

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 accept 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

DECLARATION	I
ACKNOWLEDGMENTS	III
LIST OF FIGURES	XI
I PROLOGUE	1
1 INTRODUCTION	3
1.1 Preliminaries	3
1.1.1 Type Inference	3
1.1.2 The Hindley-Milner Type System	4
1.1.3 Higher-rank Polymorphism	5
1.1.4 Implicit Polymorphism	5
1.1.5 Predicativity	6
1.2 Contribution Overview	6
1.2.1 Type Inference for Predicative Implicit Higher-rank Polymorphism	7
1.2.2 Gradually Typed Higher-rank Polymorphism	7
1.2.3 Type Promotion and Kind Inference for Datatypes	9
1.3 Contributions	10
2 BACKGROUND	15
2.1 The Hindley-Milner Type System	15
2.1.1 Declarative System	15
2.1.2 Principal Type Scheme	17
2.1.3 Algorithmic Type System	18
2.2 The Odersky-Läufer Type System	19
2.2.1 Higher-rank Types	19
2.2.2 Declarative System	20
2.2.3 Relating to HM	22

2.2.4	Discussion: Variance and Eta-equality	22
2.3	The Dunfield-Krishnaswami Type System	23
2.3.1	Bidirectional Type Checking	23
2.3.2	Declarative System	25
2.3.3	Algorithmic Type System	28
2.3.4	Discussion: lazy and eager instantiation	28
II	BIDIRECTIONAL TYPE CHECKING WITH THE APPLICATION MODE	31
3	HIGHER-RANK POLYMORPHISM WITH THE APPLICATION MODE	33
3.1	Introduction and Motivation	33
3.1.1	Revisiting Bidirectional Type Checking	33
3.1.2	Type Checking with The Application Mode	35
3.1.3	Benefits of Information Flowing from Arguments to Functions . . .	37
3.1.4	Type Inference of Higher-rank Types	38
3.2	Declarative System	40
3.2.1	Syntax	41
3.2.2	Type System	41
3.2.3	Subtyping	45
3.3	Type-directed Translation	47
3.3.1	Target Language	47
3.3.2	Subtyping Coercions	48
3.3.3	Type-Directed Translation of Typing	51
3.3.4	Type Safety	51
3.3.5	Coherence	52
3.4	Type Inference Algorithm	53
3.5	Discussion	54
3.5.1	Combining Application and Checking Modes	54
3.5.2	Additional Constructs	55
3.5.3	More Expressive Type Applications	56
III	HIGHER-RANK POLYMORPHISM AND GRADUAL TYPING	59
4	GRADUALLY TYPED HIGHER-RANK POLYMORPHISM	61
4.1	Introduction and Motivation	62
4.1.1	Background: Gradual Typing	62

4.1.2	Motivation: Gradually Typed Higher-Rank Polymorphism	64
4.1.3	Application: Efficient (Partly) Typed Encodings of ADTs	65
4.2	Revisiting Consistent Subtyping	68
4.2.1	Consistency and Subtyping	68
4.2.2	Towards Consistent Subtyping	71
4.2.3	Abstracting Gradual Typing	74
4.2.4	Directed Consistency	75
4.2.5	Consistent Subtyping Without Existentials	76
4.3	Gradually Typed Implicit Polymorphism	77
4.3.1	Typing in Detail	78
4.3.2	Type-directed Translation	80
4.3.3	Correctness Criteria	83
4.4	Algorithmic Type System	85
4.4.1	Algorithmic Consistent Subtyping	87
4.4.2	Instantiation	89
4.4.3	Algorithmic Typing	91
4.4.4	Decidability	93
4.4.5	Context Extension	95
4.4.6	Soundness	96
4.4.7	Completeness	97
4.5	Simple Extensions and Variants	98
4.5.1	Top Types	98
4.5.2	A More Declarative Type System	100
5	RESTORING THE DYNAMIC GRADUAL GUARANTEE WITH TYPE PARAMETERS	103
5.1	Declarative Type System	103
5.2	Substitutions and Representative Translations	104
5.3	Dynamic Gradual Guarantee, Reloaded	107
5.4	Extended Algorithmic Type System	108
5.4.1	Extended Algorithmic Consistent Subtyping	109
5.4.2	Extended Instantiation	112
5.4.3	Algorithmic Typing and Metatheory	112
5.4.4	Discussion	114
5.5	Restricted Generalization	114

IV	TYPE INFERENCE WITH PROMOTION	117
6	HIGHER-RANK TYPE INFERENCE WITH TYPE PROMOTION	119
6.1	Introduction and Motivation	119
6.1.1	Background: Type Inference in Context	119
6.1.2	Our Approach: Type Promotion	121
6.1.3	Polymorphic Promotion	122
6.2	Unification for the Simply Typed Lambda Calculus	124
6.2.1	Declarative System	124
6.2.2	Algorithmic System	124
6.2.3	Soundness and Completeness	126
6.3	Subtyping for Higher-Rank Polymorphism	127
6.3.1	Declarative System	127
6.3.2	Algorithmic System	127
6.3.3	Soundness and Completeness	129
6.4	Discussion	131
6.4.1	Promoting Dependent Types	131
6.4.2	Promoting Gradual Types	132
7	KIND INFERENCE FOR DATATYPES	133
7.1	Introduction and Motivation	133
7.2	Overview	135
7.2.1	Kind Inference in Haskell98	135
7.2.2	Kind Inference in Modern GHC Haskell	136
7.2.3	Desirable Properties for Kind Inference	139
7.3	Datatypes in Haskell98	140
7.3.1	Groups and Dependency Analysis	140
7.3.2	Declarative Typing Rules	140
7.4	Kind Inference for Haskell98	142
7.4.1	Syntax	142
7.4.2	Algorithmic Typing Rules	142
7.4.3	Defaulting	144
7.4.4	Checking Datatype Declarations	145
7.4.5	Kinding	145
7.4.6	Unification	147
7.4.7	Soundness and Completeness	147

7.5	Type Parameters, Principal Kinds and Completeness in Haskell98	148
7.5.1	Type Parameters	149
7.5.2	Principal Kinds and Defaulting	149
7.5.3	Completeness	150
7.6	Declarative Syntax and Semantics of PolyKinds	150
7.6.1	Groups and Dependency Analysis	150
7.6.2	Checking Kinds	155
7.7	Kind Inference for PolyKinds	156
7.7.1	Algorithmic Program Typing	156
7.7.2	The Quantification Check	158
7.7.3	Kinding	160
7.7.4	Unification	160
7.7.5	Termination	162
7.7.6	Soundness, Completeness and Principality	165
7.8	Language Extensions	166
7.8.1	Higher-Rank Polymorphism	166
7.8.2	Generalized Algebraic Datatypes (GADTs)	167
7.8.3	Type Families	168
V	EPILOGUE	169
8	RELATED WORK	171
8.1	Type Inference for Higher-Rank Types	171
8.2	Bidirectional Type Checking	172
8.3	Gradual Typing	173
8.4	Gradual Type Systems with Explicit Polymorphism	174
8.5	Gradual Type Inference	175
8.6	Haskell and GHC	175
8.7	Unification with dependent types	176
9	SUMMARY AND FUTURE DIRECTIONS	179
9.1	Dependent Type Systems with Application mode	179
9.2	Type Inference for Intersection Type Systems	180
9.3	Gradualizing Type Classes	181
9.4	Generalized Algebraic Datatypes (GADTs)	182

BIBLIOGRAPHY	185
VI TECHNICAL APPENDIX	199
A FULL RULES FOR ALGORITHMIC AP	201
B THE EXTENDED ALGORITHMIC GPC	205
B.1 Syntax	205
B.2 Type System	205
C KIND INFERENCE FOR DATATYPES	209
C.1 Other Language Extensions	209
C.1.1 Visible Dependent Quantification	209
C.1.2 Datatype Promotion	210
C.1.3 Partial Type Signatures	210
C.2 Today’s GHC	211
C.2.1 Constraint-Based Type Inference	211
C.2.2 Contexts	211
C.2.3 Unification	212
C.2.4 Promotion	213
C.2.5 Complete User-Supplied Kinds	213
C.2.6 Dependency Analysis	214
C.2.7 Approach to Kind-Checking Datatypes	214
C.2.8 Polymorphic Recursion	215
C.2.9 The Quantification Check	216
C.2.10 ScopedSort	217
C.2.11 The “Forall-or-Nothing” Rule	218
C.3 Complete Set of Rules	218
C.3.1 Declarative Haskell98	219
C.3.2 Algorithmic Haskell98	219
C.3.3 Context Application in Haskell98	220
C.3.4 Context Extension in Haskell98	221
C.3.5 Declarative PolyKinds	221
C.3.6 Algorithmic PolyKinds	223
C.3.7 Context Application in PolyKinds	228
C.3.8 Context Extension in PolyKinds	229

LIST OF FIGURES

2.1	Syntax and static semantics of the Hindley-Milner type system.	16
2.2	Declarative subtyping in the Hindley-Milner type system.	17
2.3	Syntax of the Odersky-Läufer type system.	20
2.4	Well-formedness of types in the Odersky-Läufer type system.	21
2.5	Static semantics of the Odersky-Läufer type system.	21
2.6	Syntax of the Dunfield-Krishnaswami Type System	24
2.7	Static semantics of the Dunfield-Krishnaswami type system.	26
3.1	Syntax of System AP.	41
3.2	Typing rules of System AP.	42
3.3	Syntax and typing rules of System F.	48
3.4	Subtyping translation rules of System AP.	49
3.5	Typing translation rules of System AP.	50
3.6	Type erasure and eta-id equality of System F.	52
4.1	Subtyping and type consistency in $\mathbf{FOb}_{<}^?$	62
4.2	Syntax of types, consistency, subtyping and well-formedness of types in declarative GPC.	69
4.3	Examples that break the original definition of consistent subtyping.	72
4.4	Observations of consistent subtyping	73
4.5	Example that is fixed by the new definition of consistent subtyping.	74
4.6	Consistent Subtyping for implicit polymorphism.	77
4.7	Syntax of expressions and declarative typing of declarative GPC	79
4.8	Less Precision	84
4.9	Syntax and well-formedness of the algorithmic GPC	86
4.10	Algorithmic consistent subtyping	88
4.11	Algorithmic instantiation	90
4.12	Algorithmic typing	92
4.13	Context extension	96

List of Figures

5.1	Syntax of types, and consistent subtyping in the extended declarative system.	104
5.2	Syntax of types, contexts and consistent subtyping in the extended algorithmic system.	108
5.3	Extended algorithmic consistent subtyping	110
5.4	Instantiation in the extended algorithmic system	113
6.1	Types, contexts, unification and promotion of algorithmic STLC	125
6.2	Types, contexts, subtyping and (polymorphic) promotion of the algorithmic system	128
7.1	Declarative specification of Haskell98 datatype declarations	141
7.2	Algorithmic program typing in Haskell98	143
7.3	Algorithmic kinding, unification and promotion in Haskell98.	146
7.4	Syntax of PolyKinds	151
7.5	Declarative specification of PolyKinds	152
7.6	Selected rules for declarative kind-checking in PolyKinds	153
7.7	Algorithmic syntax in PolyKinds	155
7.8	Algorithmic program typing in PolyKinds	157
7.9	Selected rules for algorithmic kinding in PolyKinds	159
7.10	Selected rules for unification, promotion, and moving in PolyKinds	161
7.11	Example of dependency graph	164

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.

1 Introduction

26 However, writing type annotations can be tedious, especially when the type annotations
27 can be *inferred* from the context. Consider the definition of `add`, which uses the built-in
28 primitive `+` : `Int → Int → Int`¹.

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

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

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

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

39 1.1.2 THE HINDLEY-MILNER TYPE SYSTEM

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

44 For example, consider the expression

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

46 There are many possible types we can give for `id`, including `Int → Int`, and `Bool → Bool`,
47 etc. In this case, HM will derive the principal type for `id`: $\forall a. a \rightarrow a$. a *polymorphic* type
48 with a universal quantifier over the type variable `a`. We call types without universal quan-
49 tifiers, like `Int → Int` and `Bool → Bool`, *monomorphic types* (i.e., *monotypes*), and types
50 like $\forall a. a \rightarrow a$ polymorphic types. For this example, from the principal type $\forall a. a \rightarrow a$,
51 other types like `Int → Int` and `Bool → Bool` can be derived by instantiating `a` to `Int` and
52 `Bool` respectively. With the principal type, we can use `id` as in the following program:

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

¹The 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.

55 1.1.3 HIGHER-RANK POLYMORPHISM

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

58 $\forall a. a \rightarrow a$

59 is a valid type, while

60 $(\forall a. a \rightarrow a) \rightarrow \text{int}$

61 is not as \forall appears inside the \rightarrow constructor.

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

68 `runST : $\forall a. (\forall s. \text{ST } s \ a) \rightarrow a$`

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

71 1.1.4 IMPLICIT POLYMORPHISM

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

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

77 `map :: $\forall a \ b. (a \rightarrow b) \rightarrow [a] \rightarrow [b]$`

78 `fst :: $\forall a \ b. (a, b) \rightarrow a$`

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

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

82 `-- [(1, 2)]`

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

1 Introduction

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

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

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

93 1.1.5 PREDICATIVITY

94 In a system with polymorphism, one important design decision to make is whether the sys-
95 tem is *predicative* or *impredicative*.

96 A system is predicative, if the type variable bound by a universal quantifier is only allowed
97 to be instantiated by a monotype; otherwise it is impredicative. For example, instantiating a
98 with \mathbf{Int} in $\forall a. a \rightarrow a$, generating $\mathbf{Int} \rightarrow \mathbf{Int}$, is predicative; while instantiating a with $\forall a.$
99 $a \rightarrow \mathbf{Int}$ in $\forall a. a \rightarrow a$, generating $(\forall a. a \rightarrow \mathbf{Int}) \rightarrow (\forall a. a \rightarrow \mathbf{Int})$, is impredicative.
100 HM is an example of predicative polymorphic system, with universal quantifiers restricted to
101 the top-level, while System F is impredicative. It is well-known that general type inference for
102 impredicativity is undecidable [Wells 1999]. The most recent line of work in impredicativity
103 can be found in work by Serrano et al. [2020, 2018].

104 Type inference for a predicative type system is much easier, while still enables most of the
105 expressiveness of higher-rank polymorphism. Thus just like Dunfield and Krishnaswami
106 [2013]; Peyton Jones et al. [2007], in this work, we focus on *predicative implicit higher-rank*
107 *polymorphism*. In the rest of this dissertation, whenever we refer to *higher-rank polymor-*
108 *phism*, unless otherwise specified, it denotes predicative implicit higher-rank polymorphism.

109 1.2 CONTRIBUTION OVERVIEW

110 The goal of this dissertation is to explore the design space of type inference for implicit pred-
111 icative higher-rank polymorphism, as well as to study the integration of techniques we have
112 developed into other advanced type system features including *gradual typing* [Siek and Taha
113 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 $\text{id} : \forall a. a \rightarrow a$, 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 $\lambda x. \lambda y. x + y$, 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 $\forall a. a \rightarrow a$, 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 $(\lambda x. x + 1) : \text{Int} \rightarrow \text{Int}$. 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, $(\lambda f. (f 1, f 'a'))$).

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., $\forall a. a \rightarrow a$) to a more specific one (e.g., `Int \rightarrow Int`) so that for example `id` can be used where an expression of

182 type **Int** \rightarrow **Int** is expected. On the other hand, gradual typing deals with the powerful un-
 183 known type, so that an expression with the unknown type can be used as an expression of any
 184 type. We show that existing design of such integration [Siek and Taha 2007] is inadequate,
 185 and we provide a generalized design that is able to deal with higher-rank polymorphism. Sec-
 186 ond, we must ensure that our system is well-designed, by showing that our system satisfies
 187 the *correctness criteria* [Siek et al. 2015]. We will show that the *dynamic gradual guarantee* is
 188 particular tricky to deal with.

189 1.2.3 TYPE PROMOTION AND KIND INFERENCE FOR DATATYPES

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

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

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

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

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

215 `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* `* → *`. 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:

- 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 Gradually Polymorphic Calculus: 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

²Note 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.

³Proofs in <https://bitbucket.org/ningningxie/let-arguments-go-first/src/master/>.

⁴Proofs in <https://github.com/xnning/Consistent-Subtyping-for-All>.

⁵Proofs in <https://github.com/xnning/Consistent-Subtyping-for-All>.

⁶Proofs in <https://xnning.github.io/papers/sanitized-type-inference-in-context.pdf>.

⁷Proofs 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 bidirectional 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 **let** $x = e_1$ **in** e_2 . Note here lambda abstractions have no type annotations, and the type information is to be reconstructed by the type system.

2 Background

Expressions	$e ::= x \mid n \mid \lambda x. e \mid e_1 e_2 \mid \mathbf{let} x = e_1 \mathbf{in} e_2$
Types	$\sigma ::= \forall \bar{a}_i^i. \tau$
Monotypes	$\tau ::= \mathbf{Int} \mid a \mid \tau_1 \rightarrow \tau_2$
Contexts	$\Psi ::= \bullet \mid \Psi, x : \sigma$

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

(Typing)

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

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

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 \mathbf{Int} , type variables a and function types $\tau_1 \rightarrow \tau_2$. 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 $\tau_1 \rightarrow \tau_2$. 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 \bar{a}_i^i which are not bound in

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

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

Figure 2.2: 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:

$$\begin{array}{c}
\vdash^{HM} \quad \text{Int} \rightarrow \text{Int} <: \text{Int} \rightarrow \text{Int} \\
\vdash^{HM} \quad \forall a. a \rightarrow a <: \text{Int} \rightarrow \text{Int}
\end{array}$$

Given the subtyping relation, now we can formally state that HM enjoys *principality*. That is, for every well-typed expression in HM, there exists one type for the expression, which is more general than any other types the expression can derive. Formally,

Theorem 2.1 (Principality for HM). *If $\Psi \vdash^{HM} e : \sigma$, then there exists σ' such that $\Psi \vdash^{HM} e : \sigma'$, and for all σ'' such that $\Psi \vdash^{HM} e : \sigma''$, we have $\vdash^{HM} \sigma' <: \sigma''$.*

Consider the expression $\lambda x. x$. It has a principal type $\forall a. a \rightarrow a$, which is more general than any other options, e.g., $\text{Int} \rightarrow \text{Int}$, $(\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})$, etc.

