Delta}$$

$$\textsc{a-u-arrow}\quad \frac{\Delta \Vdash^{\mathsf{u}} \kappa_1 \approx \kappa_3 \dashv \Theta_1 \qquad \Theta_1 \Vdash^{\mathsf{u}} [\Theta_1]\kappa_2 \approx [\Theta_1]\kappa_4 \dashv \Theta}{\Delta \Vdash^{\mathsf{u}} \kappa_1 \rightarrow \kappa_2 \approx \kappa_3 \rightarrow \kappa_4 \dashv \Theta}$$

$$\textsc{a-u-kvarL}\quad \frac{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \kappa \rightsquigarrow \kappa_2 \dashv \Theta[\widehat{\alpha}]}{\Delta[\widehat{\alpha}] \Vdash^{\mathsf{u}} \widehat{\alpha} \approx \kappa \dashv \Theta[\widehat{\alpha} = \kappa_2]}$$

$$\textsc{a-u-kvarR}\quad \frac{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \kappa \rightsquigarrow \kappa_2 \dashv \Theta[\widehat{\alpha}]}{\Delta[\widehat{\alpha}] \Vdash^{\mathsf{u}} \kappa \approx \widehat{\alpha} \dashv \Theta[\widehat{\alpha} = \kappa_2]}$$

$\boxed{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \kappa_1 \rightsquigarrow \kappa_2 \dashv \Theta}$ *(Promotion)*

$$\textsc{a-pr-star}\quad \frac{}{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \star \rightsquigarrow \star \dashv \Delta}$$

$$\textsc{a-pr-arrow}\quad \frac{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \kappa_1 \rightsquigarrow \kappa_3 \dashv \Delta_1 \qquad \Delta_1 \vdash^{\mathsf{pr}}_{\widehat{\alpha}} [\Delta_1]\kappa_2 \rightsquigarrow \kappa_4 \dashv \Theta}{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \kappa_1 \rightarrow \kappa_2 \rightsquigarrow \kappa_3 \rightarrow \kappa_4 \dashv \Theta}$$

$$\textsc{a-pr-kuvarL}\quad \frac{}{\Delta[\widehat{\beta}][\widehat{\alpha}] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\beta} \rightsquigarrow \widehat{\beta} \dashv \Delta[\widehat{\beta}][\widehat{\alpha}]}$$

$$\textsc{a-pr-kuvarR}\quad \frac{}{\Delta[\widehat{\alpha}][\widehat{\beta}] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\beta} \rightsquigarrow \widehat{\beta}_1 \dashv \Delta[\widehat{\beta}_1, \widehat{\alpha}][\widehat{\beta} = \widehat{\beta}_1]}$$

Figure 7.3: Algorithmic kinding, unification and promotion in Haskell98.

is a unification variable $\widehat{\alpha}$ (rule A-KAPP-KUVAR), we know it must be an unsolved unification variable. Since we know κ_1 must be a function kind, we solve $\widehat{\alpha}$ using $\widehat{\alpha}_1 \to \widehat{\alpha}_2$, unify $\widehat{\alpha}_1$ with the argument kind κ, and return $\widehat{\alpha}_2$. Note that the unification variables $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ are inserted in the *middle* of the context Δ; this allows us to remove the type variables from the end of the context in rule A-DT-DECL and also plays a critical role in maintaining unification variable scoping in the more complicated system we analyze later. If the kind of the function is not a unification variable, it must surely be a function kind $\kappa_1 \to \kappa_2$ (rule A-KAPP-ARROW), so we unify κ_1 with the known argument kind κ, returning κ_2.

7.4.6 Unification

The unification judgment $\Delta \Vdash^{\mathsf{u}} \kappa_1 \approx \kappa_2 \dashv \Theta$ is given in Figure 7.3. The elaborate style of this judgment (with the promotion process $\Vdash^{\mathsf{pr}}$) is overkill for Haskell98, but this design sets us up well to understand unification in the presence of our PolyKinds system, later. We require the preconditions that $[\Delta]\kappa_1 = \kappa_1$ and $[\Delta]\kappa_2 = \kappa_2$, so that every time we encounter a unification variable, we know it is unsolved. Rule A-U-REFL is our base case, and rule A-U-ARROW unifies the components of the arrow types. When unifying $\widehat{\alpha} \approx \kappa$ (rule A-U-KVARL), we cannot simply set $\widehat{\alpha}$ to κ, as κ might include variables bound to the *right* of $\widehat{\alpha}$. Instead, we need to *promote* ($\Vdash^{\mathsf{pr}}$) κ. Rule A-U-KVARL first promotes the kind κ, yielding κ_2, so that κ_2 is well-formed in the prefix context of $\widehat{\alpha}$. We can then set $\widehat{\alpha} = \kappa_2$ in the concluding context. Rule A-U-KVARR is symmetric to rule A-U-KVARL.

Promotion. As described in Chapter 6, the crucial observation of $\Vdash^{\mathsf{pr}}$ is that *the relative order between unification variables does not matter for solving a constraint.* The promotion judgment $\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \kappa_1 \rightsquigarrow \kappa_2 \dashv \Theta$ captures this observation. The judgment says that, under the context Δ, we promote the kind κ_1, yielding κ_2, so that κ_2 is well-formed in the prefix context of $\widehat{\alpha}$, while retaining $[\Theta]\kappa_1 = [\Theta]\kappa_2$. The promotion rules here are essentially the same as in Figure 6.1. Importantly, in rule A-PR-KUVARR, a unification variable $\widehat{\beta}$ bound to the right of $\widehat{\alpha}$ in Δ is replaced by a fresh variable introduced to $\widehat{\alpha}$'s left. It is this promotion algorithm that guarantees that all the $\widehat{\alpha}_i$ will be solved in rule A-DT-DECL: those variables will appear to the right of the unification variable invented in rule A-PGM-DT and will be promoted (and thus solved).

7.4.7 Soundness and Completeness

The main theorem of soundness is for program typing:

Theorem 7.1 (Soundness of $\Vdash^{\mathsf{pgm}}$). *If* Ω ok, *and* $\Omega \Vdash^{\mathsf{ectx}} \Gamma$, *and* $\Omega; \Gamma \Vdash^{\mathsf{pgm}} pgm : \sigma$, *then* $[\Omega]\Omega; [\Omega]\Gamma \vdash^{\mathsf{pgm}} pgm : \sigma$.

This lemma statement refers to judgments Ω ok and $\Omega \Vdash^{\mathsf{ectx}} \Gamma$; these basic well-formedness checks are standard. Because the declarative judgment $\vdash^{\mathsf{pgm}}$ requires declarative contexts, we write $[\Omega]\Omega$ and $[\Omega]\Gamma$ in the conclusion, applying the complete algorithmic context Ω as a substitution to form a declarative context, free of unification variables.

The statement of completeness relies on the definition of *context extension* $\Delta \longrightarrow \Theta$ [Dunfield and Krishnaswami 2013]. The judgment captures a process of *information increase*, and its definition is similar as in previous chapters. In all the algorithmic judgments, the output context is an extension of the input context.

We prove that our system is complete only up to checking *a group of datatype declarations*.

Theorem 7.2 (Completeness of $\Vdash^{\mathsf{grp}}$). *Given* Ω ok, *if* $[\Omega]\Omega \vdash^{\mathsf{grp}} \mathbf{rec}\ \overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\kappa_i}^{\,i}; \overline{\Psi_i}^{\,i}$, *then there exists* $\overline{\kappa_i'}^{\,i}$, $\overline{\Gamma_i}^{\,i}$, Θ, *and* Ω', *such that* $\Omega \Vdash^{\mathsf{grp}} \mathbf{rec}\ \overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\kappa_i'}^{\,i}; \overline{\Gamma_i}^{\,i} \dashv \Theta$, *where* $\Theta \longrightarrow \Omega'$, *and* $\overline{[\Omega']\kappa_i' = \kappa_i}^{\,i}$, *and* $\overline{\Psi_i = [\Omega']\Gamma_i}^{\,i}$.

The theorem statement uses the notational convenience for checking groups, defined in Section 7.3.2 and Section 7.4.2. The theorem states that for every possible declarative typing for a group, the algorithmic typing results can be extended to support the declarative typing.

Unfortunately, the typing program judgment $\vdash^{\mathsf{pgm}}$ is incomplete, as our algorithm models defaulting, while the declarative system does not. (For example, the *Q1*/*Q2* example of Section 7.4.3 is accepted by the declarative system but rejected by both GHC and our algorithmic system.) As straightforward as the defaulting rule may seem, it is surprisingly hard to model in a declarative system. We remedy this in the next section.

7.5 Type Parameters, Principal Kinds and Completeness in Haskell98

We have seen that our judgments for checking programs $\vdash^{\mathsf{pgm}}$ and $\Vdash^{\mathsf{pgm}}$ do not support completeness, because the declarative system cannot easily model the defaulting rule given in Section 7.4.3. In Chapter 5, we have seen that introducing type parameters [Garcia and Cimini 2015] helps resolve the dynamic gradual guarantee. Inspired by that, in this section, we introduce *kind parameters*, and relate the defaulting rule to principal kinds to recover completeness.

7.5.1 Type Parameters

Consider the datatype **data** *App f a* = *MkApp* (*f a*) again. The parameter *a* in this example can be of any kind, including $\star$, $\star \to \star$, or others. To express this polymorphism without introducing first-class polymorphism, we endow the declarative system with a set of *kind parameters*. Importantly, kind parameters live only in our reasoning; users are not allowed to write any kind parameters in the source. We amend the definition of kinds in Figure 7.1 as follows.

$$\begin{array}{llll} \text{kind parameter} & P & \in & \text{KParam} \\ \text{kind} & \kappa & ::= & \star \mid \kappa_1 \to \kappa_2 \mid \boxed{P} \end{array}$$

Kind parameters are uninterpreted kinds: there is no special treatment of kind parameters in the type system. Think of them as abstract, opaque kind constants. Kind parameters are eliminated by substitutions S, which map kind parameters to kinds, and homomorphically work on kinds themselves. For example, *App* can be assigned kind $(P \to \star) \to P \to \star$. By substituting for P, we can get, for example, $(\star \to \star) \to \star \to \star$. Indeed, from $(P \to \star) \to P \to \star$ we can get all other possible kinds of *App*. This leads to the definition of *principal kinds* for a group; and to the property that for every well-formed group, there exists a list of principal kinds.

Definition 19 (Principal Kind in Haskell98 with Kind Parameters). Given a context Σ, a group $\mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i}$, and a list of kinds $\overline{\kappa_i}^{\,i}$, we say that the $\overline{\kappa_i}^{\,i}$ are *principal kinds* of Σ and $\mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i}$, denoted as $\Sigma \vdash \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \leadsto^{\mathsf{P}} \overline{\kappa_i}^{\,i}$, if $\Sigma \vdash^{\mathsf{grp}} \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \leadsto \overline{\kappa_i}^{\,i} ; \overline{\Psi_i}^{\,i}$, and whenever $\Sigma \vdash^{\mathsf{grp}} \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \leadsto \overline{\kappa'_i}^{\,i} ; \overline{\Psi'_i}^{\,i}$ holds, there exists some substitution S, such that $\overline{S(\kappa_i) = \kappa'_i}^{\,i}$ and $\overline{S(\Psi_i) = \Psi'_i}^{\,i}$.

Theorem 7.3 (Principality of Haskell98 with Kind Parameters). *If* $\Sigma \vdash^{\mathsf{grp}} \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \leadsto \overline{\kappa_i}^{\,i} ; \overline{\Psi_i}^{\,i}$, *then there exists some* $\overline{\kappa'_i}^{\,i}$ *such that* $\Sigma \vdash \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \leadsto^{\mathsf{P}} \overline{\kappa'_i}^{\,i}$.

7.5.2 Principal Kinds and Defaulting

Using the notion of kind parameters, we can now incorporate defaulting into the declarative specification of Haskell98. To this end, we define the defaulting kind parameter substitution $S^\star$:

Definition 20 (Defaulting Kind Parameter Substitution). Let $S^\star \in \text{KParam} \to \kappa$ denote the substitution that substitutes all kind parameters to $\star$.

Using $S^\star$, we can rewrite rule PGM-DT. Noteworthy is the fact that kind parameters only live in the middle of the derivation (in the κ_i), but never appear in the results $S^\star(\kappa_i)$.

PGM-DTP

$$\frac{\Sigma \vdash^{\mathsf{grp}} \mathbf{rec}\,\overline{T_i}^{\,i} \rightsquigarrow \overline{\kappa_i}^{\,i} ; \overline{\Psi_i}^{\,i} \qquad \Sigma \vdash \mathbf{rec}\,\overline{T_i}^{\,i} \rightsquigarrow^{\mathsf{p}} \overline{\kappa_i}^{\,i} \qquad \Sigma, \overline{T_i : S^\star(\kappa_i)}^{\,i}; \Psi, \overline{S^\star(\Psi_i)}^{\,i} \vdash^{\mathsf{pgm}} pgm : \sigma}{\Sigma; \Psi \vdash^{\mathsf{pgm}} \mathbf{rec}\,\overline{T_i}^{\,i}; pgm : \sigma}$$

7.5.3 Completeness

The two versions of defaulting (the one above and $\Delta \longrightarrow\!\!\!\!\!\!\rightarrow \Omega$ of Section 7.4.2) are equivalent. This fact is embodied in the following theorem, stating that the algorithmic system is complete with respect to the declarative system with kind parameters.

Theorem 7.4 (Completeness of $\Vdash^{\mathsf{pgm}}$ with Kind Parameters). *Given algorithmic contexts Ω, Γ, and a program pgm, if $[\Omega]\Omega; [\Omega]\Gamma \vdash^{\mathsf{pgm}} pgm : \sigma$, then $\Omega; \Gamma \Vdash^{\mathsf{pgm}} pgm : \sigma$.*

7.6 Declarative Syntax and Semantics of PolyKinds

Having set the stage for kind inference for datatypes in Haskell98, we now present the declarative PolyKinds system. Our syntax is given in Figure 7.7. Compared to Haskell98, programs *pgm* now include datatype signatures $\mathcal{S}$. Data constructor declarations $\mathcal{D}$ support existential quantification. Types and kinds are collapsed into one level; σ and K are now synonymous metavariables and allow prenex polymorphism, where variables in a kind binder ϕ can optionally have kind annotations. Monotypes τ and κ allow visible kind applications $\tau_1\ @\tau_2$. Elaborated types μ, η are the result of elaboration, which decorates source types to make them fully explicit. This is done so that checking equality of elaborated types is straightforward. The syntax for elaborated types contains inferred polymorphism $\forall\{\phi^{\mathsf{c}}\}.\mu$, where complete free kind binders ϕ^{c} have all variables annotated. Elaborated monotypes ρ and ω share the same syntax as monotypes. We informally use only ρ or ω for elaborated monotypes.

7.6.1 Groups and Dependency Analysis

Decomposition of signatures and definitions allows a more fine-grained control of dependency analysis. If T has a signature, and S depends on T, then we can kind-check S without inspecting the definition of T, because we know the kind of T. In other words, S only depends on the *signature* of T, not the *definition* of T. The complete dependency analysis rule, inspired by Jones [1999, Section 11.6.3], is:

Definition 21 (Dependency Analysis in PolyKinds).

program	pgm	$::=$	$\textbf{sig}\, \mathcal{S}; pgm \mid \textbf{rec}\, \overline{\mathcal{T}_i}^{\,i}; pgm \mid e$
datatype signature	$\mathcal{S}$	$::=$	$\textbf{data}\, T : \sigma$
datatype decl.	$\mathcal{T}$	$::=$	$\textbf{data}\, T\, \overline{a_i}^{\,i} = \overline{\mathcal{D}_j}^{\,j}$
data constructor decl.	$\mathcal{D}$	$::=$	$\forall \phi.\, D\, \overline{\tau_i}^{\,i}$
type, kind	σ, K	$::=$	$\forall \phi.\, \sigma \mid \tau$
monotype, monokind	$\tau, \kappa, \rho, \omega$	$::=$	$\star \mid \mathsf{Int} \mid a \mid T \mid \tau_1\, \tau_2 \mid \tau_1\, @\tau_2 \mid \rightarrow$
elaborated type, kind	μ, η	$::=$	$\forall \{\phi^{\mathsf{c}}\}.\mu \mid \forall \phi^{\mathsf{c}}.\, \mu \mid \rho$
term context	Ψ	$::=$	$\bullet \mid \Psi, D : \mu$
type context	Σ	$::=$	$\bullet \mid \Sigma, a : \rho \mid \Sigma, T : \eta$
kind binder list	ϕ	$::=$	$\bullet \mid \phi, a \mid \phi, a : \kappa$
complete kind binder list	ϕ^{c}	$::=$	$\bullet \mid \phi^{\mathsf{c}}, a : \rho$

Figure 7.4: Syntax of PolyKinds

(i) If the signature/definition of T_1 mentions T_2, then:

a) if T_2 has a signature, the signature/definition of T_1 depends on the signature of T_2;

b) otherwise, the signature/definition of T_1 depends on the definition of T_2.

(ii) A definition depends on its signature.

To avoid a type that mentions itself in its own kind, we disallow self-dependency or mutual dependency involving signatures. For example, a group

```
data T1 :: T2 a → ⋆
data T2 :: T1 → ⋆
```

is rejected, lest $T1$ be assigned type $\forall (a :: T1).\, T2\ a \rightarrow \star$. In other words, signatures do not form groups: they are always processed individually. Moreover, the definition of a datatype which has a signature does not join others in a group, as according to Definition 21, there will be no dependency from datatypes on it. This simplifies the kinding procedure, as we will see in the coming section.

The declarative typing rules appear in Figure 7.5. The judgment $\Sigma; \Psi \vdash^{\mathsf{pgm}} pgm : \sigma$ checks the program. From now on we omit the typing rule for expressions in programs, which is essentially the same as in Haskell98. Rule PGM-SIG processes kind signatures by elaborating and generalizing the kind, then adding it to the context Σ. The helper judgment $\Sigma \vdash^{\mathsf{sig}} \mathcal{S} \rightsquigarrow T : \eta$ checks a kind signature **data** $T : \sigma$. First, it uses $\rceil \sigma \lceil$ to ensure σ returns

$\boxed{\Sigma; \Psi \vdash^{\mathsf{pgm}} pgm : \sigma}$ *(Typing Program)*

PGM-SIG

$$\frac{\Sigma \vdash^{\mathsf{sig}} \mathcal{S} \rightsquigarrow T : \eta \qquad \Sigma, T : \eta; \Psi \vdash^{\mathsf{pgm}} pgm : \mu}{\Sigma; \Psi \vdash^{\mathsf{pgm}} \mathbf{sig}\, \mathcal{S}; pgm : \mu}$$

PGM-DT-TTS

$$\frac{(T : \eta) \in \Sigma \qquad \Sigma \vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Psi_1 \qquad \Sigma; \Psi, \Psi_1 \vdash^{\mathsf{pgm}} pgm : \mu}{\Sigma; \Psi \vdash^{\mathsf{pgm}} \mathbf{rec}\, \mathcal{T}; pgm : \mu}$$

PGM-DT-TT

$$\frac{\overline{\phi^{\mathsf{c}}_i \in \mathcal{Q}(\omega_i)}^{\,i} \qquad \overline{\frac{\overline{\Sigma,\ \phi^{\mathsf{c}}_i \vdash^{\mathsf{ela}} \omega_i : \star}^{\,i}}{\Sigma,\ \cup\overline{\phi^{\mathsf{c}}_i}^{\,i},\ \overline{T_i : \omega_i}^{\,i} \vdash^{\mathsf{dt}} \mathcal{T}_i \rightsquigarrow \Psi_i}}^{\,i} \qquad \overline{\Sigma,\ \cup\overline{\phi^{\mathsf{c}}_i}^{\,i},\ \overline{T_i : \omega_i}^{\,i} \vdash^{\mathsf{gen}}_{\phi^{\mathsf{c}}_i} \Psi_i \rightsquigarrow \Psi'_i}^{\,i} \\ \Sigma,\ \overline{T_i : \forall\{\phi^{\mathsf{c}}_i\}.\omega_i}^{\,i}; \Psi,\ \overline{\Psi'_i[\overline{T_i \mapsto T_i\ @\phi^{\mathsf{c}}_i}^{\,i}]}^{\,i} \vdash^{\mathsf{pgm}} pgm : \sigma}{\Sigma; \Psi \vdash^{\mathsf{pgm}} \mathbf{rec}\, \overline{\mathcal{T}_i}^{\,i}; pgm : \sigma}$$

$\boxed{\Sigma \vdash^{\mathsf{sig}} \mathcal{S} \rightsquigarrow T : \eta}$ *(Typing Signature)*

SIG-TT

$$\frac{\rceil\sigma\lceil \qquad \phi \in \mathcal{Q}(\sigma) \qquad \phi^{\mathsf{c}}_1 \in \mathcal{Q}(\forall\phi^{\mathsf{c}}.\ \eta) \qquad \Sigma, \phi^{\mathsf{c}}_1 \vdash^{\mathsf{k}} \forall\phi.\ \sigma : \star \rightsquigarrow \forall\phi^{\mathsf{c}}.\ \eta \qquad |\phi| = |\phi^{\mathsf{c}}|}{\Sigma \vdash^{\mathsf{sig}} \mathbf{data}\, T : \sigma \rightsquigarrow T : \forall\{\phi^{\mathsf{c}}_1\}.\forall\{\phi^{\mathsf{c}}\}.\eta}$$

$\boxed{\Sigma \vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Psi}$ *(Typing Datatype Decl.)*

DT-TT

$$\frac{(T : \forall\{\phi^{\mathsf{c}}_1\}.\forall\phi^{\mathsf{c}}_2.\ \overline{\omega_i}^{\,i} \rightarrow \star) \in \Sigma \qquad \overline{\Sigma, \phi^{\mathsf{c}}_1, \phi^{\mathsf{c}}_2,\ \overline{a_i : \omega_i}^{\,i} \vdash^{\mathsf{dc}}_{(T\ @\phi^{\mathsf{c}}_1\ @\phi^{\mathsf{c}}_2\ \overline{a_i}^{\,i})} \mathcal{D}_j \rightsquigarrow \mu_j}^{\,j}}{\Sigma \vdash^{\mathsf{dt}} \mathbf{data}\, T\, \overline{a_i}^{\,i} = \overline{\mathcal{D}_j}^{\,j} \rightsquigarrow \overline{D_j : \forall\{\phi^{\mathsf{c}}_1\}.\forall\phi^{\mathsf{c}}_2.\ \forall\overline{a_i : \omega_i}^{\,i}.\ \mu_j}^{\,j}}$$

$\boxed{\Sigma \vdash^{\mathsf{dc}}_{\rho} \mathcal{D} \rightsquigarrow \mu}$ *(Typing Data Constructor Decl.)*

DC-TT

$$\frac{\phi^{\mathsf{c}} \in \mathcal{Q}(\mu\backslash_{\Sigma,\overline{\tau_i}^{\,i}}) \\ \Sigma, \phi^{\mathsf{c}} \vdash^{\mathsf{k}} \forall\phi.\ \overline{\tau_i}^{\,i} \rightarrow \rho : \star \rightsquigarrow \mu}{\Sigma \vdash^{\mathsf{dc}}_{\rho} \forall\phi.\ D\,\overline{\tau_i}^{\,i} \rightsquigarrow \forall\{\phi^{\mathsf{c}}\}.\mu}$$

$\boxed{\Sigma \vdash^{\mathsf{gen}}_{\phi^{\mathsf{c}}} \Psi_1 \rightsquigarrow \Psi_2}$ *(Generalization)*

GEN

$$\frac{\overline{\phi^{\mathsf{c}}, \phi^{\mathsf{c}}_i \in \mathcal{Q}(\mu_i)}^{\,i}}{\Sigma \vdash^{\mathsf{gen}}_{\phi^{\mathsf{c}}} \overline{D_i : \mu_i}^{\,i} \rightsquigarrow \overline{D_i : \forall\{\phi^{\mathsf{c}}, \phi^{\mathsf{c}}_i\}.\mu_i}^{\,i}}$$

Figure 7.5: Declarative specification of PolyKinds

$\boxed{\Sigma \vdash^{\mathsf{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2}$ *(Instantiation)*

$$\textsc{Inst-Refl}\quad \frac{}{\Sigma \vdash^{\mathsf{inst}} \mu : \omega <: \omega \rightsquigarrow \mu}$$

$$\textsc{Inst-Forall}\quad \frac{\Sigma \vdash^{\mathsf{ela}} \rho : \omega_1 \qquad \Sigma \vdash^{\mathsf{inst}} \mu_1 \mathbin{@}\rho : \eta[a \mapsto \rho] <: \omega_2 \rightsquigarrow \mu_2}{\Sigma \vdash^{\mathsf{inst}} \mu_1 : \forall a : \omega_1 .\, \eta <: \omega_2 \rightsquigarrow \mu_2}$$

$\boxed{\Sigma \vdash^{\mathsf{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu}$ *(Kind Checking)*

$$\textsc{KC-Sub}\quad \frac{\Sigma \vdash^{\mathsf{k}} \sigma : \eta \rightsquigarrow \mu_1 \qquad \Sigma \vdash^{\mathsf{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2}{\Sigma \vdash^{\mathsf{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu_2}$$

$\boxed{\Sigma \vdash^{\mathsf{k}} \sigma : \eta \rightsquigarrow \mu}$ *(Kinding)*

$$\textsc{KTT-Star}\quad \frac{}{\Sigma \vdash^{\mathsf{k}} \star : \star \rightsquigarrow \star}$$

$$\textsc{KTT-App}\quad \frac{\Sigma \vdash^{\mathsf{k}} \tau_1 : \eta_1 \rightsquigarrow \rho_1 \qquad \Sigma \vdash^{\mathsf{inst}} \rho_1 : \eta_1 <: (\omega_1 \rightarrow \omega_2) \rightsquigarrow \rho_2 \qquad \Sigma \vdash^{\mathsf{kc}} \tau_2 \Leftarrow \omega_1 \rightsquigarrow \rho_3}{\Sigma \vdash^{\mathsf{k}} \tau_1\, \tau_2 : \omega_2 \rightsquigarrow \rho_2\, \rho_3}$$

$$\textsc{KTT-KApp}\quad \frac{\Sigma \vdash^{\mathsf{k}} \kappa_1 : \forall a : \omega .\, \eta \rightsquigarrow \rho_1 \qquad \Sigma \vdash^{\mathsf{kc}} \kappa_2 \Leftarrow \omega \rightsquigarrow \rho_2}{\Sigma \vdash^{\mathsf{k}} \kappa_1 \mathbin{@}\kappa_2 : \eta[a \mapsto \rho_2] \rightsquigarrow \rho_1 \mathbin{@}\rho_2}$$

$$\textsc{KTT-ForallI}\quad \frac{\Sigma \vdash^{\mathsf{ela}} \omega : \star \qquad \Sigma, a : \omega \vdash^{\mathsf{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu}{\Sigma \vdash^{\mathsf{k}} \forall a .\, \sigma : \star \rightsquigarrow \forall a : \omega .\, \mu}$$

$\boxed{\Sigma \vdash^{\mathsf{ela}} \mu : \eta}$ *(Elaborated Kinding)*

$$\textsc{Ela-App}\quad \frac{\Sigma \vdash^{\mathsf{ela}} \rho_1 : \omega_1 \rightarrow \omega_2 \qquad \Sigma \vdash^{\mathsf{ela}} \rho_2 : \omega_1}{\Sigma \vdash^{\mathsf{ela}} \rho_1\, \rho_2 : \omega_2}$$

$$\textsc{Ela-KApp}\quad \frac{\Sigma \vdash^{\mathsf{ela}} \rho_1 : \forall a : \omega .\, \eta \qquad \Sigma \vdash^{\mathsf{ela}} \rho_2 : \omega}{\Sigma \vdash^{\mathsf{ela}} \rho_1 \mathbin{@}\rho_2 : \eta[a \mapsto \rho_2]}$$

Figure 7.6: Selected rules for declarative kind-checking in PolyKinds

$\star$: $\rceil\sigma\lceil$ simply traverses over arrows and foralls, checking that the final kind of σ is $\star$. Then, as σ may be an open kind signature, it extracts the free kind variables $\phi \in \mathcal{Q}(\sigma)$, where $\mathcal{Q}(\sigma)$ is the set of all well-formed orderings of the free variables (transitively looking into variables' kinds) of σ; thus, ϕ is one such ordering. As discussed in Section 7.2.2, variables in ϕ are *inferred* so we accept any relative order, as long as it features the necessary dependency between the variables. Then the rule tries to elaborate ($\vdash^{\mathsf{k}}$) the kind $\forall\phi.\ \sigma$, where ϕ and ϕ^{c} have the same length ($|\phi| = |\phi^{\mathsf{c}}|$). As the elaborated result $\forall\phi^{\mathsf{c}}.\ \eta$ can be further generalized, we bring the free variables $\phi_1^{\mathsf{c}} \in \mathcal{Q}(\forall\phi^{\mathsf{c}}.\ \eta)$ into scope when elaborating. The concluding output is $T : \forall\{\phi_1^{\mathsf{c}}\}.\forall\{\phi^{\mathsf{c}}\}.\eta$. As an example, consider a kind signature $\forall a.\ b \to \star$. We have $\phi = b$, $\phi^{\mathsf{c}} = b : \star$, and $\phi_1^{\mathsf{c}} = c : \star$, and the final kind is $\forall\{c : \star\}.\forall\{b : \star\}.\forall(a : c).\ b \to \star$. We see in this one example the three sources of quantified variables, always in this order: variables arising from generalization (c), from implicit quantification (b), and from explicit quantification (a).