2 Background

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,

$$\begin{array}{c}
 \text{HM-VAR-INST} \\
 \frac{(x : \forall \bar{a}_i^i. \tau) \in \Psi}{\Psi \vdash^{HM} x : \tau[\bar{a}_i \mapsto \bar{\tau}_i^i]}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{HM-LET-GEN} \\
 \frac{\Psi \vdash^{HM} e_1 : \tau \quad \bar{a}_i^i = \text{FV}(\tau) - \text{FV}(\Psi) \quad \Psi, x : \forall \bar{a}_i^i. \tau \vdash^{HM} e_2 : \tau}{\Psi \vdash^{HM} \text{let } x = e_1 \text{ in } e_2 : \tau}
 \end{array}$$

A syntax-directed subtyping relation can also be derived, by restricting rule [HM-S-FORALL](#) to allow only for a monotype on the right:

$$\begin{array}{c}
 \text{HM-S-A-FORALL} \\
 \frac{\vdash^{HM} \sigma[a \mapsto \tau_1] <: \tau}{\vdash^{HM} \forall a. \sigma <: \tau}
 \end{array}$$

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 program¹:

```
(\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 $\forall a. a \rightarrow a$ 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,

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

Below we give some examples:

¹For the purpose of illustration, we assume basic constructs like booleans and pairs in examples.

2 Background

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

Figure 2.3: Syntax of the Odersky-Läufer type system.

	$\text{rank}(\text{Int} \rightarrow \text{Int})$	$=$	0
	$\text{rank}(\forall a. a \rightarrow a)$	$=$	1
435	$\text{rank}(\text{Int} \rightarrow (\forall a. a \rightarrow a))$	$=$	1
	$\text{rank}((\forall a. a \rightarrow a) \rightarrow \text{Int})$	$=$	2

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

438 2.2.2 DECLARATIVE SYSTEM

439 SYNTAX. The syntax of OL is given in Figure 2.3. Comparing to HM, we observe the fol-
440 lowing differences.

441 First, expressions e include not only unannotated lambda abstractions $\lambda x. e$, but also an-
442 notated lambda abstractions $\lambda x : \sigma. e$, where the type annotation σ can be a polymorphic
443 type. Thus unlike HM, the argument type for a function is not limited to a monotype.

444 Second, the polymorphic types σ now include the integer type Int , type variables a , func-
445 tions $\sigma_1 \rightarrow \sigma_2$ and universal quantifications $\forall a. \sigma$. Since the argument type in a function can
446 be polymorphic, we see that OL supports *arbitrary* rank of types. The definition of mono-
447 types remains the same, with polymorphic types still subsuming monotypes.

448 Finally, in addition to variable types, the contexts Ψ now also keep track of type variables.
449 Note that in the original work in Odersky and Läufer [1996], the system, much like HM,
450 does not track type variables; instead, it explicitly checks that type variables are fresh with
451 respect to a context or a type when needed. The difference is more presentational rather
452 than semantic. Here we include type variables in contexts, as it sets us well for the Dunfield-
453 Krishnaswami type system to be introduced in the next section. Moreover, it provides a
454 complete view of possible formalisms of contexts in a type system with generalization.

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

458 TYPE SYSTEM. The typing rules for OL are given in Figure 2.5.

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

$$\begin{array}{c}
 \text{OL-WF-INT} \\
 \frac{}{\Psi \vdash^{OL} \text{Int}} \\
 \text{OL-WF-TVAR} \\
 \frac{a \in \Psi}{\Psi \vdash^{OL} a} \\
 \text{OL-WF-ARROW} \\
 \frac{\Psi \vdash^{OL} \sigma_1 \quad \Psi \vdash^{OL} \sigma_2}{\Psi \vdash^{OL} \sigma_1 \rightarrow \sigma_2} \\
 \text{OL-WF-FORALL} \\
 \frac{\Psi, a \vdash^{OL} \sigma}{\Psi \vdash^{OL} \forall a. \sigma}
 \end{array}$$

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

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

$$\begin{array}{c}
 \text{OL-VAR} \\
 \frac{(x : \sigma) \in \Psi}{\Psi \vdash^{OL} x : \sigma} \quad \text{OL-INT} \\
 \frac{}{\Psi \vdash^{OL} n : \text{Int}} \quad \text{OL-LAMANN} \\
 \frac{\Psi, x : \sigma_1 \vdash^{OL} e : \sigma_2}{\Psi \vdash^{OL} \lambda x : \sigma_1. e : \sigma_1 \rightarrow \sigma_2} \\
 \text{OL-LAM} \\
 \frac{\Psi \vdash^{OL} \tau \quad \Psi, x : \tau \vdash^{OL} e : \sigma}{\Psi \vdash^{OL} \lambda x. e : \tau \rightarrow \sigma} \quad \text{OL-APP} \\
 \frac{\Psi \vdash^{OL} e_1 : \sigma_1 \rightarrow \sigma_2 \quad \Psi \vdash^{OL} e_2 : \sigma_1}{\Psi \vdash^{OL} e_1 e_2 : \sigma_2} \\
 \text{OL-LET} \\
 \frac{\Psi \vdash^{OL} e_1 : \sigma_1 \quad \Psi, x : \sigma_1 \vdash^{OL} e_2 : \sigma_2}{\Psi \vdash^{OL} \text{let } x = e_1 \text{ in } e_2 : \sigma_2} \quad \text{OL-GEN} \\
 \frac{\Psi, a \vdash^{OL} e : \sigma}{\Psi \vdash^{OL} e : \forall a. \sigma} \\
 \text{OL-SUB} \\
 \frac{\Psi \vdash^{OL} e : \sigma_1 \quad \Psi \vdash^{OL} \sigma_1 <: \sigma_2}{\Psi \vdash^{OL} e : \sigma_2}
 \end{array}$$

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

$$\begin{array}{c}
 \text{OL-S-TVAR} \\
 \frac{a \in \Psi}{\Psi \vdash^{OL} a <: a} \quad \text{OL-S-INT} \\
 \frac{}{\Psi \vdash^{OL} \text{Int} <: \text{Int}} \quad \text{OL-S-ARROW} \\
 \frac{\Psi \vdash^{OL} \sigma_3 <: \sigma_1 \quad \Psi \vdash^{OL} \sigma_2 <: \sigma_4}{\Psi \vdash^{OL} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4} \\
 \text{OL-S-FORALLL} \\
 \frac{\Psi \vdash^{OL} \tau \quad \Psi \vdash^{OL} \sigma[a \mapsto \tau] <: \sigma_2}{\Psi \vdash^{OL} \forall a. \sigma_1 <: \sigma_2} \quad \text{OL-S-FORALLR} \\
 \frac{\Psi, a \vdash^{OL} \sigma_1 <: \sigma_2}{\Psi \vdash^{OL} \sigma_1 <: \forall a. \sigma_2}
 \end{array}$$

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

2 Background

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 $\forall \bar{a}_i^i. \tau <: \tau[\bar{a}_i \mapsto \bar{\tau}_i^i]$ by applying rule **HM-S-FORALLL** repeatedly.

The subtyping relation of OL $\Psi \vdash^{OL} \sigma_1 <: \sigma_2$ also generalizes the subtyping relation of HM. In particular, in rule **OL-S-ARROW**, functions are *contravariant* on arguments, and *covariant* on return types. This rule allows us to compare higher-rank polymorphic types, rather than just polymorphic types with universal quantifiers only at the top level. For example,

$$\begin{array}{ll} \Psi \vdash^{OL} \forall a. a \rightarrow a & <: \text{Int} \rightarrow \text{Int} \\ \Psi \vdash^{OL} \text{Int} \rightarrow (\forall a. a \rightarrow a) & <: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \\ \Psi \vdash^{OL} (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} & <: (\forall a. a \rightarrow a) \rightarrow \text{Int} \end{array}$$

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, $\lambda x. \text{undefined } x$, 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 referred 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	$::=$	$\text{Int} \mid a \mid \sigma_1 \rightarrow \sigma_2 \mid \forall a. \sigma$
Monotypes	τ	$::=$	$\text{Int} \mid a \mid \tau_1 \rightarrow \tau_2$
Contexts	Ψ	$::=$	$\bullet \mid \Psi, x : \sigma \mid \Psi, a$

Figure 2.6: Syntax of the Dunfield-Krishnaswami Type System

520 “... missing annotations are recovered using only information from adjacent nodes
 521 in the syntax tree, without long-distance constraints such as unification variables.”

522 Bidirectional type checking is one component of local type inference that, aided by some
 523 type annotations, enables type inference in an expressive language with polymorphism and
 524 subtyping. In its basic form typing is split into *inference* and *checking* modes. The most salient
 525 feature of a bidirectional type-checker is when information deduced from inference mode is
 526 used to guide checking of an expression in checking mode.

527 Since Pierce and Turner’s work, various other authors have proved the effectiveness of
 528 bidirectional type checking in several other settings, including many different systems with
 529 subtyping [Davies and Pfenning 2000; Dunfield and Pfenning 2004], systems with dependent
 530 types [Asperti et al. 2012; Coquand 1996; Löh et al. 2010; Xi and Pfenning 1999], etc.

531 In particular, bidirectional type checking has also been combined with HM-style tech-
 532 niques for providing type inference in the presence of higher-rank type, including DK and
 533 Peyton Jones et al. [2007]. Let’s revisit the example in Section 2.2:

534 $(\backslash f. (f\ 1, f\ 'a')) (\backslash x. x)$

535 which is not typeable in HM as it they fail to infer the type of f . In OL, it can be type-checked
 536 by adding a polymorphic type annotation on f . In DK, we can also add a polymorphic type
 537 annotation on f . But with bidirectional type checking, the type annotation can be propagated
 538 from somewhere else. For example, we can rewrite this program as:

539 $((\backslash f. (f\ 1, f\ 'c')) : (\forall a. a \rightarrow a) \rightarrow (\text{Int}, \text{Char})) (\backslash x. x)$

540 Here the type of f can be easily derived from the type signature using checking mode in
 541 bidirectional type checking.

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

548 2.3.2 DECLARATIVE SYSTEM

549 SYNTAX. The syntax of the DK is given in Figure 2.6. Comparing to OL, only the defini-
 550 tion of expressions slightly differs. First, the expressions e in DK have no let expressions.
 551 Dunfield and Krishnaswami [2013] omitted let-bindings from the formal development, but
 552 argued that restoring let-bindings is easy, as long as they get no special treatment incom-
 553 patible with substitution (e.g., a syntax-directed HM does polymorphic generalization only
 554 at let-bindings). Second, DK has annotated expressions $e : \sigma$ (instead of annotated lambda
 555 expressions $\lambda x : \sigma. e$), in which the type annotation can be propagated into the expression,
 556 as we will see shortly.

557 The definitions of types and contexts are the same as in OL. Thus, DK also shares the
 558 same well-formedness definition as in OL (Figure 2.4). We thus omit the definitions, but use
 559 $\Psi \vdash^{DK} \sigma$ to denote the corresponding judgment in DK.

560 TYPE SYSTEM. Figure 2.7 presents the typing rules for DK. The system uses bidirectional
 561 type checking to accommodate polymorphism. Traditionally, two modes are employed in
 562 bidirectional systems: the inference mode $\Psi \vdash^{DK} e \Rightarrow \sigma$, which takes a term e and produces
 563 a type σ , similar to the judgment $\Psi \vdash^{HM} e : \sigma$ or $\Psi \vdash^{OL} e : \sigma$ in previous systems; the
 564 checking mode $\Psi \vdash^{DK} e \Leftarrow \sigma$, which takes a term e and a type σ as input, and ensures that
 565 the term e checks against σ . We first discuss rules in the inference mode.

566 TYPE INFERENCE. Rule **DK-INF-VAR** and rule **DK-INF-INT** are straightforward. To infer unan-
 567 notated lambdas, rule **DK-INF-LAM** guesses a monotype. For an application $e_1 e_2$, rule **DK-**
 568 **INF-APP** first infers the type σ of the expression e_1 . The *application judgment* (discussed
 569 shortly) then takes the type σ and the argument e_2 , and returns the final result type σ_2 . For
 570 an annotated expression $e : \sigma$, rule **DK-INF-ANNO** simply checks e against σ . Both rules
 571 (rule **DK-INF-APP** and rule **DK-INF-ANNO**) have mode switched from inference to checking.

572 TYPE CHECKING. Now we turn to the checking mode. When an expression is checked
 573 against a type, the expression is expected to have that type. More importantly, the checking
 574 mode allows us to push the type information into the expressions.

575 Rule **DK-CHK-INT** checks literals against the integer type `Int`. Rule **DK-CHK-LAM** is where
 576 the system benefits from bidirectional type checking: the type information gets pushed in-
 577 side an lambda. For an unannotated lambda abstraction $\lambda x. e$, recall that in the inference
 578 mode, we can only guess a monotype for x . With the checking mode, when $\lambda x. e$ is checked
 579 against $\sigma_1 \rightarrow \sigma_2$, we do not need to guess any type. Instead, x gets directly the (possibly
 580 polymorphic) argument type σ_1 . Then the rule proceeds by checking e with σ_2 , allowing the

2 Background

$\Psi \vdash^{DK} e \Rightarrow \sigma$	<i>(Type Inference)</i>
$\frac{\text{DK-INF-VAR} \quad (x : \sigma) \in \Psi}{\Psi \vdash^{DK} x \Rightarrow \sigma}$	$\frac{\text{DK-INF-INT}}{\Psi \vdash^{DK} n \Rightarrow \text{Int}}$
$\frac{\text{DK-INF-LAM} \quad \Psi \vdash^{DK} \tau_1 \rightarrow \tau_2 \quad \Psi, x : \tau_1 \vdash^{DK} e \Rightarrow \tau_2}{\Psi \vdash^{DK} \lambda x. e \Rightarrow \tau_1 \rightarrow \tau_2}$	
$\frac{\text{DK-INF-APP} \quad \Psi \vdash^{DK} e_1 \Rightarrow \sigma \quad \Psi \vdash^{DK} \sigma \cdot e_2 \Rightarrow \sigma_2}{\Psi \vdash^{DK} e_1 e_2 \Rightarrow \sigma_2}$	$\frac{\text{DK-INF-ANNO} \quad \Psi \vdash^{DK} e \Leftarrow \sigma}{\Psi \vdash^{DK} e : \sigma \Rightarrow \sigma}$
$\Psi \vdash^{DK} e \Leftarrow \sigma$	<i>(Type Checking)</i>
$\frac{\text{DK-CHK-INT}}{\Psi \vdash^{DK} n \Leftarrow \text{Int}}$	$\frac{\text{DK-CHK-LAM} \quad \Psi, x : \sigma_1 \vdash^{DK} e \Leftarrow \sigma_2}{\Psi \vdash^{DK} \lambda x. e \Leftarrow \sigma_1 \rightarrow \sigma_2}$
$\frac{\text{DK-CHK-GEN} \quad \Psi, a \vdash^{DK} e \Leftarrow \sigma}{\Psi \vdash^{DK} e \Leftarrow \forall a. \sigma}$	
$\frac{\text{DK-CHK-SUB} \quad \Psi \vdash^{DK} e \Rightarrow \sigma_1 \quad \Psi \vdash^{DK} \sigma_1 <: \sigma_2}{\Psi \vdash^{DK} e \Leftarrow \sigma_2}$	
$\Psi \vdash^{DK} \sigma_1 \cdot e \Rightarrow \sigma_2$	<i>(Application judgment)</i>
$\frac{\text{DK-APP-FORALL} \quad \Psi \vdash^{DK} \tau \quad \Psi \vdash^{DK} \sigma[a \mapsto \tau] \cdot e \Rightarrow \sigma_1 \rightarrow \sigma_2}{\Psi \vdash^{DK} \forall a. \sigma \cdot e \Rightarrow \sigma_1 \rightarrow \sigma_2}$	$\frac{\text{DK-APP-ARR} \quad \Psi \vdash^{DK} e \Leftarrow \sigma_1}{\Psi \vdash^{DK} \sigma_1 \rightarrow \sigma_2 \cdot e \Rightarrow \sigma_2}$

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 $\Psi \vdash^{DK} \sigma_1 \cdot e \Rightarrow \sigma_2$ 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 monotone type, 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**.

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

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 $\sigma_1 \rightarrow \sigma_2$, and proceed by checking e_2 against σ_1 , and return the final result σ_2 . The definition of matching is given below.

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

$$\begin{array}{c}
 \text{DK-M-FORALL} \qquad \text{DK-M-ARR} \\
 \frac{\Psi \vdash^{DK} \tau \quad \Psi \vdash^{DK} \sigma[a \mapsto \tau] \triangleright \sigma_1 \rightarrow \sigma_2}{\Psi \vdash^{DK} \forall a. \sigma \triangleright \sigma_1 \rightarrow \sigma_2} \qquad \frac{}{\Psi \vdash^{DK} \sigma_1 \rightarrow \sigma_2 \triangleright \sigma_1 \rightarrow \sigma_2}
 \end{array}$$

2 Background

607 Matching has two straightforward rules: rule **DK-M-FORALL** instantiates a polymorphic
608 type, by substituting a with a well-formed monotype τ , and continues matching on $\sigma[a \mapsto \tau]$;
609 rule **DK-M-ARR** returns the function type directly.

610 It can be easily shown that the presentation of rule **DK-INF-APP** with the application judg-
611 ment is equivalent to that of rule **DK-INF-APP2** with matching. Essentially, they both make
612 sure that the expression being applied has an arrow type $\sigma_1 \rightarrow \sigma_2$, and then check the ar-
613 gument against σ_1 . We sometimes use the presentation of rule **DK-INF-APP2** with matching,
614 as matching is a simple and independent process whose purpose is clear. In contrast, it is
615 relatively less comprehensible with rule **DK-INF-APP** and the application judgment, where all
616 three forms of the judgment (inference, checking, application) are mutually dependent.

617 **SUBTYPING.** DK shares the same subtyping relation as of OL. We thus omit the definition
618 and use $\Psi \vdash^{DK} \sigma_1 <: \sigma_2$ to denote the subtyping relation in DK.

619 2.3.3 ALGORITHMIC TYPE SYSTEM

620 Dunfield and Krishnaswami [2013] also presented a sound and complete bidirectional algo-
621 rithmic type system. The key idea of the algorithm is using *ordered* algorithmic contexts for
622 storing existential variables and their solutions. Comparing to the algorithm for HM, they
623 argued that their algorithm is remarkably simple. The algorithm is later discussed and used
624 in Part III and Part IV. We will discuss more about it there.

625 2.3.4 DISCUSSION: LAZY AND EAGER INSTANTIATION

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

631 The differences between lazy and eager instantiation have pervasive consequences. The
632 first and direct consequence is that for the same expression lazy instantiation can derive a
633 more polymorphic type. Consider that given $\text{id}: \forall a. a \rightarrow a$, we want to infer the type of
634 id . With lazy instantiation, the algorithm can directly return $\forall a. a \rightarrow a$ unchanged, while
635 eager instantiation can only return $a \rightarrow a$ (among others). In this case, we can make two
636 types match by generalizing the final result from eager instantiation, which gives us $\forall a.$
637 $a \rightarrow a$. However, when the system features higher-rank types, after generalization eager
638 instantiation may still fail to derive the same type. For example, consider the expression

639 $\backslash x: \mathbf{Int}. \text{id}$

640 In this case, lazy instantiation returns the type $\mathbf{Int} \rightarrow \forall a. a \rightarrow a$, while eager instantiation
 641 returns the type $\mathbf{Int} \rightarrow a \rightarrow a$ which, after generalization, becomes $\forall a. \mathbf{Int} \rightarrow a \rightarrow a$,
 642 and is less polymorphic than $\mathbf{Int} \rightarrow \forall a. a \rightarrow a$, according to the subtyping relation in OL.
 643 In this sense naive eager instantiation may reject programs that lazy instantiation can accept,
 644 e.g.,

645 $\text{let } g = \backslash x: \mathbf{Int}. \text{id} \text{ in } (\backslash f: \mathbf{Int} \rightarrow \forall a. a \rightarrow a. f) g$

646 The problem can be avoided by, for example, featuring *deep skolemisation* as in Peyton Jones
 647 et al. [2007]. Deep skolemisation floats out all its universal quantifiers that appear to the
 648 right of a top level arrow, so that $\forall a. \mathbf{Int} \rightarrow a \rightarrow a$ and $\mathbf{Int} \rightarrow \forall a. a \rightarrow a$ are actually
 649 isomorphic.

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

655 $\text{if } \text{flag} \text{ then } \text{id} \text{ else } \backslash x. x + 1$

656 We expect the two branches in **if** to return the same type. Therefore when collecting the re-
 657 sult type from branches, we must instantiate the type of `id` to $\mathbf{Int} \rightarrow \mathbf{Int}$, so that it matches
 658 the type of another branch. Peyton Jones et al. [2007] discuss three possible typing formal-