Returning to the $\vdash^{\mathsf{pgm}}$ judgment, rule PGM-DT-TTS checks a datatype definition that has a kind signature. It ensures that the signature has already been checked, by fetching the kind information in the context using $(T : \eta) \in \Sigma$. Then it checks the datatype declaration, and gathers the output term context to check the rest of the program. Rule PGM-DT-TT, as in Haskell98, guesses kinds ω_i for each datatype T_i and puts $T_i : \omega_i$ in the context *before* looking at the declarations. The major difference from Haskell98 is that kinds can be generalized *after* the group is checked. We use ϕ_i^{c} to denote the free variables in each kind ω_i. After the recursive group is typed, we generalize the kind of each type constructor as well as the type of its data constructors. To generalize the type of data constructors, we use the $\vdash^{\mathsf{gen}}$ judgment. Rule GEN generalizes every data constructor in the context, where ϕ^{c} are free type variables of its corresponding type constructor, and ϕ_i^{c} are free type variables specific to the data constructor. Returning to rule PGM-DT-TT, note that since the kinds of type constructors are generalized, the occurrences of the type constructors now require more type arguments. Therefore in Ψ_i', we substitute T_i with $T_i\ @\phi_i^{\mathsf{c}}$, where T_i is applied to all the variables bound in ϕ_i^{c}.

The judgment of checking datatype declarations $\Sigma \vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Psi$ has only rule DT-TT, which expands on the rule in Haskell98, to support top-level polymorphism for the kind of T.

Rule DC-TT supports existential variables ϕ. Moreover, the elaborated type μ of $\forall\phi.\ \overline{\tau_i}^{\,i} \to \rho$ can be further generalized over ϕ^{c}. Note that ϕ^{c} (via a small abuse of notation in the rule) excludes free variables in τ_i and Σ.

elaborated monotype	ρ, ω	$::=$	$\star \mid \mathsf{Int} \mid a \mid T \mid \rho_1\,\rho_2 \mid \rho_1\,@\rho_2 \mid \to \mid \widehat{\alpha}$
term context	Γ	$::=$	$\bullet \mid \Gamma, D : \mu$
type context	Δ, Θ	$::=$	$\bullet \mid \Delta, a : \omega \mid \Delta, T : \eta$
		$\mid$	$\Delta, \widehat{\alpha} : \omega \mid \Delta, \widehat{\alpha} : \omega = \rho \mid \Delta, \{\Delta'\} \mid \Delta, \blacktriangleright_D$
complete type context	Ω	$::=$	$\bullet \mid \Omega, a : \omega \mid \Omega, T : \eta \mid \Omega, \widehat{\alpha} : \omega = \rho \mid \Omega, \{\Omega'\} \mid \Omega, \blacktriangleright_D$
kind binder list	$\widehat{\phi}^{\mathsf{c}}$	$::=$	$\bullet \mid \widehat{\phi}^{\mathsf{c}}, \widehat{\alpha} : \kappa$

Figure 7.7: Algorithmic syntax in PolyKinds

7.6.2 Checking Kinds

The kinding judgment $\vdash^{\mathsf{k}}$ appears in Figure 7.6. We only highlight selected rules. Kinding $\Sigma \vdash^{\mathsf{k}} \sigma : \eta \rightsquigarrow \mu$ infers the type σ to have kind η, and it elaborates σ to μ. The kinding rules are built upon the axiom $\Sigma \vdash^{\mathsf{k}} \star : \star \rightsquigarrow \star$ (rule KTT-STAR). While this axiom is known to violate logical consistency, as Haskell is already logically inconsistent because of its general recursion, we do not consider it as an issue here. Rule KTT-APP concerns applications $\tau_1\ \tau_2$. It first infers the kind of τ_1 to be η_1. The kind η_1 can be a polymorphic kind headed by a $\forall$, though it is expected to be a function kind. Thus the rule uses $\vdash^{\mathsf{inst}}$ to instantiate η_1 to $\omega_1 \to \omega_2$. The instantiation judgment $\Sigma \vdash^{\mathsf{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2$ instantiates a kind η to a monokind ω, where if μ_1 has kind η then μ_2 has kind ω. After instantiation, rule KTT-APP checks ($\vdash^{\mathsf{kc}}$) the argument τ_2 against the expected argument kind ω_1. The kind checking judgment $\vdash^{\mathsf{kc}}$ simply delegates the work to kinding and instantiation. Rule KTT-KAPP checks visible kind applications. Note in the return kind η, the variable a is substituted by the elaborated argument ρ_2. Rule KTT-FORALLI elaborates an unannotated type $\forall a.\,\sigma$ to $\forall a : \omega.\,\mu$, where ω is an *elaborated* kind ($\vdash^{\mathsf{ela}}$) guessed for a.

The stand-alone elaborated kinding judgment $\vdash^{\mathsf{ela}}$ type-checks elaborated types. As all necessary instantiation has been done, type-checking for elaborated types is easy. For example, rule ELA-APP concerns applications $\rho_1\ \rho_2$. Compared to rule KTT-APP, here ρ_1 has an arrow kind, and takes exactly the kind of ρ_2. All judgments output well-formed elaborated types, as the following lemma states:

Lemma 7.5 (Type Elaboration). *We have: 1. if $\Sigma \vdash^{\mathsf{k}} \sigma : \eta \rightsquigarrow \mu$, then $\Sigma \vdash^{\mathsf{ela}} \mu : \eta$; 2. if $\Sigma \vdash^{\mathsf{kc}} \sigma \Leftarrow \eta \rightsquigarrow \mu$, then $\Sigma \vdash^{\mathsf{ela}} \mu : \eta$; 3. if $\Sigma \vdash^{\mathsf{ela}} \mu_1 : \eta$, and $\Sigma \vdash^{\mathsf{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2$, then $\Sigma \vdash^{\mathsf{ela}} \mu_2 : \omega$.*

7.7 Kind Inference for PolyKinds

We now describe the *algorithmic* counterpart of the PolyKinds system. Figure 7.7 presents the syntax of kinds and contexts in the algorithmic system for PolyKinds. Elaborated monotypes are extended with unification variables $\widehat{\alpha}$. Echoing the algorithm for Haskell98, type contexts are extended with unification variables, which now have kinds ($\widehat{\alpha} : \omega$ and $\widehat{\alpha} : \omega = \rho$). Also added to contexts are local scopes $\{\Delta\}$. These are special type contexts, where *variables can be reordered*. Recall the kind $\forall\,(a :: (f\;\;b))\;\;(c :: k).\,f\;\;c\;\rightarrow\;\star$ in Section 7.2.2, where f and b appear before k, but end up depending on k. In which order should we put f, b and k in the algorithmic context to kind-check the signature? We cannot have a correct order before completing inference. Therefore, we put them into a local scope, knowing we can reorder the variables during kind-checking according to the dependency information. The well-formedness judgment for local scopes requires them to be well-scoped, leading to the fact that $\Delta, \{\Delta'\}$ is well-formed iff Δ, Δ' is. The marker $\blacktriangleright_D$, subscripted by the name of a data constructor, is used only in and explained with rule A-DC-TT.

7.7.1 Algorithmic Program Typing

The algorithmic typing rules appear in Figure 7.8. The judgment $\Omega; \Gamma \Vdash^{\mathsf{pgm}} pgm : \mu$ checks the program. The rule A-PGM-SIG and rule A-PGM-DT-TTS correspond directly to the declarative rules. Note that as the datatype declaration in rule A-PGM-DT-TTS already has a signature, the output type context remains unchanged. Rule A-PGM-DT-TT concerns a group (without kind signatures). Like in Haskell98, it first assigns a fresh unification variable $\widehat{\alpha}_i : \star$ as the kind of each type constructor, and then type-checks each datatype declaration, yielding the output context Θ_{n+1}. Unlike Haskell98 which then uses defaulting, here from each $\widehat{\alpha}_i$ we get their unsolved unification variables $\widehat{\phi}_i^{\mathsf{c}}$ and generalize the kind of each type constructor as well as the type of each data constructor. The **unsolved** (Δ) metafunction simply extracts a set of free unification variables in Δ, with their kinds substituted by Δ. Before generalization, we apply Θ_{n+1} to the results so all solved unification variables get substituted away. We use the notation $\widehat{\phi}_i^{\mathsf{c}} \mapsto \phi_i^{\mathsf{c}}$ to mean that all unification variables in $\widehat{\phi}_i^{\mathsf{c}}$ are replaced by fresh type variables in ϕ_i^{c}. The algorithmic generalization judgment $\Vdash^{\mathsf{gen}}$ corresponds straightforwardly to the declarative rule, and thus is omitted. Though they appear daunting, the extended contexts used in the last premise to this rule are unsurprising: they just apply the relevant substitutions (the solved unification variables in Θ_{n+1}, the replacement of unification variables with fresh proper type variables $\widehat{\phi}_i^{\mathsf{c}} \mapsto \phi_i^{\mathsf{c}}$, and the generalization of the kinds of the group of datatypes $T_i \mapsto T_i\;@\phi_i^{\mathsf{c}}$).

$\boxed{\Omega; \Gamma \Vdash^{\mathsf{pgm}} pgm : \mu}$ *(Typing Program)*

A-PGM-SIG

$$\frac{\Omega \Vdash^{\mathsf{sig}} \mathcal{S} \rightsquigarrow T : \eta \qquad \Omega, T : \eta; \Gamma \Vdash^{\mathsf{pgm}} pgm : \mu}{\Omega; \Gamma \Vdash^{\mathsf{pgm}} \mathbf{sig}\,\mathcal{S}; pgm : \mu}$$

A-PGM-DT-TTS

$$\frac{(T : \eta) \in \Omega \qquad \Omega \Vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Gamma_1 \dashv \Omega \qquad \Omega; \Gamma, \Gamma_1 \Vdash^{\mathsf{pgm}} pgm : \mu}{\Omega; \Gamma \Vdash^{\mathsf{pgm}} \mathbf{rec}\,\mathcal{T}; pgm : \mu}$$

A-PGM-DT-TT

$$\frac{\begin{array}{c} \Theta_1 = \Omega, \overline{\widehat{\alpha}_i : \star}^{\,i}, \overline{T_i : \widehat{\alpha}_i}^{\,i} \qquad \overline{\Theta_i \Vdash^{\mathsf{dt}} \mathcal{T}_i \rightsquigarrow \Gamma_i \dashv \Theta_{i+1}}^{\,i} \\ \overline{\widehat{\phi}^{\mathsf{c}}_i = \mathbf{unsolved}([\Theta_{n+1}]\widehat{\alpha}_i)}^{\,i} \qquad \overline{\Theta_{n+1} \Vdash^{\mathsf{gen}}_{\phi^{\mathsf{c}}_i} ([\Theta_{n+1}](\Gamma_i[\overline{\widehat{\phi}^{\mathsf{c}}_i \mapsto \phi^{\mathsf{c}}_i}^{\,i}])) \rightsquigarrow \Gamma'_i}^{\,i} \\ \overline{\Omega,\ T_i : \forall\{\phi^{\mathsf{c}}_i\}.(([\Theta_{n+1}]\widehat{\alpha}_i)[\overline{\widehat{\phi}^{\mathsf{c}}_i \mapsto \phi^{\mathsf{c}}_i}^{\,i}])}^{\,i} ; \Gamma,\ \overline{\Gamma'_i[\overline{T_i \mapsto T_i\ @\phi^{\mathsf{c}}_i}^{\,i}]}^{\,i} \Vdash^{\mathsf{pgm}} pgm : \mu \end{array}}{\Omega; \Gamma \Vdash^{\mathsf{pgm}} \mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i \in 1..n}; pgm : \mu}$$

$\boxed{\Omega \Vdash^{\mathsf{sig}} \mathcal{S} \rightsquigarrow T : \eta}$ *(Typing Signature)*

A-SIG-TT

$$\frac{\begin{array}{c} \rceil\sigma\lceil \qquad \overline{a_i}^{\,i} = \mathsf{fkv}(\sigma) \qquad \Omega, \{\overline{\widehat{\alpha}_i : \star, a_i : \widehat{\alpha}_i}^{\,i}\} \Vdash^{\mathsf{k}} \sigma : \star \rightsquigarrow \eta \dashv \Delta \\ \phi^{\mathsf{c}}_1 = \mathsf{scoped\_sort}(\overline{a_i : [\Delta]\widehat{\alpha}_i}^{\,i}) \qquad \widehat{\phi}^{\mathsf{c}}_2 = \mathbf{unsolved}(\Delta) \qquad \Delta \hookrightarrow \overline{a_i}^{\,i} \end{array}}{\Omega \Vdash^{\mathsf{sig}} \mathbf{data}\,T : \sigma \rightsquigarrow T : \forall\{\phi^{\mathsf{c}}_2\}.((\forall\{\phi^{\mathsf{c}}_1\}.[\Delta]\eta)[\widehat{\phi}^{\mathsf{c}}_2 \mapsto \phi^{\mathsf{c}}_2])}$$

$\boxed{\Delta \Vdash^{\mathsf{dt}} \mathcal{T} \rightsquigarrow \Gamma \dashv \Theta}$ *(Typing Datatype Decl.)*

A-DT-TT

$$\frac{\begin{array}{c} (T : \forall\{\phi^{\mathsf{c}}_1\}.\forall\phi^{\mathsf{c}}_2.\ \omega) \in \Delta \\ \Delta, \phi^{\mathsf{c}}_1, \phi^{\mathsf{c}}_2, \overline{\widehat{\alpha}_i : \star}^{\,i} \Vdash^{\mathsf{u}} [\Delta]\omega \approx (\overline{\widehat{\alpha}_i}^{\,i} \to \star) \dashv \Theta_1, \phi^{\mathsf{c}}_1, \phi^{\mathsf{c}}_2, \overline{\widehat{\alpha}_i : \star = \omega_i}^{\,i} \\ \overline{\Theta_j, \phi^{\mathsf{c}}_1, \phi^{\mathsf{c}}_2, \overline{a_i : \omega_i}^{\,i} \Vdash^{\mathsf{dc}}_{(T\ @\phi^{\mathsf{c}}_1\ @\phi^{\mathsf{c}}_2\ \overline{a_i}^{\,i})} \mathcal{D}_j \rightsquigarrow \mu_j \dashv \Theta_{j+1}, \phi^{\mathsf{c}}_1, \phi^{\mathsf{c}}_2, \overline{a_i : \omega_i}^{\,i}}^{\,j} \end{array}}{\Delta \Vdash^{\mathsf{dt}} \mathbf{data}\,T\,\overline{a_i}^{\,i} = \overline{\mathcal{D}_j}^{\,j \in 1..n} \rightsquigarrow \overline{D_j : \forall\{\phi^{\mathsf{c}}_1\}.\forall\phi^{\mathsf{c}}_2.\ \forall\overline{a_i : \omega_i}^{\,i}.\ \mu_j}^{\,j} \dashv \Theta_{n+1}}$$

$\boxed{\Delta \Vdash^{\mathsf{dc}}_{\rho} \mathcal{D} \rightsquigarrow \mu \dashv \Theta}$ *(Typing Data Constructor Decl.)*

A-DC-TT

$$\frac{\Delta, \blacktriangleright_D \Vdash^{\mathsf{k}} \forall\phi.\ (\overline{\tau_i}^{\,i} \to \rho) : \star \rightsquigarrow \mu \dashv \Theta_1, \blacktriangleright_D, \Theta_2 \qquad \widehat{\phi}^{\mathsf{c}} = \mathbf{unsolved}(\Theta_2)}{\Delta \Vdash^{\mathsf{dc}}_{\rho} \forall\phi.\ D\,\overline{\tau_i}^{\,i} \rightsquigarrow \forall\{\phi^{\mathsf{c}}\}.(([\Theta_2]\mu)[\widehat{\phi}^{\mathsf{c}} \mapsto \phi^{\mathsf{c}}]) \dashv \Theta_1}$$

Figure 7.8: Algorithmic program typing in PolyKinds

The judgment $\Omega \Vdash^{\mathsf{sig}} \mathcal{S} \rightsquigarrow T : \eta$ type-checks a signature definition. We get all free variables in σ using $\mathsf{fkv}(\sigma)$ and assign each variable a_i a kind $\widehat{\alpha}_i : \star$. Those variables are put into a local scope to kind-check σ. Then, we use scoped_sort—a standard topological sort—to return an ordering of the variables that respects dependencies. Finally, we substitute away solved unification variables in the result kind μ and generalize over the unsolved variables $\widehat{\phi}_2^{\mathsf{c}}$ in Δ. As $\widehat{\phi}_2^{\mathsf{c}}$ is generalized outside ϕ_1^{c}, we use the *quantification check* $\Delta \hookrightarrow \overline{a_i}^{\,i}$ (Section 7.7.2) to ensure the result kind is well-ordered.

Rule A-DT-TT is a straightforward generalization of rule A-DT-DECL to polymorphic kinds. Here T can have a polymorphic kind from kind signatures.

Rule A-DC-TT checks a data constructor declaration. It first puts a marker into the context before kinding. After kinding, it substitutes away all the solved unification variables to the right of the marker, and generalizes over all unsolved unification variables to the right of the marker. The fact that the context is ordered gives us precise control over variables that need generalization.

7.7.2 The Quantification Check

Ill-ordered kinds are rejected. Consider the following example:

$$\mathbf{data}\ \mathit{Proxy} :: \forall k.\ k \to \star$$
$$\mathbf{data}\ \mathit{Relate} :: \forall a\ (b :: a).\ a \to \mathit{Proxy}\ b \to \star$$
$$\mathbf{data}\ T :: \forall (a :: \star)\ (b :: a)\ (c :: a)\ d.\ \mathit{Relate}\ b\ d \to \star$$

Proxy just gives us a way to write a type whose kind is not $\star$. The $\mathit{Relate}\ \tau_1\ \tau_2$ type forces the kind of τ_2 to depend on that of τ_1, giving rise to the unusual dependency in T. The definition of T then introduces a, b, c and d. The kinds of a, b and c are known, but the kind of d must be inferred; call it $\widehat{\alpha}$. We discover that $\widehat{\alpha} = \mathit{Proxy}\ \widehat{\beta}$, where $\widehat{\beta} :: a$. There are no further constraints on $\widehat{\beta}$. Naïvely, we would generalize over $\widehat{\beta}$, but that would be disastrous, as a is locally bound. Instead, we must reject this definition, as our declarative specification always puts inferred variables (such as the type variable $\widehat{\beta}$ would become if generalized) before other ones.

The quantification-checking metafunction $\Delta \hookrightarrow \phi$, defined as $\mathbf{fkv}(\mathbf{unsolved}\,(\Delta)) \sharp \phi$, ensures that free variables in $\mathbf{unsolved}\,(\Delta)$ are disjoint ($\sharp$) with ϕ, so that we can safely generalize $\mathbf{unsolved}\,(\Delta)$ outside ϕ.[3]

$\boxed{\Delta \Vdash^{\mathsf{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2 \dashv \Theta}$ *(Instantiation)*

$$\frac{\Delta \Vdash^{\mathsf{u}} \omega_1 \approx \omega_2 \dashv \Theta}{\Delta \Vdash^{\mathsf{inst}} \mu : \omega_1 <: \omega_2 \rightsquigarrow \mu \dashv \Theta}\ \textsc{A-Inst-Refl}$$

$$\frac{\Delta, \widehat{\alpha} : \omega_1 \Vdash^{\mathsf{inst}} \mu_1 \mathbin{@}\widehat{\alpha} : \eta[a \mapsto \widehat{\alpha}] <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta}{\Delta \Vdash^{\mathsf{inst}} \mu_1 : \forall a : \omega_1 .\, \eta <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta}\ \textsc{A-Inst-Forall}$$

$\boxed{\Delta \Vdash^{\mathsf{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu \dashv \Theta}$ *(Kind Checking)*

$$\frac{\Delta \Vdash^{\mathsf{k}} \sigma : \eta \rightsquigarrow \mu_1 \dashv \Delta_1 \qquad \Delta_1 \Vdash^{\mathsf{inst}} \mu_1 : [\Delta_1]\eta <: [\Delta_1]\omega \rightsquigarrow \mu_2 \dashv \Delta_2}{\Delta \Vdash^{\mathsf{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu_2 \dashv \Delta_2}\ \textsc{A-KC-Sub}$$

$\boxed{\Delta \Vdash^{\mathsf{k}} \sigma : \eta \rightsquigarrow \mu \dashv \Theta}$ *(Kinding)*

$$\frac{}{\Delta \Vdash^{\mathsf{k}} \star : \star \rightsquigarrow \star \dashv \Delta}\ \textsc{A-KTT-Star}$$

$$\frac{\Delta \Vdash^{\mathsf{k}} \tau_1 : \eta_1 \rightsquigarrow \rho_1 \dashv \Delta_1 \qquad \Delta_1 \Vdash^{\mathsf{kapp}} (\rho_1 : [\Delta_1]\eta_1) \bullet \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\mathsf{k}} \tau_1\, \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}\ \textsc{A-KTT-App}$$

$$\frac{\Delta, \widehat{\alpha} : \star, a : \widehat{\alpha} \Vdash^{\mathsf{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu \dashv \Delta_2, a : \widehat{\alpha}, \Delta_3 \qquad \Delta_3 \hookrightarrow a}{\Delta \Vdash^{\mathsf{k}} \forall a .\, \sigma : \star \rightsquigarrow \forall a : \widehat{\alpha} .\, [\Delta_3]\mu \dashv \Delta_2, \mathbf{unsolved}\,(\Delta_3)}\ \textsc{A-KTT-ForallI}$$

$\boxed{\Delta \Vdash^{\mathsf{kapp}} (\rho_1 : \eta) \bullet \tau : \omega \rightsquigarrow \rho_2 \dashv \Theta}$ *(Application Kinding)*

$$\frac{\Delta \Vdash^{\mathsf{kc}} \tau \Leftarrow \omega_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta \Vdash^{\mathsf{kapp}} (\rho_1 : \omega_1 \to \omega_2) \bullet \tau : \omega_2 \rightsquigarrow \rho_1\, \rho_2 \dashv \Theta}\ \textsc{A-KApp-TT-Arrow}$$

$$\frac{\Delta, \widehat{\alpha} : \omega_1 \Vdash^{\mathsf{kapp}} (\rho_1 \mathbin{@}\widehat{\alpha} : \eta[a \mapsto \widehat{\alpha}]) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\mathsf{kapp}} (\rho_1 : \forall a : \omega_1 .\, \eta) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}\ \textsc{A-KApp-TT-Forall}$$

$$\frac{\Delta_1, \widehat{\alpha}_1 : \star, \widehat{\alpha}_2 : \star, \widehat{\alpha} : \omega = (\widehat{\alpha}_1 \to \widehat{\alpha}_2), \Delta_2 \Vdash^{\mathsf{kc}} \tau \Leftarrow \widehat{\alpha}_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta_1, \widehat{\alpha} : \omega, \Delta_2 \Vdash^{\mathsf{kapp}} (\rho_1 : \widehat{\alpha}) \bullet \tau : \widehat{\alpha}_2 \rightsquigarrow \rho_1\, \rho_2 \dashv \Theta}\ \textsc{A-KApp-TT-KUVar}$$

$\boxed{\Delta \Vdash^{\mathsf{ela}} \mu : \eta}$ *(Elaborated Kinding)*

$$\frac{\Delta \Vdash^{\mathsf{ela}} \rho_1 : \omega_1 \to \omega_2 \qquad \Delta \Vdash^{\mathsf{ela}} \rho_2 : \omega_1}{\Delta \Vdash^{\mathsf{ela}} \rho_1\, \rho_2 : \omega_2}\ \textsc{A-Ela-App}$$

$$\frac{\Delta \Vdash^{\mathsf{ela}} \rho_1 : \forall a : \omega .\, \eta \qquad \Delta \Vdash^{\mathsf{ela}} \rho_2 : \omega}{\Delta \Vdash^{\mathsf{ela}} \rho_1 \mathbin{@}\rho_2 : \eta[a \mapsto [\Delta]\rho_2]}\ \textsc{A-Ela-KApp}$$

Figure 7.9: Selected rules for algorithmic kinding in PolyKinds

7.7.3 Kinding

Figure 7.9 presents the selected rules for kinding judgment $\Vdash^{\mathsf{k}}$, along with the auxiliary judgments. Full rules can be found in Appendix C.3. Most rules correspond directly to their declarative counterparts. For applications $\tau_1\ \tau_2$, rule A-KTT-APP first synthesizes the kind of τ_1 to be η_1, then uses $\Vdash^{\mathsf{kapp}}$ to type-check τ_2. The judgment $\Delta \Vdash^{\mathsf{kapp}} (\rho_1 : \eta) \bullet \tau : \omega \rightsquigarrow \rho_2 \dashv \Theta$ is interpreted as, under context Δ, applying the type ρ_1 of kind η to the type τ returns kind ω, the elaboration result ρ_2, and an output context Θ. When η_1 is polymorphic (rule A-KAPP-TT-FORALL), we instantiate it with a fresh unification variable. Rule A-KTT-FORALLI checks a polymorphic type. We assign a unification variable as the kind of a, bring $\widehat{\alpha} : \star, a : \widehat{\alpha}$ into scope to check the body against $\star$, yielding the output context $\Delta_2, a : \widehat{\alpha}, \Delta_3$. As a goes out of the scope in the conclusion, we need to drop a in the concluding context. To make sure that dropping a outputs a well-formed context, we substitute away all solved unification variables in Δ_3 for the return kind, and keep only **unsolved** (Δ_3), which are ensured $(\Delta_3 \hookrightarrow a)$ to have no dependency on a.