 659 izations of **if** expressions under eager instantiation. To adopt either formalization in lazy
 660 instantiation systems, instantiation of the return types of the branches must be performed.

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

665 $\text{let } f = \backslash x. x$
 666 $\text{in let } g = \backslash y. f$
 667 $\text{in } e$

668 If we generalize the type of `f` as in HM, then we get $f: \forall a. a \rightarrow a$. Now we continue type-
 669 checking `g`, which requires another step of generalization. With lazy instantiation, we get
 670 $g: \forall b. b \rightarrow \forall a. a \rightarrow a$, which is again, more polymorphic than the type $g: \forall b a. b \rightarrow$
 671 $a \rightarrow a$ we get from eager instantiation. However, if we inline the definition of `f`, then the
 672 program becomes

2 Background

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

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

680 In this thesis, the choice does not matter so much for the key contributions of the work.
681 As we do not feature deep skolemisation, we mostly follow the idea of lazy instantiation as
682 in DK. Nevertheless, we believe that deep skolemisation is compatible with our subtyping
683 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 $(\lambda f : \text{Int} \rightarrow \text{Int}. f) (\lambda y. y)$, 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:

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

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

One guideline for designing bidirectional type checking rules [Dunfield and Pfenning 2004] is to distinguish introduction rules from elimination rules. Constructs which correspond to introduction forms are *checked* against a given type, while constructs corresponding to elimination forms *infer* (or synthesize) their types. For instance, under this design principle, the introduction rule for literals is supposed to be in the checking mode, as in the rule [DK-CHK-INT](#):

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

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

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](#).

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

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.

$$\begin{array}{c} \text{INF-PAIR} \\ \frac{\Psi \vdash e_1 \Rightarrow \sigma_1 \quad \Psi \vdash e_2 \Rightarrow \sigma_2}{\Psi \vdash (e_1, e_2) \Rightarrow (\sigma_1, \sigma_2)} \end{array} \quad \begin{array}{c} \text{CHECK-PAIR} \\ \frac{\Psi \vdash e_1 \Leftarrow \sigma_1 \quad \Psi \vdash e_2 \Leftarrow \sigma_2}{\Psi \vdash (e_1, e_2) \Leftarrow (\sigma_1, \sigma_2)} \end{array}$$

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

$(\lambda x. x) 1$

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 required¹. For example, $(\lambda x. x) : \text{Int} \rightarrow \text{Int} \vdash 1$.

In this example we can observe that if the type of the argument is accounted for in inferring the type of $\lambda x. x$, then it is actually possible to deduce that the lambda expression has type $\text{Int} \rightarrow \text{Int}$, 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,

$$\frac{\text{APP} \quad \Psi \vdash e_2 \Rightarrow \sigma_1 \quad \Psi; \Sigma, \sigma_1 \vdash e_1 \Rightarrow \sigma \rightarrow \sigma_2}{\Psi; \Sigma \vdash e_1 e_2 \Rightarrow \sigma_2}$$

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

¹It 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:

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

The type σ that appears last in the application context serves as the type for x , and type checking continues with a smaller application context and $x : \sigma$ in the typing context. Therefore, using the 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 con-

779 text means that an expression is not being applied to anything, which allows us to model the
 780 inference mode as a particular case².

781 **PARTIAL TYPE CHECKING.** The inference mode synthesizes the type of an expression, and
 782 the checking mode checks an expression against some type. A natural question is how do
 783 these modes compare to the application mode. An answer is that, in some sense: the ap-
 784 plication mode is stronger than the inference mode, but weaker than the checking mode.
 785 Specifically, the inference mode means that we know nothing about the type of an expres-
 786 sion before hand. The checking mode means that the whole type of the expression is already
 787 known before hand. With the application mode we know some partial type information
 788 about the type of an expression: we know some of its argument types (since it must be a
 789 function type when the application context is non-empty), but not the return type.

790 Instead of nothing or all, this partialness gives us a finer grain notion on how much we
 791 know about the type of an expression. For example, assume $e : \sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3$. In the
 792 inference mode, we only have e . In the checking mode, we have both e and $\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3$.
 793 In the application mode, we have e , and maybe an empty context (which degenerates into
 794 the inference mode), or an application context σ_1 (we know the type of first argument), or
 795 an application context σ_1, σ_2 (we know the type of both arguments). Partial type checking
 796 has also been used in techniques like *colored local type inference* [Odersky et al. 2001] and
 797 *boxy types* [Vytiniotis et al. 2006].

798 **TRADE-OFFS.** Note that the application mode is *not* conservative over traditional bidirec-
 799 tional type checking due to the different information flow. However, it provides a new design
 800 choice for type inference/checking algorithms, especially for those where the information
 801 about arguments is useful. Therefore we next discuss some benefits of the application mode
 802 for two interesting cases where functions are either variables; or lambda (or type) abstrac-
 803 tions.

804 3.1.3 BENEFITS OF INFORMATION FLOWING FROM ARGUMENTS TO FUNCTIONS

805 **LOCAL CONSTRAINT SOLVER FOR FUNCTION VARIABLES.** Many type systems, including
 806 type systems with *implicit polymorphism* and/or *static overloading*, need information about
 807 the types of the arguments when type checking function variables. For example, in conven-
 808 tional functional languages with implicit polymorphism, function calls such as $(\text{id } 1)$ where

²Although 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.

809 $\text{id} : \forall a. (a \rightarrow a)$, are *pervasive*. In such a function call the type system must instantiate
 810 a to Int . Dealing with such implicit instantiation gets trickier in systems with *higher-rank*
 811 *types*. For example, Peyton Jones et al. [2007] require additional syntactic forms and rela-
 812 tions, whereas DK adds a special-purpose application judgment.

813 With the application mode, all the type information about the arguments being applied
 814 is available in the application context and can be used to solve instantiation constraints. To
 815 exploit such information, the type system employs a special subtyping judgment called *ap-*
 816 *plication subtyping*, with the form $\Sigma \vdash \sigma_1 <: \sigma_2$. Unlike conventional subtyping, compu-
 817 tationally Σ and σ_1 are interpreted as inputs and σ_2 as output. In the example above, we
 818 have that $\text{Int} \vdash \forall a. a \rightarrow a <: \sigma$ and we can determine that $a = \text{Int}$ and $\sigma = \text{Int} \rightarrow \text{Int}$.
 819 In this way, the type system is able to solve the constraints *locally* according to the applica-
 820 tion context since we no longer need to propagate the instantiation constraints to the typing
 821 process.

822 **DECLARATION DESUGARING FOR LAMBDA ABSTRACTIONS.** An interesting consequence of
 823 the usage of an application mode is that it enables the following **let** sugar:

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

824 Such syntactic sugar for **let** is, of course, standard. However, in the context of implementa-
 825 tions of typed languages it normally requires extra type annotations or a more sophisticated
 826 type-directed translation. Type checking $(\lambda x. e_2) e_1$ would normally require annotations (for
 827 example a higher-rank type annotation for x as in OL and DK), or otherwise such annotation
 828 should be inferred first. Nevertheless, with the application mode no extra annotations/in-
 829 ference is required, since from the type of the argument e_1 it is possible to deduce the type
 830 of x . Generally speaking, with the application mode *annotations are never needed for ap-*
 831 *plied lambdas*. Thus **let** can be the usual sugar from the untyped lambda calculus, including
 832 HM-style **let** expression and even type declarations.

3.1.4 TYPE INFERENCE OF HIGHER-RANK TYPES

834 We believe the application mode can be integrated into many traditional bidirectional type
 835 systems. In this chapter, we focus on integrating the application mode into a bidirectional
 836 type system with higher-rank types. Our paper [Xie and Oliveira 2018] includes another
 837 application to System F.

838 Consider again the motivation example used in Section 2.2:

839 $(\lambda f. (f \ 1, f \ 'a')) (\lambda x. x)$

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:

$(\lambda f: (\forall a. a \rightarrow a). (f\ 1, f\ 'c')) (\lambda x. x)$

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 $\lambda x. x$, which is of type $\forall a. a \rightarrow a$.

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: $(\lambda f. (f\ 1, f\ 'c')) (\lambda x. x)$.

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 $\forall a. a \rightarrow a$ 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:

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

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	$\sigma ::= \text{Int} \mid a \mid \sigma_1 \rightarrow \sigma_2 \mid \forall a. \sigma$
Monotypes	$\tau ::= \text{Int} \mid a \mid \tau_1 \rightarrow \tau_2$
Contexts	$\Psi ::= \bullet \mid \Psi, x : \sigma$
Application Contexts	$\Sigma ::= \bullet \mid \Sigma, \sigma$

Figure 3.1: Syntax of System AP.

903 a combination of inference and application modes; 2) the novel subtyping relation under an
 904 application context.

905 3.2.1 SYNTAX

906 The syntax of the language is given in Figure 3.1.

907 **EXPRESSIONS.** The definition of expressions e include variables (x), integers (n), annotated
 908 lambda abstractions ($\lambda x : \sigma. e$), lambda abstractions ($\lambda x. e$), and applications ($e_1 e_2$). No-
 909 tably, the syntax does not include a **let** expression (**let** $x = e_1$ **in** e_2). Let expressions can be
 910 regarded as the standard syntax sugar $(\lambda x. e_2) e_1$, as illustrated in more detail later.

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

913 **CONTEXTS.** Typing contexts Ψ are standard: they map a term variable x to its type σ . Again,
 914 we implicitly assume that all variables in Ψ are distinct. In this system, the context is modeled
 915 as the HM-style context (Section 2.1), which does not contain type variables; so the type
 916 system needs to explicitly ensure freshness of type variables.

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

923 3.2.2 TYPE SYSTEM

924 The top part of Figure 3.2 gives the typing rules for our language. The judgment $\Psi; \Sigma \vdash^{AP}$
 925 $e \Rightarrow \sigma$ is read as: under typing context Ψ , and application context Σ , e has type σ . The

3 Higher-Rank Polymorphism with the Application Mode

$\boxed{\Psi \vdash^{AP} e \Rightarrow \sigma}$			(Typing Inference)
$\frac{\text{AP-INF-INT}}{\Psi \vdash^{AP} n \Rightarrow \text{Int}}$	$\frac{\text{AP-INF-LAM} \quad \Psi, x : \tau \vdash^{AP} e \Rightarrow \sigma}{\Psi \vdash^{AP} \lambda x. e \Rightarrow \tau \rightarrow \sigma}$	$\frac{\text{AP-INF-LAMANN} \quad \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_2}{\Psi \vdash^{AP} \lambda x : \sigma_1. e \Rightarrow \sigma_1 \rightarrow \sigma_2}$	
$\boxed{\Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma}$			(Typing Application Mode)
$\frac{\text{AP-APP-VAR} \quad (x : \sigma_1) \in \Psi \quad \Sigma \vdash^{AP} \sigma_1 <: \sigma_2}{\Psi; \Sigma \vdash^{AP} x \Rightarrow \sigma_2}$	$\frac{\text{AP-APP-LAM} \quad \Psi, x : \sigma_1; \Sigma \vdash^{AP} e \Rightarrow \sigma_2}{\Psi; \Sigma, \sigma_1 \vdash^{AP} \lambda x. e \Rightarrow \sigma_1 \rightarrow \sigma_2}$		
	$\frac{\text{AP-APP-LAMANN} \quad \vdash^{AP} \sigma_2 <: \sigma_1 \quad \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_3}{\Psi; \Sigma, \sigma_2 \vdash^{AP} \lambda x : \sigma_1. e \Rightarrow \sigma_2 \rightarrow \sigma_3}$		
	$\frac{\text{AP-APP-APP} \quad \Psi \vdash^{AP} e_2 \Rightarrow \sigma_1 \quad \bar{a}_i^i = \text{FV}(\sigma_1) - \text{FV}(\Psi) \quad \sigma_2 = \forall \bar{a}_i^i. \sigma_1 \quad \Psi; \Sigma, \sigma_2 \vdash^{AP} e_1 \Rightarrow \sigma_2 \rightarrow \sigma_3}{\Psi; \Sigma \vdash^{AP} e_1 e_2 \Rightarrow \sigma_3}$		
$\boxed{\vdash^{AP} \sigma_1 <: \sigma_2}$			(Subtyping)
$\frac{\text{AP-S-TVAR}}{\vdash^{AP} a <: a}$	$\frac{\text{AP-S-INT}}{\vdash^{AP} \text{Int} <: \text{Int}}$	$\frac{\text{AP-S-ARROW} \quad \vdash^{AP} \sigma_3 <: \sigma_1 \quad \vdash^{AP} \sigma_2 <: \sigma_4}{\vdash^{AP} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4}$	
$\frac{\text{AP-S-FORALLL} \quad \vdash^{AP} \sigma_1[a \mapsto \tau] <: \sigma_2}{\vdash^{AP} \forall a. \sigma_1 <: \sigma_2}$	$\frac{\text{AP-S-FORALLR} \quad a \notin \text{FV}(\sigma_1) \quad \vdash^{AP} \sigma_1 <: \sigma_2}{\vdash^{AP} \sigma_1 <: \forall a. \sigma_2}$		
$\boxed{\Sigma \vdash^{AP} \sigma_1 <: \sigma_2}$			(Application Subtyping)
$\frac{\text{AP-AS-EMPTY}}{\bullet \vdash^{AP} \sigma <: \sigma}$	$\frac{\text{AP-AS-FORALL} \quad \Sigma, \sigma_3 \vdash^{AP} \sigma_1[a \mapsto \tau] <: \sigma_2}{\Sigma, \sigma_3 \vdash^{AP} \forall a. \sigma_1 <: \sigma_2}$	$\frac{\text{AP-AS-ARROW} \quad \vdash^{AP} \sigma_3 <: \sigma_1 \quad \Sigma \vdash^{AP} \sigma_2 <: \sigma_4}{\Sigma, \sigma_3 \vdash^{AP} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4}$	

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 reformatting to match AP, we get (without the gray-shaded parts):

$$\frac{\Psi \vdash e_1 \Rightarrow \sigma_1 \quad \overline{a_i}^i = \text{FV}(\tau) - \text{FV}(\Psi) \quad \sigma_2 = \forall \overline{a_i}^i. \sigma_1 \quad \Psi, x : \sigma_2; \Sigma \vdash e_2 \Rightarrow \sigma_3}{\Psi; \Sigma \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow \sigma_3}$$

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 $(\text{let } x = e_1 \text{ in } e_2)$ to $(\lambda x. e_2) e_1$, we have the following derivation:

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

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 ($\Sigma \rightarrow \sigma$). If $\Sigma = \sigma_1, \sigma_2, \dots, \sigma_n$, then $\Sigma \rightarrow \sigma$ means the function type $\sigma_n \rightarrow \dots \rightarrow \sigma_2 \rightarrow \sigma_1 \rightarrow \sigma$.

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 $\sigma = \Sigma \rightarrow \sigma'$.*

Having this lemma, we can always use the judgment $\Psi; \Sigma \vdash^{AP} e \Rightarrow \Sigma \rightarrow \sigma'$ 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 ($\Psi \vdash^{AP} e \Rightarrow$ to $\Psi; \Sigma \vdash^{AP} e \Rightarrow$). *If $\Psi \vdash^{AP} e \Rightarrow \Sigma \rightarrow \sigma$, then $\Psi; \Sigma \vdash^{AP} e \Rightarrow \Sigma \rightarrow \sigma$.*

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

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

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

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

993 3.2.3 SUBTYPING

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

996 **SUBTYPING.** The first subtyping judgment $\vdash^{AP} \sigma_1 <: \sigma_2$ follows OL, but in HM-style; that
 997 is, without tracking type variables. Rule **AP-S-FORALLR** states σ_1 is subtype of $\forall a. \sigma_2$ only
 998 if σ_1 is a subtype of σ_2 , with the assumption a is a fresh variable. Rule **AP-S-FORALLL** says
 999 $\forall a. \sigma_1$ is a subtype of σ_2 if we can instantiate it with some τ and show the result is a subtype
 1000 of σ_2 .

1001 **APPLICATION SUBTYPING.** The typing rule **AP-APP-VAR** uses the second subtyping judg-
 1002 ment $\Sigma \vdash^{AP} \sigma_1 <: \sigma_2$. To motivate this new kind of judgment, consider the expression $\text{id } 1$
 1003 for example, whose derivation is stuck at rule **AP-APP-VAR** (here we assume $\text{id} : \forall a. a \rightarrow a \in \Psi$):
 1004 Ψ):

$$\frac{\Psi \vdash^{AP} 1 \Rightarrow \text{Int} \quad \emptyset = \text{FV}(\text{Int}) - \text{FV}(\Psi) \quad \frac{\text{id} : \forall a. a \rightarrow a \in \Psi \quad ???}{\Psi; \text{Int} \vdash^{AP} \text{id} \Rightarrow ?} \text{AP-APP-VAR}}{\Psi \vdash^{AP} \text{id } 1 \Rightarrow ?} \text{AP-APP-APP}$$

1005 Here we know that $\text{id} : \forall a. a \rightarrow a$ and also, from the application context, that id is applied
 1006 to an argument of type Int . Thus we need a mechanism for solving the instantiation $a = \text{Int}$
 1007 and returning a supertype $\text{Int} \rightarrow \text{Int}$ as the type of id . This is precisely what the application
 1008 subtyping achieves: resolving instantiation constraints according to the application context.
 1009 Notice that unlike existing works (Peyton Jones et al. [2007] or DK), application subtyping
 1010 provides a way to solve instantiation more *locally*, since it does not mutually depend on typ-
 1011 ing.

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 $\sigma_1 \rightarrow \sigma_2$ 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 $\sigma_3 \rightarrow \sigma_4$ 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 , $\sigma_2 = \Sigma \rightarrow \sigma_3$.*

Therefore we can always use the judgment $\Sigma \vdash^{AP} \sigma_1 <: \Sigma \rightarrow \sigma_2$, 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). $\Sigma \vdash^{AP} \Sigma \rightarrow \sigma <: \Sigma \rightarrow \sigma$.

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

1046 Finally, we can convert between subtyping and application subtyping. We can remove the
1047 application context and still get a subtyping relation:

1048 **Lemma 3.8** ($\Sigma \vdash^{AP} <: \text{to } \vdash^{AP} <:$). *If $\Sigma \vdash^{AP} \sigma_1 <: \sigma_2$, then $\vdash^{AP} \sigma_1 <: \sigma_2$.*

1049 Transferring from subtyping to application subtyping will result in a more general type.

1050 **Lemma 3.9** ($\vdash^{AP} <: \text{to } \Sigma \vdash^{AP} <:$). *If $\vdash^{AP} \sigma_1 <: \Sigma \rightarrow \sigma_2$, then for some σ_3 , we have $\Sigma \vdash^{AP}$
1051 $\sigma_1 <: \Sigma \rightarrow \sigma_3$, and $\vdash^{AP} \sigma_3 <: \sigma_2$.*