In the algorithmic elaborated kinding judgment $\Delta \Vdash^{\mathsf{ela}} \mu : \eta$, we keep the invariant: $[\Delta]\eta = \eta$. That is why in rule A-ELA-APP we substitute a with $[\Delta]\rho_2$.

Instantiation ($\Vdash^{\mathsf{inst}}$) contains the only entry to unification (rule A-INST-REFL).

7.7.4 Unification

The judgments of unification and promotion are excerpted in Figure 7.10. Most rules are natural extensions of those in Haskell98.

Promotion The promotion judgment $\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta$ is extended with kind annotations for unification variables. As our unification variables have kinds now, rule A-PR-KUVARR-TT must also promote the kind of $\widehat{\beta}$, so that $\widehat{\beta}_1 : \rho_1$ in the context is well-formed. Promotion now has a new failure mode: it cannot move proper quantified type variables. In rule A-PR-TVAR, the variable a must be to the left of $\widehat{\alpha}$.

Unfortunately, now we cannot easily tell whether promoting is terminating. In particular, the convergence of promotion in Haskell98 is built upon the obvious fact that the size of the kind being promoted always gets smaller from the conclusion to the hypothesis. However, rule A-PR-KUVARR-TT breaks this invariant, as the judgment recurs into the kinds of unification variables, and the size of the kinds may be larger than the unification variables. As shown in Section 7.7.5, we prove that promotion is terminating.

[3] See also the alternative design in Appendix C.2.9.

$\boxed{\Delta \Vdash^{\mathsf{u}} \omega_1 \approx \omega_2 \dashv \Theta}$ *(Unification)*

$$\frac{}{\Delta \Vdash^{\mathsf{u}} \omega \approx \omega \dashv \Delta}\ \textsc{a-u-refl-tt}$$

$$\frac{\Delta \Vdash^{\mathsf{u}} \rho_1 \approx \rho_3 \dashv \Delta_1 \qquad \Delta_1 \Vdash^{\mathsf{u}} [\Delta_1]\rho_2 \approx [\Delta_1]\rho_4 \dashv \Theta}{\Delta \Vdash^{\mathsf{u}} \rho_1\, \rho_2 \approx \rho_3\, \rho_4 \dashv \Theta}\ \textsc{a-u-app}$$

$$\frac{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \widehat{\alpha} : \omega_1, \Theta_2 \qquad \Theta_1 \Vdash^{\mathsf{ela}} \rho_2 : \omega_2 \qquad \Theta_1 \Vdash^{\mathsf{u}} [\Theta_1]\omega_1 \approx \omega_2 \dashv \Theta_3}{\Delta \Vdash^{\mathsf{u}} \widehat{\alpha} \approx \rho_1 \dashv \Theta_3, \widehat{\alpha} : \omega_1 = \rho_2, \Theta_2}\ \textsc{a-u-kvarL-tt}$$

$$\frac{\begin{array}{c}\Delta_1, \Delta_2 +\!\!+^{\mathsf{mv}} \widehat{\alpha} : \omega_1 \leadsto \Theta \qquad \Delta[\{\Theta\}] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \{\Theta_2, \widehat{\alpha} : \omega_1, \Theta_3\}, \Theta_4 \\ \Theta_1, \{\Theta_2\} \Vdash^{\mathsf{ela}} \rho_2 : \omega_2 \qquad \Theta_1, \{\Theta_2\} \Vdash^{\mathsf{u}} [\Theta_1, \Theta_2]\omega_1 \approx \omega_2 \dashv \Theta_5, \{\Theta_6\}\end{array}}{\Delta[\{\Delta_1, \widehat{\alpha} : \omega_1, \Delta_2\}] \Vdash^{\mathsf{u}} \widehat{\alpha} \approx \rho_1 \dashv \Theta_5, \{\Theta_6, \widehat{\alpha} : \omega_1 = \rho_2, \Theta_3\}, \Theta_4}\ \textsc{a-u-kvarL-lo-tt}$$

$\boxed{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta}$ *(Promotion)*

$$\frac{}{\Delta[a][\widehat{\alpha}] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} a \rightsquigarrow a \dashv \Delta[a][\widehat{\alpha}]}\ \textsc{a-pr-tvar}$$

$$\frac{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} [\Delta]\rho \rightsquigarrow \rho_1 \dashv \Theta[\widehat{\alpha}][\widehat{\beta} : \rho]}{\Delta[\widehat{\alpha}][\widehat{\beta} : \rho] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\beta} \rightsquigarrow \widehat{\beta}_1 \dashv \Theta[\widehat{\beta}_1 : \rho_1, \widehat{\alpha}][\widehat{\beta} : \rho = \widehat{\beta}_1]}\ \textsc{a-pr-kuvarR-tt}$$

$\boxed{\Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \leadsto \Theta}$ *(Moving)*

$$\frac{}{\bullet +\!\!+^{\mathsf{mv}} \Delta \leadsto \Delta}\ \textsc{a-mv-empty}$$

$$\frac{\mathbf{var}(\omega) \sharp \mathbf{dom}(\Delta_2) \qquad \Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \leadsto \Theta}{\widehat{\alpha} : \omega, \Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \leadsto \widehat{\alpha} : \omega, \Theta}\ \textsc{a-mv-kuvar}$$

$$\frac{\neg(\mathbf{var}(\omega) \sharp \mathbf{dom}(\Delta_2)) \qquad \Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2, \widehat{\alpha} : \omega \leadsto \Theta}{\widehat{\alpha} : \omega, \Delta_1 +\!\!+^{\mathsf{mv}} \Delta \leadsto \Theta}\ \textsc{a-mv-kuvarM}$$

Figure 7.10: Selected rules for unification, promotion, and moving in PolyKinds

Unification The unification judgment $\Delta \Vdash^{\mu} \omega_1 \approx \omega_2 \dashv \Theta$ for PolyKinds features *heterogeneous constraints*. Recall the definition of X and Y discussed in Section 7.2.2. When unifying $\widehat{\alpha}\,\widehat{\beta}$ with $\mathsf{Maybe\ Bool}$, setting $\widehat{\alpha} = \mathsf{Maybe}$ and $\widehat{\beta} = \mathsf{Bool}$ results in ill-kinded results. This suggests that when solving a unification variable, we need to first unify the kinds of both sides, as shown in rule A-U-KVARL-TT. When unifying $\widehat{\alpha}$ with ρ_1, we first promote ρ_1, yielding ρ_2. Now ρ_2 must be well-formed under Θ_1, so we can get its kind ω_1. We then unify the kinds of both sides. If everything succeeds, we set $\widehat{\alpha} : \omega_1 = \rho_2$. Under this rule, the unification $\widehat{\alpha}\,\widehat{\beta} \approx \mathsf{Maybe\ Bool}$ would be rejected correctly.

Rule A-U-KVARL-LO-TT is essentially the same as rule A-U-KVARL-TT, but deals with unification variables in a local scope. We thus need an extra step to *move* $\widehat{\alpha}$ towards the end of the local scope.

Local scopes and moving As we have mentioned, a local scope can be reordered as long as the context is well-formed. Consider unifying $\{\widehat{\alpha} : \star, a : \star, b : \widehat{\alpha}, c : \star\} \vdash \widehat{\alpha} \approx a$. We see that a is not well-formed under the context before $\widehat{\alpha}$, and thus we cannot rewrite $\widehat{\alpha} : \star$ with $\widehat{\alpha} = a : \star$. However, we *can* reorder the context to put $\widehat{\alpha}$ to the right of a. In fact, to maximize the prefix context of $\widehat{\alpha}$, we can move $\widehat{\alpha}$ to the end of the context, yielding $\{a : \star, c : \star, \widehat{\alpha} : \star, b : \widehat{\alpha}\}$. As b depends on $\widehat{\alpha}$, b is also moved to the end of the context. The final context is now $\{a : \star, c : \star, \widehat{\alpha} : \star = a, b : \widehat{\alpha}\}$.

The *moving* judgment $\Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \rightsquigarrow \Theta$ reorders the context, by appending Δ_2 to the end of Δ_1, yielding Θ. As we have emphasized, reordering must preserve a well-formed context. Therefore, every term that depends on Δ_2 (rule A-MV-KUVARM) needs to be placed at the end, along with Δ_2.

In rule A-U-KVARL-LO-TT, we begin by reordering the local scope to put $\widehat{\alpha}$ as far to the right as possible. The rest of the rule is essentially the same as rule A-U-KVARL-TT: the added complication stems from the need to keep track of what bindings in the context are a part of the current local scope.

7.7.5 Termination

Now the challenge is to prove that our unification algorithm terminates, which relies on the convergence of the promotion algorithm. Next, we first discuss the termination of unification, and then move to the more complicated proof for promotion. Let $\langle\Delta\rangle$ denote the number of unsolved unification variables in Δ.

Lemma 7.6 (Promotion Preserves $\langle\Delta\rangle$). *If* $\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta$, *then* $\langle\Delta\rangle = \langle\Theta\rangle$.

Lemma 7.7 (Unification Makes Progress). *If $\Delta \Vdash^{\mu} \omega_1 \approx \omega_2 \dashv \Theta$, then either $\Theta = \Delta$, or $\langle\Theta\rangle < \langle\Delta\rangle$.*

Now we measure unification $\Delta \Vdash^{\mu} \omega_1 \approx \omega_2 \dashv \Theta$ using the lexicographic order of the pair $(\langle\Delta\rangle, |\omega_1|)$, where $|\omega_1|$ computes the standard size of ω_1. We prove the pair always gets smaller from the conclusion to the hypothesis. Formally, assuming promotion terminates, we have

Theorem 7.8 (Unification Terminates). *Given a context Δ ok, and kinds ρ_1 and ρ_2, where $[\Delta]\rho_1 = \rho_1$, and $[\Delta]\rho_2 = \rho_2$, it is decidable whether there exists Θ such that $\Delta \Vdash^{\mu} \rho_1 \approx \rho_2 \dashv \Theta$.*

We are not yet done, since Theorem 7.8 depends on the convergence of promotion. As observed in rule A-PR-KUVARR, the size of the type being promoted increases from the conclusion to the hypothesis. Worse, the context never decreases. How do we prove promotion terminates? The crucial observation for rule A-PR-KUVARR is that, when we move from the conclusion to the hypothesis, we also move from a unification variable to its kind. Since the kind is well-formed under the prefix context of the variable, we are somehow moving leftward in the context.

To formalize the observation, we define the *dependency graph* of a context.

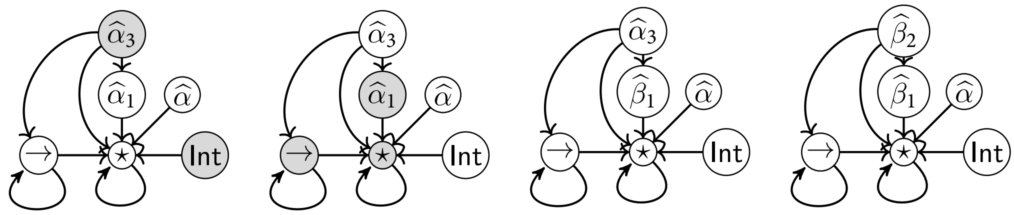
Definition 22 (Dependency Graph). The dependency graph of a context Δ is a *directed* graph where:

1. Nodes are all type variables and unsolved unification variables of Δ, and the terminal symbols $\star$, $\rightarrow$ and Int.

2. Edges indicate the dependency from a type to its substituted kind. For example, if $\widehat{\alpha} : \omega$, then there is a directed edge from $\widehat{\alpha}$ to all the nodes appearing in $[\Delta]\omega$.

As an illustration, consider the context $\Delta = \widehat{\alpha} : \star, \widehat{\alpha}_1 : \star, \widehat{\alpha}_2 : \star = \widehat{\alpha}_1, \widehat{\alpha}_3 : \star \rightarrow \widehat{\alpha}_2$, whose dependency graph is given in Figure 7.11a (the reader is advised to ignore the color for now). There are several notable properties. First, as long as the context is well-formed, the graph is *acyclic* except for the self-loop of $\star$ and $\rightarrow$. Second, solved unification variables never appear in the graph. The kind of $\widehat{\alpha}_3$ depends on $\widehat{\alpha}_2$, which is already solved by $\widehat{\alpha}_1$, so the dependency goes from $\widehat{\alpha}_3$ to $\widehat{\alpha}_1$.

Now let us consider how promotion works in terms of the dependency graph, by trying to unify $\Delta \vdash \widehat{\alpha} \approx \widehat{\alpha}_3\ \mathsf{Int}$. We start by promoting $\widehat{\alpha}_3\ \mathsf{Int}$. The derivation of the promotion is given at the bottom of Figure 7.11. We omit some details via $(\cdots)$ as promoting constants ($\star$,$\rightarrow$ and Int) is trivial. At the top of Figure 7.11 we give the dependency graph at certain



(a) Promote $\widehat{\alpha}_3$ Int (b) Promote $\star \to \widehat{\alpha}_1$ (c) Solve $\widehat{\alpha}_1 = \widehat{\beta}_1$ (d) Solve $\widehat{\alpha}_3 = \widehat{\beta}_2$

$$\Theta_1 = \widehat{\beta}_1 : \star, \widehat{\alpha} : \star, \widehat{\alpha}_1 : \star = \widehat{\beta}_1, \widehat{\alpha}_2 : \star = \widehat{\alpha}_1, \widehat{\alpha}_3 : \star \to \widehat{\alpha}_2$$

$$\Theta_2 = \widehat{\beta}_1 : \star, \widehat{\beta}_2 : \star \to \widehat{\beta}_1, \widehat{\alpha} : \star, \widehat{\alpha}_1 : \star = \widehat{\beta}_1, \widehat{\alpha}_2 : \star = \widehat{\alpha}_1, \widehat{\alpha}_3 : \star \to \widehat{\alpha}_2 = \widehat{\beta}_2$$

$$\dfrac{\dfrac{\dfrac{\cdots \quad \dfrac{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \star \leadsto \star \dashv \Delta}{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\alpha}_1 \leadsto \widehat{\beta}_1 \dashv \Theta_1 \quad \boxed{7.11c}}\ \textsc{a-pr-kuvarR}}{\boxed{7.11b} \quad \Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \star \to \widehat{\alpha}_1 \leadsto \star \to \widehat{\beta}_1 \dashv \Theta_1}\ \textsc{a-pr-app}}{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\alpha}_3 \leadsto \widehat{\beta}_2 \dashv \Theta_2 \quad \boxed{7.11d}}\ \textsc{a-pr-kuvarR} \qquad \cdots}{\boxed{7.11a} \quad \Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\alpha}_3\ \mathsf{Int} \leadsto \widehat{\beta}_2\ \mathsf{Int} \dashv \Theta_2}\ \textsc{a-pr-app}$$

Figure 7.11: Example of dependency graph

points in the derivation, where the part being promoted is highlighted in gray. At the beginning we are at Figure 7.11a. For $\widehat{\alpha}_3$, by rule A-PR-KUVARR, we first promote the kind of $\widehat{\alpha}_3$, which is (after context application) $\star \to \widehat{\alpha}_1$ (Figure 7.11b). As $\star$ and $\to$ are always well-formed, we then promote $\widehat{\alpha}_1$ whose kind is the well-formed $\star$. Now we create a fresh variable $\widehat{\beta}_1 : \star$, and solve $\widehat{\alpha}_1$ with $\widehat{\beta}_1$ (Figure 7.11c). Note since $\widehat{\alpha}_1$ is solved, the dependency from $\widehat{\alpha}_3$ goes to $\widehat{\beta}_1$. Finally, we create a fresh variable $\widehat{\beta}_2$ with kind $\star \to \widehat{\beta}_1$, and solve $\widehat{\alpha}_3$ with $\widehat{\beta}_2$ (Figure 7.11d). Going back to unification, we solve $\widehat{\alpha} = \widehat{\beta}_2$ Int.

We have several key observations. First, when we move from Figure 7.11a to Figure 7.11b via rule A-PR-KUVARR, we are actually moving from the current node ($\widehat{\alpha}_3$) to its adjacent nodes ($\star$, $\to$, and $\widehat{\alpha}_1$). In other words, we are going down in this graph. Moreover, promotion terminates immediately at type constants, so we never fall into the trap of loop. Further, when we solve variables with fresh ones (Figure 7.11c and Figure 7.11d), the shape of the graph never changes.

With all those in mind, we conclude that *the promotion process goes top-down via rule A-PR-KUVARR in the dependency graph until it terminates at types that are already well-formed.* Based on this conclusion, we can formally prove that promotion terminates.

Theorem 7.9 (Promotion Terminates). *Given a context $\Delta[\widehat{\alpha}]$ ok, and a kind ω_1 with $[\Delta]\omega_1 = \omega_1$, it is decidable whether there exists Θ such that $\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta$.*

7.7.6 Soundness, Completeness and Principality

We prove our algorithm is sound:

Theorem 7.10 (Soundness of $\Vdash^{\mathsf{pgm}}$). *If $\Omega; \Gamma \Vdash^{\mathsf{pgm}} pgm : \mu$, then $[\Omega]\Omega; [\Omega]\Gamma \vdash^{\mathsf{pgm}} pgm : [\Omega]\mu$.*

Unfortunately, we lose completeness. Recall the example in Section 7.7.2. This definition of T is rejected by the algorithmic quantification check as the kind of d cannot be determined. However, the declarative system can guess correctly, e.g., $Proxy\ b$ or $Proxy\ c$. Unfortunately, different choices lead to incomparable kinds for T. Thus we argue such programs must be rejected.

Nevertheless, if the user explicitly writes down $d :: Proxy\ \ b$ or $d :: Proxy\ \ c$, then the program will be accepted by the algorithm. Thus, we conjecture that if all local dependencies are annotated by the user, we can regain completeness. This, however, is a bit annoying to users, because it means that we cannot accept definitions like the one below, even though the dependency is clear.

$$\mathbf{data}\ Eq :: \forall k.\ k \to k \to \star$$
$$\mathbf{data}\ P :: \forall k\ (a :: k)\ b.\ Eq\ a\ b \to \star$$

We do not consider the incompleteness as a problematic issue in practice, as this scenario is quite contrived and (we expect) will rarely occur "in the wild". See more discussion of this point in Section 8.7.

Although the algorithm is incomplete, we offer the following guarantee: *if the algorithm accepts a definition, then that definition has a principal kind, and the algorithm infers the principal kind.*

Definition 23 (Kind Instantiation). Under context Σ, a kind $\eta = \forall\{\phi_1\}.\forall\phi_2.\omega_1$, where ϕ's can be empty, instantiates to ω, denoted as $\Sigma \vdash \quad \eta <: \omega$, if $\omega_1[\phi_1 \mapsto \overline{\rho_1}][\phi_2 \mapsto \overline{\rho_2}] = \omega$ for some $\overline{\rho_1}$ and $\overline{\rho_2}$.

The relation is embedded in $\Sigma \vdash^{\mathsf{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2$ (Figure 7.6), where we ignore μ_1 and μ_2.

Definition 24 (Partial Order of Kinds in PolyKinds). Under context Σ, a kind η_1 is *more general than* η_2, denoted as $\Sigma \vdash \eta_1 \preceq \eta_2$, if for all ω such that $\Sigma \vdash \quad \eta_2 <: \omega$, we have $\Sigma \vdash \quad \eta_1 <: \omega$.

To understand the definition, consider that if the program type-checks under $T : \eta_2$, then it must type-check under $T : \eta_1$, as $T : \eta_1$ can be instantiated to all monokinds that $T : \eta_2$ is used at.

Now we lift the definition of $\Vdash^{\mathsf{grp}}$ to be the generalized result of kinds and contexts.

Theorem 7.11 (Principality of $\Vdash^{\mathsf{grp}}$). *If* $\Omega \Vdash^{\mathsf{grp}}$ $\mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\eta_i}^{\,i} ; \overline{\Gamma_i}^{\,i}$*, then whenever* $[\Omega]\Omega \vdash^{\mathsf{grp}}$ $\mathbf{rec}\,\overline{\mathcal{T}_i}^{\,i} \rightsquigarrow \overline{\eta'_i}^{\,i} ; \overline{\Psi_i}^{\,i}$ *holds, we have* $[\Omega]\Omega \vdash [\Omega]\eta_i \preceq \eta'_i$.

This result echoes the result in the term-level type inference algorithm for Haskell ([Vytiniotis et al. 2011, Section 6.5]): our algorithm does not infer every kind acceptable by the declarative system, but the kinds it does infer are always the best (principal) ones.

7.8 Language Extensions

We have seen that the PolyKinds system incorporates many features and enjoys desirable properties. In this section, we discuss how the PolyKinds system can be extended with more related language features. Appendix C.1 contains a few more, less impactful extensions.

7.8.1 Higher-Rank Polymorphism

The system can be extended naturally to support higher-rank polymorphism [Dunfield and Krishnaswami 2013; Peyton Jones et al. 2007]. With higher-rank polymorphism, every type can have a polymorphic kind. For example, data constructor declarations become $\forall \phi.\, D\, \overline{\sigma_i}^{\,i}$ instead of $\forall \phi.\, D\, \overline{\tau_i}^{\,i}$.

Unfortunately, higher-rank polymorphism breaks principality. Consider:

$$\mathbf{data}\ \mathit{Q1} :: \forall k_1\ k_2.\, k_1 \to \star$$
$$\mathbf{data}\ \mathit{Q2} :: (\forall (k_1 : \star)\ (k_2 : k_1).\, k_1 \to \star) \to \star$$

First, we modify the definition of partial order of kinds (Definition 24) to state that one kind is more general than another if it can be instantiated to all *polykinds* that the other kind can be instantiated to. Now consider the kind of $\mathit{Q1}$, which under the algorithm is generalized to $\forall \{k3 : \star\}\ (k_1 : \star)\ (k_2 : k3).\, k_1 \to \star$. In Theorem 7.11, we guarantee that this kind is a principal kind as it can be instantiated to all monokinds that other possible kinds for $\mathit{Q1}$, e.g., $\forall (k_1 :: \star)\ (k_2 :: k_1).\, k_1 \to \star$, can be instantiated to. However, under the new definition, $\forall \{k3 :: \star\}\ (k_1 :: \star)\ (k_2 :: k3).\, k_1 \to \star$ is no longer more general than $\forall (k_1 :: \star)\ (k_2 :: k_1).\, k_1 \to \star$, as there is no way to instantiate the former to the latter. To see why we need to modify the definition at all, consider the rank-2 kind of $\mathit{Q2}$, which expects exactly an argument of kind $\forall (k_1 :: \star)\ (k_2 :: k_1).\, k_1 \to \star$.

We do not consider the absence of principality in the setting of higher-rank polymorphism to be a severe issue in practice, for two reasons: to our knowledge, higher-rank polymorphism for datatypes is not heavily used; and it may be possible to recover principality through the use of a more generous type-subsumption relation. Currently, GHC (and our model of it) does not support first-class type-level abstraction (i.e., Λ in types) [Jones 1995]. This means that we cannot introduce new variables (also called *skolemization* [Peyton Jones et al. 2007, Section 4.6.2]) in an attempt to equate one type with another. Returning to the example above, we *could* massage $\forall\{k3{::}\star\}\ (k_1{::}\star)\ (k_2{::}k3).\ k_1 \to \star$ to $\forall(k_1{::}\star)\ (k_2{::}k_1).\ k_1 \to \star$ if we could abstract over the k_1 in the target type. Recent advances in type-level programming in Haskell [Kiss et al. 2019] suggest we may be able to add first-class abstraction, meaning that type-subsumption can use both instantiation *and* skolemization. We conjecture that this development would recover principal types.

7.8.2 Generalized Algebraic Datatypes (GADTs)

The focus of this work has been on uniform datatypes, where every constructor's type matches exactly the datatype head: this fact allows us to easily choose the subscript to the $\vdash^{\mathsf{dc}}$ judgment in, e.g., rule DT-TT. Programmers in modern Haskell, however, often use *generalized* algebraic datatypes [Peyton Jones et al. 2006; Xi et al. 2003]. There are two impacts of adding these, both of which we found surprising.

Equality constraints The power of GADTs arises from how they encode local equality constraints. Any GADT can be rewritten to a uniform datatype with equality constraints [Vytiniotis et al. 2011, Section 4.1]. For example, we can rewrite

```
data G a where
  MkG :: G Bool
```

to be

```
data G a = (a ~ Bool) => MkG
```

where $\sim$ describes an equality constraint. For our purposes of doing kind inference, these equality constraints are uninteresting: the $\sim$ operator simply relates two types of the same kind and can be processed as any polykinded type constructor would be. Modeling constraints to the left of a $\Rightarrow$ similarly would add a little clutter to our rules, but would offer no real challenges.

The unexpected simplicity of adding GADTs to our system arises from a key fact: we do not ever allow *pattern-matching*. A GADT pattern-match brings a local equality assumption into scope, which would influence the unification algorithm. However, as pattern matching does not happen in the context of datatype declarations, we avoid this wrinkle here.

Syntax The implementation of GADTs in GHC has an unusual syntax:

$$\textbf{data}\ G\ a\ \textbf{where}$$
$$\quad MkG :: a \to G\ Int$$

The surprising aspect of this syntax is that the two as above are *different*: the a in the header is unrelated to the a in the data constructor. This seemingly inconsequential design choice makes kind inference for GADTs very challenging, as constructors have no way to refer back to the datatype parameters. Given that this aspect of GADTs is a quirk of GHC's design—and is not repeated in other languages that support GADTs—we remark here that it is odd and perhaps should be remedied. We will return back to this discussion in Section 9.4.

7.8.3 Type Families

Type families [Chakravarty et al. 2005] are, effectively, type-level functions. Kind inference of type families thus can be designed much like type inference for ordinary functions. However, as they can have dependency, the complications we describe in this paper would arise here, too. In particular, unification would have to be kind-directed, as we have described. The current syntax for closed type families [Eisenberg et al. 2014] shares the same scoping problem as the syntax for GADTs, so our arguments above apply to closed type families equally.

The challenge with type families is that they indeed do pattern-matching, and thus (in concert with GADTs) can bring local equalities into scope. A full analysis of the ramifications here is beyond the scope of this paper, but we believe the literature on type inference in the presence of local equalities would be helpful. Principal among these is the work of Vytiniotis et al. [2011], but Gundry [2013] and Eisenberg [2016] also approach this problem in the context of dependent types.

Part V

Epilogue

8 Related Work

There is a great deal of work related to this thesis. Along the way we have discussed some of the most relevant work. In this chapter, we briefly review more related work.

8.1 Type Inference for Higher-Rank Types

Predicative higher-rank type inference. Odersky and Läufer [1996] introduced a type system for higher-rank implicit polymorphic types. Based on that, Peyton Jones et al. [2007] developed an approach for type inference for higher-rank types using traditional bidirectional type checking. They use a more general subtyping relation, inspired by the type containment relation by Mitchell [1988], which supports *deep skolemisation*. With deep skolemization, examples like $\forall a. \mathsf{Int} \to \mathsf{Int} <: \mathsf{Int} \to \forall a. a$ are allowed. We believe deep skolemization is compatible with our subtyping definition.

Dunfield and Krishnaswami [2013] build a simple and concise algorithm for higher-rank polymorphism based on traditional bidirectional type checking. They deal with the same language of Peyton Jones et al. [2007], except they do not have **let** expressions nor generalization (though it is discussed in design variations). Built upon some of these techniques, Dunfield and Krishnaswami [2019] extend the system to a much richer type language that includes existentials, indexed types, and equations over type variables.

Impredicative higher-rank type inference. While our work focuses on predicative higher-rank types, there are also a lot of work on type inference for *impredicative* higher-rank types. Many of these work replies on new forms of types. ML^F [Le Botlan and Rémy 2003, 2009; Rémy and Yakobowski 2008] generalizes ML with first-class polymorphism. ML^F introduces a new type of bounded quantification (either rigid or flexible) for polymorphic types so that instantiation of polymorphic bindings is delayed until a principal type is found. higher-rank types. The HML system [Leijen 2009] is proposed as a simplification and restriction of ML^F. HML only uses flexible types, which simplifies the type inference algorithm, but retains many interesting properties and features.

The FPH system [Vytiniotis et al. 2008] introduces boxy monotypes into System F types. One critique of boxy type inference is that the impredicativity is deeply hidden in the algorithmic type inference rules, which makes it hard to understand the interaction between its predicative constraints and impredicative instantiations [Rémy 2005].

Recently, Serrano et al. [2020, 2018] exploit impredicative instantiations of type variables that appears under a type constructor (i.e., type variables are *guarded*). Serrano et al. [2018] distinguish variables using three *sorts*, so that certain sorts of variables can be instantiated with higher-rank polymorphic types. Serrano et al. [2020] inspect the function arguments and assign impredicative instantiations before monomorphic ones.

8.2 Bidirectional Type Checking

Bidirectional type checking was popularized by the work of Pierce and Turner [2000]. It has since been applied to many type systems with advanced features. The alternative application mode introduced in Chapter 3 enables a variant of bidirectional type checking. There are many other efforts to refine bidirectional type checking.

Colored local type inference [Odersky et al. 2001] allows partial type information to be propagated, by distinguishing inherited types (known from the context) and synthesized types (inferred from terms). A similar distinction is achieved in Dunfield and Krishnaswami [2013] by manipulating type variables.

Tridirectional type checking [Dunfield and Pfenning 2004] is based on bidirectional type checking and has a rich set of property types including intersections, unions and quantified dependent types, but without parametric polymorphism. Tridirectional type checking has a new direction for supporting type checking unions and existential quantification.

Greedy bidirectional polymorphism [Dunfield 2009] adopts a greedy idea from Cardelli [1993] on bidirectional type checking with higher-rank types, where type variables in instantiations are determined by their first constraint. In this way, they support some uses of impredicative polymorphism. However, the greediness also makes many obvious programs rejected.

A detailed survey of the development of bidirectional type checking is given by Dunfield and Krishnaswami [2020], which collect and explain the design principles of bidirectional type checking, and summarize past research related to bidirectional type checking.

8.3 Gradual Typing

The seminal paper by Siek and Taha [2006] is the first to propose gradual typing, which enables programmers to mix static and dynamic typing in a program by providing a mechanism to control which parts of a program are statically checked. The original proposal extends the simply typed lambda calculus by introducing the unknown type $?$ and replacing type equality with type consistency. Casts are introduced to mediate between statically and dynamically typed code. Later Siek and Taha [2007] incorporated gradual typing into a simple object oriented language, and showed that subtyping and consistency are orthogonal – an insight that partly inspired our work on GPC. We show that subtyping and consistency are orthogonal in a much richer type system with higher-rank polymorphism. Siek et al. [2009] explores the design space of different dynamic semantics for simply typed lambda calculus with casts and unknown types. In the light of the ever-growing popularity of gradual typing, and its somewhat murky theoretical foundations, Siek et al. [2015] felt the urge to have a complete formal characterization of what it means to be gradually typed. They proposed a set of criteria that provides important guidelines for designers of gradually typed languages. Cimini and Siek [2016] introduced the *Gradualizer*, a general methodology for generating gradual type systems from static type systems. Later they also develop an algorithm to generate dynamic semantics [Cimini and Siek 2017]. Garcia et al. [2016] introduced the AGT approach based on abstract interpretation. As we discussed, none of these approaches instructed us how to define consistent subtyping for polymorphic types.

There is some work on integrating gradual typing with rich type disciplines. Bañados Schwerter et al. [2014] establish a framework to combine gradual typing and effects, with which a static effect system can be transformed to a dynamic effect system or any intermediate blend. Jafery and Dunfield [2017] present a type system with *gradual sums*, which combines refinement and imprecision. We have discussed the interesting definition of *directed consistency* in Section 4.2. Castagna and Lanvin [2017] develop a gradual type system with intersection and union types, with consistent subtyping defined by following the idea of Garcia et al. [2016]. Eremondi et al. [2019] develop a gradual dependently-typed language, where compile-time normalization and run-time execution are distinguished to account for nontermination and failure. TypeScript [Bierman et al. 2014] has a distinguished dynamic type, written any, whose fundamental feature is that any type can be implicitly converted to and from any. Our treatment of the unknown type in Figure 4.6 is similar to their treatment of any. However, their type system does not have polymorphic types. Also, unlike our consistent subtyping which inserts runtime casts, in TypeScript, type information is erased after compilation so there are no runtime casts, which makes runtime type errors possible.

8.4 Gradual Type Systems with Explicit Polymorphism

Morris [1973] dynamically enforces parametric polymorphism and uses *sealing* functions as the dynamic type mechanism. More recent works on integrating gradual typing with parametric polymorphism include the dynamic type of Abadi et al. [1995] and the *Sage* language of Gronski et al. [2006]. None of these has carefully studied the interaction between statically and dynamically typed code.

Ahmed et al. [2009] proposed λB that extends the blame calculus [Wadler and Findler 2009] to incorporate polymorphism. The key novelty of their work is to use dynamic sealing to enforce parametricity. As such, they end up with a sophisticated dynamic semantics. Later, Ahmed et al. [2017] prove that with more restrictions, λB satisfies parametricity. Compared to their work, our GPC type system can catch more errors earlier since, as we argued, their notion of *compatibility* is too permissive. For example, the following is rejected (more precisely, the corresponding source program never gets elaborated) by our type system:

$$(\lambda x : ?.\, x + 1) : \forall a.\, a \to a \rightsquigarrow \langle ? \to \mathsf{Int} \hookrightarrow \forall a.\, a \to a \rangle (\lambda x : ?.\, x + 1)$$

while the type system of λB would accept the translation, though at runtime, the program would result in a cast error as it violates parametricity. We emphasize that it is the combination of our powerful type system together with the powerful dynamic semantics of λB that makes it possible to have implicit higher-rank polymorphism in a gradually typed setting. Devriese et al. [2017] proved that embedding of System F terms into λB is not fully abstract. Igarashi et al. [2017] also studied integrating gradual typing with parametric polymorphism. They proposed System F_G, a gradually typed extension of System F, and System F_C, a new polymorphic blame calculus. As has been discussed extensively, their definition of type consistency does not apply to our setting (implicit polymorphism). All of these approaches mix consistency with subtyping to some extent, which we argue should be orthogonal. On a side note, it seems that our calculus can also be safely translated to System F_C. However we do not understand all the tradeoffs involved in the choice between λB and System F_C as a target.

Recently, Toro et al. [2019] applied AGT to designing a gradual language with explicit parametric polymorphism, claiming that graduality and parametricity are inherently incompatible. However, later New et al. [2019] show that by modifying System F's syntax to make the sealing visible, both graduality and parametricity can be achieved.

8.5 Gradual Type Inference

Siek and Vachharajani [2008] studied unification-based type inference for gradual typing, where they show why three straightforward approaches fail to meet their design goals. One of their main observations is that simply ignoring dynamic types during unification does not work. Therefore, their type system assigns unknown types to type variables and infers gradual types, which results in a complicated type system and inference algorithm. In our algorithm presented in Chapter 5, comparisons between existential variables and unknown types are emphasized by the distinction between static existential variables and gradual existential variables. By syntactically refining unsolved gradual existential variables with unknown types, we gain a similar effect as assigning unknown types, while keeping the algorithm relatively simple. Garcia and Cimini [2015] presented a new approach where gradual type inference only produces static types, which is adopted in our type system. They also deal with let-polymorphism (rank 1 types). They proposed the distinction between static and gradual type parameters, which inspired our extension to restore the dynamic gradual guarantee. Although those existing works all involve gradual types and inference, none of these works deal with higher-rank implicit polymorphism.

8.6 Haskell and GHC

The Glasgow Haskell Compiler. The systems we present in Chapter 7 are inspired by the algorithms implemented in GHC. However, our goal in the design of these systems is to produce a sound and (nearly) complete pair of specification and implementation, not simply to faithfully record what is implemented. We have identified ways that the GHC implementation can improve in the future. For example, GHC quantifies over local scopes as *specified* where we believe they should be *inferred*; and the tight connection in our system between unification and promotion may improve upon GHC's approach, which separates the two. The details of the relationship between our work and GHC (including a myriad of ways our design choices differ in small ways from GHC's) appear in Appendix C.2.

Type inference in Haskell. Type inference in Haskell is inspired by Damas and Milner [1982] and Pottier and Rémy [2005], extended with various type features, including higher rank polymorphism [Peyton Jones et al. 2007] and local assumptions [Schrijvers et al. 2009; Simonet and Pottier 2007; Vytiniotis et al. 2011], among others. However, none of these works describe an inference algorithm for datatypes, nor do they formalize type variables of varying kinds or polymorphic recursion.

Dependent Haskell. Our PolyKinds system merges types and kinds, a key feature of *Dependent Haskell* (DH) [Eisenberg 2016; Gundry 2013; Weirich et al. 2013, 2017]. There is ongoing work dedicated to its implementation [Xie and Eisenberg 2018]. The most recent work by Weirich et al. [2019] integrates *roles* Breitner et al. [2016] with dependent types. Our work is the first presentation of unification for DH, and our system may be useful in designing DH's term-level type inference.

Polymorphic recursion. Mycroft [1984] presented a semi-algorithm for polymorphic recursion. Jim [1996] and Damiani [2003] studied typing rules for recursive definitions based on rank-2 intersection types. Comini et al. [2008] studied recursive definitions in a type system that corresponds to the abstract interpreter in Gori and Levi [2002, 2003]. Our system PolyKinds does not infer polymorphic recursion; instead, we exploit kind annotations to guide the acceptance of polymorphic recursion.

Constraint-solving approaches. Many systems (e.g. [Pottier and Rémy 2005]) adopt a modular presentation of type inference, which consists of a constraint generator and a constraint solver. For simplicity, we have presented an eager unification algorithm instead of using a separate constraint solver. However, we believe changing to a constraint-solving approach should not change any of our main results. Xie et al. [2019b] considers this point further.

8.7 Unification with dependent types

While full higher-order unification is undecidable [Goldfarb 1981], the *pattern* fragment [Miller 1991] is a well-known decidable fragment. Much literature [Abel and Pientka 2011; Gundry and McBride 2013; Reed 2009] is built upon the pattern fragment.

Unification in a dependently typed language features *heterogeneous constraints*. To prove correctness, Reed [2009] used a weaker invariant on homogeneous equality, *typing modulo*, which states that two sides are well typed up to the equality of the constraint yet to be solved. Gundry and McBride [2013] observed the same problem, and use *twin variables* to explicitly represent the same variable at different types, where twin variables are eliminated once the heterogeneous constraint is solved. In both approaches the well-formedness of a constraint depends on other constraints. Cockx et al. [2016] proposed a proof-relevant unification that keeps track of the dependencies between equations. Different from their approaches, our algorithm unifies the kinds when solving unification variables. This guarantees that our unification always outputs well-formed solutions.

Ziliani and Sozeau [2015] present the higher-order unification algorithm for CIC, the base logic of Coq. They favor syntactic equality by trying first-order unification, as they argue the first-order solution gives the most *natural* solution. However, they omit a correctness proof for their algorithm. Coen [2004] also considers first-order unification, but only the soundness lemma is proved. Different from their systems, our system is based on the novel promotion judgment, and correctness including soundness and termination is proved.

The technique of *suspended substitutions* [Eisenberg 2016; Gundry and McBride 2013] is widely adopted in unification algorithms. Our system provides a design alternative, our *quantification check*. Choosing between suspended substitutions and the quantification check is a user-facing language design decision, as suspended substitutions can accept some more programs. The quantification check means that the kind of a locally quantified variable a must be fully determined in a's scope; it may *not* be influenced by usage sites of the construct that depends on a. Suspended substitutions relax this restriction. We conjecture that suspended substitutions can yield a complete algorithm. However, that mechanism is complex. Moreover, unification based on suspended substitutions is only decidable for the pattern fragment. Our system, in contrast, avoids all the complication introduced by suspended substitutions through its quantification check. Our unification terminates for all inputs, preserving backward compatibility to Hindley-Milner-style inference. Although we reject the definition of T (Section 7.7.2), we can solve more constraints outside the pattern fragment. We conjecture that those constraints are much more common than definitions like T. Suspended substitutions often come with a *pruning* process [Abel and Pientka 2011], which produces a valid solution before solving a unification variable. Our promotion process has a similar effect.

Homogeneous kind-preserving unification. Jones [1995] proposed a homogeneous kind-preserving unification between two types. Kinds κ are defined only as $\star$ or $\kappa_1 \to \kappa_2$. As the kind system is much simpler, kind-preserving unification $\sim_\kappa$ is simply subscripted by the kind, and working out the kinds is straightforward. Our unification subsumes Jones's algorithm.

Context extension. Our approach of recording unification variables and their solutions in the contexts is inspired by Gundry et al. [2010] and Dunfield and Krishnaswami [2013]. Gundry and McBride [2013] applied the approach to unification in dependent types, where the context also records constraints; constraints also appear in context in Eisenberg [2016]. Further, in PolyKinds, we extend the context extension approach with local scopes, supporting groups of order-insensitive variables.

9 Summary and Future Directions

In summary, this dissertation has pushed the research on predicative implicit higher-rank polymorphism further, and we believe that contributions in this dissertation can be used to guide the continued evolution of (functional) programming language design and implementations. Specifically, with the new bidirectional type checking algorithm using the application mode, we were able to type-check programs that traditional type inference algorithms cannot, and thus provide new insights for inference algorithm design with bidirectional type checking. With the integration of higher-rank polymorphism and gradual typing, we provided a step forward in gradualizing modern functional programming languages like Haskell. Moreover, the work on *type promotion* simplified type inference algorithms with tricky dependency and scoping issues, and the kind inference for datatypes presented a first known, detailed account of datatypes, which can serve as a guide for future development of datatypes.

In this section we discuss some future directions we would like to pursue.

9.1 Dependent Type Systems with Application mode

The application mode 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.

9.2 Type Inference for Intersection Type Systems

Another type system that could possibly benefit from the application mode is intersection type systems [Coppo et al. 1979; Pottinger 1980; Salle 1978]. In particular, we consider intersection type systems with an explicit *merge operator* [Dunfield 2014]. In such a system, we can construct terms of an intersection type, like $1 \,,,\, \mathsf{true}$ of type $\mathsf{Int} \,\&\, \mathsf{Bool}$. Thanks to *subtyping*, a term of type $\mathsf{Int} \,\&\, \mathsf{Bool}$ can also be used as if it had type Int, or as if it had type Bool. Calculi with *disjoint intersection types* [Alpuim et al. 2017; Bi et al. 2019; Oliveira et al. 2016] incorporate a *coherent* merge operator. In such calculi the merge operator can merge two terms with *arbitrary* types as long as their types are disjoint; disjointness conflicts are reported as type-errors. As illustrated by Xie et al. [2020], the expressive power of disjoint intersection types can encode diverse programming language features, promising an economy of theory and implementation.

Disjoint intersection types also pose challenges to type inference. Supposing that we have $\mathsf{succ} : \mathsf{Int} \to \mathsf{Int}$ and $\mathsf{not} : \mathsf{Bool} \to \mathsf{Bool}$, consider the following term:

$$(\mathsf{succ} \,,,\, \mathsf{not})\ 3$$

We expect the expression to type-check, as according to subtyping, the term $(\mathsf{succ} \,,,\, \mathsf{not})$ of type $(\mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Bool})$ can also be used as type $\mathsf{Int} \to \mathsf{Int}$. Thus we expect typing to automatically pick succ and apply it to 3. To this end, we need to push the type information of the argument (3) into the function $(\mathsf{succ} \,,,\, \mathsf{not})$.

Future work is required to explore how well the application mode can be used for type inference in intersection type systems, and whether it can be integrated with the distributivity subtyping rules of intersection types [Bi et al. 2019].

9.3 Gradualizing Type Classes

In Section 4.1.2, we discussed about gradualizing modern functional programming languages like Haskell. One of its core abstraction features in Haskell is *type classes*. Type classes [Wadler and Blott 1989] were initially introduced in Haskell to make ad-hoc overloading less ad-hoc, and since then have been adopted in many languages including Mercury [Henderson et al. 1996], Coq [Sozeau and Oury 2008], PureScript [Freeman 2017], and Lean [de Moura et al. 2015]. An interesting future direction then is to gradualizing type classes.

Consider again the example used in Section 4.1.2:

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

While `f : ∀a. a → a` is of course a valid type annotation, it unfortunately rules out many valid arguments that may have type class constraints in their types, e.g.,

```
show     :: Show a ⇒ a → String

(\f :: ∀a. a → a. (f 1, f 'a')) show     -- rejected
```

With gradual typing, if we annotation `f` with the the unknown type ?, we expect that the following expression can type-check.

```
(\f :: ?. (f 1, f 'a')) show
```

However, a nontrivial challenge in gradualizing type classes is that the dynamic semantics of type classes is not expressed directly but rather by type-directed elaboration into a simpler language without type classes. Thus the dynamic semantics of type classes is given indirectly as the dynamic semantics of their elaborated forms. Consider `show` as an example. The *dictionary-passing* elaboration of type-classes translates the type of `show` into the following one, supposing `ShowD` is the dictionary type of the type class `show`.

```
show :: ShowD a → a → String
```

Now with the unknown type, we cannot predict how to elaborate the original expression. In particular, if `f` is applied to `show`, it means that `f` needs to be elaborated into a function that actually takes two arguments, first the dictionary and then the argument.

```
(\f. (f showInt 1, f showChar 'a')) show
```

This kind of uncertainty in elaboration brings extra complexity and may interact with explicit casts in the target blame calculi.

9.4 Generalized Algebraic Datatypes (GADTs)

A natural extension of PolyKinds is to include GADTs. We have briefly discussed GADTs in Section 7.8.2. In particular, we are interested in finding the right formalization of GADTs.

Haskell's *syntax* for GADT declarations is quite troublesome. Consider these examples:

```
data R a where
  MkR :: b → R b
```

```
data S a where
  MkS :: S b
```

```
data T a where
  MkT :: ∀(k :: ⋆) (b :: k). T b
```

In GHC's implementation of GADTs, any variables declared in the header (between **data** and **where**) *do not scope*. In all the examples above, the type variable a does not scope over the constructor declarations. This is why we have written the variable b in those types, to make it clear that b is distinct from a. We could have written a—it would still be a distinct a from that in the header—but it would be more confusing.

The question is: how do we determine the kind of the parameter to the datatype? One possibility is to look only in the header. In all cases above, we would infer no constraints and would give each type a kind of $\forall (k :: \star).\ k \to \star$. This is unfortunate, as it would make R a kind-indexed GADT: the MkR constructor would carry a proof that the kind of its type parameter is $\star$. This, in turn, wreaks havoc with type inference, as it is hard to infer the result type of a pattern-match against a GADT Vytiniotis et al. [2011].

Furthermore, this approach might accept *more* programs than the user wants. Consider this definition:

```
data P a where
  MkP1 :: b → P b
  MkP2 :: f a → P f
```

Does the user want a kind-indexed GADT, noting that b and f have different kinds? Or would the user want this rejected? If we make the fully general kind $\forall k.\ k \to \star$ for P, this would be accepted, perhaps surprising users.

It thus seems we wish to look at the data constructors when inferring the kind of the datatype. The challenge in looking at data constructors is that their variables are *locally* bound. In MkR and MkS, we implicitly quantify over b. In MkR, we discover that $b :: \star$, and thus that R must have kind $\star \to \star$. In MkS, we find no constraints on b's kind, and thus no constraints on S's argument's kind, and so we can generalize to get $S :: \forall (k :: \star).\ k \to \star$. Let us now examine MkT: it explicitly brings k and b into scope. Thus, the argument to T has *local* kind k. It would be impossible to unify the kind of T's argument—call it $\widehat{\alpha}$—with k, because k would be bound to the *right* of $\widehat{\alpha}$ in an inference context. Thus it seems we would reject T.

Our conclusion here is that the design of GADTs in GHC/Haskell is flawed: the type variables mentioned in the header should indeed scope over the constructors. This would mean we could reject T: if the user wanted to explicitly make T polymorphically kinded, they could do so right in the header. So one possible application of our work is to apply our insights in the scoping (order in the context) and unification into formalizing GADTs.

Bibliography

[Citing pages are listed after each reference.]