1052 This lemma may not seem intuitive at first glance. Consider a concrete example. Consider
1053 the derivation for $\vdash^{AP} \text{Int} \rightarrow \forall a. a <: \text{Int} \rightarrow \text{Int}$:

$$\frac{\frac{\vdash^{AP} \text{Int} <: \text{Int}}{\vdash^{AP} \text{Int} <: \text{Int}} \text{AP-S-INT} \quad \frac{\frac{\vdash^{AP} \text{Int} <: \text{Int}}{\vdash^{AP} \forall a. a <: \text{Int}} \text{AP-S-FORALLL}}{\vdash^{AP} \text{Int} \rightarrow \forall a. a <: \text{Int} \rightarrow \text{Int}} \text{AP-S-ARROW}$$

1054 and for $\text{Int} \vdash^{AP} \text{Int} \rightarrow \forall a. a <: \text{Int} \rightarrow \forall a. a$:

$$\frac{\frac{\vdash^{AP} \text{Int} <: \text{Int}}{\text{Int} \vdash^{AP} \text{Int} \rightarrow \forall a. a <: \text{Int} \rightarrow \forall a. a} \text{AP-S-INT} \quad \frac{\vdash^{AP} \forall a. a <: \forall a. a}{\text{Int} \vdash^{AP} \text{Int} \rightarrow \forall a. a <: \text{Int} \rightarrow \forall a. a} \text{AP-AS-EMPTY}}{\text{Int} \vdash^{AP} \text{Int} \rightarrow \forall a. a <: \text{Int} \rightarrow \forall a. a} \text{AP-AS-ARROW}$$

1055 The former one, holds because we have $\vdash^{AP} \forall a. a <: \text{Int}$ in the return type. But in the latter
1056 one, after Int is consumed from application context, we eventually reach rule **AP-AS-EMPTY**,
1057 which always returns the original type back.

1058 3.3 TYPE-DIRECTED TRANSLATION

1059 This section discusses the type-directed translation of System AP into a variant of System F
1060 that is also used in Peyton Jones et al. [2007]. The translation is shown to be coherent and
1061 type safe. The later result implies the type-safety of the source language. To prove coherency,
1062 we need to decide when two translated terms are the same using η -id equality, and show that
1063 the translation is unique up to this definition.

1064 3.3.1 TARGET LANGUAGE

1065 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	$\sigma ::= \text{Int} \mid a \mid \sigma_1 \rightarrow \sigma_2 \mid \forall a. \sigma$
Contexts	$\Psi ::= \bullet \mid \Psi, x : \sigma$

$\Psi \vdash^F s : \sigma$

(Typing)

$\frac{\text{F-VAR} \quad (x : \sigma) \in \Psi}{\Psi \vdash^F x : \sigma}$	$\frac{\text{F-INT}}{\Psi \vdash^F n : \text{Int}}$	$\frac{\text{F-LAMANN} \quad \Psi, x : \sigma_1 \vdash^F s : \sigma_2}{\Psi \vdash^F \lambda x : \sigma_1. s : \sigma_1 \rightarrow \sigma_2}$
$\frac{\text{F-APP} \quad \Psi \vdash^F s_1 : \sigma_1 \rightarrow \sigma_2 \quad \Psi \vdash^F s_2 : \sigma_1}{\Psi \vdash^F s_1 s_2 : \sigma_2}$	$\frac{\text{F-TABS} \quad \Psi \vdash^F s : \sigma \quad a \notin \text{FV}(\Psi)}{\Psi \vdash^F \Lambda a. s : \forall a. \sigma}$	
$\frac{\text{F-TAPP} \quad \Psi \vdash^F s : \forall a. \sigma_1}{\Psi \vdash^F s \sigma_2 : \sigma_1[a \mapsto \sigma_2]}$		

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 $\bullet \vdash^F f : \sigma_1 \rightarrow \sigma_2$.

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 $\sigma_3 \rightarrow \sigma_1$ 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

$$\frac{}{\vdash^{AP} \text{Int} <: \text{Int} \rightsquigarrow \lambda x : \text{Int}. x}$$

AP-S-ARROW

$$\frac{\vdash^{AP} \sigma_3 <: \sigma_1 \rightsquigarrow f_1 \quad \vdash^{AP} \sigma_2 <: \sigma_4 \rightsquigarrow f_2}{\vdash^{AP} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4 \rightsquigarrow \lambda x : \sigma_1 \rightarrow \sigma_2. \lambda y : \sigma_3. f_2(x(f_1 y))}$$

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

$$\frac{a \notin \text{FV}(\sigma_1) \quad \vdash^{AP} \sigma_1 <: \sigma_2 \rightsquigarrow f}{\vdash^{AP} \sigma_1 <: \forall a. \sigma_2 \rightsquigarrow \lambda x : \sigma_1. \Lambda a. f x}$$

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

$$\frac{\vdash^{AP} \sigma_3 <: \sigma_1 \rightsquigarrow f_1 \quad \Sigma \vdash^{AP} \sigma_2 <: \sigma_4 \rightsquigarrow f_2}{\Sigma, \sigma_3 \vdash^{AP} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4 \rightsquigarrow \lambda x : \sigma_1 \rightarrow \sigma_2. \lambda y : \sigma_3. f_2(x(f_1 y))}$$

Figure 3.4: Subtyping translation rules of System AP.

$\Psi \vdash^{AP} e \Rightarrow \sigma \rightsquigarrow s$

(Typing Inference)

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

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

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

$\Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma \rightsquigarrow s$

(Typing Application Mode)

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

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

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

$$\frac{\text{AP-APP-APP} \quad \Psi \vdash^{AP} e_2 \Rightarrow \sigma_1 \rightsquigarrow s_2 \quad \overline{a_i}^i = \text{FV}(\sigma_1) - \text{FV}(\Psi) \quad \sigma_2 = \forall \overline{a_i}^i. \sigma_1 \quad \Psi; \Sigma, \sigma_2 \vdash^{AP} e_1 \Rightarrow \sigma_2 \rightarrow \sigma_3 \rightsquigarrow s_1}{\Psi; \Sigma \vdash^{AP} e_1 e_2 \Rightarrow \sigma_3 \rightsquigarrow s_1 (\Lambda \overline{a_i}^i. s_2)}$$

Figure 3.5: Typing translation rules of System AP.

of type σ_4 is computed. In rule **A-PS-FORALL**, 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-FORALL** 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 $\sigma_1 \rightarrow \forall a. \sigma_2$.

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

$ 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 $

$f_1 =_{\eta id} f_2$

(Eta-id Equality)

$\frac{\text{ETA-REDUCE}}{x \notin \text{FV}(f)} \quad \frac{}{\lambda x. f x =_{\eta id} f}$	$\frac{\text{ETA-ID}}{} \quad \frac{}{(\lambda x. x) f =_{\eta id} f}$	$\frac{\text{ETA-APP}}{f_1 =_{\eta id} f'_1 \quad f_2 =_{\eta id} f'_2} \quad \frac{}{f_1 f_2 =_{\eta id} f'_1 f'_2}$
$\frac{\text{ETA-LAM}}{f =_{\eta id} f'} \quad \frac{}{\lambda x. f =_{\eta id} \lambda x. f'}$	$\frac{\text{ETA-REFL}}{} \quad \frac{}{f =_{\eta id} f}$	$\frac{\text{ETA-SYMM}}{f =_{\eta id} f'} \quad \frac{}{f' =_{\eta id} f}$
$\frac{\text{ETA-TRANS}}{f_1 =_{\eta id} f_2 \quad f_2 =_{\eta id} f_3} \quad \frac{}{f_1 =_{\eta id} f_3}$		

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

1113 3.3.5 COHERENCE

1114 One problem with the translation is that there are multiple targets corresponding to one ex-
 1115 pression. This is because in original system there are multiple choices when instantiating a
 1116 polymorphic type, or guessing the annotation for unannotated lambda abstraction (rule [AP-](#)
 1117 [INF-LAM](#)). For each choice, the corresponding target will be different. For example, expres-
 1118 sion $\lambda x. x$ can be type checked with $\text{Int} \rightarrow \text{Int}$, or $a \rightarrow a$, and the corresponding targets are
 1119 $\lambda x : \text{Int}. x$, and $\lambda x : a. x$.

1120 Therefore, in order to prove the translation is coherent, we turn to prove that all the trans-
 1121 lations have the same operational semantics. There are two steps towards the goal: type
 1122 erasure, and considering η expansion and identity functions.

1123 **TYPE ERASURE.** Since type information is useless after type-checking, we erase the type in-
 1124 formation of the targets for comparison. The erasure process is given at the top of Figure 3.6.

1125 The erasure process is standard, where we erase the type annotation in abstractions, and
 1126 remove type abstractions and type applications. The calculus after erasure is the untyped
 1127 lambda calculus.

1128 **ETA-ID EQUALITY.** However, even if we have type erasure, multiple targets for one expres-
 1129 sion can still be syntactically different. The problem is that we can insert more coercion
 1130 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 $\lambda x. (\lambda y. y) f x =_{\eta id} f$.

$$\frac{\frac{\frac{}{f =_{\eta id} f} \text{ETA-REFL}}{(\lambda y. y) f =_{\eta id} f} \text{ETA-ID}}{\lambda x. (\lambda y. y) f x =_{\eta id} f} \text{ETA-REDUCE}$$

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 $\hat{\alpha}, \hat{\beta}$ 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

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

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 $(\square, N_0); \Psi \vdash^{AP} e \Rightarrow \sigma \hookrightarrow (S_1, N_1)$, then for any substitution V with $\text{dom}(V) = \text{fv}(S_1 \Psi, S_1 \sigma)$, 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 $(\square, N_0); \Psi \vdash^{AP} e \Rightarrow \sigma_2 \hookrightarrow (S_1, N_1)$, and for some S_2 , if $\overline{a}_i^i = \text{FV}(\Psi) - \text{FV}(S_2 S_1 \sigma_2)$, and $\overline{b}_i^i = \text{FV}(\Psi) - \text{FV}(\sigma)$, 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:

$$\frac{\text{AP-APP-ANNO} \quad \Psi \vdash^{AP} e \Leftarrow \sigma_1 \quad \Sigma \vdash^{AP} \sigma_1 <: \sigma_2}{\Psi; \Sigma \vdash^{AP} e : \sigma_1 \Rightarrow \sigma_2}$$

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 .

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

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

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

Note that adding annotated expressions might bring convenience for programmers, since annotations can be more freely placed in a program. For example, $(\lambda x. x \ 1) : (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$ 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 $(\lambda x : \text{Int} \rightarrow \text{Int}. x \ 1)$.

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.

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

The application mode can apply to the elimination constructs of pairs. If one component of the pair is a function, for example, $\text{fst } (\lambda x. x, 1) \ 2$, then it is possible to have a judgment

3 Higher-Rank Polymorphism with the Application Mode

with a non-empty application context. Therefore, we can use the application subtyping to account for the application contexts:

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

However, in polymorphic type systems, we need to take the subsumption rule into consideration. For example, in the expression $(\lambda x : \forall a. (a, b). \mathbf{fst} x)$, \mathbf{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 $\mathbf{fst} e$ as an expression form, where e is itself an argument, we could regard \mathbf{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 \mathbf{fst} and \mathbf{snd} can be modeled as:

$$\begin{array}{c}
 \text{AP-APP-FST-VAR} \\
 \frac{\Sigma \vdash^{AP} \forall a. \forall b. (a, b) \rightarrow a <: \sigma}{\Psi; \Sigma \vdash^{AP} \mathbf{fst} \Rightarrow \sigma} \\
 \\
 \text{AP-APP-SND-VAR} \\
 \frac{\Sigma \vdash^{AP} \forall a. \forall b. (a, b) \rightarrow b <: \sigma}{\Psi; \Sigma \vdash^{AP} \mathbf{snd} \Rightarrow \sigma}
 \end{array}$$

1199 Note that another way to model those two rules would be to simply have an initial typing
 1200 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
 1201 elimination of pairs be dealt directly by the rule for variables.

1202 An extended version of the calculus extended with rules for pairs (rule [AP-INF-PAIR](#), rule [AP-](#)
 1203 [S-PAIR](#), rule [AP-APP-FST-VAR](#) and rule [AP-APP-SND-VAR](#)), has been formally studied. All the
 1204 theorems presented before hold with the extension of pairs.

1205 3.5.3 MORE EXPRESSIVE TYPE APPLICATIONS

1206 The design choice of propagating arguments to functions was subject to consideration in
 1207 the original work on local type inference [Pierce and Turner 2000], but was rejected due to
 1208 possible non-determinism introduced by explicit type applications:

1209 *“It is possible, of course, to come up with examples where it would be beneficial to*
 1210 *synthesize the argument types first and then use the resulting information to avoid*

1211 *type annotations in the function part of an application expression....Unfortunately*
 1212 *this refinement does not help infer the type of polymorphic functions. For example,*
 1213 *we cannot uniquely determine the type of x in the expression $(\text{fun}[X](x) e) [\text{Int}] 3$."*

1214 As a response to this challenge, we also present an application of the application mode to a
 1215 variant of System F [Xie and Oliveira 2018]. The development of the calculus shows that the
 1216 application mode can actually work well with calculi with explicit type applications. Here
 1217 we explain the key ideas of the design of the system, but refer to Xie and Oliveira [2018] for
 1218 more details.

1219 To explain the new design, consider the expression:

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

1220 which is not typeable in the traditional type system for System F. In System F the lambda
 1221 abstractions do not account for the context of possible function applications. Therefore when
 1222 type checking the inner body of the lambda abstraction, the expression $x + 1$ is ill-typed,
 1223 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 $a = \text{Int}$ 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:

$$\frac{\Psi; \Sigma, [[\Psi]\sigma_1] \vdash^{AP} e \Rightarrow \sigma_2}{\Psi; \Sigma \vdash^{AP} e \sigma_1 \Rightarrow \sigma_2} \text{AP-APP-TAPP} \qquad \frac{\Psi, a = \sigma_1; \Sigma \vdash^{AP} e \Rightarrow \sigma_2}{\Psi; \Sigma, [\sigma_1] \vdash^{AP} \Lambda a. e \Rightarrow \sigma_2} \text{AP-APP-TLAM}$$

1224 For type applications, rule **AP-APP-TAPP** stores the type argument σ_1 into the application
 1225 context. Since Ψ tracks type equalities, we apply Ψ as a type substitution to σ_1 (i.e., $[\Psi]\sigma_1$)
 1226 Moreover, to distinguish between type arguments and types of term arguments, we put type
 1227 arguments in brackets (i.e., $[[\Psi]\sigma_1]$). For type abstractions (rule **AP-APP-TLAM**), if the appli-
 1228 cation context is non-empty, we put a new type equality between the type variable a and the
 1229 type argument σ_1 into the context.

1230 Now, back to the problematic expression $(\text{fun}[X](x) e) [\text{Int}] 3$, the type of x can be inferred
 1231 as either X or Int since they are actually equivalent.

1232 **SUGAR FOR TYPE SYNONYMS.** In the same way that we can regard **let** expressions as syntactic
 1233 sugar, in the new type system we further *gain built-in type synonyms*. A *type synonym* is a

1234 new name for an existing type. Type synonyms are common in languages such as Haskell.
 1235 In our calculus a simple form of type synonyms can be desugared as follows:

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

1236 One practical benefit of such syntactic sugar is that it enables a direct encoding of a System F-
 1237 like language with declarations (including type-synonyms). Although declarations are often
 1238 viewed as a routine extension to a calculus, and are not formally studied, they are highly
 1239 relevant in practice. Therefore, a more realistic formalization of a programming language
 1240 should directly account for declarations. By providing a way to encode declarations, our
 1241 new calculus enables a simple way to formalize declarations.

1242 **TYPE ABSTRACTION.** The type equalities introduced by type applications may seem like
 1243 we are breaking System F type abstraction. However, we argue that *type abstraction* is still
 1244 supported by our System F variant. For example:

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

1245 (after desugaring) does *not* type-check, as in a System-F like language. In our type system
 1246 lambda abstractions that are immediately applied to an argument, and unapplied lambda
 1247 abstractions behave differently. Unapplied lambda abstractions are just like System F ab-
 1248 stractions and retain type abstraction. The example above illustrates this. In contrast the
 1249 typeable example $(\Lambda a. \lambda x : a. x + 1) \ \mathbf{Int}$, which uses a lambda abstraction directly applied to
 1250 an argument, can be regarded as the desugared expression for $\mathbf{type} \ a = \mathbf{Int} \ \mathbf{in} \ \lambda x : a. x + 1$.

PART III

HIGHER-RANK POLYMORPHISM AND GRADUAL TYPING

4 GRADUALLY TYPED HIGHER-RANK POLYMORPHISM

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 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].

In this chapter, we present GPC, a gradually typed calculus for implicit higher-rank polymorphism. Integrating gradual typing with higher-rank polymorphism is challenging. In particular, gradual typing calculi employ *type consistency* to validate type conversions. Polymorphic types à la System F also induce a subtyping relation that relates polymorphic types to their instantiations. The original definition of *consistent subtyping* by Siek and Taha [2007] serves as a guideline for designing gradual type systems with subtyping. However Siek and Taha’s definition is not adequate for polymorphic subtyping. Therefore, 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 develop GPC on top of 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.

$$\begin{array}{c}
 \boxed{\sigma_1 \sim \sigma_2} \qquad \qquad \qquad \text{(Type Consistency)} \\
 \\
 \overline{\sigma \sim \sigma} \quad \overline{\sigma \sim ?} \quad \overline{? \sim \sigma} \quad \frac{\sigma_1 \sim \sigma_3 \quad \sigma_2 \sim \sigma_4}{\sigma_1 \rightarrow \sigma_2 \sim \sigma_3 \rightarrow \sigma_4} \quad \frac{\overline{\sigma_i \sim \sigma'_i}^i}{\overline{[l_i : \sigma_i]^i} \sim [l_i : \sigma'_i]^i} \\
 \\
 \boxed{\sigma_1 <: \sigma_2} \qquad \qquad \qquad \text{(Subtyping)} \\
 \\
 \overline{\text{Int} <: \text{Int}} \quad \overline{\text{Bool} <: \text{Bool}} \quad \overline{\text{Float} <: \text{Float}} \quad \overline{\text{Int} <: \text{Float}} \\
 \\
 \frac{\sigma_3 <: \sigma_1 \quad \sigma_2 <: \sigma_4}{\sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4} \quad \frac{\overline{\sigma_i <: \sigma'_i}^i}{\overline{[l_i : \sigma_i]^i} <: [l_i : \sigma'_i]^i} \quad \overline{? <: ?}
 \end{array}$$

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

1278 4.1 INTRODUCTION AND MOTIVATION

1279 4.1.1 BACKGROUND: GRADUAL TYPING

1280 As is well known, a gradually typed language supports both fully static and fully dynamic
 1281 checking of program properties, as well as the continuum between these two extremes. It
 1282 also offers programmers fine-grained control over the static-to-dynamic spectrum, i.e., a
 1283 program can be evolved by introducing more or less precise types as needed [Garcia et al.
 1284 2016]. The untyped part of a program is dynamically checked, while the typed part is stati-
 1285 cally checked. Therefore, an expression like $(\lambda x : ?. x + 1) \text{ true}$ is dynamically rejected, while
 1286 an expression like $(\lambda x : \text{Int}. x + 1) \text{ true}$ is statically rejected. Here in the example we see the
 1287 use of the *unknown type* $?$, which indicates missing of static type information and thus the
 1288 use of dynamic typing.

1289 Gradual type calculi formalize the interaction between static types and dynamic type by