Martin Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. 1995. Dynamic Typing in Polymorphic Languages. *Journal of Functional Programming* 5, 1 (1995), 111–130. [cited on page 174]

Andreas Abel and Brigitte Pientka. 2011. Higher-order dynamic pattern unification for dependent types and records. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 10–26. [cited on pages 176 and 177]

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* (Genova, Italy) *(STOP '09)*. Association for Computing Machinery, New York, NY, USA, 1–13. `https://doi.org/10.1145/1570506.1570507` [cited on pages 11, 68, 70, 78, 80, 81, 83, 107, and 174]

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, With and Without Types. In *Proceedings of the 22nd International Conference on Functional Programming*. [cited on page 174]

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint polymorphism. In *European Symposium on Programming (ESOP)*. [cited on page 180]

P. B. Andrews. 1971. Resolution in type Theory. *Journal of Symbolic Logic* 36 (1971), 414–432. [cited on page 137]

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 24]

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A Theory of Gradual Effect Systems. In *Proceedings of the 19th International Conference on Functional Programming*. [cited on pages 62 and 173]

Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive disjoint polymorphism for compositional programming. In *European Symposium on Programming (ESOP)*. [cited on page 180]

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *Proceedings of the 28th European Conference on Object-Oriented Programming*. [cited on pages 8, 61, and 173]

Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C#. In *Proceedings of the European Conference on Object-Oriented Programming*. [cited on page 61]

Richard S. Bird and Lambert Meertens. 1998. Nested datatypes. In *LNCS 1422: Proceedings of Mathematics of Program Construction*, Johan Jeuring (Ed.). Springer-Verlag, Marstrand, Sweden, 52–67. `http://www.cs.ox.ac.uk/people/richard.bird/online/BirdMeertens98Nested.pdf` [cited on page 138]

Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure. In *Programming Languages and Systems*. [cited on pages 8 and 61]

Joachim Breitner, Richard A Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016. Safe zero-cost coercions for Haskell. *Journal of Functional Programming* 26 (2016). [cited on page 176]

L. Cardelli. 1986. *A polymorphic lambda-calculus with Type:Type*. Technical Report 10. SRC. [cited on page 136]

Luca Cardelli. 1993. *An implementation of FSub*. Technical Report. Research Report 97, Digital Equipment Corporation Systems Research Center. [cited on pages 89 and 172]

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 62, 74, 99, and 173]

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated type synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia) *(ICFP '05)*. ACM, New York, NY, USA, 241–253. `https://doi.org/10.1145/1086365.1086397` [cited on page 168]

Gang Chen. 2003. Coercive Subtyping for the Calculus of Constructions *(POPL '03)*. 10. [cited on page 53]

Alonzo Church. 1941. *The calculi of lambda-conversion*. Number 6. Princeton University Press. [cited on page 65]

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 61, 74, 82, and 173]

Matteo Cimini and Jeremy G. Siek. 2017. Automatically Generating the Dynamic Semantics of Gradually Typed Languages. In *Proceedings of the 44th Symposium on Principles of Programming Languages*. [cited on pages 61 and 173]

Jesper Cockx, Dominique Devriese, and Frank Piessens. 2016. Unifiers as equivalences: proof-relevant unification of dependently typed data. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. ACM, New York, NY, USA, 270–283. `https://doi.org/10.1145/2951913.2951917` [cited on page 176]

Claudio Sacerdoti Coen. 2004. *Mathematical knowledge management and interactive theorem proving*. Ph.D. Dissertation. University of Bologna, 2004. Technical Report UBLCS 2004-5. [cited on page 177]

Marco Comini, Ferruccio Damiani, and Samuel Vrech. 2008. On polymorphic recursion, type systems, and abstract interpretation. In *International Static Analysis Symposium*. Springer, 144–158. [cited on page 176]

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Patrick Sallé. 1979. Functional characterization of some semantic equalities inside λ-calculus. In *International Colloquium on Automata, Languages, and Programming*. Springer, 133–146. [cited on page 180]

Thierry Coquand. 1996. An algorithm for type-checking dependent types. *Science of Computer Programming* 26, 1-3 (1996), 167–177. [cited on page 24]

Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. 1958. *Combinatory logic*. Vol. 1. North-Holland Amsterdam. [cited on pages 64 and 65]

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* (Albuquerque, New Mexico) *(POPL '82)*. Association for Computing Machinery, New York, NY, USA, 207–212. `https://doi.org/10.1145/582153.582176` [cited on pages 3, 4, 15, 18, 133, 136, and 175]

Ferruccio Damiani. 2003. Rank 2 intersection types for local definitions and conditional expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 4 (2003), 401–451. [cited on page 176]

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 24 and 99]

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover. (2015). [cited on page 181]

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2017. Parametricity versus the universal type. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 38. [cited on page 174]

Joshua Dunfield. 2009. Greedy Bidirectional Polymorphism. In *Workshop on ML*. [cited on page 172]

Joshua Dunfield. 2014. Elaborating intersection and union types. *Journal of Functional Programming (JFP)* 24, 2-3 (2014), 133–165. [cited on page 180]

Jana Dunfield and Neel Krishnaswami. 2020. Bidirectional Typing. arXiv:1908.05839 [cs.PL] [cited on page 172]

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* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 429–442. `https://doi.org/10.1145/2500365.2500582` [cited on pages 3, 6, 7, 12, 23, 25, 28, 30, 61, 80, 85, 95, 109, 114, 119, 120, 124, 127, 142, 148, 166, 171, 172, and 177]

Joshua Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types. *Proc. ACM Program. Lang.* 3, POPL, Article 9 (Jan. 2019), 28 pages. `https://doi.org/10.1145/3290322` [cited on page 171]

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 24, 34, 35, 36, and 172]

Richard A Eisenberg. 2016. *Dependent types in haskell: Theory and practice*. Ph.D. Dissertation. University of Pennsylvania. [cited on pages 168, 176, 177, and 211]

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 671–683. `https://doi.org/10.1145/2535838.2535856` [cited on page 168]

Richard A Eisenberg, Stephanie Weirich, and Hamidhasan G Ahmed. 2016. Visible type application. In *European Symposium on Programming*. Springer, 229–254. [cited on page 138]

Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019. Approximate Normalization for Gradual Dependent Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 88 (July 2019), 30 pages. `https://doi.org/10.1145/3341692` [cited on page 173]

Phil Freeman. 2017. *PureScript by Example*. Leanpub. `https://leanpub.com/purescript`. [cited on page 181]

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* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 303–315. `https://doi.org/10.1145/2676726.2676992` [cited on pages 12, 27, 85, 89, 103, 104, 109, 148, and 175]

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 61, 63, 67, 68, 73, 74, 75, and 173]

Andrew Gill, John Launchbury, and Simon L Peyton Jones. 1993. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*. 223–232. [cited on page 5]

Jean-Yves Girard. 1986. The System F of Variable Types, Fifteen Years Later. *Theoretical computer science* 45 (1986), 159–192. [cited on pages 5, 23, and 51]

Warren D Goldfarb. 1981. The undecidability of the second-order unification problem. *Theoretical Computer Science* 13, 2 (1981), 225–230. [cited on pages 137 and 176]

Roberta Gori and Giorgio Levi. 2002. An experiment in type inference and verification by abstract interpretation. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 225–239. [cited on page 176]

Roberta Gori and Giorgio Levi. 2003. Properties of a type abstract interpreter. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 132–145. [cited on page 176]

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N Freund, and Cormac Flanagan. 2006. Sage: Hybrid Checking for Flexible Specifications. In *Scheme and Functional Programming Workshop*. [cited on page 174]

Adam Gundry and Conor McBride. 2013. A tutorial implementation of dynamic pattern unification. *Unpublished draft* (2013). [cited on pages 176 and 177]

Adam Gundry, Conor McBride, and James McKinna. 2010. Type inference in context. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*. ACM, 43–54. [cited on pages 12, 119, 120, 121, 122, and 177]

Adam Michael Gundry. 2013. *Type inference, Haskell and dependent types*. Ph.D. Dissertation. University of Strathclyde. [cited on pages 137, 168, and 176]

Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, and Chris Speirs. 1996. *The Mercury Language Reference Manual*. Technical Report. [cited on page 181]

Fritz Henglein. 1993. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.* 15, 2 (April 1993), 253–289. `https://doi.org/10.1145/169701.169692` [cited on pages 138 and 216]

J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969), 29–60. [cited on pages 3, 4, 15, and 133]

G. Huet. 1973. A unification algorithm for typed lambda calculus. *Theoretical Computer Science* 1, 1 (1973), 27–57. [cited on page 137]

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 68, 70, 75, 103, 107, and 174]

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 pages 62, 74, and 173]

Trevor Jim. 1996. What are principal typings and what are they good for?. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 42–53. [cited on page 176]

Mark P Jones. 1995. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming* 5, 1 (1995), 1–35. [cited on pages 134, 137, 167, and 177]

Mark P. Jones. 1996. Using Parameterized Signatures to Express Modular Structure. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL ' 96)*. 68–78. `https://doi.org/10.1145/237721.237731` [cited on page 5]

Mark P. Jones. 1999. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop (Haskell '99)*, Erik Meijer (Ed.). Paris, France, pp. 9–22. University of Utrecht Technical Report UU-CS-1999-28. [cited on page 150]

Mark P Jones. 2000. Type classes with functional dependencies. In *European Symposium on Programming*. Springer, 230–244. [cited on page 67]

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 19]

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 67]

Csongor Kiss, Susan Eisenbach, Tony Field, and Simon Peyton Jones. 2019. Higher-order type-level programming in Haskell. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming (ICFP 2019)*. ACM. [cited on page 167]

Didier Le Botlan and Didier Rémy. 2003. MLF: Raising ML to the Power of System F *(ICFP '03)*. 12. [cited on pages 6 and 171]

Didier Le Botlan and Didier Rémy. 2009. Recasting MLF. *Information and Computation* 207, 6 (2009), 726–785. [cited on page 171]

Jukka Lehtosalo et al. 2006. Mypy. `http://www.mypy-lang.org/` [cited on pages 8 and 61]

Daan Leijen. 2009. Flexible Types: Robust Type Inference for First-class Polymorphism *(POPL '09)*. 12. [cited on pages 6 and 171]

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 24]

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 80]

Conor McBride. 2002. Faking it Simulating dependent types in Haskell. *Journal of functional programming* 12, 4-5 (2002), 375–392. [cited on page 67]

McCracken. 1984. The typechecking of programs with implicit type structure. In *Lecture Notes in Computer Science (Semantics of Data Types)*, Vol. 173. [cited on page 5]

Dale Miller. 1991. Unification of simply typed lambda-terms as logic programming. (1991). [cited on page 176]

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375. [cited on page 15]

James H. Morris, Jr. 1973. Types Are Not Sets. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. ACM, New York, NY, USA, 120–124. `https://doi.org/10.1145/512927.512938` [cited on page 174]

James Hiram Morris Jr. 1969. *Lambda-calculus models of programming languages.* Ph.D. Dissertation. Massachusetts Institute of Technology. [cited on page 81]

Alan Mycroft. 1984. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*. Springer, 217–228. [cited on page 176]

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2009. Non-parametric Parametricity. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. ACM, New York, NY, USA, 135–148. `https://doi.org/10.1145/1596550.1596572` [cited on page 80]

Max S. New, Dustin Jamner, and Amal Ahmed. 2019. Graduality and Parametricity: Together Again for the First Time. *Proc. ACM Program. Lang.* 4, POPL, Article 46 (Dec. 2019), 32 pages. `https://doi.org/10.1145/3371114` [cited on page 174]

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* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 54–67. `https://doi.org/10.1145/237721.237729` [cited on pages 3, 7, 15, 18, 19, 20, 68, 133, and 171]

Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored Local Type Inference *(POPL '01)*. 13. [cited on pages 37 and 172]

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*. [cited on page 180]

Michel Parigot. 1992. Recursive programming with proofs. *Theoretical Computer Science* 94, 2 (1992), 335–356. [cited on pages 64 and 65]

Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press. [cited on page 135]

Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: exploring the design space. In *Haskell workshop*, Vol. 1997. [cited on page 67]

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 3, 6, 7, 15, 18, 24, 28, 29, 35, 38, 39, 45, 46, 47, 53, 80, 133, 137, 166, 167, 171, and 175]

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA) *(ICFP '06)*. ACM, New York, NY, USA, 50–61. `https://doi.org/10.1145/1159803.1159811` [cited on pages 133 and 167]

Benjamin C Pierce. 2002. *Types and programming languages*. [cited on page 80]

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 7, 11, 23, 56, and 172]

François Pottier and Didier Rémy. 2005. The essence of ML type inference. *Advanced Topics in Types and Programming Languages* (2005). [cited on pages 175, 176, and 211]

Garrel Pottinger. 1980. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* (1980), 561–577. [cited on page 180]

Jason Reed. 2009. Higher-order constraint simplification in dependent type theory. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*. ACM, 49–56. [cited on page 176]

Didier Rémy. 2005. Simple, Partial Type-inference for System F Based on Type-containment *(ICFP '05)*. 14. [cited on page 172]

Didier Rémy and Boris Yakobowski. 2008. From ML to MLF: Graphic Type Constraints with Efficient Type Inference *(ICFP '08)*. 12. [cited on page 171]

John C Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium*. Springer, 408–425. [cited on page 5]

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Proceedings of the IFIP 9th World Computer Congress*. [cited on page 80]

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 81]

Patrick Salle. 1978. Une Extension De La Theorie Des Types En lambda-Calcul. In *Proceedings of the Fifth Colloquium on Automata, Languages and Programming*. Springer-Verlag, London, UK, UK, 398–410. [cited on page 180]

Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. ACM, New York, NY, USA, 341–352. `https://doi.org/10.1145/1596550.1596599` [cited on page 175]

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A Quick Look at Impredicativity. *Proc. ACM Program. Lang.* 4, ICFP, Article 89 (Aug. 2020), 29 pages. `https://doi.org/10.1145/3408971` [cited on pages 6, 22, and 172]

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 783–796. `https://doi.org/10.1145/3192366.3192389` [cited on pages 6, 22, and 172]

Paula Severi and Erik Poll. 1994. Pure Type Systems with Definitions. *Logical Foundations of Computer Science* (1994), 316–328. [cited on page 179]

Jeremy Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the design space of higher-order casts. In *European Symposium on Programming*. 17–31. [cited on page 173]

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings of the 2006 Scheme and Functional Programming Workshop*. [cited on pages 8, 11, 61, 101, and 173]

Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *European Conference on Object-Oriented Programming*. [cited on pages 6, 9, 11, 61, 62, 63, 67, 70, 73, 98, 99, 101, and 173]

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-based Inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*. [cited on page 175]

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 9, 11, 12, 75, 78, 82, 83, and 173]

Vincent Simonet and François Pottier. 2007. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 1 (2007), 1. [cited on page 175]

Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *TPHOLs '08* (Montreal, P.Q., Canada). Springer-Verlag, 278–293. [cited on page 181]

Sam Tobin-Hochstadt. 2019. Gradual Typing from Theory to Practice. `https://blog.sigplan.org/2019/07/12/gradual-typing-theory-practice/` [cited on page 8]

Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual Parametricity, Revisited. *Proc. ACM Program. Lang.* 3, POPL, Article 17 (Jan. 2019), 30 pages. `https://doi.org/10.1145/3290330` [cited on page 174]

Julien Verlaguet. 2013. Facebook: Analyzing PHP statically. In *Proceedings of Commercial Users of Functional Programming*. [cited on page 61]

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Proceedings of the 10th Symposium on Dynamic languages*. [cited on pages 8 and 61]

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn (X) Modular type inference with local assumptions. *Journal of functional programming* 21, 4-5 (2011), 333–412. [cited on pages 137, 166, 167, 168, 175, 182, 211, and 212]

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2006. Boxy types: inference for higher-rank types and impredicativity. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA) *(ICFP '06)*. ACM, New York, NY, USA. `https://doi.org/10.1145/1159803.1159811` [cited on page 37]

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: First-class Polymorphism for Haskell *(ICFP '08)*. 12. [cited on page 172]

P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *POPL '89*. ACM. [cited on page 181]

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*. [cited on page 174]

Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. 2019. A Role for dependent types in Haskell. *Proc. ACM Program. Lang.* 3, ICFP, Article 101 (July 2019), 29 pages. `https://doi.org/10.1145/3341705` [cited on page 176]

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. ACM, New York, NY, USA, 275–286. `https://doi.org/10.1145/2500365.2500599` [cited on pages 134, 136, and 176]

Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A Eisenberg. 2017. A specification for dependent types in Haskell. In *Proceedings of the 22th ACM SIGPLAN International Conference on Functional Programming (ICFP '17)*. ACM. [cited on page 176]

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 6, 7, 19, and 91]

Thomas Winant, Dominique Devriese, Frank Piessens, and Tom Schrijvers. 2014. Partial type signatures for haskell. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 17–32. [cited on page 210]

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (Nov. 1994), 38–94. https://doi.org/10.1006/inco.1994.1093 [cited on page 51]

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. ACM, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150 [cited on page 167]

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* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560 [cited on pages 24 and 179]

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 pages 13 and 27]

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 pages 13 and 27]

Ningning Xie, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers. 2020. Row and Bounded Polymorphism via Disjoint Polymorphism. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs))*, Robert Hirschfeld and Tobias Pape (Eds.), Vol. 166. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 27:1–27:30. https://doi.org/10.4230/LIPIcs.ECOOP.2020.27 [cited on page 180]

Ningnign Xie and Richard A Eisenberg. 2018. Coercion Quantification. In *Haskell Implementors' Workshop*. [cited on page 176]

Ningning Xie, Richard A Eisenberg, and Bruno CDS Oliveira. 2019b. Kind Inference for Datatypes: Technical Supplement. *arXiv preprint arXiv:1911.06153* (2019). `https://arxiv.org/abs/1911.06153` [cited on pages 136 and 176]

Ningning Xie, Richard A. Eisenberg, and Bruno C. d. S. Oliveira. 2019c. 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 13]

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 pages 13 and 132]

Ningning Xie and Bruno C d S Oliveira. 2018. Let Arguments Go First. In *European Symposium on Programming*. Springer, 272–299. [cited on pages 13, 38, and 57]

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(TLDI '12)*. ACM, New York, NY, USA, 53–66. `https://doi.org/10.1145/2103786.2103795` [cited on pages 134, 136, and 210]

Beta Ziliani and Matthieu Sozeau. 2015. A Unification Algorithm for Coq Featuring Universe Polymorphism and Overloading. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. ACM, New York, NY, USA, 179–191. `https://doi.org/10.1145/2784731.2784751` [cited on page 176]

Part VI

Technical Appendix

A Full Rules for Algorithmic AP

$\boxed{(S_1, N_1) \vdash^{AP} \sigma <: \sigma_2 \hookrightarrow (S_2, N_2)}$ *(Algorithmic Subtyping)*

$$\frac{S_0 \vdash^{AP} \tau_1 \approx \tau_2 \hookrightarrow S_1}{(S_0, N_0) \vdash^{AP} \tau_1 <: \tau_2 \hookrightarrow (S_1, N_0)} \textsc{ ap-a-s-mono}$$

$$\frac{\begin{array}{c}(S_0, N_0) \vdash^{AP} \sigma \rhd \sigma_3 \to \sigma_4 \hookrightarrow (S_1, N_1) \\ (S_1, N_1) \vdash^{AP} \sigma_3 <: \sigma_1 \hookrightarrow (S_2, N_2) \qquad (S_2, N_2) \vdash^{AP} \sigma_2 <: \sigma_4 \hookrightarrow (S_3, N_3)\end{array}}{(S_0, N_0) \vdash^{AP} \sigma_1 \to \sigma_2 <: \sigma \hookrightarrow (S_3, N_3)} \textsc{ ap-a-s-arrowL}$$

$$\frac{\begin{array}{c}(S_0, N_0) \vdash^{AP} \sigma \rhd \sigma_1 \to \sigma_2 \hookrightarrow (S_1, N_1) \\ (S_1, N_1) \vdash^{AP} \sigma_3 <: \sigma_1 \hookrightarrow (S_2, N_2) \qquad (S_2, N_2) \vdash^{AP} \sigma_2 <: \sigma_4 \hookrightarrow (S_3, N_3)\end{array}}{(S_0, N_0) \vdash^{AP} \sigma <: \sigma_3 \to \sigma_4 \hookrightarrow (S_3, N_3)} \textsc{ ap-a-s-arrowR}$$

$$\frac{(S_0, N_0) \vdash^{AP} \sigma_1[a \mapsto \widehat{\beta}] <: \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0\, \widehat{\beta}) \vdash^{AP} \forall a.\, \sigma_1 <: \sigma_2 \hookrightarrow (S_1, N_1)} \textsc{ ap-a-s-forallL}$$

$$\frac{(S_0, N_0) \vdash^{AP} \sigma_1 <: \sigma_2[a \mapsto b] \hookrightarrow (S_1, N_1) \qquad b \notin \textsc{fv}\,(S(\sigma_1)) \qquad b \notin \textsc{fv}\,(S(\forall a.\, \sigma_2))}{(S_0, N_0\, b) \vdash^{AP} \sigma_1 <: \forall a.\, \sigma_2 \hookrightarrow (S_1, N_1)} \textsc{ ap-a-s-forallR}$$

$\boxed{(S_1, N_1); \Sigma \vdash^{AP} \sigma <: \sigma_2 \hookrightarrow (S_2, N_2)}$ *(Algorithmic Application Subtyping)*

$$\frac{}{(S_0, N_0); \bullet \vdash^{AP} \sigma <: \sigma \hookrightarrow (S_0, N_0)} \textsc{ ap-a-as-empty}$$

$$\frac{(S_0, N_0); \Sigma, \sigma_3 \vdash^{AP} \sigma_1[a \mapsto \widehat{\beta}] <: \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0\, \widehat{\beta}); \Sigma, \sigma_3 \vdash^{AP} \forall a.\, \sigma_1 <: \sigma_2 \hookrightarrow (S_1, N_1)} \textsc{ ap-a-as-forall}$$

$$\frac{(S_0, N_0) \vdash^{AP} \sigma_3 <: \sigma_1 \hookrightarrow (S_1, N_1) \qquad (S_1, N_1); \Sigma \vdash^{AP} \sigma_2 <: \sigma_4 \hookrightarrow (S_2, N_2)}{(S_0, N_0); \Sigma, \sigma_3 \vdash^{AP} \sigma_1 \to \sigma_2 <: \sigma_3 \to \sigma_4 \hookrightarrow (S_2, N_2)}\ \textsc{ap-a-as-arrow}$$

$$\frac{\begin{array}{c}(S_0, N_0) \vdash^{AP} \tau \rhd \tau_1 \to \tau_2 \hookrightarrow (S_1, N_1) \\ (S_1, N_1); \Sigma, \sigma_3 \vdash^{AP} \tau_1 \to \tau_2 <: \sigma \hookrightarrow (S_2, N_2)\end{array}}{(S_0, N_0\, \widehat{\beta}); \Sigma, \sigma_3 \vdash^{AP} \tau <: \sigma \hookrightarrow (S_2, N_2)}\ \textsc{ap-a-as-mono}$$

$\boxed{(S_1, N_1) \vdash^{AP} \sigma \rhd \sigma_1 \to \sigma_2 \hookrightarrow (S_2, N_2)}$ *(Matching)*

$$\frac{S_0 \vdash^{AP} \widehat{\alpha} \approx \widehat{\alpha}_1 \to \widehat{\alpha}_2 \hookrightarrow S_1}{(S_0, N_0\, \widehat{\alpha}_1\, \widehat{\alpha}_2) \vdash^{AP} \widehat{\alpha} \rhd \widehat{\alpha}_1 \to \widehat{\alpha}_2 \hookrightarrow (S_1, N_0)}\ \textsc{ap-a-m-tvar}$$

$$\frac{}{(S_0, N_0) \vdash^{AP} \sigma_1 \to \sigma_2 \rhd \sigma_1 \to \sigma_2 \hookrightarrow (S_0, N_0)}\ \textsc{ap-a-m-arrow}$$

$\boxed{S_1 \vdash^{AP} \tau_1 \approx \tau_2 \hookrightarrow S_2}$ *(Unification)*

$$\frac{}{S_0 \vdash^{AP} \tau \approx \tau \hookrightarrow S_0}\ \textsc{ap-a-u-refl}$$

$$\frac{\widehat{\alpha} \in S_0 \qquad S_0 \vdash^{AP} S_0(\widehat{\alpha}) \approx \tau \hookrightarrow S_1}{S_0 \vdash^{AP} \widehat{\alpha} \approx \tau \hookrightarrow S_1}\ \textsc{ap-a-u-solvedEvarL}$$

$$\frac{\widehat{\alpha} \notin S_0 \qquad \widehat{\alpha} \notin \textsc{fv}\,(S_0(\tau))}{S_0 \vdash^{AP} \widehat{\alpha} \approx \tau \hookrightarrow [\widehat{\alpha} \mapsto S_0(\tau)] \cdot S_1}\ \textsc{ap-a-u-evarL}$$

$$\frac{\widehat{\alpha} \in S_0 \qquad S_0 \vdash^{AP} \tau \approx S_0(\widehat{\alpha}) \hookrightarrow S_1}{S_0 \vdash^{AP} \tau \approx \widehat{\alpha} \hookrightarrow S_1}\ \textsc{ap-a-u-solvedEvarR}$$

$$\frac{\widehat{\alpha} \notin S_0 \qquad \widehat{\alpha} \notin \textsc{fv}\,(S_0(\tau))}{S_0 \vdash^{AP} \tau \approx \widehat{\alpha} \hookrightarrow [\widehat{\alpha} \mapsto S_0(\tau)] \cdot S_1}\ \textsc{ap-a-u-evarR}$$

$$\frac{S_0 \vdash^{AP} \tau_1 \approx \tau_3 \hookrightarrow S_1 \qquad S_1 \vdash^{AP} \tau_2 \approx \tau_4 \hookrightarrow S_2}{S_0 \vdash^{AP} \tau_1 \to \tau_2 \approx \tau_3 \to \tau_4 \hookrightarrow S_2}\ \textsc{ap-a-u-arrow}$$

$\boxed{(S_1, N_1); \Psi \vdash^{AP} e \Rightarrow \sigma \hookrightarrow (S_2, N_2)}$ *(Algorithmic Typing Inference)*

$$\frac{\textsc{ap-a-inf-int}}{(S_0, N_0); \Psi \vdash^{AP} n \Rightarrow \mathsf{Int} \hookrightarrow (S_0, N_0)}$$

$$\textsc{ap-a-inf-lam}\quad \frac{(S_0, N_0); \Psi, x : \widehat{\beta} \vdash^{AP} e \Rightarrow \sigma \hookrightarrow (S_1, N_1)}{(S_0, N_0\, \widehat{\beta}); \Psi \vdash^{AP} \lambda x.\, e \Rightarrow \widehat{\beta} \to \sigma \hookrightarrow (S_1, N_1)}$$

$$\textsc{ap-a-inf-lamann}\quad \frac{(S_0, N_0); \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0); \Psi \vdash^{AP} \lambda x : \sigma_1.\, e \Rightarrow \sigma_1 \to \sigma_2 \hookrightarrow (S_1, N_1)}$$

$\boxed{(S_1, N_1); \Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma \hookrightarrow (S_2, N_2)}$ *(Algorithmic Typing Application Mode)*

$$\textsc{ap-a-app-var}\quad \frac{(x : \sigma_1) \in \Psi \qquad (S_0, N_0); \Sigma \vdash^{AP} \sigma_1 <: \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0); \Psi; \Sigma \vdash^{AP} x \Rightarrow \sigma_2 \hookrightarrow (S_1, N_1)}$$

$$\textsc{ap-a-app-lam}\quad \frac{(S_0, N_0); \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0); \Psi; \Sigma, \sigma_1 \vdash^{AP} \lambda x.\, e \Rightarrow \sigma_1 \to \sigma_2 \hookrightarrow (S_1, N_1)}$$

$$\textsc{ap-a-app-lamann}\quad \frac{(S_0, N_0) \vdash^{AP} \sigma_2 <: \sigma_1 \hookrightarrow (S_1, N_1) \qquad (S_1, N_1); \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_3 \hookrightarrow (S_2, N_2)}{(S_0, N_0); \Psi; \Sigma, \sigma_2 \vdash^{AP} \lambda x : \sigma_1.\, e \Rightarrow \sigma_2 \to \sigma_3 \hookrightarrow (S_2, N_2)}$$

$$\textsc{ap-a-app-app}\quad \frac{\begin{array}{c}(S_0, N_0); \Psi \vdash^{AP} e_2 \Rightarrow \sigma_1 \hookrightarrow (S_1, N_1\, \overline{a_i}^{\,i}) \qquad \overline{\widehat{\alpha}_i}^{\,i} = \textsc{fv}\,(S_1(\sigma_1)) - \textsc{fv}\,(S_1(\Psi)) \\ \sigma_2 = \forall \overline{a_i}^{\,i}.\, \sigma_1[\,\overline{\widehat{\alpha}_i \mapsto a_i}^{\,i}\,] \qquad (S_1, N_1); \Psi; \Sigma, \sigma_2 \vdash^{AP} e_1 \Rightarrow \sigma_2 \to \sigma_3 \hookrightarrow (S_2, N_2)\end{array}}{(S_0, N_0); \Psi; \Sigma \vdash^{AP} e_1\, e_2 \Rightarrow \sigma_3 \hookrightarrow (S_2, N_2)}$$

B The Extended Algorithmic GPC

B.1 Syntax

Expressions	e	$::=$	$x \mid n \mid \lambda x : \sigma.\, e \mid \lambda x.\, e \mid e_1\, e_2 \mid e : \sigma \mid \textbf{let}\, x = e_1\, \textbf{in}\, e_2$
Types	σ	$::=$	$\mathsf{Int} \mid a \mid \widehat{\alpha} \mid \sigma_1 \rightarrow \sigma_2 \mid \forall a.\, \sigma \mid ? \mid \mathcal{S} \mid \mathcal{G}$
Monotypes	τ	$::=$	$\mathsf{Int} \mid a \mid \widehat{\alpha} \mid \tau_1 \rightarrow \tau_2 \mid \mathcal{S} \mid \mathcal{G}$
Existential variables	$\widehat{\alpha}$	$::=$	$\widehat{\alpha}_S \mid \widehat{\alpha}_G$
Castable Types	$\mathbb{C}$	$::=$	$\mathsf{Int} \mid a \mid \widehat{\alpha} \mid \mathbb{C}_1 \rightarrow \mathbb{C}_2 \mid \forall a.\, \mathbb{C} \mid ? \mid \mathcal{G}$
Castable Monotypes	t	$::=$	$\mathsf{Int} \mid a \mid \widehat{\alpha} \mid t_1 \rightarrow t_2 \mid \mathcal{G}$
Algorithmic Contexts	Γ, Δ, Θ	$::=$	$\bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \widehat{\alpha} \mid \Gamma, \widehat{\alpha}_S = \tau \mid \Gamma, \widehat{\alpha}_G = t \mid \Gamma, \blacktriangleright_{\widehat{\alpha}}$
Complete Contexts	Ω	$::=$	$\bullet \mid \Omega, x : \sigma \mid \Omega, a \mid \Omega, \widehat{\alpha}_S = \tau \mid \Omega, \widehat{\alpha}_G = t \mid \Omega, \blacktriangleright_{\widehat{\alpha}}$

B.2 Type System

$\boxed{\Gamma \vdash^G \sigma \lesssim aB \dashv \Delta}$ *(Algorithmic Consistent Subtyping)*

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

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

$$\textsc{gpc-as-forall}\mathrm{LL}\quad \frac{\Gamma, \blacktriangleright_{\widehat{\alpha}_S}, \widehat{\alpha}_S \vdash^G \sigma_1[a \mapsto \widehat{\alpha}_S] \lesssim \sigma_2 \dashv \Delta, \blacktriangleright_{\widehat{\alpha}_S}, \Theta}{\Gamma \vdash^G \forall a.\, \sigma_1 \lesssim \sigma_2 \dashv \Delta} \qquad \textsc{gpc-as-spar}\quad \frac{}{\Gamma \vdash^G \mathcal{S} \lesssim \mathcal{S} \dashv \Gamma}$$

$$\textsc{gpc-as-gpar}\quad \frac{}{\Gamma \vdash^G \mathcal{G} \lesssim \mathcal{G} \dashv \Gamma} \qquad \textsc{gpc-as-unknown}\mathrm{LL}\quad \frac{}{\Gamma \vdash^G ? \lesssim \mathbb{C} \dashv \mathsf{contaminate}(\Gamma, \mathbb{C})}$$

$$\textsc{gpc-as-unknownRR}\quad \frac{}{\Gamma \vdash^{G} \mathbb{C} \lesssim \textbf{?} \dashv \mathsf{contaminate}(\Gamma, \mathbb{C})}$$

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

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

$\boxed{\Gamma \vdash^{G} \widehat{\alpha} \lessapprox \sigma \dashv \Delta}$ *(Instantiation I)*

$$\textsc{gpc-instL-solveS}\quad \frac{\Gamma \vdash^{G} \tau}{\Gamma, \widehat{\alpha}_S, \Gamma' \vdash^{G} \widehat{\alpha}_S \lessapprox \tau \dashv \Gamma, \widehat{\alpha}_S = \tau, \Gamma'}$$

$$\textsc{gpc-instL-solveG}\quad \frac{\Gamma \vdash^{G} t}{\Gamma, \widehat{\alpha}_G, \Gamma' \vdash^{G} \widehat{\alpha}_G \lessapprox t \dashv \Gamma, \widehat{\alpha}_G = t, \Gamma'}$$

$$\textsc{gpc-instL-solveUS}\quad \frac{}{\Gamma[\widehat{\alpha}_S] \vdash^{G} \widehat{\alpha}_S \lessapprox \textbf{?} \dashv \Gamma[\widehat{\alpha}_G, \widehat{\alpha}_S = \widehat{\alpha}_G]}$$

$$\textsc{gpc-instL-solveUG}\quad \frac{}{\Gamma[\widehat{\alpha}_G] \vdash^{G} \widehat{\alpha}_G \lessapprox \textbf{?} \dashv \Gamma[\widehat{\alpha}_G]}$$

$$\textsc{gpc-instL-reachSG1}\quad \frac{}{\Gamma[\widehat{\alpha}_S][\widehat{\beta}_G] \vdash^{G} \widehat{\alpha}_S \lessapprox \widehat{\beta}_G \dashv \Gamma[\widehat{\alpha}_G, \widehat{\alpha}_S = \widehat{\alpha}_G][\widehat{\beta}_G = \widehat{\alpha}_G]}$$

$$\textsc{gpc-instL-reachSG2}\quad \frac{}{\Gamma[\widehat{\beta}_S][\widehat{\alpha}_G] \vdash^{G} \widehat{\alpha}_G \lessapprox \widehat{\beta}_S \dashv \Gamma[\widehat{\beta}_G, \widehat{\beta}_S = \widehat{\beta}_G][\widehat{\alpha}_G = \widehat{\beta}_G]}$$

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

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

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

$\boxed{\Gamma \vdash^{G} \sigma \lessapprox \widehat{\alpha} \dashv \Delta}$ *(Instantiation II)*

$$\frac{\Gamma \vdash^{G} \tau}{\Gamma, \widehat{\alpha}_S, \Gamma' \vdash^{G} \tau \lessapprox \widehat{\alpha}_S \dashv \Gamma, \widehat{\alpha}_S = \tau, \Gamma'}\;\textsc{gpc-instr-solveS} \qquad \frac{\Gamma \vdash^{G} t}{\Gamma, \widehat{\alpha}_G, \Gamma' \vdash^{G} t \lessapprox \widehat{\alpha}_G \dashv \Gamma, \widehat{\alpha}_G = t, \Gamma'}\;\textsc{gpc-instr-solveG}$$

$$\frac{}{\Gamma[\widehat{\alpha}_S] \vdash^{G} \textit{?} \lessapprox \widehat{\alpha}_S \dashv \Gamma[\widehat{\alpha}_G, \widehat{\alpha}_S = \widehat{\alpha}_G]}\;\textsc{gpc-instr-solveUS} \qquad \frac{}{\Gamma[\widehat{\alpha}_G] \vdash^{G} \textit{?} \lessapprox \widehat{\alpha}_G \dashv \Gamma[\widehat{\alpha}_G]}\;\textsc{gpc-instr-solveUG}$$

$$\frac{}{\Gamma[\widehat{\alpha}_S][\widehat{\beta}_G] \vdash^{G} \widehat{\beta}_G \lessapprox \widehat{\alpha}_S \dashv \Gamma[\widehat{\alpha}_G, \widehat{\alpha}_S = \widehat{\alpha}_G][\widehat{\beta}_G = \widehat{\alpha}_G]}\;\textsc{gpc-instr-reachSG1}$$

$$\frac{}{\Gamma[\widehat{\beta}_S][\widehat{\alpha}_G] \vdash^{G} \widehat{\beta}_S \lessapprox \widehat{\alpha}_G \dashv \Gamma[\widehat{\beta}_G, \widehat{\beta}_S = \widehat{\beta}_G][\widehat{\alpha}_G = \widehat{\beta}_G]}\;\textsc{gpc-instr-reachSG2}$$

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

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

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

$\boxed{\Gamma \vdash^{G} e \Rightarrow \sigma \dashv \Delta}$ *(Inference)*

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

$$\frac{\Gamma \vdash^{G} \sigma \qquad \Gamma, \widehat{\beta}_S, x : \sigma \vdash^{G} e \Leftarrow \widehat{\beta}_S \dashv \Delta, x : \sigma, \Theta}{\Gamma \vdash^{G} \lambda x : \sigma.\, e \Rightarrow \sigma \to \widehat{\beta}_S \dashv \Delta}\;\textsc{gpc-inf-lamann2}$$

$$\frac{\Gamma, \widehat{\alpha}_S, \widehat{\beta}_S, x : \widehat{\alpha}_S \vdash^{G} e \Leftarrow \widehat{\beta}_S \dashv \Delta, x : \widehat{\alpha}_S, \Theta}{\Gamma \vdash^{G} \lambda x.\, e \Rightarrow \widehat{\alpha}_S \to \widehat{\beta}_S \dashv \Delta}\;\textsc{gpc-inf-lam2} \qquad \frac{\Gamma \vdash^{G} \sigma \qquad \Gamma \vdash^{G} e \Leftarrow \sigma \dashv \Delta}{\Gamma \vdash^{G} e : \sigma \Rightarrow \sigma \dashv \Delta}\;\textsc{gpc-inf-anno}$$

GPC-INF-APP

$$\frac{\Gamma \vdash^G e_1 \Rightarrow \sigma \dashv \Theta_1 \qquad \Theta_1 \vdash^G [\Theta_1]\sigma \triangleright \sigma_1 \to \sigma_2 \dashv \Theta_2 \qquad \Theta_2 \vdash^G e_2 \Leftarrow [\Theta_2]\sigma_1 \dashv \Delta}{\Gamma \vdash^G e_1\, e_2 \Rightarrow \sigma_2 \dashv \Delta}$$

GPC-INF-LET2

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

$\boxed{\Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta}$ *(Checking)*

GPC-CHK-LAM

$$\frac{\Gamma, x : \sigma_1 \vdash^G e \Leftarrow \sigma_2 \dashv \Delta, x : \sigma_1, \Theta}{\Gamma \vdash^G \lambda x.\, e \Leftarrow \sigma_1 \to \sigma_2 \dashv \Delta}$$

GPC-CHK-GEN

$$\frac{\Gamma, a \vdash^G e \Leftarrow \sigma \dashv \Delta, a, \Theta}{\Gamma \vdash^G e \Leftarrow \forall a.\, \sigma \dashv \Delta}$$

GPC-CHK-SUB

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

$\boxed{\Gamma \vdash^G \sigma \triangleright \sigma_1 \to \sigma_2 \dashv \Delta}$ *(Algorithmic Matching)*

GPC-AM-FORALLL

$$\frac{\Gamma, \widehat{\alpha}_S \vdash^G \sigma[a \mapsto \widehat{\alpha}_S] \triangleright \sigma_1 \to \sigma_2 \dashv \Delta}{\Gamma \vdash^G \forall a.\, \sigma \triangleright \sigma_1 \to \sigma_2 \dashv \Delta}$$

GPC-AM-ARR

$$\frac{}{\Gamma \vdash^G \sigma_1 \to \sigma_2 \triangleright \sigma_1 \to \sigma_2 \dashv \Gamma}$$

GPC-AM-UNKNOWN

$$\frac{}{\Gamma \vdash^G \mathord{?} \triangleright \mathord{?} \to \mathord{?} \dashv \Gamma}$$

GPC-AM-VAR

$$\frac{}{\Gamma[\widehat{\alpha}] \vdash^G \widehat{\alpha} \triangleright \widehat{\alpha}_1 \to \widehat{\alpha}_2 \dashv \Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2, \widehat{\alpha} = \widehat{\alpha}_1 \to \widehat{\alpha}_2]}$$

C Kind Inference for Datatypes

C.1 Other Language Extensions

This section accompanies Section 7.8 of the main paper, including discussion about more related language extensions. These extensions affect kind inference, but not in a fundamental way.

C.1.1 Visible Dependent Quantification

Besides specified type variables for which users can optionally provide type arguments, Haskell also incorporates *visible dependent quantification (VDQ)*[1], e.g., **type** $T :: \forall (k :: \star) \to k \to \star$, with which users are forced to provide type arguments to T. That is, one would use T with, e.g., $T\ \star\ Int$ and $T\ (\star \to \star)\ Maybe$, never just $T\ Int$. Visible dependent quantification is Haskell's equivalent to routine dependent quantification in dependently typed languages.

To support VDQ, rule DT-TT needs to be extended, as VDQ brings variables into scope for later reference. For example, given

$$\begin{array}{l} \textbf{data}\ T :: \forall (k :: \star) \to k \to \star \\ \textbf{data}\ T\ k\ a = MkT \end{array}$$

We should get a context $k :: \star, a :: k$ when checking MkT.

VDQ opens an interesting design choice: should unannotated type variables be able to introduce VDQ? For example, in the definition of P below, we use f and a as the arguments to T. To make it type-check, we need to infer $P :: \forall (f :: \star) \to f \to \star$.

$$\textbf{data}\ P\ f\ a = MkP\ (T\ f\ a)$$

However, the tricky part with inferring the kind of P is that we cannot have a fixed initial form of the kind of P, i.e., $\widehat{\alpha} \to \widehat{\beta} \to \star$ or $\forall (f : \widehat{\alpha}) \to \widehat{\beta} \to \star$, when type-checking the **rec** group of P, until we type-check P's body. In order to avoid this challenge, we support GHC's current ruling on the matter: *dependent variables must be manifestly so*. That is, the

[1] VDQ is implemented in GHC 8.10.

initial kind of a datatype includes VDQ only for those variables that appear, lexically, in the kind of a variable; other type parameters are reflected in a datatype's initial kind with a regular (non-dependent) arrow. This guideline rejects P as an example of non-manifest dependency.

C.1.2 Datatype Promotion

Haskellers can use datatypes as kinds and can write data constructors in types [Yorgey et al. 2012]. In the PolyKinds system, types and kinds are mixed (allowing datatypes to be used as kinds), but there is no facility to use a data constructor in a type.

To support such usage, the kinding judgment must now use the term context to fetch the type of data constructors. Moreover, dependency analysis needs to take dependencies on data constructors into account.

Definition 25 (Dependency Analysis with Type-Level Data)**.** We extend Definition 21 with

(iii) The definition of T_1 depends on the definition of T_2 if T_1 uses data constructors of T_2.

While the appearance of data constructors in types enriches the type language considerably, they do not pose a particular challenge for inference; the rest of our presentation would remain unaffected.

C.1.3 Partial Type Signatures

For quite some time, GHC has supported kind signatures on a subset of a datatype's parameters, much like the partial type signatures described by Winant et al. [2014]. For example, App, below, does not have a signature but still has a kind annotation for f.

$$\mathbf{data}\ App\ (f :: \star \to \star)\ a = A\ (f\ a)$$

To deal with such a construct we first need to amend the syntax of a datatype declaration to support kind annotations for variables.

datatype decl.	$\mathcal{T}$	$::=$	$\mathbf{data}\, T\, \phi = \overline{\mathcal{D}_j}^{\,j}$

Kind annotations can also contain free variables, which need to be generalized in a similar way as signatures. For example, $T2$ has kind $\forall\{k :: \star\}.\ \forall(f :: k).\ \star$.

$$\mathbf{data}\ T2\ (f :: k) = MkT2$$

Supporting these partial signatures adds complication to rule PGM-DT-TT (and its algorithmic counterpart) to bring the kind variables into scope. However, and critically, a partial signature will still go via rule PGM-DT-TT, never rule PGM-DT-TTS, used for full signatures only. This means that a partial type signature does *not* unlock polymorphic recursion: the datatype will considered monomorphic and ungeneralized within its own recursive group.

C.2 Today's GHC

Our Chapter 7 describes, in depth, how kind inference can work for datatype declarations. Here, we review how our work relates to GHC. To make the claims concrete, this section contains references to specific stretches of code within GHC.

C.2.1 Constraint-Based Type Inference

Type inference in GHC is based on generating and solving constraints [Pottier and Rémy 2005; Vytiniotis et al. 2011], distinct from our approach here, where we unify on the fly. Despite this different architecture, our results carry over to the constraint-based style. Instead of using eager unification, we can imagine accumulating constraints in output contexts Θ, and then invoking a solver to extend the context with solutions. This approach is taken by Eisenberg [2016].

In thinking about the change from eager unification to delayed constraints, one might worry about information loss around any place where we apply a context as a substitution, as these substitutions would be empty in a constraint-solving approach without eager unification. At top-level (Figure 7.8), a constraint-solving approach would run the constraint solver, and the substitutions would contain the same mappings as our approach provides. Conversely, the relations in Figure 7.10 would become part of the constraint solver, so substituting here is safe, too. A potential problem arises in rule A-KTT-APP (Figure 7.9), where we substitute in the function's kind before running the kind-directed $\Vdash^{\mathsf{kapp}}$ judgment. However, our system is predicative: it never unifies a type variable with a polytype. Thus, the substitution in rule A-KTT-APP can never trigger a new usage of rule A-KAPP-TT-FORALL. It *can* distinguish between rule A-KAPP-TT-ARROW and rule A-KAPP-TT-KUVAR, but we conjecture that the choice between these rules is irrelevant: both will lead to equivalent substitutions in the end.

C.2.2 Contexts

A typing context is *not* maintained in much of GHC's inference algorithm. Instead, a variable's kind is stored in the data structure representing the variable. This is very convenient, as it means that looking up a variable's type or kind is a pure, fast operation. One downside is that the compiler must maintain an extra invariant that all occurrences of a variable store the same kind; this is straightforward to maintain in practice.

Beyond just storing variables' kinds, the typing context in this work also critically stores variables' ordering. Lacking contexts, GHC uses a different mechanism: *level numbers*, originally invented to implement untouchability [Vytiniotis et al. 2011, Section 5.1]. Every type variable in GHC is assigned a level number during inference. Type variables contain a structure that includes level numbers. Roughly, the level number of a type variable a corresponds to the number of type variables in scope before a. Accordingly, we can tell the relative order (in a hypothetical context, according to the systems in this work) of two variables simply by comparing their level numbers. One of GHC's invariants is that a unification variable at level n is never unified with a type that mentions a variable with a level number $m > n$; this is much like the extra checks in the unification judgments in our work.

The *local scopes* of this work are also tracked by GHC. All the variables in the same local scope are assigned the same level number, and they are flagged as reorderable. After inference is complete, GHC does a topological sort to get the final order.

A final role that contexts play in our formalism is that they store solutions for unification variables; we apply contexts as a substitution. In GHC, unification variables store mutable cells that get filled in. It has a process called *zonking*[2], which is exactly analogous to our use of contexts as substitutions. Zonking a unification variable replaces the variable with its solution, if any.

C.2.3 Unification

The solver in GHC still has to carry out unification, much along the lines of the unification judgment we present here. This algorithm has to deal with the heterogeneous unification problems we consider, as well. Indeed, GHC's unification algorithm recurs into the kinds of a unification variable and the type it is unifying with, just as ours does. As implied by our focus on decidability of unification, there have been a number of bugs in GHC's implementation that led to loops in the type checker; the most recent is #16902.

GHC actually uses several unification algorithms internally. It has an eager unifier, much like the one we describe. When that unifier fails, it generates the constraint that is sent to the solver. (The eager unifier is meant solely to be an optimization.) There is also a unifier meant to work after type inference is complete; it checks for instance overlap, for example. All the unifiers recur into kinds:

[2] There are actually two variants of zonking in GHC: we can zonk during type-checking or at the end. The difference between the variants is chiefly what to do for an unfilled unification variable. The former leaves them alone, while the latter has to default them somehow; details are beyond our scope here.

- The eager unifier recurs into kinds.
- The unifier in the solver recurs into kinds.
- The pure unifier uses an invariant that the kinds are related before looking at the types. It must recur when decomposing applications.

In addition, GHC also has an overlap problem within unification, as exhibited in our work by the overlap between rules A-U-KVARL and A-U-KVARR in Figure 7.3. Both the eager unifier and the constraint-solver unifier deal with this ambiguity by using heuristics to choose which variable might be more suitable for unification. This particular issue—which variable to unify when there is a choice—has been the subject of some amount of churn over the years.

C.2.4 Promotion

The promotion operation, too, is present in GHC, though its form is quite different than what we have presented. Instead of promoting during unification, GHC simply refuses to solve a unification variable if any of the free variables of its supposed solution lives to the right of the variable in the context. Because GHC is working with constraints, it just leaves the unification problem as an unsolved constraint. If there remain unsolved constraints, GHC then promotes the variables it can: some cannot be promoted because they depend on locally bound quantified (not unification) type variables.

C.2.5 Complete User-Supplied Kinds

Along with stand-alone kind signatures, as described in this work, GHC supports *complete user-supplied kinds*, or CUSKs. A datatype has a CUSK when certain syntactic conditions are satisfied; GHC detects these conditions *before* doing any kind inference. These CUSKs are a poor substitute for proper kind signatures, as the syntactic cues are fragile and unexpected: users sometimes write a CUSK without meaning to, and also sometimes leave out a necessary part of a CUSK when they intend to specify the kind. Stand-alone kind signatures are a new feature; they begin with the keyword **type** instead of **data**, as we have used in our work.

Interestingly, it would be wrong to support CUSKs in a system without polymorphic kinds. Consider this example:

$$\textbf{data}\ S1\ a = MkT1\ S2$$
$$\textbf{data}\ S2 = MkS2\ (S1\ Maybe)$$

The types $S1$ and $S2$ form a group. We put $S2$ (which has a CUSK) into the context with kind $\star$. When we check $S1$, we find no constraints on a (in the constraint-generation pass;

see the general approach below). The kind of $S1$ is then defaulted to $\star \to \star$. Checking $S2$ fails. Instead, we wish to pretend that $S2$ does not have a CUSK. This would mean that constraint-generation happens for all the constructors in both $S1$ and $S2$, and $S1$ would get its correct kind $(\star \to \star) \to \star$.

With kind-polymorphism, we have no problem because the kind of $T1$ will be generalized to $\forall (k :: \star).\ k \to \star$.

This was reported as bug #16609.

C.2.6 Dependency Analysis

The algorithm implemented in GHC for processing datatype declarations starts with dependency analysis, as ours does. The dependency analysis is less fine-grained than what we have proposed in this work: signatures are ignored in the dependency analysis, and so datatypes with signatures are processed alongside all the others. This means that the kinds in the example below have more restrictive kinds in GHC than they do in our system:

$$\begin{array}{l}
\textbf{data}\ S1 :: \forall k.\ k \to \star \\
\textbf{data}\ S1\ a = MkS1\ (S2\ Int) \\
\textbf{data}\ S2\ a = MkS2\ (S3\ Int) \\
\textbf{data}\ S3\ a = MkS3\ (S1\ Int)
\end{array}$$

A naïve dependency analysis would put all three definitions in the same group. The kind for $S1$ is given; it would indeed have that kind. The parameters of $S2$ and $S3$ would initially have an unknown kind, but when occurrences of $S2$ and $S3$ are processed (in the definitions of $S1$ and $S2$, respectively), this unknown kind would become $\star$. Neither $S2$ nor $S3$ would be generalized.

There is a ticket to improve the dependency analysis: #9427.

C.2.7 Approach to Kind-Checking Datatypes

In GHC's approach, after dependency analysis, so-called *initial kinds* are produced for all the datatypes in the group. These either come from a datatype's CUSK or from a simple analysis of the header of the datatype (without looking at constructors). This step corresponds to our algorithm's placing a binding for the datatype in the context, either with the kind signature or with a unification variable (rules A-PGM-DT-TTS and A-PGM-DT-TT).

If there is no CUSK, GHC then passes over all the datatype's constructors, collecting constraints on unification variables. After solving these constraints, GHC generalizes the datatype kind.

For all datatypes, now with generalized kinds, all data constructors are checked (again, for non-CUSK types). Because the kinds of the types are now generalized, a pass infers any invisible parameters to polykinded types. For non-CUSK types, this second pass using generalized kinds replaces the $T_i \mapsto T_i\,@\phi_i^c$ substitution in the context in the last premise to rule A-PGM-DT-TT. Performing a substitution—instead of re-generating and solving constraints—may be an opportunity for improvement in GHC.

C.2.8 Polymorphic Recursion

One challenge in kind inference is in the handling of polymorphic recursion. Although non-CUSK types are indeed monomorphic during the constraint-generation pass, some limited form of polymorphic recursion can get through. This is because all type variables are represented by a special form of unification variable called a TyVarTv. TyVarTvs can unify only with other type variables. This design is motivated by the following examples:

$$\mathbf{data}\ T1\ (a :: k)\ b = MkT1\ (T2\ a\ b)$$
$$\mathbf{data}\ T2\ (c :: j)\ d = MkT2\ (T1\ c\ d)$$

$$\mathbf{data}\ T3\ a\ \mathbf{where}$$
$$\quad MkT3 :: \forall (k :: \star)\ (b :: k).\ T3\ b$$

We want to accept all of these definitions. The first two, $T1$ and $T2$, form a mutually recursive group. Neither has a CUSK. However, the recursive occurrences are not polymorphically recursive: both recursive occurrences are at the *same* kind as the definition. Yet the first parameter to $T1$ is declared to have kind k while the first parameter to $T2$ is declared to have kind j. The solution: allow k to unify with j during the constraint-generation pass. We would *not* want to allow either k or j to unify with a non-variable, as that would seem to go against the user's wishes. But they must be allowed to unify with each other to accept this example.

With $T3$ (identical to T from Section 9.4), we have a different motivation. During inference, we will guess the kind of a; call it $\widehat{\alpha}$. When checking the $MkT3$ constructor, we will need to unify $\widehat{\alpha}$ with the locally bound k. We cannot set $\widehat{\alpha} := k$, as that will fill $\widehat{\alpha}$ with a k, bound to $\widehat{\alpha}$'s *right* in the context. Instead, we must set $k := \widehat{\alpha}$. This is possible only if k is represented by a unification variable.

There are two known problems with this approach:

1. It sometimes accepts polymorphic recursion, even without a CUSK. Here is an example:

$$\mathbf{data}\ T4\ a = \forall (k :: \star)\ (b :: k).\ MkT4\ (T4\ b)$$

The definition of $T4$ is polymorphically recursive: the occurrence $T4\ b$ is specialized to a kind other than the kind of a. Yet this definition is accepted. The two kinds unify (as k becomes a unification variable, set to the guessed kind of a) during the constraint-generation pass. Then, $T4$ is generalized to get the kind $\forall k.\ k \to \star$, at which point the last pass goes through without a hitch.

The reason this acceptance is troublesome is not that $T4$ is somehow dangerous or unsafe. It is that we know that polymorphic recursion cannot be inferred Henglein [1993], and yet GHC does it. Invariably, this must mean that GHC's algorithm will be hard to specify beyond its implementation.