 1290 featuring the unknown type $?$, as well as a key concept, *consistency*, which weakens type
 1291 equality to allow for the presence of the unknown type $?$. Essentially, if σ_1 is consistent with
 1292 σ_2 (written $\sigma_1 \sim \sigma_2$), then a function e_1 expecting an argument of type σ_1 can also accept
 1293 an argument e_2 of type σ_2 . Importantly, the unknown type $?$ is consistent with any type.
 1294 Therefore, the expression $(\lambda x : ?. x + 1) \text{ true}$ type checks and only later gets rejected at
 1295 runtime.

1296 Siek and Taha [2007] developed a gradual type system for object-oriented languages that
 1297 they call $\mathbf{FOb}_{<}^?$. The definition of type consistency, $\sigma_1 \sim \sigma_2$, is given at the top of Figure 4.1.

1298 The intuition is that consistency relaxes the structure of a type system to tolerate unknown
 1299 positions in a gradual type. Consider some examples:

$$\begin{array}{lcl}
 & \text{Int} & \sim \text{Int} \\
 & \text{Int} & \sim ? \\
 1300 & ? & \sim \text{Int} \\
 & \text{Int} \rightarrow ? & \sim ? \rightarrow \text{Int}
 \end{array}$$

1301 $\text{FOb}_{\lesssim}^?$ also features subtyping. Siek and Taha defined the subtyping relation in a way that
 1302 static type safety is preserved. Their key insight is that the unknown type $?$ is neutral to sub-
 1303 typing, with only $? <: ?$. The subtyping relation $\sigma_1 <: \sigma_2$ relations are defined in Figure 4.1.
 1304 With subtyping, if $\sigma_1 <: \sigma_2$, then a function e_1 expecting an argument of type σ_1 can also
 1305 accept an argument e_2 of type σ_2 . For example, a function expecting `Float` can accept an
 1306 integer.

1307 However, the orthogonality of consistency and subtyping does not lead to a deterministic
 1308 relation. Now the question is that if we know a function e_1 expecting an argument of type
 1309 σ_1 , and an argument e_2 with type σ_2 , how can we determine if $e_1 e_2$ is well-typed? To this
 1310 end, Siek and Taha defined *consistent subtyping*, such that $e_1 e_2$ is well-typed only if σ_2 is a
 1311 consistent subtype of σ_1 (written $\sigma_2 \lesssim \sigma_1$). They defined algorithmic consistent subtyping
 1312 based on a *restriction operator*, written $\sigma_1|_{\sigma_2}$ that “masks off” the parts of type σ_1 that are
 1313 unknown in type σ_2 . For example,

$$\begin{array}{lcl}
 1314 & \text{Int} \rightarrow \text{Int}|_{\text{Bool} \rightarrow \text{Bool}} & = \text{Int} \rightarrow ? \\
 & \text{Bool} \rightarrow ?|_{\text{Int} \rightarrow \text{Int}} & = \text{Bool} \rightarrow ?
 \end{array}$$

1315 The definition of the restriction operator is given below:

$$\begin{array}{l}
 \sigma|_{\sigma'} = \text{case } (\sigma, \sigma') \text{ of} \\
 | (_, ?) \Rightarrow ? \\
 | (\sigma_1 \rightarrow \sigma_2, \sigma'_1 \rightarrow \sigma'_2) \Rightarrow \sigma_1|_{\sigma'_1} \rightarrow \sigma_2|_{\sigma'_2} \\
 | ([l_1 : \sigma_1, \dots, l_n : \sigma_n], [l_1 : \sigma'_1, \dots, l_m : \sigma'_m]) \text{ if } n \leq m \Rightarrow [l_1 : \sigma_1|_{\sigma'_1}, \dots, l_n : \sigma_n|_{\sigma'_n}] \\
 | ([l_1 : \sigma_1, \dots, l_n : \sigma_n], [l_1 : \sigma'_1, \dots, l_m : \sigma'_m]) \text{ if } n > m \Rightarrow [l_1 : \sigma_1|_{\sigma'_1}, \dots, l_m : \sigma_m|_{\sigma'_m}, \dots, l_n : \sigma_n] \\
 | (_, _) \Rightarrow \sigma
 \end{array}$$

1316

1317 With the restriction operator, consistent subtyping is simply defined as:

1318 **Definition 3** (Algorithmic Consistent Subtyping of Siek and Taha [2007]). $\sigma_1 \lesssim \sigma_2 \equiv$
 1319 $\sigma_1|_{\sigma_2} <: \sigma_2|_{\sigma_1}$.

A primary contribution of their work is to show that, as Siek and Taha [2007] put it, “gradual typing and subtyping are orthogonal and can be combined in a principled fashion”. To compose subtyping and consistency, Siek and Taha showed a proposition that algorithmic consistent subtyping is equivalent to two declarative definitions, which we refer to as the strawman for *declarative* consistent subtyping because it serves 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).

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

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 $\text{Int} \rightarrow \text{Int}$, or $\text{Char} \rightarrow \text{Char}$, but not $\forall a. a \rightarrow a$. Finding such manual polymorphic annotations can be non-trivial, especially when the program scales up and the annotation is long and complicated.

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

1353 4.1.3 APPLICATION: EFFICIENT (PARTLY) TYPED ENCODINGS OF ADTs

1354 We illustrate two concrete applications of gradually typed higher-rank polymorphism related
 1355 to algebraic datatypes. The first application shows how gradual typing helps in defining Scott
 1356 encodings of algebraic datatypes [Curry et al. 1958; Parigot 1992], which are impossible to
 1357 encode in plain System F. The second application shows how gradual typing makes it easy to
 1358 model and use heterogeneous containers.

1359 Our calculus does not provide built-in support for algebraic datatypes (ADTs). Neverthe-
 1360 less, the calculus is expressive enough to support efficient function-based encodings of (op-
 1361 tionally polymorphic) ADTs¹. This offers an immediate way to model algebraic datatypes
 1362 in our calculus without requiring extensions to our calculus or, more importantly, to its
 1363 target—the polymorphic blame calculus. While we believe that such extensions are possible,
 1364 they would likely require non-trivial extensions to the polymorphic blame calculus. Thus
 1365 the alternative of being able to model algebraic datatypes without extending the target cal-
 1366 culus is appealing. The encoding also paves the way to provide built-in support for algebraic
 1367 datatypes in the source language, while elaborating them via the encoding into λB .

1368 CHURCH AND SCOTT ENCODINGS. It is well-known that polymorphic calculi such as System
 1369 F can encode datatypes via Church encodings. However these encodings have well-known
 1370 drawbacks [Koopman et al. 2014]. In particular, some operations are hard to define, and
 1371 they can have a time complexity that is greater than that of the corresponding functions
 1372 for built-in algebraic datatypes. A well-known example is the definition of the predecessor
 1373 function for Church numerals [Church 1941]. Its definition requires significant ingenuity
 1374 (while it is trivial with built-in algebraic datatypes), and it has *linear* time complexity (versus
 1375 the *constant* time complexity for a definition using built-in algebraic datatypes).

1376 An alternative to Church encodings are the so-called Scott encodings [Curry et al. 1958].
 1377 They address the two drawbacks of Church encodings: they allow simple definitions that
 1378 directly correspond to programs implemented with built-in algebraic datatypes, and those
 1379 definitions have the same time complexity to programs using algebraic datatypes.

1380 Unfortunately, Scott encodings, or more precisely, their typed variant [Parigot 1992], can-
 1381 not be expressed in System F: in the general case they require recursive types, which System
 1382 F does not support. However, with gradual typing, we can remove the need for recursive
 1383 types, thus enabling Scott encodings in our calculus.

¹In 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.

A SCOTT ENCODING OF PARAMETRIC LISTS. Consider for instance the typed Scott encoding of parametric lists in a system with implicit polymorphism:

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

This encoding requires both polymorphic and recursive types². 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 $?$:

$$\text{List}_? a \triangleq \forall b. b \rightarrow (a \rightarrow ? \rightarrow b) \rightarrow b$$

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

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

1384 With the reinterpretation of **fold** and **unfold** as functions instead of built-in primitives, we
1385 have exactly the same definitions of $\text{nil}_?$ and $\text{cons}_?$.

1386 Note that when we elaborate our calculus into the polymorphic blame calculus, the above
1387 type annotations give rise to explicit casts. For instance, after elaboration $\mathbf{fold}_{\text{List}_? a} e$ results
1388 in the cast $\langle (\forall b. b \rightarrow (a \rightarrow \text{List}_? a \rightarrow b) \rightarrow b) \hookrightarrow \text{List}_? a \rangle s$ 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.

$$\text{fix} \triangleq \lambda f. (\lambda x : ?. f(x \ x)) (\lambda x : ?. f(x \ x)) : \forall a. (a \rightarrow a) \rightarrow a$$

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

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

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

Here $\text{zero}_? : \text{Int}_?$ and $\text{succ}_? : \text{Int}_? \rightarrow \text{Int}_?$ are the encodings of the constructors for natural numbers $\text{Int}_?$. 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: a list of type $\text{List}_? a$ requires every element to have type a . For instance, a heterogeneous list like $\text{cons}_? \text{zero}_? (\text{cons}_? \text{true}_? \text{nil}_?)$, is rejected because $\text{zero}_? : \text{Int}_?$ and $\text{true}_? : \text{Bool}_?$ 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 lists³. 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, with the cost paid through gradual typing via explicit casts during runtime.

An example of such gradually typed heterogeneous collections is:

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

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

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

1419 implementation of the source language, we could write more succinct programs akin to
 1420 Haskell’s syntax, such as `[0, True]`.

1421 4.2 REVISITING CONSISTENT SUBTYPING

1422 In this section we explore the design space of consistent subtyping. We start with the defini-
 1423 tions of consistency and subtyping for polymorphic types, and compare with some relevant
 1424 work. We then discuss the design decisions involved in our new definition of consistent sub-
 1425 typing, and justify the new definition by demonstrating its equivalence with that of Siek and
 1426 Taha [2007] and the AGT approach [Garcia et al. 2016] on simple types.

1427 The syntax of types is given at the top of Figure 4.2. Types σ are either the integer type `Int`,
 1428 type variables a , function types $\sigma_1 \rightarrow \sigma_2$, universal quantification $\forall a. \sigma$, or the unknown
 1429 type `?`. Note that monotypes τ contain all types other than the universal quantifier and the
 1430 unknown type `?`. We will discuss this restriction when we present the subtyping rules. Con-
 1431 texts Ψ are *ordered* lists of type variable declarations and term variables.

1432 4.2.1 CONSISTENCY AND SUBTYPING

1433 We start by giving the definitions of consistency and subtyping for polymorphic types, and
 1434 comparing our definitions with the compatibility relation by Ahmed et al. [2009] and type
 1435 consistency by Igarashi et al. [2017].

1436 **CONSISTENCY.** The key observation here is that consistency, $\sigma \sim \sigma_2$, is mostly a struc-
 1437 tural relation, except that the unknown type `?` can be regarded as any type. In other words,
 1438 consistency is an equivalence relation lifted from static types to gradual types [Garcia et al.
 1439 2016]. Following this observation, we naturally extend the definition from Figure 4.1 with
 1440 polymorphic types, as shown in the middle of Figure 4.2. In particular a polymorphic type
 1441 $\forall a. \sigma$ is consistent with another polymorphic type $\forall a. \sigma_2$ if σ is consistent with σ_2 .

1442 **SUBTYPING.** We express the fact that one type is a polymorphic generalization of another
 1443 by means of the subtyping judgment $\Psi \vdash^G \sigma <: \sigma_2$. Compared with the subtyping rules
 1444 of Odersky and Läufer [1996] in Figure 2.5, the only addition is the neutral subtyping of `?`.
 1445 In rule **GPC-S-FORALL**, the judgment $\Psi \vdash^G \sigma$ checks whether all the type variables in σ are
 1446 bound in the context Ψ . Notice that, in this rule, the universal quantifier is only allowed to
 1447 be instantiated with a *monotype*. According to the syntax in Figure 4.2, monotypes do not
 1448 include the unknown type `?`. This is because if we were to allow the unknown type to be used
 1449 for instantiation, we could have $\forall a. a \rightarrow a <: ? \rightarrow ?$ by instantiating a with `?`. Since $? \rightarrow ?$ is

Types	$\sigma ::=$	$\text{Int} \mid a \mid \sigma_1 \rightarrow \sigma_2 \mid \forall a. \sigma \mid ?$	
Monotypes	$\tau ::=$	$\text{Int} \mid a \mid \tau_1 \rightarrow \tau_2$	
Contexts	$\Psi ::=$	$\bullet \mid \Psi, x : \sigma \mid \Psi, a$	

$\sigma \sim \sigma_2$

(Type Consistency)

$\frac{}{\sigma \sim \sigma}$
 $\frac{}{\sigma \sim ?}$
 $\frac{}{? \sim \sigma}$
 $\frac{\sigma_1 \sim \sigma_3 \quad \sigma_2 \sim \sigma_4}{\sigma_1 \rightarrow \sigma_2 \sim \sigma_3 \rightarrow \sigma_4}$
 $\frac{\sigma \sim \sigma_2}{\forall a. \sigma \sim \forall a. \sigma_2}$

$\Psi \vdash^G \sigma <: \sigma_2$

(Subtyping)

$\frac{a \in \Psi}{\Psi \vdash^G a <: a} \text{ GPC-S-TVAR}$
 $\frac{}{\Psi \vdash^G \text{Int} <: \text{Int}} \text{ GPC-S-INT}$
 $\frac{\Psi \vdash^G \sigma_3 <: \sigma_1 \quad \Psi \vdash^G \sigma_2 <: \sigma_4}{\Psi \vdash^G \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4} \text{ GPC-S-ARROW}$
 $\frac{\Psi \vdash^G \tau \quad \Psi \vdash^G \sigma[a \mapsto \tau] <: \sigma_2}{\Psi \vdash^G \forall a. \sigma <: \sigma_2} \text{ GPC-S-FORALLL}$
 $\frac{\Psi, a \vdash^G \sigma <: \sigma_2}{\Psi \vdash^G \sigma <: \forall a. \sigma_2} \text{ GPC-S-FORALLR}$
 $\frac{}{\Psi \vdash^G ? <: ?} \text{ GPC-S-UNKNOWN}$

$\Psi \vdash^G \sigma$

(Well-formedness of types)

$\frac{}{\Psi \vdash^G \text{Int}}$
 $\frac{}{\Psi \vdash^G ?}$
 $\frac{a \in \Psi}{\Psi \vdash^G a}$
 $\frac{\Psi \vdash^G \sigma \quad \Psi \vdash^G \sigma_2}{\Psi \vdash^G \sigma \rightarrow \sigma_2}$
 $\frac{\Psi, a \vdash^G \sigma}{\Psi \vdash^G \forall a. \sigma}$

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

1450 consistent with any functions $\sigma_1 \rightarrow \sigma_2$, for instance, $\text{Int} \rightarrow \text{Bool}$, this means that we could
 1451 provide an expression of type $\forall a. a \rightarrow a$ to a function where the input type is supposed to be
 1452 $\text{Int} \rightarrow \text{Bool}$. However, as we know, $\forall a. a \rightarrow a$ is definitely not compatible with $\text{Int} \rightarrow \text{Bool}$.
 1453 Indeed, this does not hold in any polymorphic type systems without gradual typing. So the
 1454 gradual type system should not accept it either. (This is the *conservative extension* property
 1455 that will be made precise in Section 4.3.3.)

1456 Importantly there is a subtle distinction between a type variable and the unknown type,
 1457 although they both represent a kind of “arbitrary” type. The unknown type stands for the
 1458 absence of type information: it could be *any type at any instance*. Therefore, the unknown
 1459 type is consistent with any type, and additional type-checks have to be performed at runtime.
 1460 On the other hand, a type variable indicates *parametricity*. In other words, a type variable can
 1461 only be instantiated to a single type. For example, in the type $\forall a. a \rightarrow a$, the two occurrences
 1462 of a represent an arbitrary but single type (e.g., $\text{Int} \rightarrow \text{Int}$, $\text{Bool} \rightarrow \text{Bool}$), while $? \rightarrow ?$ could
 1463 be an arbitrary function (e.g., $\text{Int} \rightarrow \text{Bool}$) at runtime.

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

$$\frac{\sigma_1 \prec \sigma_2}{\sigma_1 \prec \forall a. \sigma_2} \text{ COMP-ALLR} \qquad \frac{\sigma_1[a \mapsto ?] \prec \sigma_2}{\forall a. \sigma_1 \prec \sigma_2} \text{ COMP-ALLL}$$

1464 Notice that, in rule **COMP-ALLL**, the universal quantifier is *always* instantiated to $?$. How-
 1465 ever, this way, λB allows $\forall a. a \rightarrow a \prec \text{Int} \rightarrow \text{Bool}$, which as we discussed before might
 1466 not be what we expect. Indeed λB relies on sophisticated runtime checks to rule out such
 1467 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 follows⁴:

$$\frac{\sigma \sim \sigma_2}{\forall a. \sigma \sim \forall a. \sigma_2} \qquad \frac{\sigma \sim \sigma_2 \quad \sigma_2 \neq \forall a. \sigma'_2 \quad ? \in \text{Types}(\sigma_2)}{\forall a. \sigma \sim \sigma_2}$$

⁴This 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*.

1468 Compared with our consistency definition in Figure 4.2, their first rule is the same as ours.
 1469 The second rule says that a non \forall -type can be consistent with a \forall -type only if it contains $?$.
 1470 In this way, their type system is able to reject $\forall a. a \rightarrow a \sim \text{Int} \rightarrow \text{Bool}$. However, in order
 1471 to keep conservativity, they also reject $\forall a. a \rightarrow a \sim \text{Int} \rightarrow \text{Int}$, which is perfectly sensible
 1472 in their setting of explicit polymorphism. However with implicit polymorphism, we would
 1473 expect $\forall a. a \rightarrow a$ to be related with $\text{Int} \rightarrow \text{Int}$, since a can be instantiated to Int .

1474 Nonetheless, when it comes to interactions between dynamically typed and polymorphi-
 1475 cally typed terms, both relations allow $\forall a. a \rightarrow \text{Int}$ to be related with $? \rightarrow \text{Int}$ for example
 1476 (i.e., $\forall a. a \rightarrow \text{Int} \prec ? \rightarrow \text{Int}$ in λB and $\forall a. a \rightarrow \text{Int} \sim ? \rightarrow \text{Int}$ in Igarashi et al. [2017]),
 1477 which in our view, is a kind of (implicit) polymorphic subtyping combined with type consis-
 1478 tency, and that should be derivable by the more primitive notions in the type system (instead
 1479 of inventing new relations). One of our design principles is that subtyping and consistency
 1480 are *orthogonal*, and can be naturally superimposed, echoing the opinion of Siek and Taha
 1481 [2007]. In this case, we can for example derive the relation between $\forall a. a \rightarrow \text{Int}$ and $? \rightarrow \text{Int}$
 1482 via an intermediate type $\text{Int} \rightarrow \text{Int}$, and by the subtyping relation $\forall a. a \rightarrow \text{Int} \prec: \text{Int} \rightarrow \text{Int}$
 1483 and the consistency relation $\text{Int} \rightarrow \text{Int} \sim ? \rightarrow \text{Int}$.

1484 4.2.2 TOWARDS CONSISTENT SUBTYPING

1485 With the definitions of consistency and subtyping, the question now is how to compose the
 1486 two relations so that two types can be compared in a way that takes both relations into ac-
 1487 count.