2. In rare cases, the constraint-generation pass will succeed, while the final pass—meant to be redundant—will fail. Here is an example:

$$\begin{array}{l}
\mathbf{data}\ SameKind :: k \to k \to Type \\
\mathbf{data}\ Bad\ a\ \mathbf{where} \\
\quad MkBad :: \forall k_1\ k_2\ (a :: k_1)\ (b :: k_2).\ Bad\ (SameKind\ a\ b)
\end{array}$$

During the constraint-generation pass, the kinds k_1 and k_2 are allowed to unify, accepting the definition of Bad. During the final pass, however, k_1 and k_2 are proper quantified type variables, always distinct. Thus the $SameKind\ a\ b$ type is ill-kinded and rejected.

The fact that this final pass can fail means that we cannot implement it via a simple substitution, as we do in rule A-PGM-DT-TT. One possible solution is our suggestion to change the scoping of type parameters to GADT-syntax datatype declarations. With that change, our second motivation above for TyVarTvs would disappear. GHC could then use TyVarTvs only for kind variables in the head of a datatype declaration, using proper quantified type variables in constructors. Of course, this change would break much code in the wild, and we do not truly expect it to ever be adopted.

C.2.9 The Quantification Check

Our quantification check (Section 7.7.2) also has a parallel in GHC, but GHC's solution to the problem differed from ours. Instead of rejecting programs that fail the quantification check, GHC accepted them, replacing the variables that would be (but cannot be) quantified with its constant $Any :: \forall k.\ k$. The Any type is uninhabited, but exists at all kinds. As such, it is an appropriate replacement for unquantifiable, unconstrained unification variables. Yet this decision in GHC had unfortunate consequences: the Any type can appear in error messages, and its introduction induces hard-to-understand type errors.

We have later implemented our quantification check in GHC; see #16775.

Another design alternative is to generalize the variable to the leftmost position where it is still well-formed. Recall the example in Section 7.7.2:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Proxy} :: \forall k.\, k \to \star \\
&\mathbf{data}\ \mathit{Relate} :: \forall a\ (b :: a).\, a \to \mathit{Proxy}\ b \to \star \\
&\mathbf{data}\ T :: \forall (a :: \star)\ (b :: a)\ (c :: a)\ d.\, \mathit{Relate}\ b\ d \to \star
\end{aligned}$$

We have $d :: \widehat{\alpha}$, and $\widehat{\alpha} = \mathit{Proxy}\ \widehat{\beta}$, with $\widehat{\beta} :: a$. As there are no further constraints on $\widehat{\beta}$, the definition of T is rejected by the quantification check.

Instead of rejecting the program, or solving $\widehat{\beta}$ using Any, we can generalize over $\widehat{\beta}$ as a fresh variable f, which is put after a to make it well-kinded. Namely, we get

$$\mathbf{data}\ T :: \forall (a :: \star)\ \{f :: a\}\ (b :: a)\ (c :: a)\ (d :: \mathit{Proxy}\ f).\, \mathit{Relate}\ @a\ @f\ b\ d \to \star$$

However, this ordering of the variables violates our declarative specification. Moreover, this type requires an inferred variable to be between specified variables. With higher-rank polymorphism, due to the fact that GHC does not support first-class type-level abstraction (i.e., Λ in types), this type cannot be instantiated to

$$\forall (a :: \star)\ (b :: a)\ (c :: a)\ (d :: \mathit{Proxy}\ f).\, \mathit{Relate}\ @a\ @b\ b\ d \to \star$$

or

$$\forall (a :: \star)\ (b :: a)\ (c :: a)\ (d :: \mathit{Proxy}\ f).\, \mathit{Relate}\ @a\ @c\ b\ d \to \star$$

which makes the generalization less useful.

C.2.10 ScopedSort

When GHC deals with a local scope—a set of variables that may be reordered—it does a topological sort on the variables at the end. However, not any topological sort will do: it must use one that preserves the left-to-right ordering of the variables as much as possible. This is because GHC considers these implicitly bound variables to be *specified*: they are available for visible type application. For example, recall the example from Section 7.2.2, modified slightly:

$$\mathbf{data}\ Q\ (a :: (f\ b))\ (c :: k)\ (x :: f\ c)$$

Inference will tell us that k must come before f and b, but the order of f and b is immaterial. Our approach here is to make f, b, and k *inferred* variables: users of Q will not be able to instantiate these parameters with visible type application. However, GHC takes a different view: because the user has written the names of f, b, and k, they will be *specified*. This choice means that the precise sorting algorithm GHC uses to fix the order of local scopes becomes part of the *specification* of the language. Indeed, GHC documents the precise algorithm in its manual. If we followed suit, the algorithm would have to appear in our declarative specification, which goes against the philosophy of a declarative system.

Some recent debate led to a conclusion (see #16726) that we would change the interpretation of the Q example from the main work, meaning that its kind variables would indeed become *inferred*. However, the problem with ScopedSort still exists in type signatures, where type variables may be implicitly bound.

C.2.11 The "Forall-or-Nothing" Rule

GHC implements the so-called *forall-or-nothing* rule, which states that either *all* variables are quantified by a user-written $\forall$ or none are. These examples illustrate the effect:

$$
\begin{aligned}
&\mathit{ex1} :: a \to b \to a\\
&\mathit{ex2} :: \forall a\ b.\ a \to b \to a\\
&\mathit{ex3} :: \forall a.\ a \to b \to a\\
&\mathit{ex4} :: (\forall a.\ a \to b \to a)
\end{aligned}
$$

The signatures for both $\mathit{ex1}$ and $\mathit{ex2}$ are accepted: $\mathit{ex1}$ quantifies none, while $\mathit{ex2}$ quantifies all. The signature for $\mathit{ex3}$ is rejected, as GHC rejects a mixed economy. However, and perhaps surprisingly, $\mathit{ex4}$ is accepted. The only difference between $\mathit{ex3}$ and $\mathit{ex4}$ is the seemingly-redundant parentheses. However, because the forall-or-nothing rule applies only at the top level of a signature, the rule is not in effect for the $\forall$ in $\mathit{ex4}$.

This rule interacts with the main work only in that our formalism (and some of our examples) does not respect it. This may be the cause of differing behavior between GHC and the examples we present.

C.3 Complete Set of Rules

In this section we include the complete set of rules for Chapter 7. Some of the rules are repeated from those in the chapter.

C.3.1 Declarative Haskell98

$\boxed{\Sigma \vdash^{\mathsf{k}} \sigma : \kappa}$ *(Kinding for Polymorphic Types)*

$$\textsc{k-forall}\quad \frac{\Sigma, a : \kappa \vdash^{\mathsf{k}} \sigma : \star}{\Sigma \vdash^{\mathsf{k}} \forall a : \kappa .\, \sigma : \star}$$

$\boxed{\Sigma \vdash \Psi}$ *(Well-formed Term Contexts)*

$$\textsc{ectx-empty}\quad \frac{}{\Sigma \vdash \bullet} \qquad \textsc{ectx-dcon}\quad \frac{\Sigma \vdash \Psi \qquad \Sigma \vdash^{\mathsf{k}} \sigma : \star}{\Sigma \vdash \Psi, D : \sigma}$$

C.3.2 Algorithmic Haskell98

$\boxed{\Delta \Vdash^{\mathsf{k}} \sigma : \kappa \dashv \Theta}$ *(Kinding for Polymorphic Types)*

$$\textsc{a-k-forall}\quad \frac{\Delta \Vdash^{\mathsf{kv}} \kappa \qquad \Delta, a : \kappa \Vdash^{\mathsf{k}} \sigma : \kappa_2 \dashv \Theta, a : \kappa \qquad [\Theta]\kappa_2 = \star}{\Delta \Vdash^{\mathsf{k}} \forall a : \kappa .\, \sigma : \star \dashv \Theta}$$

$\boxed{\Delta \Vdash^{\mathsf{kc}} \sigma \Leftarrow \kappa}$ *(Checking)*

$$\textsc{a-kc-eq}\quad \frac{\Delta \Vdash^{\mathsf{k}} \sigma : \kappa_1 \dashv \Delta \qquad [\Delta]\kappa_1 = [\Delta]\kappa_2}{\Delta \Vdash^{\mathsf{kc}} \sigma \Leftarrow \kappa_2}$$

$\boxed{\Delta \Vdash^{\mathsf{kv}} \kappa}$ *(Well-formed Kinds)*

$$\textsc{a-kv-star}\quad \frac{}{\Delta \Vdash^{\mathsf{kv}} \star} \qquad \textsc{a-kv-arrow}\quad \frac{\Delta \Vdash^{\mathsf{kv}} \kappa_1 \qquad \Delta \Vdash^{\mathsf{kv}} \kappa_2}{\Delta \Vdash^{\mathsf{kv}} \kappa_1 \rightarrow \kappa_2} \qquad \textsc{a-kv-kuvar}\quad \frac{\widehat{\alpha} \in \Delta}{\Delta \Vdash^{\mathsf{kv}} \widehat{\alpha}}$$

$\boxed{\Delta\ \mathsf{ok}}$ *(Well-formed Type Contexts)*

$$\textsc{a-tctx-empty}\quad \frac{}{\bullet\ \mathsf{ok}} \qquad \textsc{a-tctx-tvar}\quad \frac{\Delta\ \mathsf{ok} \qquad \Delta \Vdash^{\mathsf{kv}} \kappa}{\Delta, a : \kappa\ \mathsf{ok}} \qquad \textsc{a-tctx-tcon}\quad \frac{\Delta\ \mathsf{ok} \qquad \Delta \Vdash^{\mathsf{kv}} \kappa}{\Delta, T : \kappa\ \mathsf{ok}} \qquad \textsc{a-tctx-kuvar}\quad \frac{\Delta\ \mathsf{ok}}{\Delta, \widehat{\alpha}\ \mathsf{ok}}$$

$$\textsc{a-tctx-kuvarSolved}\quad \frac{\Delta\ \mathsf{ok} \qquad \Delta \Vdash^{\mathsf{kv}} \kappa}{\Delta, \widehat{\alpha} = \kappa\ \mathsf{ok}}$$

$\boxed{\Delta \Vdash^{\mathsf{ectx}} \Gamma}$ *(Well-formed Term Contexts)*

$$\frac{}{\Delta \Vdash^{\mathsf{ectx}} \bullet}\ \textsc{a-ectx-empty} \qquad \frac{\Delta \Vdash^{\mathsf{ectx}} \Gamma \qquad \Delta \Vdash^{\mathsf{kc}} \sigma \Leftarrow \star}{\Delta \Vdash^{\mathsf{ectx}} \Gamma, D : \sigma}\ \textsc{a-ectx-dcon}$$

$\boxed{\Delta \longrightarrow\!\!\!\!\!\rightarrow \Omega}$ *(Defaulting)*

$$\frac{}{\bullet \twoheadrightarrow \bullet}\ \textsc{a-ctxde-empty} \qquad \frac{\Delta \twoheadrightarrow \Omega}{\Delta, a : \kappa \twoheadrightarrow \Omega, a : \kappa}\ \textsc{a-ctxde-tvar} \qquad \frac{\Delta \twoheadrightarrow \Omega}{\Delta, T : \kappa \twoheadrightarrow \Omega, T : \kappa}\ \textsc{a-ctxde-tcon}$$

$$\frac{\Delta \twoheadrightarrow \Omega}{\Delta, \widehat{\alpha} = \kappa \twoheadrightarrow \Omega, \widehat{\alpha} = \kappa}\ \textsc{a-ctxde-kuvarSolved} \qquad \frac{\Delta \twoheadrightarrow \Omega}{\Delta, \widehat{\alpha} \twoheadrightarrow \Omega, \widehat{\alpha} = \star}\ \textsc{a-ctxde-solve}$$

C.3.3 Context Application in Haskell98

$[\Delta]\kappa$ applies Δ as a substitution to κ.

$$\begin{array}{lcl}
[\Delta]\star & = & \star \\
[\Delta]\kappa_1 \to \kappa_2 & = & [\Delta]\kappa_1 \to [\Delta]\kappa_2 \\
[\Delta[\widehat{\alpha}]]\widehat{\alpha} & = & \widehat{\alpha} \\
[\Delta[\widehat{\alpha} = \kappa]]\widehat{\alpha} & = & [\Delta[\widehat{\alpha} = \kappa]]\kappa
\end{array}$$

$[\Delta]\Gamma$ applies Δ as a substitution to Γ.

$$\begin{array}{lcl}
[\Delta]\bullet & = & \bullet \\
[\Delta](\Gamma, D : \sigma) & = & [\Delta]\Gamma, D : [\Delta]\sigma
\end{array}$$

$[\Omega]\Delta$ applies Ω as a substitution to Δ.

$$\begin{array}{lcll}
[\bullet]\bullet & = & \bullet & \\
[\Omega, a : \kappa](\Delta, a : \kappa) & = & [\Omega]\Delta, a : [\Omega]\kappa & \\
[\Omega, T : \kappa](\Delta, T : \kappa) & = & [\Omega]\Delta, T : [\Omega]\kappa & \\
[\Omega, \widehat{\alpha} = \kappa](\Delta, \widehat{\alpha}) & = & [\Omega]\Delta & \\
[\Omega, \widehat{\alpha} = \kappa](\Delta, \widehat{\alpha} = \kappa') & = & [\Omega]\Delta & \text{if } [\Omega]\kappa = [\Omega]\kappa' \\
[\Omega, \widehat{\alpha} = \kappa]\Delta & = & [\Omega]\Delta & \text{if } \widehat{\alpha} \notin \Delta
\end{array}$$

C.3.4 Context Extension in Haskell98

$\boxed{\Delta \longrightarrow \Theta}$ *(Context Extension)*

$$\frac{}{\bullet \longrightarrow \bullet}\ \textsc{a-ctxe-empty} \qquad \frac{\Delta \longrightarrow \Theta}{\Delta, a : \kappa \longrightarrow \Theta, a : \kappa}\ \textsc{a-ctxe-tvar} \qquad \frac{\Delta \longrightarrow \Theta}{\Delta, T : \kappa \longrightarrow \Theta, T : \kappa}\ \textsc{a-ctxe-tcon}$$

$$\frac{\Delta \longrightarrow \Theta}{\Delta, \widehat{\alpha} \longrightarrow \Theta, \widehat{\alpha}}\ \textsc{a-ctxe-kuvar} \qquad \frac{\Delta \longrightarrow \Theta \qquad [\Theta]\kappa_1 = [\Theta]\kappa_2}{\Delta, \widehat{\alpha} = \kappa_1 \longrightarrow \Theta, \widehat{\alpha} = \kappa_2}\ \textsc{a-ctxe-kuvarSolved} \qquad \frac{\Delta \longrightarrow \Theta \qquad \Theta \Vdash^{\mathsf{kv}} \kappa}{\Delta, \widehat{\alpha} \longrightarrow \Theta, \widehat{\alpha} = \kappa}\ \textsc{a-ctxe-solve}$$

$$\frac{\Delta \longrightarrow \Theta}{\Delta \longrightarrow \Theta, \widehat{\alpha}}\ \textsc{a-ctxe-add} \qquad \frac{\Delta \longrightarrow \Theta \qquad \Theta \Vdash^{\mathsf{kv}} \kappa}{\Delta \longrightarrow \Theta, \widehat{\alpha} = \kappa}\ \textsc{a-ctxe-addSolved}$$

C.3.5 Declarative PolyKinds

$\boxed{\rceil \sigma \lceil}$ *(Kind results in $\star$)*

$$\frac{}{\rceil \star \lceil}\ \textsc{sr-star} \qquad \frac{\rceil \kappa_2 \lceil}{\rceil \kappa_1 \to \kappa_2 \lceil}\ \textsc{sr-arrow} \qquad \frac{\rceil \sigma \lceil}{\rceil \forall \phi.\ \sigma \lceil}\ \textsc{sr-forall}$$

$\boxed{\Sigma \vdash^{\mathsf{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2}$ *(Instantiation)*

$$\frac{}{\Sigma \vdash^{\mathsf{inst}} \mu : \omega <: \omega \rightsquigarrow \mu}\ \textsc{inst-refl} \qquad \frac{\Sigma \vdash^{\mathsf{ela}} \rho : \omega_1 \qquad \Sigma \vdash^{\mathsf{inst}} \mu_1\ @\rho : \eta[a \mapsto \rho] <: \omega_2 \rightsquigarrow \mu_2}{\Sigma \vdash^{\mathsf{inst}} \mu_1 : \forall a : \omega_1.\, \eta <: \omega_2 \rightsquigarrow \mu_2}\ \textsc{inst-forall}$$

$$\frac{\Sigma \vdash^{\mathsf{ela}} \rho : \omega_1 \qquad \Sigma \vdash^{\mathsf{inst}} \mu_1\ @\rho : \eta[a \mapsto \rho] <: \omega_2 \rightsquigarrow \mu_2}{\Sigma \vdash^{\mathsf{inst}} \mu_1 : \forall \{a : \omega_1\}.\eta <: \omega_2 \rightsquigarrow \mu_2}\ \textsc{inst-forall-infer}$$

$\boxed{\Sigma \vdash^{\mathsf{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu}$ *(Kind Checking)*

$$\frac{\Sigma \vdash^{\mathsf{k}} \sigma : \eta \rightsquigarrow \mu_1 \qquad \Sigma \vdash^{\mathsf{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2}{\Sigma \vdash^{\mathsf{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu_2}\ \textsc{kc-sub}$$

$\boxed{\Sigma \vdash^{\mathsf{k}} \sigma : \eta \leadsto \mu}$ *(Kinding)*

$$\textsc{ktt-star}\quad \frac{}{\Sigma \vdash^{\mathsf{k}} \star : \star \leadsto \star} \qquad \textsc{ktt-nat}\quad \frac{}{\Sigma \vdash^{\mathsf{k}} \mathsf{Int} : \star \leadsto \mathsf{Int}} \qquad \textsc{ktt-var}\quad \frac{(a : \omega) \in \Sigma}{\Sigma \vdash^{\mathsf{k}} a : \omega \leadsto a}$$

$$\textsc{ktt-arrow}\quad \frac{}{\Sigma \vdash^{\mathsf{k}} \rightarrow : \star \rightarrow \star \rightarrow \star \leadsto \rightarrow} \qquad \textsc{ktt-tcon}\quad \frac{(T : \eta) \in \Sigma}{\Sigma \vdash^{\mathsf{k}} T : \eta \leadsto T}$$

$$\textsc{ktt-app}\quad \frac{\Sigma \vdash^{\mathsf{k}} \tau_1 : \eta_1 \leadsto \rho_1 \qquad \Sigma \vdash^{\mathsf{inst}} \rho_1 : \eta_1 <: (\omega_1 \rightarrow \omega_2) \leadsto \rho_2 \qquad \Sigma \vdash^{\mathsf{kc}} \tau_2 \Leftarrow \omega_1 \leadsto \rho_3}{\Sigma \vdash^{\mathsf{k}} \tau_1\, \tau_2 : \omega_2 \leadsto \rho_2\, \rho_3}$$

$$\textsc{ktt-kapp}\quad \frac{\Sigma \vdash^{\mathsf{k}} \kappa_1 : \forall a : \omega.\, \eta \leadsto \rho_1 \qquad \Sigma \vdash^{\mathsf{kc}} \kappa_2 \Leftarrow \omega \leadsto \rho_2}{\Sigma \vdash^{\mathsf{k}} \kappa_1\, @\kappa_2 : \eta[a \mapsto \rho_2] \leadsto \rho_1\, @\rho_2}$$

$$\textsc{ktt-kapp-infer}\quad \frac{\begin{array}{c}\Sigma \vdash^{\mathsf{k}} \kappa_1 : \forall \{\overline{a_i : \omega_i}^{\,i}\}.\forall a : \omega.\, \eta \leadsto \rho_1' \\ \overline{\Sigma \vdash^{\mathsf{ela}} \rho_i : \omega_i[\overline{a_i \mapsto \rho_i}^{\,i}]}^{\,i} \qquad \Sigma \vdash^{\mathsf{kc}} \kappa_2 \Leftarrow \omega[\overline{a_i \mapsto \rho_i}^{\,i}] \leadsto \rho_2'\end{array}}{\Sigma \vdash^{\mathsf{k}} \kappa_1\, @\kappa_2 : \eta[\overline{a_i \mapsto \rho_i}^{\,i}][a \mapsto \rho_2] \leadsto \rho_1'\, @\overline{\rho_i}^{\,i}\, @\rho_2'}$$

$$\textsc{ktt-forall}\quad \frac{\Sigma \vdash^{\mathsf{kc}} \kappa \Leftarrow \star \leadsto \omega \qquad \Sigma, a : \omega \vdash^{\mathsf{kc}} \sigma \Leftarrow \star \leadsto \mu}{\Sigma \vdash^{\mathsf{k}} \forall a : \kappa.\, \sigma : \star \leadsto \forall a : \omega.\, \mu}$$

$$\textsc{ktt-foralli}\quad \frac{\Sigma \vdash^{\mathsf{ela}} \omega : \star \qquad \Sigma, a : \omega \vdash^{\mathsf{kc}} \sigma \Leftarrow \star \leadsto \mu}{\Sigma \vdash^{\mathsf{k}} \forall a.\, \sigma : \star \leadsto \forall a : \omega.\, \mu}$$

$\boxed{\Sigma \vdash^{\mathsf{ela}} \mu : \eta}$ *(Elaborated Kinding)*

$$\textsc{ela-star}\quad \frac{}{\Sigma \vdash^{\mathsf{ela}} \star : \star} \qquad \textsc{ela-nat}\quad \frac{}{\Sigma \vdash^{\mathsf{ela}} \mathsf{Int} : \star} \qquad \textsc{ela-var}\quad \frac{(a : \omega) \in \Sigma}{\Sigma \vdash^{\mathsf{ela}} a : \omega} \qquad \textsc{ela-tcon}\quad \frac{(T : \eta) \in \Sigma}{\Sigma \vdash^{\mathsf{ela}} T : \eta}$$

$$\textsc{ela-arrow}\quad \frac{}{\Sigma \vdash^{\mathsf{ela}} \rightarrow : \star \rightarrow \star \rightarrow \star} \qquad \textsc{ela-app}\quad \frac{\Sigma \vdash^{\mathsf{ela}} \rho_1 : \omega_1 \rightarrow \omega_2 \qquad \Sigma \vdash^{\mathsf{ela}} \rho_2 : \omega_1}{\Sigma \vdash^{\mathsf{ela}} \rho_1\, \rho_2 : \omega_2}$$

$$\textsc{ela-kapp}\quad \frac{\Sigma \vdash^{\mathsf{ela}} \rho_1 : \forall a : \omega.\, \eta \qquad \Sigma \vdash^{\mathsf{ela}} \rho_2 : \omega}{\Sigma \vdash^{\mathsf{ela}} \rho_1\ @\rho_2 : \eta[a \mapsto \rho_2]} \qquad \textsc{ela-kapp-infer}\quad \frac{\Sigma \vdash^{\mathsf{ela}} \rho_1 : \forall \{a : \omega\}.\eta \qquad \Sigma \vdash^{\mathsf{ela}} \rho_2 : \omega}{\Sigma \vdash^{\mathsf{ela}} \rho_1\ @\rho_2 : \eta[a \mapsto \rho_2]}$$

$$\textsc{ela-forall}\quad \frac{\Sigma \vdash^{\mathsf{ela}} \omega : \star \qquad \Sigma, a : \omega \vdash^{\mathsf{ela}} \mu : \star}{\Sigma \vdash^{\mathsf{ela}} \forall a : \omega.\, \mu : \star} \qquad \textsc{ela-forall-infer}\quad \frac{\Sigma \vdash^{\mathsf{ela}} \omega : \star \qquad \Sigma, a : \omega \vdash^{\mathsf{ela}} \mu : \star}{\Sigma \vdash^{\mathsf{ela}} \forall \{a : \omega\}.\mu : \star}$$

$\boxed{\Sigma\ \mathsf{ok}}$ *(Well-formed Type Contexts)*

$$\textsc{tctx-empty}\quad \frac{}{\bullet\ \mathsf{ok}} \qquad \textsc{tctx-tvar-tt}\quad \frac{\Sigma\ \mathsf{ok} \qquad \Sigma \vdash^{\mathsf{ela}} \rho : \star}{\Sigma, a : \rho\ \mathsf{ok}} \qquad \textsc{tctx-tcon-tt}\quad \frac{\Sigma\ \mathsf{ok} \qquad \Sigma \vdash^{\mathsf{ela}} \eta : \star}{\Sigma, T : \eta\ \mathsf{ok}}$$

$\boxed{\Sigma \vdash \Psi}$ *(Well-formed Term Contexts)*

$$\textsc{ectx-empty}\quad \frac{}{\Sigma \vdash \bullet} \qquad \textsc{ectx-dcon-tt}\quad \frac{\Sigma \vdash \Psi \qquad \Sigma \vdash^{\mathsf{ela}} \mu : \star}{\Sigma \vdash \Psi, D : \mu}$$

C.3.6 Algorithmic PolyKinds

$\boxed{\Delta \Vdash^{\mathsf{inst}} \mu_1 : \eta <: \omega \leadsto \mu_2 \dashv \Theta}$ *(Instantiation)*

$$\textsc{a-inst-refl}\quad \frac{\Delta \Vdash^{\mathsf{u}} \omega_1 \approx \omega_2 \dashv \Theta}{\Delta \Vdash^{\mathsf{inst}} \mu : \omega_1 <: \omega_2 \leadsto \mu \dashv \Theta} \qquad \textsc{a-inst-forall}\quad \frac{\Delta, \widehat{\alpha} : \omega_1 \Vdash^{\mathsf{inst}} \mu_1\ @\widehat{\alpha} : \eta[a \mapsto \widehat{\alpha}] <: \omega_2 \leadsto \mu_2 \dashv \Theta}{\Delta \Vdash^{\mathsf{inst}} \mu_1 : \forall a : \omega_1.\, \eta <: \omega_2 \leadsto \mu_2 \dashv \Theta}$$

$$\textsc{a-inst-forall-infer}\quad \frac{\Delta, \widehat{\alpha} : \omega_1 \Vdash^{\mathsf{inst}} \mu_1\ @\widehat{\alpha} : \eta[a \mapsto \widehat{\alpha}] <: \omega_2 \leadsto \mu_2 \dashv \Theta}{\Delta \Vdash^{\mathsf{inst}} \mu_1 : \forall \{a : \omega_1\}.\eta <: \omega_2 \leadsto \mu_2 \dashv \Theta}$$