1488 Unfortunately, the strawman version of consistent subtyping (Definition 4) does not work
 1489 well with our definitions of consistency and subtyping for polymorphic types. Consider two
 1490 types: $(\forall a. a \rightarrow \text{Int}) \rightarrow \text{Int}$, and $(? \rightarrow \text{Int}) \rightarrow \text{Int}$. The first type can only reach the second
 1491 type in one way (first by applying consistency, then subtyping), but not the other way, as
 1492 shown in Figure 4.3a. We use $????$ to mean that we cannot find such a type. Similarly, there
 1493 are situations where the first type can only reach the second type by the other way (first
 1494 applying subtyping, and then consistency), as shown in Figure 4.3b.

1495 What is worse, if those two examples are composed in a way that those types all appear
 1496 co-variantly, then the resulting types cannot reach each other in either way. For example,
 1497 Figure 4.3c shows two such types by putting a Bool type in the middle, and neither definition
 1498 of consistent subtyping in Definition 4 works.

1499 OBSERVATIONS ON CONSISTENT SUBTYPING BASED ON INFORMATION PROPAGATION. In
 1500 order to develop a correct definition of consistent subtyping for polymorphic types, we need
 1501 to understand how consistent subtyping works. We first review two important properties of

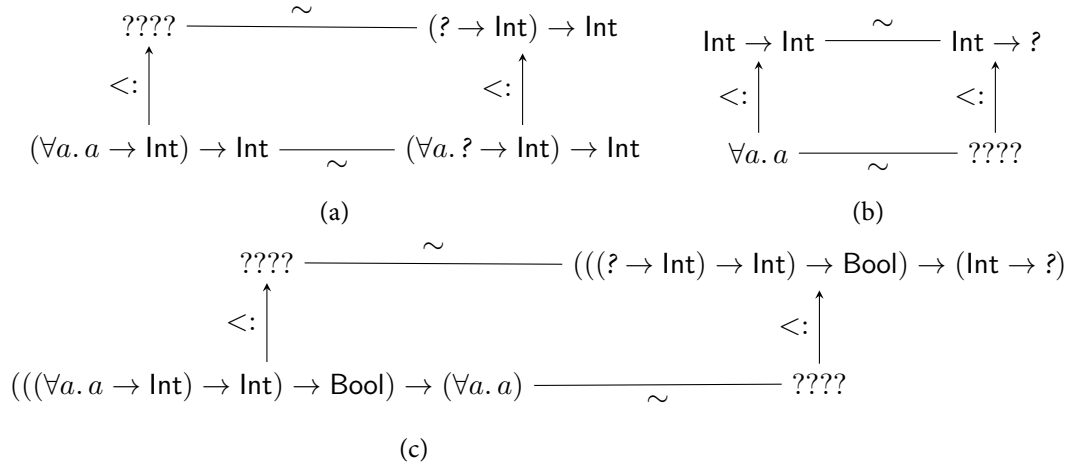


Figure 4.3: Examples that break the original definition of consistent subtyping.

subtyping: (1) subtyping induces the subsumption rule used in typing: 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 consideration, 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, e.g., we could derive $\text{Int} \sim \text{Bool}$ from $\text{Int} \sim ?$ and $? \sim \text{Bool}$. 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. We thus have the following desirable property:

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

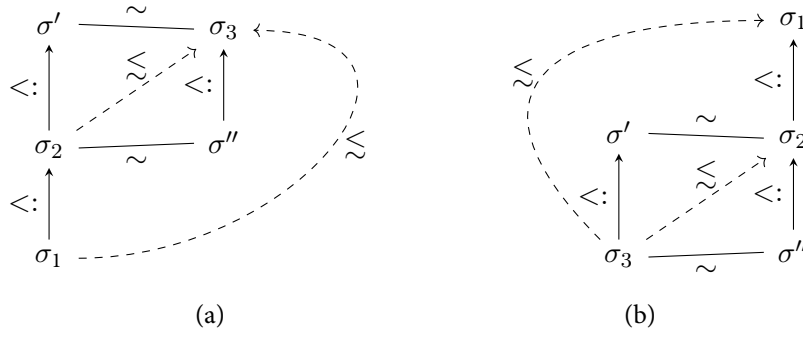


Figure 4.4: Observations of consistent subtyping

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:

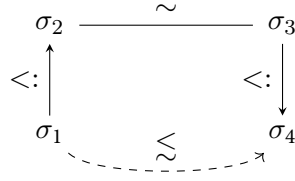
Definition 5 (Consistent Subtyping). $\Psi \vdash^G \sigma_1 \lesssim \sigma_2$ if and only if $\Psi \vdash^G \sigma_1 <: \sigma', \sigma' \sim \sigma''$ and $\Psi \vdash^G \sigma'' <: \sigma_2$ for some σ' and σ'' .

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 works well for with polymorphic subtyping. Furthermore, as we shall see in Section 4.5.1, this definition also works for 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 $\sigma'' <: \sigma_2$.
- *In the system of Siek and Taha, Definition 4 is equivalent to Definition 5.*
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 $\sigma_3 <: \sigma_2$.



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

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

1545 4.2.3 ABSTRACTING GRADUAL TYPING

1546 Garcia et al. [2016] presented a new foundation for gradual typing that they call the *Abstract-*
 1547 *ing Gradual Typing* (AGT) approach. In the AGT approach, gradual types are interpreted as
 1548 sets of static types, where static types refer to types containing no unknown types. In this
 1549 interpretation, predicates and functions on static types can then be lifted to apply to gradual
 1550 types. Central to their approach is the so-called *concretization* function. For simple types, a
 1551 concretization γ from gradual types to a set of static types is defined as follows:

1552 **Definition 6** (Concretization).

$$\begin{aligned}
 \gamma(\text{Int}) &= \{\text{Int}\} \\
 \gamma(\sigma_1 \rightarrow \sigma_2) &= \{\sigma'_1 \rightarrow \sigma'_2 \mid \sigma'_1 \in \gamma(\sigma_1), \sigma'_2 \in \gamma(\sigma_2)\} \\
 \gamma(?) &= \{\text{All static types}\}
 \end{aligned}$$

1554 Based on the concretization function, subtyping between static types can be lifted to grad-
 1555 ual types, resulting in the consistent subtyping relation:

1556 **Definition 7** (Consistent Subtyping in AGT). $\sigma_1 \widetilde{<} \sigma_2$ if and only if $\sigma'_1 <: \sigma'_2$ for some *static*
 1557 *types* σ'_1 and σ'_2 such that $\sigma'_1 \in \gamma(\sigma_1)$ and $\sigma'_2 \in \gamma(\sigma_2)$.

1558 Later they proved that this definition of consistent subtyping coincides with that of Def-
 1559 inition 4. By Proposition 4.1, we can directly conclude that our definition coincides with
 1560 AGT:

1561 **Corollary 4.2** (Equivalence to AGT on Simple Types). $\sigma_1 \lesssim \sigma_2$ if and only if $\sigma_1 \widetilde{<} \sigma_2$.

1562 However, AGT does not show how to deal with polymorphism (e.g. the interpretation of
 1563 type variables) yet. Still, as noted by Garcia et al. [2016], this is a promising line of future
 1564 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:

$$\frac{\sigma'_1 \sqsubseteq \sigma_1 \quad \sigma_1 <: \sigma_2 \quad \sigma'_2 \sqsubseteq \sigma_2}{\sigma'_1 \lesssim \sigma'_2}$$

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 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.

⁵Jafery 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.

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, $\forall a. a \rightarrow \text{Int}$ and $(\text{Int} \rightarrow ?) \rightarrow \text{Int}$. 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 $\text{Int} \rightarrow ? \rightarrow \text{Bool}$ and $\text{Int} \rightarrow \text{Int} \rightarrow ?$. Since the known parts have the relation that $\text{Int} \rightarrow ? \rightarrow ? \leq: \text{Int} \rightarrow ? \rightarrow ?$, we conclude that $\text{Int} \rightarrow ? \rightarrow \text{Bool} \lesssim \text{Int} \rightarrow \text{Int} \rightarrow ?$.

$$\begin{array}{l} \text{Int} \rightarrow \boxed{?} \rightarrow \boxed{\text{Bool}} \\ \text{Int} \rightarrow \boxed{\text{Int}} \rightarrow \boxed{?} \end{array} \left| \begin{array}{l} \text{Int} \rightarrow \text{Int} \rightarrow ? = \text{Int} \rightarrow ? \rightarrow ? \\ \text{Int} \rightarrow ? \rightarrow \text{Bool} = \text{Int} \rightarrow ? \rightarrow ? \end{array} \right) \leq:$$

1623

$\boxed{\Psi \vdash^G \sigma_1 \lesssim \sigma_2}$				(Consistent Subtyping)
$\frac{\text{GPC-CS-TVAR} \quad a \in \Psi}{\Psi \vdash^G a \lesssim a}$	$\frac{\text{GPC-CS-INT}}{\Psi \vdash^G \text{Int} \lesssim \text{Int}}$	$\frac{\text{GPC-CS-ARROW} \quad \Psi \vdash^G \sigma_3 \lesssim \sigma_1 \quad \Psi \vdash^G \sigma_2 \lesssim \sigma_4}{\Psi \vdash^G \sigma_1 \rightarrow \sigma_2 \lesssim \sigma_3 \rightarrow \sigma_4}$	$\frac{\text{GPC-CS-FORALLR} \quad \Psi, a \vdash^G \sigma_1 \lesssim \sigma_2}{\Psi \vdash^G \sigma_1 \lesssim \forall a. \sigma_2}$	
$\frac{\text{GPC-CS-FORALLL} \quad \Psi \vdash^G \tau \quad \Psi \vdash^G \sigma_1[a \mapsto \tau] \lesssim \sigma_2}{\Psi \vdash^G \forall a. \sigma_1 \lesssim \sigma_2}$	$\frac{\text{GPC-CS-UNKNOWNL}}{\Psi \vdash^G ? \lesssim \sigma}$	$\frac{\text{GPC-CS-UNKNOWNR}}{\Psi \vdash^G \sigma \lesssim ?}$		

Figure 4.6: Consistent Subtyping for implicit polymorphism.

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

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

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

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

4.3 GRADUALLY TYPED IMPLICIT POLYMORPHISM

In Section 4.2 we introduced our consistent subtyping relation that accommodates polymorphic types. In this section we continue with the development by giving a declarative type

1646 system for predicative implicit polymorphism, GPC, that employs the consistent subtyping
 1647 relation. The declarative system itself is already quite interesting as it is equipped with both
 1648 higher-rank polymorphism and the unknown type.

1649 The syntax of expressions in the declarative system is given at the top of Figure 4.7. The
 1650 definition of expressions are the same as of OL in Figure 2.3. Meta-variable e ranges over
 1651 expressions. Expressions include variables x , integers n , annotated lambda abstractions $\lambda x : \sigma. e$,
 1652 un-annotated lambda abstractions $\lambda x. e$, applications $e_1 e_2$, and let expressions **let** $x = e_1$ **in** e_2 .
 1653

1654 4.3.1 TYPING IN DETAIL

1655 Figure 4.7 gives the typing rules for our declarative system (the reader is advised to ignore
 1656 the gray-shaded parts for now). Rule **GPC-VAR** extracts the type of the variable from the
 1657 typing context. Rule **GPC-INT** always infers integer types. Rule **GPC-LAMANN** puts x with type
 1658 annotation σ into the context, and continues type checking the body e . Rule **GPC-LAM** assigns
 1659 a monotype τ to x , and continues type checking the body e . Gradual types and polymorphic
 1660 types are introduced via explicit annotations. Rule **GPC-GEN** puts a fresh type variable a into
 1661 the type context and generalizes the typing result σ to $\forall a. \sigma$. Rule **GPC-LET** infers the type
 1662 σ of e_1 , then puts $x : \sigma$ in the context to infer the type of e_2 . Rule **GPC-APP** first infers the
 1663 type of e_1 , then the matching judgment $\Psi \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2$ extracts the domain type σ_1 and
 1664 the codomain type σ_2 from type σ . The type σ_3 of the argument e_2 is then compared with σ_1
 1665 using the consistent subtyping judgment.

1666 **MATCHING.** The matching judgment of Siek et al. [2015] is extended to polymorphic types
 1667 naturally, resulting in $\Psi \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2$. Note that the matching rules generalize that of DK
 1668 in Section 2.3.2 with the unknown type. In rule **GPC-M-FORALL**, a monotype τ is guessed
 1669 to instantiate the universal quantifier a . If σ is a polymorphic type, the judgment works by
 1670 guessing instantiations until it reaches an arrow type. Rule **GPC-M-ARR** returns the domain
 1671 type σ_1 and range type σ_2 as expected. If the input is $?$, then rule **GPC-M-UNKNOWN** returns
 1672 $?$ as both the type for the domain and the range.

1673 Note that in GPC, matching saves us from having a subsumption rule (rule **OL-SUB** in
 1674 Figure 2.5). The subsumption rule is incompatible with consistent subtyping, since the latter
 1675 is not transitive. A discussion of a subsumption rule based on normal subtyping can be found
 1676 in Section 4.5.2.

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

$\Psi \vdash^G e : \sigma \rightsquigarrow s$

(Typing)

GPC-VAR

$$\frac{(x : \sigma) \in \Psi}{\Psi \vdash^G x : \sigma \rightsquigarrow x}$$

GPC-INT

$$\frac{}{\Psi \vdash^G n : \text{Int} \rightsquigarrow n}$$

GPC-GEN

$$\frac{\Psi, a \vdash^G e : \sigma \rightsquigarrow s}{\Psi \vdash^G e : \forall a. \sigma \rightsquigarrow \Lambda a. s}$$

GPC-LAMANN

$$\frac{\Psi, x : \sigma \vdash^G e : \sigma_2 \rightsquigarrow s}{\Psi \vdash^G \lambda x : \sigma. e : \sigma \rightarrow \sigma_2 \rightsquigarrow \lambda x : \sigma. s}$$

GPC-LAM

$$\frac{\Psi, x : \tau \vdash^G e : \sigma_2 \rightsquigarrow s}{\Psi \vdash^G \lambda x. e : \tau \rightarrow \sigma_2 \rightsquigarrow \lambda x : \tau. s}$$

GPC-LET

$$\frac{\Psi \vdash^G e_1 : \sigma \rightsquigarrow s_1 \quad \Psi, x : \sigma \vdash^G e_2 : \sigma_2 \rightsquigarrow s_2}{\Psi \vdash^G \text{let } x = e_1 \text{ in } e_2 : \sigma_2 \rightsquigarrow (\lambda x : \sigma. s_2) s_1}$$

GPC-APP

$$\frac{\Psi \vdash^G e_1 : \sigma \rightsquigarrow s_1 \quad \Psi \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \quad \Psi \vdash^G e_2 : \sigma_3 \rightsquigarrow s_2 \quad \Psi \vdash^G \sigma_3 \lesssim \sigma_1}{\Psi \vdash^G e_1 e_2 : \sigma_2 \rightsquigarrow ((\sigma \hookrightarrow \sigma_1 \rightarrow \sigma_2) s_1) ((\sigma_3 \hookrightarrow \sigma_1) s_2)}$$

$\Psi \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2$

(Matching)

GPC-M-FORALL

$$\frac{\Psi \vdash^G \tau \quad \Psi \vdash^G \sigma[a \mapsto \tau] \triangleright \sigma_1 \rightarrow \sigma_2}{\Psi \vdash^G \forall a. \sigma \triangleright \sigma_1 \rightarrow \sigma_2}$$

GPC-M-ARR

$$\frac{}{\Psi \vdash^G \sigma_1 \rightarrow \sigma_2 \triangleright \sigma_1 \rightarrow \sigma_2}$$

GPC-M-UNKNOWN

$$\frac{}{\Psi \vdash^G ? \triangleright ? \rightarrow ?}$$

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

1677 4.3.2 TYPE-DIRECTED TRANSLATION

1678 We give the dynamic semantics of our language by translating it to λB [Ahmed et al. 2009].
 1679 Below we show a subset of the terms in λB that are used in the translation:

$$1680 \quad \lambda B \text{ Terms } \quad s ::= x \mid n \mid \lambda x : \sigma. s \mid \Lambda a. s \mid s_1 s_2 \mid \langle \sigma_1 \hookrightarrow \sigma_2 \rangle s$$

1681 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
 1682 to σ_2 is permitted only if the types are *compatible*, written $\sigma_1 \prec \sigma_2$, as briefly mentioned in
 1683 Section 4.2.1. The syntax of types in λB is the same as ours.

1684 The translation is given in the gray-shaded parts in Figure 4.7. The only interesting case
 1685 here is to insert explicit casts in the application rule. Note that there is no need to translate
 1686 matching or consistent subtyping. Instead we insert the source and target types of a cast
 1687 directly in the translated expressions, thanks to the following two lemmas:

1688 **Lemma 4.5** (\triangleright to \prec). *If $\Psi \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2$, then $\sigma \prec \sigma_1 \rightarrow \sigma_2$.*

1689 **Lemma 4.6** (\lesssim to \prec). *If $\Psi \vdash^G \sigma_1 \lesssim \sigma_2$, then $\sigma_1 \prec \sigma_2$.*

1690 In order to show the correctness of the translation, we prove that our translation always
 1691 produces well-typed expressions in λB . By Lemmas 4.5 and 4.6, we have the following the-
 1692 orem:

1693 **Theorem 4.7** (Type Safety). *If $\Psi \vdash^G e : \sigma \rightsquigarrow s$, then $\Psi \vdash^B s : \sigma$.*

1694 **PARAMETRICITY.** An important semantic property of polymorphic types is *relational para-*
 1695 *metricity* [Reynolds 1983]. The parametricity property says that all instances of a poly-
 1696 morphic function should behave *uniformly*. A classic example is a function with the type
 1697 $\forall a. a \rightarrow a$. The parametricity property guarantees that a value of this type must be either
 1698 the identity function (i.e., $\lambda x. x$) or the undefined function (one which never returns a value).
 1699 However, with the addition of the unknown type $?$, careful measures are to be taken to ensure
 1700 parametricity. Our translation target λB is taken from Ahmed et al. [2009], where relational
 1701 parametricity is enforced by dynamic sealing [Matthews and Ahmed 2008; Neis et al. 2009],
 1702 but there is no rigorous proof. Later, Ahmed et al. [2009] imposed a syntactic restriction on
 1703 terms of λB , where all type abstractions must have *values* as their body. With this invari-
 1704 ant, they proved that the restricted λB satisfies relational parametricity. It remains to see if
 1705 our translation process can be adjusted to target restricted λB . One possibility is to impose
 1706 similar restriction to the rule **GPC-GEN**:

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

1707 where we only generate type abstractions if the inner body is a value. However, the type
 1708 system with this rule is a weaker calculus, which is not a conservative extension of the OL
 1709 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-FORALL](#), 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 $(\lambda x : ?.fx) : ? \rightarrow \text{Int}$, where the type of f is instantiated to $\text{Int} \rightarrow \text{Int}$ and $\text{Bool} \rightarrow \text{Int}$, respectively:

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

1710 If we apply $\lambda x : ?.fx$ to 3, which is fine since the function can take any input, the first
 1711 translation runs smoothly in λB , while the second one will raise a cast error (Int cannot be
 1712 cast to Bool). Similarly, if we apply it to `true`, then the second succeeds while the first fails.
 1713 The culprit lies in the highlighted parts where the instantiation of a appears in the explicit
 1714 cast. More generally, any choice introduces an explicit cast to that type in the translation,
 1715 which causes a runtime cast error if the function is applied to a value whose type does not
 1716 match the guessed type. Note that this does not compromise the type safety of the translated
 1717 expressions, since cast errors are part of the type safety guarantees.

1718 The semantic discrepancy is due to the guessing nature of the *declarative* system. As far
 1719 as the static semantics is concerned, both $\text{Int} \rightarrow \text{Int}$ and $\text{Bool} \rightarrow \text{Int}$ are equally acceptable.
 1720 But this is not the case at runtime. The astute reader may have found that the *only* appropriate
 1721 choice is to instantiate the type of f to $? \rightarrow \text{Int}$ in the matching judgment. However,
 1722 as specified by rule [GPC-M-FORALL](#) in Figure 4.7, we can only instantiate type variables to
 1723 monotypes, but $?$ is *not* a monotype! We will get back to this issue in Chapter 5.

1724 COHERENCE. The ambiguity of translation seems to imply that the declarative system is
 1725 *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).

$$\begin{aligned} \Psi \vdash s_1 \preceq_{ctx} s_2 : \sigma &\triangleq \Psi \vdash^B s_1 : \sigma \wedge \Psi \vdash^B s_2 : \sigma \wedge \\ &\text{for all } \mathcal{C}. \mathcal{C} : (\Psi \vdash^B \sigma) \rightsquigarrow (\bullet \vdash^B \text{Int}) \implies \\ &\mathcal{C}\{s_1\} \Downarrow n \implies (\mathcal{C}\{s_2\} \Downarrow n \vee \mathcal{C}\{s_2\} \Downarrow \text{blame}) \\ \Psi \vdash s_1 \simeq_{ctx} s_2 : \sigma &\triangleq \Psi \vdash s_1 \preceq_{ctx} s_2 : \sigma \wedge \Psi \vdash s_2 \preceq_{ctx} s_1 : \sigma \end{aligned}$$

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 $[s]$ 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 $[s_1] \equiv_\alpha [s_2]$.*

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 $\Psi \vdash s_1 \simeq_{ctx} s_2 : \sigma$.*

1754 4.3.3 CORRECTNESS CRITERIA

1755 Siek et al. [2015] present a set of properties, the *refined criteria*, that a well-designed gradual
 1756 typing calculus must have. Among all the criteria, those related to the static aspects of gradual
 1757 typing are well summarized by Cimini and Siek [2016]. Here we review those criteria and
 1758 adapt them to our notation. We have proved in Coq that our type system satisfies all these
 1759 criteria.

1760 **Lemma 4.10** (Correctness Criteria).

- 1761 • **Conservative extension:** for all static Ψ , e , and σ_1 ,
 - 1762 – if $\Psi \vdash^{OL} e : \sigma_1$, then there exists σ_2 , such that $\Psi \vdash^G e : \sigma_2$, and $\Psi \vdash^G \sigma_2 <: \sigma_1$.
 - 1763 – if $\Psi \vdash^G e : \sigma$, then $\Psi \vdash^{OL} e : \sigma$
- 1764 • **Monotonicity w.r.t. precision:** for all Ψ , e , e' , σ_1 , if $\Psi \vdash^G e : \sigma_1$, and $e' \sqsubseteq e$, then
 1765 $\Psi \vdash^G e' : \sigma_2$, and $\sigma_2 \sqsubseteq \sigma_1$ for some σ_2 .
- 1766 • **Type Preservation of cast insertion:** for all Ψ , e , σ , if $\Psi \vdash^G e : \sigma$, then $\Psi \vdash^G e : \sigma \rightsquigarrow s$,
 1767 and $\Psi \vdash^B s : \sigma$ for some s .
- 1768 • **Monotonicity of cast insertion:** for all Ψ , e_1 , e_2 , s_1 , s_2 , σ , if $\Psi \vdash^G e_1 : \sigma \rightsquigarrow s_1$, and
 1769 $\Psi \vdash^G e_2 : \sigma \rightsquigarrow s_2$, and $e_1 \sqsubseteq e_2$, then $\Psi \vdash s_1 \sqsubseteq^B s_2$.

1770 The first criterion states that the gradual type system should be a conservative extension
 1771 of the original system. In other words, a *static* program is typeable in the OL type system if
 1772 and only if it is typeable in the gradual type system. A static program is one that does not
 1773 contain any type ⁶. However since our gradual type system does not have the subsumption
 1774 rule, it produces more general types.

1775 The second criterion states that if a typeable expression loses some type information, it
 1776 remains typeable. This criterion depends on the definition of the precision relation, written
 1777 $\sigma_1 \sqsubseteq \sigma_2$, which is given in Figure 4.8. The relation intuitively captures a notion of types con-
 1778 taining more or less unknown types (?). The precision relation over types lifts to programs,
 1779 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
 1780 types.

1781 The first two criteria are fundamental to gradual typing. They explain for example why
 1782 these two programs $\lambda x : \text{Int}. x + 1$ and $\lambda x : ?. x + 1$ are typeable, as the former is typeable in
 1783 the OL type system and the latter is a less-precise version of it.

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

$\sigma_1 \sqsubseteq \sigma_2$		(Type Precision)	
GPC-L-UNKNOWN	GPC-L-INT	GPC-L-ARROW	GPC-L-TVAR
$\frac{}{? \sqsubseteq \sigma}$	$\frac{}{\text{Int} \sqsubseteq \text{Int}}$	$\frac{\sigma_1 \sqsubseteq \sigma_3 \quad \sigma_2 \sqsubseteq \sigma_4}{\sigma_1 \rightarrow \sigma_2 \sqsubseteq \sigma_3 \rightarrow \sigma_4}$	$\frac{}{a \sqsubseteq a}$
		GPC-L-FORALL	
		$\frac{\sigma_1 \sqsubseteq \sigma_2}{\forall a. \sigma_1 \sqsubseteq \forall a. \sigma_2}$	
$e_1 \sqsubseteq e_2$		(Term Precision)	
GPC-LE-REFL	GPC-LE-LAMANN	GPC-LE-APP	
$\frac{}{e \sqsubseteq e}$	$\frac{\sigma_1 \sqsubseteq \sigma_2 \quad e_1 \sqsubseteq e_2}{\lambda x : \sigma_1. e_1 \sqsubseteq \lambda x : \sigma_2. e_2}$	$\frac{e_1 \sqsubseteq e_3 \quad e_2 \sqsubseteq e_4}{e_1 e_2 \sqsubseteq e_3 e_4}$	
$s_1 \sqsubseteq s_2$		(Term Precision in λB)	
B-LE-VAR	B-LE-NAT	B-LE-LAMANN	B-LE-TABS
$\frac{}{x \sqsubseteq x}$	$\frac{}{n \sqsubseteq n}$	$\frac{\sigma_1 \sqsubseteq \sigma_2 \quad s_1 \sqsubseteq s_2}{\lambda x : \sigma_1. s_1 \sqsubseteq \lambda x : \sigma_2. s_2}$	$\frac{s_1 \sqsubseteq s_2}{\Lambda a. s_1 \sqsubseteq \Lambda a. s_2}$
B-LE-APP		B-LE-CAST	
$\frac{s_1 \sqsubseteq s_3 \quad s_2 \sqsubseteq s_4}{s_1 s_2 \sqsubseteq s_3 s_4}$		$\frac{\sigma_1 \sqsubseteq \sigma_3 \quad \sigma_2 \sqsubseteq \sigma_4 \quad s_1 \sqsubseteq s_2}{\langle \sigma_1 \hookrightarrow \sigma_2 \rangle s_1 \sqsubseteq \langle \sigma_3 \hookrightarrow \sigma_4 \rangle s_2}$	

Figure 4.8: Less Precision

The last two criteria relate the compilation to the cast calculus. The third criterion is essentially the same as Theorem 4.7, given that a target expression should always exist, which can be easily seen from 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 $\bullet \vdash^G e' : \sigma' \rightsquigarrow s'$,

- 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 $s \Downarrow \text{blame}$. If $s' \Uparrow$ then $s \Uparrow$ or $s \Downarrow \text{blame}$.

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:

$$(\lambda f : \forall a. a \rightarrow \text{Int}. \lambda x : \text{Int}. f x) (\lambda x. 1) \ 3 \qquad (\lambda f : \forall a. a \rightarrow \text{Int}. \lambda x : ?. f x) (\lambda x. 1) \ 3$$

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.

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	$\sigma ::= \text{Int} \mid a \mid \hat{\alpha} \mid \sigma_1 \rightarrow \sigma_2 \mid \forall a. \sigma \mid ?$
Monotypes	$\tau ::= \text{Int} \mid a \mid \hat{\alpha} \mid \tau_1 \rightarrow \tau_2$
Algorithmic Contexts Γ, Δ, Θ	$::= \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \hat{\alpha} \mid \Gamma, \hat{\alpha} = \tau \mid \Gamma, \blacktriangleright_{\hat{\alpha}}$
Complete Contexts Ω	$::= \bullet \mid \Omega, x : \sigma \mid \Omega, a \mid \Omega, \hat{\alpha} = \tau \mid \Omega, \blacktriangleright_{\hat{\alpha}}$

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

GPC-AD-INT

 $\overline{\Gamma \vdash^G \text{Int}}$

GPC-AD-UNKNOWN

 $\overline{\Gamma \vdash^G ?}$

GPC-AD-TVAR

 $\overline{\Gamma[a] \vdash^G a}$

GPC-AD-EVAR

 $\overline{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha}}$

GPC-AD-SOLVED

 $\overline{\Gamma[\hat{\alpha} = \tau] \vdash^G \hat{\alpha}}$

GPC-AD-ARROW

 $\frac{\Gamma \vdash^G \sigma_1 \quad \Gamma \vdash^G \sigma_2}{\Gamma \vdash^G \sigma_1 \rightarrow \sigma_2}$

GPC-AD-FORALL

 $\frac{\Gamma, a \vdash^G \sigma}{\Gamma \vdash^G \forall a. \sigma}$

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

GPC-WF-EMPTY

 $\overline{\vdash^G \bullet}$

GPC-WF-VAR

 $\frac{\vdash^G \Gamma \quad x \notin \text{FV}(\Gamma) \quad \Gamma \vdash^G \sigma}{\vdash^G \Gamma, x : \sigma}$

GPC-WF-TVAR

 $\frac{\vdash^G \Gamma \quad a \notin \text{FV}(\Gamma)}{\vdash^G \Gamma, a}$

GPC-WF-EVAR

 $\frac{\vdash^G \Gamma \quad \hat{\alpha} \notin \text{FV}(\Gamma)}{\vdash^G \Gamma, \hat{\alpha}}$

GPC-WF-SOLVED

 $\frac{\vdash^G \Gamma \quad \hat{\alpha} \notin \text{FV}(\Gamma) \quad \Gamma \vdash^G \tau}{\vdash^G \Gamma, \hat{\alpha} = \tau}$

GPC-WF-MARKER

 $\frac{\vdash^G \Gamma \quad \blacktriangleright_{\hat{\alpha}} \notin \text{FV}(\Gamma)}{\vdash^G \Gamma, \blacktriangleright_{\hat{\alpha}}}$

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

1809 **ALGORITHMIC CONTEXTS.** The top of Figure 4.9 shows the syntax of the algorithmic sys-
 1810 tem. A noticeable difference are the algorithmic contexts Γ , which are represented as an *or-*
 1811 *dered* list containing declarations of type variables a and term variables $x : \sigma$. Unlike declar-
 1812 ative contexts, algorithmic contexts also contain declarations of existential type variables $\hat{\alpha}$,
 1813 which can be either unsolved (written $\hat{\alpha}$) or solved to some monotype (written $\hat{\alpha} = \tau$).
 1814 Finally, algorithmic contexts include a *marker* $\blacktriangleright_{\hat{\alpha}}$ (read “marker $\hat{\alpha}$ ”), which is used to de-
 1815 lineate existential variables created by the algorithm. We will have more to say about markers
 1816 when we examine the rules. Complete contexts Ω are the same as contexts, except that they
 1817 contain no unsolved variables.

1818 Apart from expressions in the declarative system, we add annotated expressions $e : \sigma$. The
 1819 well-formedness judgments for types and contexts are shown in Figure 4.9.

1820 NOTATIONAL CONVENIENCE. Following DK's system, we use contexts as substitutions on
 1821 types. We write $[\Gamma]\sigma$ to mean Γ applied as a substitution to type σ , whose definition is given
 1822 below. We also use a hole notation, which is useful when manipulating contexts by inserting
 1823 and replacing declarations in the middle. The hole notation is used extensively in proving
 1824 soundness and completeness. For example, $\Gamma[\Theta]$ means Γ has the form $\Gamma_L, \Theta, \Gamma_R$; if we have
 1825 $\Gamma[\hat{\alpha}] = (\Gamma_L, \hat{\alpha}, \Gamma_R)$, then $\Gamma[\hat{\alpha} = \tau] = (\Gamma_L, \hat{\alpha} = \tau, \Gamma_R)$. Occasionally, we will see a context
 1826 with two *ordered* holes, e.g., $\Gamma = \Gamma_0[\Theta_1][\Theta_2]$ means Γ has the form $\Gamma_L, \Theta_1, \Gamma_M, \Theta_2, \Gamma_R$.

	$[\Delta]\text{Int}$	$=$	Int
	$[\Delta]a$	$=$	a
1827	$[\Delta[\hat{\alpha}]]\hat{\alpha}$	$=$	$\hat{\alpha}$
	$[\Delta[\hat{\alpha} = \tau]]\hat{\alpha}$	$=$	$[\Delta[\hat{\alpha} = \tau]]\tau$
	$[\Delta]\sigma_1 \rightarrow \sigma_2$	$=$	$[\Delta]\sigma_1 \rightarrow [\Delta]\sigma_2$
	$[\Delta](\forall a. \sigma)$	$=$	$\forall a. [\Delta]\sigma$

1828 INPUT AND OUTPUT CONTEXTS. The algorithmic system, compared with the declarative
 1829 system, includes similar judgment forms, except that we replace the declarative context Ψ
 1830 with an algorithmic context Γ (the *input context*), and add an *output context* Δ after a back-
 1831 ward turnstile, e.g., $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$ is the judgment form for the algorithmic consistent
 1832 subtyping. All algorithmic rules manipulate input and output contexts in a way that is con-
 1833 sistent with the notion of *context extension*, which will be described in Section 4.4.5.

1834 We start with the explanation of the algorithmic consistent subtyping as it involves ma-
 1835 nipulating existential type variables explicitly (and solving them if possible).

1836 4.4.1 ALGORITHMIC CONSISTENT SUBTYPING

1837 Figure 4.10 presents the rules of algorithmic consistent subtyping $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$, which
 1838 says that under input context Γ , σ_1 is a consistent subtype of σ_2 , with output context Δ . The
 1839 first five rules do not manipulate contexts, but illustrate how contexts are propagated.

1840 Rule [GPC-AS-TVAR](#) and rule [GPC-AS-INT](#) do not involve existential variables, so the output
 1841 contexts remain unchanged. Rule [GPC-AS-EVAR](#) says that any unsolved existential variable is a
 1842 consistent subtype of itself. The output is still the same as the input context as the rule gives no
 1843 clue as to what is the solution of that existential variable. Rules [GPC-AS-UNKNOWNL](#) and [AS-](#)
 1844 [UNKNOWNR](#) are the counterparts of rule [GPC-CS-UNKNOWNL](#) and rule [GPC-CS-UNKNOWNR](#).

1845 Rule [GPC-AS-ARROW](#) is a natural extension of its declarative counterpart. The output con-
 1846 text of the first premise is used by the second premise, and the output context of the second
 1847 premise is the output context of the conclusion. Note that we do not simply check $\sigma_2 \lesssim \sigma_4$,

<div style="border: 1px solid black; padding: 5px; display: inline-block;"> $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$ </div> (Under input context Γ, σ_1 is a consistent subtype of σ_2, with output context Δ)		
GPC-AS-TVAR	GPC-AS-INT	GPC-AS-EVAR
$\overline{\Gamma[a] \vdash^G a \lesssim a \dashv \Gamma[a]}$	$\overline{\Gamma \vdash^G \text{Int} \lesssim \text{Int} \dashv \Gamma}$	$\overline{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \hat{\alpha} \dashv \Gamma[\hat{\alpha}]}$
GPC-AS-UNKNOWNL		GPC-AS-UNKNOWNR
$\overline{\Gamma \vdash^G ? \lesssim \sigma \dashv \Gamma}$		$\overline{\Gamma \vdash^G \sigma \lesssim ? \dashv \Gamma}$
GPC-AS-ARROW		GPC-AS-FORALLR
$\frac{\Gamma \vdash^G \sigma_3 \lesssim \sigma_1 \dashv \Theta \quad \Theta \vdash^G [\Theta]\sigma_2 \lesssim [\Theta]\sigma_4 \dashv \Delta}{\Gamma \vdash^G \sigma_1 \rightarrow \sigma_2 \lesssim \sigma_3 \rightarrow \sigma_4 \dashv \Delta}$		$\frac{\Gamma, a \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta, a, \Theta}{\Gamma \vdash^G \sigma_1 \lesssim \forall a. \sigma_2 \dashv \Delta}$
GPC-AS-FORALLL	GPC-AS-INSTL	
$\frac{\Gamma, \blacktriangleright_{\hat{\alpha}}, \hat{\alpha} \vdash^G \sigma_1[a \mapsto \hat{\alpha}] \lesssim \sigma_2 \dashv \Delta, \blacktriangleright_{\hat{\alpha}}, \Theta}{\Gamma \vdash^G \forall a. \sigma_1 \lesssim \sigma_2 \dashv \Delta}$	$\frac{\hat{\alpha} \notin \text{FV}(\sigma) \quad \Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta}$	
GPC-AS-INSTR		
$\frac{\hat{\alpha} \notin \text{FV}(\sigma) \quad \Gamma[\hat{\alpha}] \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta}$		

Figure 4.10: Algorithmic consistent subtyping

but apply Θ (the input context of the second premise) to both types (e.g., $[\Theta]\sigma_2$). This is to maintain an important invariant: whenever $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$ holds, the types σ_1 and σ_2 are fully applied under input context Γ (they contain no existential variables already solved in Γ). The same invariant applies to every algorithmic judgment.

Rule **GPC-AS-FORALLR**, similar to the declarative rule **GPC-CS-FORALLR**, adds a to the input context. Note that the output context of the premise allows additional existential variables to appear after the type variable a , in a trailing context Θ . These existential variables could depend on a ; since a goes out of scope in the conclusion, we need to drop them from the concluding output, resulting in Δ . The next rule is essential to eliminating the guessing work. Instead of guessing a monotype τ out of thin air, rule **GPC-AS-FORALLL** generates a fresh existential variable $\hat{\alpha}$, and replaces a with $\hat{\alpha}$ in the body σ . The new existential variable $\hat{\alpha}$ is then added to the input context, just before the marker $\blacktriangleright_{\hat{\alpha}}$. The output context $(\Delta, \blacktriangleright_{\hat{\alpha}}, \Theta)$ allows additional existential variables to appear after $\blacktriangleright_{\hat{\alpha}}$ 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 $\Gamma \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta$ and $\Gamma \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta$, defined in Figure 4.11, instantiate unsolved existential variables. They read “under input context Γ , instantiate $\hat{\alpha}$ 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 $\hat{\alpha}$ 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 .

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

$$\begin{array}{c}
 \text{GPC-INSTL-SOLVE} \\
 \frac{\Gamma \vdash^G \tau}{\Gamma, \hat{\alpha}, \Gamma' \vdash^G \hat{\alpha} \lesssim \tau \dashv \Gamma, \hat{\alpha} = \tau, \Gamma'} \\
 \\
 \text{GPC-INSTL-SOLVEU} \\
 \frac{}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim ? \dashv \Gamma[\hat{\alpha}]} \\
 \\
 \text{GPC-INSTL-REACH} \\
 \frac{}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash^G \hat{\alpha} \lesssim \hat{\beta} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]} \\
 \\
 \text{GPC-INSTL-FORALLR} \\
 \frac{\Gamma[\hat{\alpha}], b \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta, b, \Theta}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \forall b. \sigma \dashv \Delta} \\
 \\
 \text{GPC-INSTL-ARR} \\
 \frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \vdash^G \sigma_1 \lesssim \hat{\alpha}_1 \dashv \Theta \quad \Theta \vdash^G \hat{\alpha}_2 \lesssim [\Theta]\sigma_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \sigma_1 \rightarrow \sigma_2 \dashv \Delta}
 \end{array}$$

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

$$\begin{array}{c}
 \text{GPC-INSTR-SOLVE} \\
 \frac{\Gamma \vdash^G \tau}{\Gamma, \hat{\alpha}, \Gamma' \vdash^G \tau \lesssim \hat{\alpha} \dashv \Gamma, \hat{\alpha} = \tau, \Gamma'} \\
 \\
 \text{GPC-INSTR-SOLVEU} \\
 \frac{}{\Gamma[\hat{\alpha}] \vdash^G ? \lesssim \hat{\alpha} \dashv \Gamma[\hat{\alpha}]} \\
 \\
 \text{GPC-INSTR-REACH} \\
 \frac{}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash^G \hat{\beta} \lesssim \hat{\alpha} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]} \\
 \\
 \text{GPC-INSTR-FORALLL} \\
 \frac{\Gamma[\hat{\alpha}], \blacktriangleright_{\hat{\beta}}, \hat{\beta} \vdash^G \sigma[b \mapsto \hat{\beta}] \lesssim \hat{\alpha} \dashv \Delta, \blacktriangleright_{\hat{\beta}}, \Theta}{\Gamma[\hat{\alpha}] \vdash^G \forall b. \sigma \lesssim \hat{\alpha} \dashv \Delta} \\
 \\
 \text{GPC-INSTR-ARR} \\
 \frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \vdash^G \hat{\alpha}_1 \lesssim \sigma_1 \dashv \Theta \quad \Theta \vdash^G [\Theta]\sigma_2 \lesssim \hat{\alpha}_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \sigma_1 \rightarrow \sigma_2 \lesssim \hat{\alpha} \dashv \Delta}
 \end{array}$$

Figure 4.11: Algorithmic instantiation

1881 Rule **GPC-INSTL-SOLVEU** is similar to rule **GPC-AS-UNKNOWNR** in that we put no con-
 1882 straint⁷ on $\hat{\alpha}$ when it meets the unknown type $?$. This design decision reflects the point
 1883 that type inference only produces static types [Garcia and Cimini 2015].

1884 Rule **GPC-INSTL-REACH** deals with the situation where two existential variables meet. Re-
 1885 call that $\Gamma[\hat{\alpha}][\hat{\beta}]$ denotes a context where some unsolved existential variable $\hat{\alpha}$ is declared
 1886 before $\hat{\beta}$. In this situation, the only logical thing we can do is to set the solution of one exis-
 1887 tential variable to the other one, depending on which one is declared before. For example, in
 1888 the output context of rule **GPC-INSTL-REACH**, we have $\hat{\beta} = \hat{\alpha}$ because in the input context, $\hat{\alpha}$
 1889 is declared before $\hat{\beta}$.

1890 Rule **GPC-INSTL-FORALLR** is the instantiation version of rule **GPC-AS-FORALLR**. Since our
 1891 system is predicative, $\hat{\alpha}$ cannot be instantiated to $\forall b. \sigma$, but we can decompose $\forall b. \sigma$ in the
 1892 same way as in rule **GPC-AS-FORALLR**. Rule **GPC-INSTL-FORALLL** is the instantiation version
 1893 of rule **GPC-AS-FORALLL**.

1894 Rule **GPC-INSTL-ARR** applies when $\hat{\alpha}$ meets an arrow type. It follows that the solution
 1895 must also be an arrow type. This is why, in the first premise, we generate two fresh existential
 1896 variables $\hat{\alpha}_1$ and $\hat{\alpha}_2$, and insert them just before $\hat{\alpha}$ in the input context, so that we can solve
 1897 $\hat{\alpha}$ to $\hat{\alpha}_1 \rightarrow \hat{\alpha}_2$. Note that the first premise $\sigma_1 \lesssim \hat{\alpha}_1$ switches to the other instantiation
 1898 judgment.

1899 4.4.3 ALGORITHMIC TYPING

1900 We now turn to the algorithmic typing rules in Figure 4.12. Because general type infer-
 1901 ence for System F is undecidable [Wells 1999], our algorithmic system uses bidirectional
 1902 type checking to accommodate (first-class) polymorphism. Traditionally, two modes are
 1903 employed in bidirectional systems: the checking mode $\Gamma \vdash^G e \Leftarrow \sigma \dashv \Theta$, which takes a
 1904 term e and a type σ as input, and ensures that the term e checks against σ ; the inference
 1905 mode $\Gamma \vdash^G e \Rightarrow \sigma \dashv \Theta$, which takes a term e and produces a type σ . We first discuss rules
 1906 in the inference mode.

1907 Rule **GPC-INF-VAR** and rule **GPC-INF-INT** do not generate any new information and sim-
 1908 ply propagate the input context. Rule **GPC-INF-ANNO** is standard, switching to the checking
 1909 mode in the premise.

1910 In rule **GPC-INF-LAMANN**, we generate a fresh existential variable $\hat{\beta}$ for the function codomain,