$\boxed{\Delta \Vdash^{\mathsf{kc}} \sigma \Leftarrow \eta \leadsto \mu \dashv \Theta}$ *(Kind Checking)*

$$\textsc{a-kc-sub}\quad \frac{\Delta \Vdash^{\mathsf{k}} \sigma : \eta \leadsto \mu_1 \dashv \Delta_1 \qquad \Delta_1 \Vdash^{\mathsf{inst}} \mu_1 : [\Delta_1]\eta <: [\Delta_1]\omega \leadsto \mu_2 \dashv \Delta_2}{\Delta \Vdash^{\mathsf{kc}} \sigma \Leftarrow \omega \leadsto \mu_2 \dashv \Delta_2}$$

$\boxed{\Delta \Vdash^{\mathsf{k}} \sigma : \eta \rightsquigarrow \mu \dashv \Theta}$ *(Kinding)*

A-KTT-STAR
$$\frac{}{\Delta \Vdash^{\mathsf{k}} \star : \star \rightsquigarrow \star \dashv \Delta}$$

A-KTT-NAT
$$\frac{}{\Delta \Vdash^{\mathsf{k}} \mathsf{Int} : \star \rightsquigarrow \mathsf{Int} \dashv \Delta}$$

A-KTT-VAR
$$\frac{(a : \omega) \in \Delta}{\Delta \Vdash^{\mathsf{k}} a : \omega \rightsquigarrow a \dashv \Delta}$$

A-KTT-TCON
$$\frac{(T : \eta) \in \Delta}{\Delta \Vdash^{\mathsf{k}} T : \eta \rightsquigarrow T \dashv \Delta}$$

A-KTT-ARROW
$$\frac{}{\Delta \Vdash^{\mathsf{k}} \rightarrow : \star \rightarrow \star \rightarrow \star \rightsquigarrow \rightarrow \dashv \Delta}$$

A-KTT-FORALL
$$\frac{\Delta \Vdash^{\mathsf{kc}} \kappa \Leftarrow \star \rightsquigarrow \omega \dashv \Delta_1 \qquad \Delta_1, a : \omega \Vdash^{\mathsf{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu \dashv \Delta_2, a : \omega, \Delta_3 \qquad \Delta_3 \hookrightarrow a}{\Delta \Vdash^{\mathsf{k}} \forall a : \kappa .\, \sigma : \star \rightsquigarrow \forall a : \omega .\, [\Delta_3]\mu \dashv \Delta_2, \mathbf{unsolved}\,(\Delta_3)}$$

A-KTT-APP
$$\frac{\Delta \Vdash^{\mathsf{k}} \tau_1 : \eta_1 \rightsquigarrow \rho_1 \dashv \Delta_1 \qquad \Delta_1 \Vdash^{\mathsf{kapp}} (\rho_1 : [\Delta_1]\eta_1) \bullet \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\mathsf{k}} \tau_1\, \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}$$

A-KTT-FORALLI
$$\frac{\Delta, \widehat{\alpha} : \star, a : \widehat{\alpha} \Vdash^{\mathsf{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu \dashv \Delta_2, a : \widehat{\alpha}, \Delta_3 \qquad \Delta_3 \hookrightarrow a}{\Delta \Vdash^{\mathsf{k}} \forall a .\, \sigma : \star \rightsquigarrow \forall a : \widehat{\alpha} .\, [\Delta_3]\mu \dashv \Delta_2, \mathbf{unsolved}\,(\Delta_3)}$$

A-KTT-KAPP
$$\frac{\Delta \Vdash^{\mathsf{k}} \tau_1 : \eta \rightsquigarrow \rho_1 \dashv \Delta_1 \qquad [\Delta_1]\eta = \forall a : \omega .\, \eta_2 \qquad \Delta_1 \Vdash^{\mathsf{kc}} \tau_2 \Leftarrow \omega \rightsquigarrow \rho_2 \dashv \Delta_2}{\Delta \Vdash^{\mathsf{k}} \tau_1 \mathbin{@}\tau_2 : \eta_2[a \mapsto \rho_2] \rightsquigarrow \rho_1 \mathbin{@}\rho_2 \dashv \Delta_2}$$

A-KTT-KAPP-INFER
$$\frac{\begin{array}{c}\Delta \Vdash^{\mathsf{k}} \tau_1 : \eta \rightsquigarrow \rho_1 \dashv \Delta_1 \qquad [\Delta_1]\eta = \forall \{\overline{a_i : \omega_i}^{\,i}\}. \forall a : \omega .\, \eta_2 \\ \Delta_1, \overline{\widehat{\alpha}_i : \omega_i[\overline{a_i \mapsto \widehat{\alpha}_i}^{\,i}]}^{\,i} \Vdash^{\mathsf{kc}} \tau_2 \Leftarrow \omega[\overline{a_i \mapsto \widehat{\alpha}_i}^{\,i}] \rightsquigarrow \rho_2 \dashv \Delta_2\end{array}}{\Delta \Vdash^{\mathsf{k}} \tau_1 \mathbin{@}\tau_2 : \eta_2[\overline{a_i \mapsto \widehat{\alpha}_i}^{\,i}][a \mapsto \rho_2] \rightsquigarrow \rho_1 \mathbin{@}\overline{\widehat{\alpha}_i}^{\,i} \mathbin{@}\rho_2 \dashv \Delta_2}$$

$\boxed{\Delta \Vdash^{\mathsf{kapp}} (\rho_1 : \eta) \bullet \tau : \omega \rightsquigarrow \rho_2 \dashv \Theta}$ *(Application Kinding)*

A-KAPP-TT-ARROW
$$\frac{\Delta \Vdash^{\mathsf{kc}} \tau \Leftarrow \omega_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta \Vdash^{\mathsf{kapp}} (\rho_1 : \omega_1 \rightarrow \omega_2) \bullet \tau : \omega_2 \rightsquigarrow \rho_1\, \rho_2 \dashv \Theta}$$

A-KAPP-TT-FORALL
$$\frac{\Delta, \widehat{\alpha} : \omega_1 \Vdash^{\mathsf{kapp}} (\rho_1 \mathbin{@}\widehat{\alpha} : \eta[a \mapsto \widehat{\alpha}]) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\mathsf{kapp}} (\rho_1 : \forall a : \omega_1 .\, \eta) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}$$

A-KAPP-TT-FORALL-INFER
$$\frac{\Delta, \widehat{\alpha} : \omega_1 \Vdash^{\mathsf{kapp}} (\rho_1 \,@\widehat{\alpha} : \eta[a \mapsto \widehat{\alpha}]) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\mathsf{kapp}} (\rho_1 : \forall\{a : \omega_1\}.\eta) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}$$

A-KAPP-TT-KUVAR
$$\frac{\Delta_1, \widehat{\alpha}_1 : \star, \widehat{\alpha}_2 : \star, \widehat{\alpha} : \omega = (\widehat{\alpha}_1 \to \widehat{\alpha}_2), \Delta_2 \Vdash^{\mathsf{kc}} \tau \Leftarrow \widehat{\alpha}_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta_1, \widehat{\alpha} : \omega, \Delta_2 \Vdash^{\mathsf{kapp}} (\rho_1 : \widehat{\alpha}) \bullet \tau : \widehat{\alpha}_2 \rightsquigarrow \rho_1\, \rho_2 \dashv \Theta}$$

$\boxed{\Delta \Vdash^{\mathsf{ela}} \mu : \eta}$ *(Elaborated Kinding)*

A-ELA-STAR
$$\frac{}{\Delta \Vdash^{\mathsf{ela}} \star : \star}$$

A-ELA-KUVAR
$$\frac{(\widehat{\alpha} : \omega) \in \Delta}{\Delta \Vdash^{\mathsf{ela}} \widehat{\alpha} : [\Delta]\omega}$$

A-ELA-NAT
$$\frac{}{\Delta \Vdash^{\mathsf{ela}} \mathsf{Int} : \star}$$

A-ELA-VAR
$$\frac{(a : \omega) \in \Delta}{\Delta \Vdash^{\mathsf{ela}} a : [\Delta]\omega}$$

A-ELA-TCON
$$\frac{(T : \eta) \in \Delta}{\Delta \Vdash^{\mathsf{ela}} T : [\Delta]\eta}$$

ELA-ARROW
$$\frac{}{\Delta \Vdash^{\mathsf{ela}} \to : \star \to \star \to \star}$$

A-ELA-FORALL
$$\frac{\Delta \Vdash^{\mathsf{ela}} \omega : \star \qquad \Delta, a : \omega \Vdash^{\mathsf{ela}} \mu : \star}{\Delta \Vdash^{\mathsf{ela}} \forall a : \omega.\, \mu : \star}$$

A-ELA-FORALL-INFER
$$\frac{\Delta \Vdash^{\mathsf{ela}} \omega : \star \qquad \Delta, a : \omega \Vdash^{\mathsf{ela}} \mu : \star}{\Delta \Vdash^{\mathsf{ela}} \forall\{a : \omega\}.\mu : \star}$$

A-ELA-APP
$$\frac{\Delta \Vdash^{\mathsf{ela}} \rho_1 : \omega_1 \to \omega_2 \qquad \Delta \Vdash^{\mathsf{ela}} \rho_2 : \omega_1}{\Delta \Vdash^{\mathsf{ela}} \rho_1\, \rho_2 : \omega_2}$$

A-ELA-KAPP
$$\frac{\Delta \Vdash^{\mathsf{ela}} \rho_1 : \forall a : \omega.\, \eta \qquad \Delta \Vdash^{\mathsf{ela}} \rho_2 : \omega}{\Delta \Vdash^{\mathsf{ela}} \rho_1 \,@\rho_2 : \eta[a \mapsto [\Delta]\rho_2]}$$

A-ELA-KAPP-INFER
$$\frac{\Delta \Vdash^{\mathsf{ela}} \rho_1 : \forall\{a : \omega\}.\eta \qquad \Delta \Vdash^{\mathsf{ela}} \rho_2 : \omega}{\Delta \Vdash^{\mathsf{ela}} \rho_1 \,@\rho_2 : \eta[a \mapsto [\Delta]\rho_2]}$$

$\boxed{\Delta \Vdash^{\mathsf{gen}}_{\phi^{\mathsf{c}}} \Gamma_1 \rightsquigarrow \Gamma_2}$ *(Generalization)*

A-GEN
$$\frac{\overline{\widehat{\phi}^{\mathsf{c}}_i = \mathbf{unsolved}(\mu_i)}^{\,i}}{\Delta \Vdash^{\mathsf{gen}}_{\phi^{\mathsf{c}}} \overline{D_i : \mu_i}^{\,i} \rightsquigarrow \overline{D : \forall\{\phi^{\mathsf{c}}\}.\forall\{\phi^{\mathsf{c}}_i\}.(\mu[\widehat{\phi}^{\mathsf{c}}_i \mapsto \phi^{\mathsf{c}}_i])}^{\,i}}$$

$\boxed{\Delta\ \mathsf{ok}}$ *(Well-formed Type Contexts)*

$$\textsc{a-tctx-empty}\quad \frac{}{\bullet\ \mathsf{ok}} \qquad \textsc{a-tctx-tvar-tt}\quad \frac{\Delta\ \mathsf{ok} \qquad \Delta \Vdash^{\mathsf{ela}} \omega : \star}{\Delta, a : \omega\ \mathsf{ok}} \qquad \textsc{a-tctx-tcon-tt}\quad \frac{\Delta\ \mathsf{ok} \qquad \Delta \Vdash^{\mathsf{ela}} \eta : \star}{\Delta, T : \eta\ \mathsf{ok}}$$

$$\textsc{a-tctx-kuvar-tt}\quad \frac{\Delta\ \mathsf{ok} \qquad \Delta \Vdash^{\mathsf{ela}} \omega : \star}{\Delta, \widehat{\alpha} : \omega\ \mathsf{ok}} \qquad \textsc{a-tctx-kuvarSolved-tt}\quad \frac{\Delta\ \mathsf{ok} \qquad \Delta \Vdash^{\mathsf{ela}} \omega_2 : [\Delta]\omega_1}{\Delta, \widehat{\alpha} : \omega_1 = \omega_2\ \mathsf{ok}} \qquad \textsc{a-tctx-lo}\quad \frac{\Delta, \Delta^{\mathsf{lo}}\ \mathsf{ok}}{\Delta, \{\Delta^{\mathsf{lo}}\}\ \mathsf{ok}}$$

$$\textsc{a-tctx-marker}\quad \frac{\Delta\ \mathsf{ok}}{\Delta, \blacktriangleright_D\ \mathsf{ok}}$$

$\boxed{\Delta \Vdash^{\mathsf{ectx}} \Gamma}$ *(Well-formed Term Contexts)*

$$\textsc{a-ectx-empty}\quad \frac{}{\Delta \Vdash^{\mathsf{ectx}} \bullet} \qquad \textsc{a-ectx-dcon-tt}\quad \frac{\Delta \Vdash^{\mathsf{ectx}} \Gamma \qquad \Delta \Vdash^{\mathsf{ela}} \mu : \star}{\Delta \Vdash^{\mathsf{ectx}} \Gamma, D : \mu}$$

$\boxed{\Delta \Vdash^{\mathsf{u}} \omega_1 \approx \omega_2 \dashv \Theta}$ *(Unification)*

$$\textsc{a-u-refl-tt}\quad \frac{}{\Delta \Vdash^{\mathsf{u}} \omega \approx \omega \dashv \Delta} \qquad \textsc{a-u-app}\quad \frac{\Delta \Vdash^{\mathsf{u}} \rho_1 \approx \rho_3 \dashv \Delta_1 \qquad \Delta_1 \Vdash^{\mathsf{u}} [\Delta_1]\rho_2 \approx [\Delta_1]\rho_4 \dashv \Theta}{\Delta \Vdash^{\mathsf{u}} \rho_1\ \rho_2 \approx \rho_3\ \rho_4 \dashv \Theta}$$

$$\textsc{a-u-kapp}\quad \frac{\Delta \Vdash^{\mathsf{u}} \rho_1 \approx \rho_3 \dashv \Delta_1 \qquad \Delta_1 \Vdash^{\mathsf{u}} [\Delta_1]\rho_2 \approx [\Delta_1]\rho_4 \dashv \Theta}{\Delta \Vdash^{\mathsf{u}} \rho_1\ @\rho_2 \approx \rho_3\ @\rho_4 \dashv \Theta}$$

$$\textsc{a-u-kvarL-tt}\quad \frac{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \widehat{\alpha} : \omega_1, \Theta_2 \qquad \Theta_1 \Vdash^{\mathsf{ela}} \rho_2 : \omega_2 \qquad \Theta_1 \Vdash^{\mathsf{u}} [\Theta_1]\omega_1 \approx \omega_2 \dashv \Theta_3}{\Delta \Vdash^{\mathsf{u}} \widehat{\alpha} \approx \rho_1 \dashv \Theta_3, \widehat{\alpha} : \omega_1 = \rho_2, \Theta_2}$$

$$\textsc{a-u-kvarR-tt}\quad \frac{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \widehat{\alpha} : \omega_1, \Theta_2 \qquad \Theta_1 \Vdash^{\mathsf{ela}} \rho_2 : \omega_2 \qquad \Theta_1 \Vdash^{\mathsf{u}} [\Theta_1]\omega_1 \approx \omega_2 \dashv \Theta_3}{\Delta \Vdash^{\mathsf{u}} \rho_1 \approx \widehat{\alpha} \dashv \Theta_3, \widehat{\alpha} : \omega_1 = \rho_2, \Theta_2}$$

A-U-KVARL-LO-TT

$$\frac{\Delta_1, \Delta_2 +\!\!+^{\mathsf{mv}} \widehat{\alpha} : \omega_1 \leadsto \Theta \qquad \Delta[\{\Theta\}] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \{\Theta_2, \widehat{\alpha} : \omega_1, \Theta_3\}, \Theta_4 \qquad \Theta_1, \{\Theta_2\} \Vdash^{\mathsf{ela}} \rho_2 : \omega_2 \qquad \Theta_1, \{\Theta_2\} \Vdash^{\mathsf{u}} [\Theta_1, \Theta_2]\omega_1 \approx \omega_2 \dashv \Theta_5, \{\Theta_6\}}{\Delta[\{\Delta_1, \widehat{\alpha} : \omega_1, \Delta_2\}] \Vdash^{\mathsf{u}} \widehat{\alpha} \approx \rho_1 \dashv \Theta_5, \{\Theta_6, \widehat{\alpha} : \omega_1 = \rho_2, \Theta_3\}, \Theta_4}$$

A-U-KVARR-LO-TT

$$\frac{\Delta_1, \Delta_2 +\!\!+^{\mathsf{mv}} \widehat{\alpha} : \omega_1 \leadsto \Theta \qquad \Delta[\{\Theta\}] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \{\Theta_2, \widehat{\alpha} : \omega_1, \Theta_3\}, \Theta_4 \qquad \Theta_1, \{\Theta_2\} \Vdash^{\mathsf{ela}} \rho_2 : \omega_2 \qquad \Theta_1, \{\Theta_2\} \Vdash^{\mathsf{u}} [\Theta_1, \Theta_2]\omega_1 \approx \omega_2 \dashv \Theta_5, \{\Theta_6\}}{\Delta[\{\Delta_1, \widehat{\alpha} : \omega_1, \Delta_2\}] \Vdash^{\mathsf{u}} \rho_1 \approx \widehat{\alpha} \dashv \Theta_5, \{\Theta_6, \widehat{\alpha} : \omega_1 = \rho_2, \Theta_3\}, \Theta_4}$$

$\boxed{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta}$ *(Promotion)*

A-PR-STAR

$$\frac{}{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \star \rightsquigarrow \star \dashv \Delta}$$

A-PR-ARROW

$$\frac{}{\Delta \Vdash^{\mathsf{pr}}_{\widehat{\alpha}} \to \rightsquigarrow \to \dashv \Delta}$$

A-PR-TCON

$$\frac{}{\Delta[T][\widehat{\alpha}] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} T \rightsquigarrow T \dashv \Delta[T][\widehat{\alpha}]}$$

A-PR-NAT

$$\frac{}{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \mathsf{Int} \rightsquigarrow \mathsf{Int} \dashv \Delta}$$

A-PR-APP

$$\frac{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \omega_1 \rightsquigarrow \rho_1 \dashv \Delta_1 \qquad \Delta_1 \vdash^{\mathsf{pr}}_{\widehat{\alpha}} [\Delta_1]\omega_2 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \omega_1\, \omega_2 \rightsquigarrow \rho_1\, \rho_2 \dashv \Theta}$$

A-PR-KAPP

$$\frac{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \omega_1 \rightsquigarrow \rho_1 \dashv \Delta_1 \qquad \Delta_1 \vdash^{\mathsf{pr}}_{\widehat{\alpha}} [\Delta_1]\omega_2 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \omega_1\, @\omega_2 \rightsquigarrow \rho_1\, @\rho_2 \dashv \Theta}$$

A-PR-TVAR

$$\frac{}{\Delta[a][\widehat{\alpha}] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} a \rightsquigarrow a \dashv \Delta[a][\widehat{\alpha}]}$$

A-PR-KUVARL

$$\frac{}{\Delta[\widehat{\beta}][\widehat{\alpha}] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\beta} \rightsquigarrow \widehat{\beta} \dashv \Delta[\widehat{\beta}][\widehat{\alpha}]}$$

A-PR-KUVARR-TT

$$\frac{\Delta \vdash^{\mathsf{pr}}_{\widehat{\alpha}} [\Delta]\rho \rightsquigarrow \rho_1 \dashv \Theta[\widehat{\alpha}][\widehat{\beta} : \rho]}{\Delta[\widehat{\alpha}][\widehat{\beta} : \rho] \vdash^{\mathsf{pr}}_{\widehat{\alpha}} \widehat{\beta} \rightsquigarrow \widehat{\beta}_1 \dashv \Theta[\widehat{\beta}_1 : \rho_1, \widehat{\alpha}][\widehat{\beta} : \rho = \widehat{\beta}_1]}$$

$\boxed{\Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \leadsto \Theta}$ *(Moving)*

A-MV-EMPTY

$$\frac{}{\bullet +\!\!+^{\mathsf{mv}} \Delta \leadsto \Delta}$$

A-MV-KUVAR

$$\frac{\mathbf{var}(\omega) \sharp \mathbf{dom}(\Delta_2) \qquad \Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \leadsto \Theta}{\widehat{\alpha} : \omega, \Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \leadsto \widehat{\alpha} : \omega, \Theta}$$

A-MV-KUVARM

$$\frac{\neg(\mathbf{var}(\omega) \sharp \mathbf{dom}(\Delta_2)) \qquad \Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2, \widehat{\alpha} : \omega \leadsto \Theta}{\widehat{\alpha} : \omega, \Delta_1 +\!\!+^{\mathsf{mv}} \Delta \leadsto \Theta}$$

$$\frac{\textsc{a-mv-tvar}\qquad \mathbf{var}(\omega) \sharp \mathbf{dom}(\Delta_2) \qquad \Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \rightsquigarrow \Theta}{a : \omega, \Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \rightsquigarrow a : \omega, \Theta}$$

$$\frac{\textsc{a-mv-tvarM}\qquad \neg(\mathbf{var}(\omega) \sharp \mathbf{dom}(\Delta_2)) \qquad \Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2, a : \omega \rightsquigarrow \Theta}{a : \omega, \Delta_1 +\!\!+^{\mathsf{mv}} \Delta_2 \rightsquigarrow \Theta}$$

C.3.7 Context Application in PolyKinds

$[\Delta]\eta$ applies Δ as a substitution to η.

$[\Delta]\star$	=	$\star$
$[\Delta]\mathsf{Int}$	=	Int
$[\Delta]a$	=	a
$[\Delta]T$	=	T
$[\Delta] \rightarrow$	=	$\rightarrow$
$[\Delta]\forall a : \omega.\, \eta$	=	$\forall a : [\Delta]\omega.\, [\Delta]\eta$
$[\Delta]\forall\{a : \omega\}.\eta$	=	$\forall\{a : [\Delta]\omega\}.[\Delta]\eta$
$[\Delta](\rho_1\, \rho_2)$	=	$([\Delta]\rho_1)\, ([\Delta]\rho_2)$
$[\Delta](\rho_1\, @\rho_2)$	=	$([\Delta]\rho_1)\, @([\Delta]\rho_2)$
$[\Delta[\widehat{\alpha}]]\widehat{\alpha}$	=	$\widehat{\alpha}$
$[\Delta[\widehat{\alpha} : \omega = \rho]]\widehat{\alpha}$	=	$[\Delta[\widehat{\alpha} : \omega = \rho]]\rho$

$[\Delta]\Gamma$ applies Δ as a substitution to Γ.

$[\Omega]\bullet$	=	$\bullet$
$[\Omega](\Gamma, D : \mu)$	=	$[\Omega]\Gamma, D : [\Omega]\mu$

$[\Omega]\Delta$ applies Ω as a substitution to Δ.

$[\Omega]\bullet$	=	$\bullet$	
$[\Omega, a : \omega](\Delta, a : \omega)$	=	$[\Omega]\Delta, a : [\Omega]\omega$	
$[\Omega, T : \omega](\Delta, T : \omega)$	=	$[\Omega]\Delta, T : [\Omega]\omega$	
$[\Omega, \widehat{\alpha} : \omega = \rho](\Delta, \widehat{\alpha} : \omega)$	=	$[\Omega]\Delta$	
$[\Omega, \widehat{\alpha} : \omega = \rho_1](\Delta, \widehat{\alpha} : \omega = \rho_2)$	=	$[\Omega]\Delta$	if $[\Omega]\rho_1 = [\Omega]\rho_2$
$[\Omega, \widehat{\alpha} : \omega = \rho]\Delta$	=	$[\Omega]\Delta$	if $\widehat{\alpha} \notin \Delta$
$[\Omega, \blacktriangleright_D](\Delta, \blacktriangleright_D)$	=	$[\Omega]\Delta$	
$[\Omega, \{\Omega_1\}](\Delta, \{\Delta_1\})$	=	$[\Omega, \Omega_1](\Delta, \Delta')$ where $\Delta' = \mathsf{topo}\,(\Delta_1)$	

C.3.8 Context Extension in PolyKinds

$\boxed{\Delta \longrightarrow \Theta}$ *(Context Extension)*

$$\textsc{a-ctxe-empty}\quad \frac{}{\bullet \longrightarrow \bullet} \qquad \textsc{a-ctxe-tvar-tt}\quad \frac{\Delta \longrightarrow \Theta}{\Delta, a : \omega \longrightarrow \Theta, a : \omega} \qquad \textsc{a-ctxe-tcon-tt}\quad \frac{\Delta \longrightarrow \Theta}{\Delta, T : \eta \longrightarrow \Theta, T : \eta}$$

$$\textsc{a-ctxe-kuvar-tt}\quad \frac{\Delta \longrightarrow \Theta}{\Delta, \widehat{\alpha} : \omega \longrightarrow \Theta, \widehat{\alpha} : \omega} \qquad \textsc{a-ctxe-kuvarSolved-tt}\quad \frac{\Delta \longrightarrow \Theta \qquad [\Theta]\rho_1 = [\Theta]\rho_2}{\Delta, \widehat{\alpha} : \omega = \rho_1 \longrightarrow \Theta, \widehat{\alpha} : \omega = \rho_2}$$

$$\textsc{a-ctxe-solve-tt}\quad \frac{\Delta \longrightarrow \Theta \qquad \Theta \Vdash^{\mathsf{ela}} \rho : [\Theta]\omega}{\Delta, \widehat{\alpha} : \omega \longrightarrow \Theta, \widehat{\alpha} : \omega = \rho} \qquad \textsc{a-ctxe-add-tt}\quad \frac{\Delta \longrightarrow \Theta \qquad \Theta \Vdash^{\mathsf{ela}} \omega : \star}{\Delta \longrightarrow \Theta, \widehat{\alpha} : \omega}$$

$$\textsc{a-ctxe-addSolved-tt}\quad \frac{\Delta \longrightarrow \Theta \qquad \Theta \Vdash^{\mathsf{ela}} \rho : [\Theta]\omega}{\Delta \longrightarrow \Theta, \widehat{\alpha} : \omega = \rho} \qquad \textsc{a-ctxe-marker}\quad \frac{\Delta \longrightarrow \Theta}{\Delta, \blacktriangleright_D \longrightarrow \Theta, \blacktriangleright_D}$$

$$\textsc{a-ctxe-lo}\quad \frac{\Delta \longrightarrow \Theta \qquad \Delta, \mathsf{topo}\,(\Delta_1) \longrightarrow \Theta, \Theta_1}{\Delta, \{\Delta_1\} \longrightarrow \Theta, \{\Theta_1\}}$$