 1911 and check the function body against $\hat{\beta}$. Note that it is tempting to write $\Gamma, x : \sigma \vdash^G e \Rightarrow$
 1912 $\sigma_2 \dashv \Delta, x : \sigma, \Theta$ as the premise (in the hope of better matching its declarative counterpart
 1913 rule **GPC-LAMANN**), which has a subtle consequence. Consider the expression $\lambda x : \text{Int}. \lambda y. y$.

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

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

$$\begin{array}{c}
 \text{GPC-INF-VAR} \\
 \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash^G x \Rightarrow \sigma \dashv \Gamma} \\
 \\
 \text{GPC-INF-INT} \\
 \frac{}{\Gamma \vdash^G n \Rightarrow \text{Int} \dashv \Gamma} \\
 \\
 \text{GPC-INF-ANNO} \\
 \frac{\Gamma \vdash^G \sigma \quad \Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta}{\Gamma \vdash^G e : \sigma \Rightarrow \sigma \dashv \Delta} \\
 \\
 \text{GPC-INF-LAMANN} \\
 \frac{\Gamma \vdash^G \sigma \quad \Gamma, \widehat{\beta}, x : \sigma \vdash^G e \Leftarrow \widehat{\beta} \dashv \Delta, x : \sigma, \Theta}{\Gamma \vdash^G \lambda x : \sigma. e \Rightarrow \sigma \rightarrow \widehat{\beta} \dashv \Delta} \\
 \\
 \text{GPC-INF-LAM} \\
 \frac{\Gamma, \widehat{\alpha}, \widehat{\beta}, x : \widehat{\alpha} \vdash^G e \Leftarrow \widehat{\beta} \dashv \Delta, x : \widehat{\alpha}, \Theta}{\Gamma \vdash^G \lambda x. e \Rightarrow \widehat{\alpha} \rightarrow \widehat{\beta} \dashv \Delta} \\
 \\
 \text{GPC-INF-LET} \\
 \frac{\Gamma \vdash^G e_1 \Rightarrow \sigma \dashv \Theta_1 \quad \Theta_1, \widehat{\alpha}, x : \sigma \vdash^G e_2 \Leftarrow \widehat{\alpha} \dashv \Delta, x : \sigma, \Theta_2}{\Gamma \vdash^G \text{let } x = e_1 \text{ in } e_2 \Rightarrow \widehat{\alpha} \dashv \Delta} \\
 \\
 \text{GPC-INF-APP} \\
 \frac{\Gamma \vdash^G e_1 \Rightarrow \sigma \dashv \Theta_1 \quad \Theta_1 \vdash^G [\Theta_1] \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Theta_2 \quad \Theta_2 \vdash^G e_2 \Leftarrow [\Theta_2] \sigma_1 \dashv \Delta}{\Gamma \vdash^G e_1 e_2 \Rightarrow \sigma_2 \dashv \Delta}
 \end{array}$$

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

$$\begin{array}{c}
 \text{GPC-CHK-LAM} \\
 \frac{\Gamma, x : \sigma_1 \vdash^G e \Leftarrow \sigma_2 \dashv \Delta, x : \sigma_1, \Theta}{\Gamma \vdash^G \lambda x. e \Leftarrow \sigma_1 \rightarrow \sigma_2 \dashv \Delta} \\
 \\
 \text{GPC-CHK-GEN} \\
 \frac{\Gamma, a \vdash^G e \Leftarrow \sigma \dashv \Delta, a, \Theta}{\Gamma \vdash^G e \Leftarrow \forall a. \sigma \dashv \Delta} \\
 \\
 \text{GPC-CHK-SUB} \\
 \frac{\Gamma \vdash^G e \Rightarrow \sigma_1 \dashv \Theta \quad \Theta \vdash^G [\Theta] \sigma_1 \lesssim [\Theta] \sigma_2 \dashv \Delta}{\Gamma \vdash^G e \Leftarrow \sigma_2 \dashv \Delta}
 \end{array}$$

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

$$\begin{array}{c}
 \text{GPC-AM-FORALL} \\
 \frac{\Gamma, \widehat{\alpha} \vdash^G \sigma[a \mapsto \widehat{\alpha}] \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta}{\Gamma \vdash^G \forall a. \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta} \\
 \\
 \text{GPC-AM-ARR} \\
 \frac{}{\Gamma \vdash^G \sigma_1 \rightarrow \sigma_2 \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Gamma} \\
 \\
 \text{GPC-AM-UNKNOWN} \\
 \frac{}{\Gamma \vdash^G ? \triangleright ? \rightarrow ? \dashv \Gamma} \\
 \\
 \text{GPC-AM-VAR} \\
 \frac{}{\Gamma[\widehat{\alpha}] \vdash^G \widehat{\alpha} \triangleright \widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2 \dashv \Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2, \widehat{\alpha} = \widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2]}
 \end{array}$$

Figure 4.12: Algorithmic typing

Under the new premise, this is untypable because of $\bullet \vdash^G \lambda x : \text{Int}. \lambda y. y \Rightarrow \text{Int} \rightarrow \hat{\alpha} \rightarrow \hat{\alpha} \dashv$ where $\hat{\alpha}$ is not found in the output context. This explains why we put $\hat{\beta}$ 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 $\hat{\alpha}$ (for function domain) and $\hat{\beta}$ (for function codomain) and check the function body against $\hat{\beta}$. Rule **GPC-INF-LET** is similar to rule **GPC-INF-LAMANN**.

ALGORITHMIC MATCHING. Rule **GPC-INF-APP** deserves attention. It relies on the algorithmic matching judgment $\Gamma \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta$. The matching judgment algorithmically synthesizes an arrow type from an arbitrary type. Rule **GPC-AM-FORALL** replaces a with a fresh existential variable $\hat{\alpha}$, 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-INSTL-ARR**: we create $\hat{\alpha}_1$ and $\hat{\alpha}_2$ and solve $\hat{\alpha}$ to $\hat{\alpha}_1 \rightarrow \hat{\alpha}_2$ in the output context.

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

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

4.4.4 DECIDABILITY

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

DECIDABILITY OF INSTANTIATION. The basic idea is that we need to show σ in the instantiation judgments $\Gamma \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta$ and $\Gamma \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta$ 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 $\hat{\alpha}$ 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 $\hat{\alpha}$ 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 $\Gamma = \Gamma_0[\hat{\alpha}]$ and $\Gamma \vdash^G \sigma$ such that $[\Gamma]\sigma = \sigma$ and $\hat{\alpha} \notin \text{FV}(\sigma)$ then:*

1. *Either there exists Δ such that $\Gamma \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta$, or not.*
2. *Either there exists Δ such that $\Gamma \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta$, 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) $|\text{UNSOLVED}(\Gamma)|$: 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).

$$\begin{aligned}
 |\Gamma \vdash^G \text{Int}| &= 1 \\
 |\Gamma \vdash^G ?| &= 1 \\
 |\Gamma \vdash^G a| &= 1 \\
 |\Gamma \vdash^G \hat{\alpha}| &= 1 \\
 |\Gamma[\hat{\alpha} = \tau] \vdash^G \hat{\alpha}| &= 1 + |\Gamma[\hat{\alpha} = \tau] \vdash^G \tau| \quad (*) \\
 |\Gamma \vdash^G \forall a. \sigma| &= 1 + |\Gamma, a \vdash^G \sigma| \\
 |\Gamma \vdash^G \sigma_1 \rightarrow \sigma_2| &= 1 + |\Gamma \vdash^G \sigma_1| + |\Gamma \vdash^G \sigma_2|
 \end{aligned}$$

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 $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$.*

1973 DECIDABILITY OF ALGORITHMIC TYPING. Similar to proving decidability of algorithmic
 1974 consistent subtyping, the key is to come up with a correct measure. Since the typing rules
 1975 depend on the matching judgment, we first show decidability of the algorithmic matching.

1976 **Lemma 4.13** (Decidability of Algorithmic Matching). *Given a context Γ and a type σ it is*
 1977 *decidable whether there exist types σ_1, σ_2 and a context Δ such that $\Gamma \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta$.*

1978 Now we are ready to show decidability of typing. The proof is obtained by induction on
 1979 the lexicographically ordered triple: size of e , typing judgment (where the inference mode
 1980 \Rightarrow is considered smaller than the checking mode \Leftarrow) and contextual size.

$$1981 \quad \left\langle e, \begin{array}{c} \Rightarrow \\ \Leftarrow \end{array}, |\Gamma \vdash^G \sigma| \right\rangle$$

1982 The above measure is much simpler than the corresponding one in the DK system, where
 1983 they also need to consider the application judgment together with the inference and checking
 1984 judgments. This shows another benefit (besides the independence from typing) of adopting
 1985 the matching judgment.

1986 **Theorem 4.14** (Decidability of Algorithmic Typing).

- 1987 1. *Inference: Given a context Γ and a term e , it is decidable whether there exist a type σ*
 1988 *and a context Δ such that $\Gamma \vdash^G e \Rightarrow \sigma \dashv \Delta$.*
- 1989 2. *Checking: Given a context Γ , a term e and a type σ such that $\Gamma \vdash^G \sigma$, it is decidable*
 1990 *whether there exists a context Δ such that $\Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta$.*

1991 4.4.5 CONTEXT EXTENSION

1992 To be confident that our algorithmic type system and the declarative type system agree with
 1993 each other, we need to prove that the algorithmic rules are sound and complete with respect
 1994 to the declarative specification. Before we give the formal statements of the soundness and
 1995 completeness theorems, we need a meta-theoretical device, called *context extension* [Dun-
 1996 field and Krishnaswami 2013], to capture a notion of information increase from input con-
 1997 texts to output contexts.

1998 A context extension judgment $\Gamma \longrightarrow \Delta$ reads “ Γ is extended by Δ ”. Intuitively, this judg-
 1999 ment says that Δ has at least as much information as Γ : some unsolved existential variables
 2000 in Γ may be solved in Δ . The full inductive definition can be found Figure 4.13.

$\boxed{\Gamma \longrightarrow \Delta}$				(Context extension)
GPC-EXT-ID $\frac{}{\bullet \longrightarrow \bullet}$	GPC-EXT-VAR $\frac{\Gamma \longrightarrow \Delta \quad [\Delta]\sigma = [\Delta]\sigma'}{\Gamma, x : \sigma \longrightarrow \Delta, x : \sigma'}$	GPC-EXT-TVAR $\frac{\Gamma \longrightarrow \Delta}{\Gamma, a \longrightarrow \Delta, a}$	GPC-EXT-EVAR $\frac{\Gamma \longrightarrow \Delta}{\Gamma, \hat{\alpha} \longrightarrow \Delta, \hat{\alpha}}$	
GPC-EXT-SOLVED $\frac{\Gamma \longrightarrow \Delta \quad [\Delta]\tau = [\Delta]\tau'}{\Gamma, \hat{\alpha} = \tau \longrightarrow \Delta, \hat{\alpha} = \tau'}$	GPC-EXT-SOLVE $\frac{\Gamma \longrightarrow \Delta}{\Gamma, \hat{\alpha} \longrightarrow \Delta, \hat{\alpha} = \tau}$	GPC-EXT-ADD $\frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \hat{\alpha}}$		
	GPC-EXT-ADDSOLVE $\frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \hat{\alpha} = \tau}$	GPC-EXT-MARKER $\frac{\Gamma \longrightarrow \Delta}{\Gamma, \blacktriangleright_{\hat{\alpha}} \longrightarrow \Delta, \blacktriangleright_{\hat{\alpha}}}$		

Figure 4.13: Context extension

2001 4.4.6 SOUNDNESS

2002 Roughly speaking, soundness of the algorithmic system says that given a derivation of an
 2003 algorithmic judgment with input context Γ , output context Δ , and a complete context Ω that
 2004 extends Δ , applying Ω throughout the given algorithmic judgment should yield a derivable
 2005 declarative judgment. For example, let us consider an algorithmic typing judgment $\bullet \vdash^G$
 2006 $\lambda x. x \Rightarrow \hat{\alpha} \rightarrow \hat{\alpha} \dashv \hat{\alpha}$, and any complete context, say, $\Omega = (\hat{\alpha} = \text{Int})$, then applying Ω to the
 2007 above judgment yields $\bullet \vdash^G \lambda x. x : \text{Int} \rightarrow \text{Int}$, which is derivable in the declarative system.

2008 However there is one complication: applying Ω to the algorithmic expression does not
 2009 necessarily yield a typable declarative expression. For example, by rule [GPC-CHK-LAM](#) we
 2010 have $\lambda x. x \Leftarrow (\forall a. a \rightarrow a) \rightarrow (\forall a. a \rightarrow a)$, but $\lambda x. x$ itself cannot have type $(\forall a. a \rightarrow$
 2011 $a) \rightarrow (\forall a. a \rightarrow a)$ in the declarative system. To circumvent that, we add an annotation to
 2012 the lambda abstraction, resulting in $\lambda x : (\forall a. a \rightarrow a). x$, which is typeable in the declarative
 2013 system with the same type. To relate $\lambda x. x$ and $\lambda x : (\forall a. a \rightarrow a). x$, we erase all annotations
 2014 on both expressions.

2015 **Definition 11** (Type annotation erasure). The erasure function is denoted as $|\cdot|$, and defined
 2016 as follows:

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

2018 **Theorem 4.15** (Instantiation Soundness). Given $\Delta \longrightarrow \Omega$ and $[\Gamma]\sigma = \sigma$ and $\hat{\alpha} \notin \text{FV}(\sigma)$:

- 2019 1. If $\Gamma \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta$ then $[\Omega]\Delta \vdash^G [\Omega]\hat{\alpha} \lesssim [\Omega]\sigma$.

2020 2. If $\Gamma \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta$ then $[\Omega]\Delta \vdash^G [\Omega]\sigma \lesssim [\Omega]\hat{\alpha}$.

2021 Notice that the declarative judgment uses $[\Omega]\Delta$, an operation that applies a complete con-
 2022 text Ω to the algorithmic context Δ , essentially plugging in all known solutions and removing
 2023 all declarations of existential variables (both solved and unsolved), resulting in a declarative
 2024 context.

2025 With instantiation soundness, next we show that the algorithmic consistent subtyping is
 2026 sound:

2027 **Theorem 4.16** (Soundness of Algorithmic Consistent Subtyping). *If $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$
 2028 where $[\Gamma]\sigma_1 = \sigma_1$ and $[\Gamma]\sigma_2 = \sigma_2$ and $\Delta \longrightarrow \Omega$ then $[\Omega]\Delta \vdash^G [\Omega]\sigma_1 \lesssim [\Omega]\sigma_2$.*

2029 Finally the soundness theorem of algorithmic typing is:

2030 **Theorem 4.17** (Soundness of Algorithmic Typing). *Given $\Delta \longrightarrow \Omega$:*

2031 1. If $\Gamma \vdash^G e \Rightarrow \sigma \dashv \Delta$ then $\exists e'$ such that $[\Omega]\Delta \vdash^G e' : [\Omega]\sigma$ and $|e| = |e'|$.

2032 2. If $\Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta$ then $\exists e'$ such that $[\Omega]\Delta \vdash^G e' : [\Omega]\sigma$ and $|e| = |e'|$.

2033 4.4.7 COMPLETENESS

2034 Completeness of the algorithmic system is the reverse of soundness: given a declarative judg-
 2035 ment of the form $[\Omega]\Gamma \vdash^G [\Omega] \dots$, we want to get an algorithmic derivation of $\Gamma \vdash^G \dots \dashv \Delta$.
 2036 It turns out that completeness is a bit trickier to state in that the algorithmic rules generate
 2037 existential variables on the fly, so Δ could contain unsolved existential variables that are not
 2038 found in Γ , nor in Ω . Therefore the completeness proof must produce another complete con-
 2039 text Ω' that extends both the output context Δ , and the given complete context Ω . As with
 2040 soundness, we need erasure to relate both expressions.

2041 **Theorem 4.18** (Instantiation Completeness). *Given $\Gamma \longrightarrow \Omega$ and $\sigma = [\Gamma]\sigma$ and $\hat{\alpha} \in$
 2042 $UNSOLVED(\Gamma)$ and $\hat{\alpha} \notin FV(\sigma)$:*

2043 1. If $[\Omega]\Gamma \vdash^G [\Omega]\hat{\alpha} \lesssim [\Omega]\sigma$ then there are Δ, Ω' such that $\Omega \longrightarrow \Omega'$ and $\Delta \longrightarrow \Omega'$ and
 2044 $\Gamma \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta$.

2045 2. If $[\Omega]\Gamma \vdash^G [\Omega]\sigma \lesssim [\Omega]\hat{\alpha}$ then there are Δ, Ω' such that $\Omega \longrightarrow \Omega'$ and $\Delta \longrightarrow \Omega'$ and
 2046 $\Gamma \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta$.

2047 Next is the completeness of consistent subtyping:

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

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

Theorem 4.20 (Matching Completeness). *Given $\Gamma \longrightarrow \Omega$ and $\Gamma \vdash^G \sigma$, if $[\Omega]\Gamma \vdash^G [\Omega]\sigma \triangleright \sigma_1 \rightarrow \sigma_2$ then there exist Δ , Ω' , σ'_1 and σ'_2 such that $\Gamma \vdash^G [\Gamma]\sigma \triangleright \sigma'_1 \rightarrow \sigma'_2 \dashv \Delta$ and $\Delta \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$ and $\sigma_1 = [\Omega']\sigma'_1$ and $\sigma_2 = [\Omega']\sigma'_2$.*

Finally here is the completeness theorem of the algorithmic typing:

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

4.5 SIMPLE EXTENSIONS AND VARIANTS

This section considers two simple variations 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, $\text{Int} \rightarrow ? \lesssim \top$.

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

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

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

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

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

SIEK AND TAHA'S DEFINITION OF CONSISTENT SUBTYPING DOES NOT WORK FOR \top . As with the analysis in Section 4.2.2, $\text{Int} \rightarrow ? \lesssim \top$ only holds when we first apply consistency, then subtyping. However we cannot find a type σ such that $\text{Int} \rightarrow ? <: \sigma$ and $\sigma \sim \top$. The following diagram depicts the situation:

$$\begin{array}{ccc} \emptyset & \xrightarrow{\sim} & \top \\ \uparrow & & \uparrow \\ <: & & <: \\ \text{Int} \rightarrow ? & \xrightarrow{\sim} & \text{Int} \rightarrow \text{Int} \end{array}$$

Additionally we have a similar problem in extending the restriction operator: *non-structural* masking between $\text{Int} \rightarrow ?$ 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.

2102 4.5.2 A MORE DECLARATIVE TYPE SYSTEM

2103 In Section 4.3 we present our declarative system in terms of the matching and consistent
 2104 subtyping judgments. The rationale behind this design choice is that the resulting declara-
 2105 tive system combines subtyping and type consistency in the application rule, thus making
 2106 it easier to design an algorithmic system accordingly. Still, one may wonder if it is possible
 2107 to design a more declarative specification. For example, even though we mentioned that the
 2108 subsumption rule is incompatible with consistent subtyping, it might be possible to accom-
 2109 modate a subsumption rule for normal subtyping (instead of consistent subtyping). In this
 2110 section, we discuss an alternative for the design of the declarative system.

2111 **WRONG DESIGN.** A naive design that does not work is to replace rule **GPC-APP** in Figure 4.7
 2112 with the following two rules:

$$\begin{array}{c}
 \text{GPC-V-SUB} \\
 \frac{\Psi \vdash^G e : \sigma \quad \sigma <: \sigma_2}{\Psi \vdash^G e : \sigma_2}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{GPC-V-APP1} \\
 \frac{\Psi \vdash^G e_1 : \sigma \quad \Psi \vdash^G e_2 : \sigma_1 \quad \sigma \sim \sigma_1 \rightarrow \sigma_2}{\Psi \vdash^G e_1 e_2 : \sigma_2}
 \end{array}$$

2113 Rule **GPC-V-SUB** is the standard subsumption rule: if an expression e has type σ , then it
 2114 can be assigned some type σ_2 that is a supertype of σ . Rule **GPC-V-APP1** first infers that e_1
 2115 has type σ , and e_2 has type σ_1 , then it finds some σ_2 so that σ is consistent with $\sigma_1 \rightarrow \sigma_2$.

2116 There would be two obvious benefits of this variant if it did work: firstly this approach
 2117 closely resembles the traditional declarative type systems for calculi with subtyping; secondly
 2118 it saves us from discussing various forms of σ in rule **GPC-V-APP1**, leaving the job to the
 2119 consistency judgment.

2120 The design is wrong because of the information loss caused by the choice of σ_2 in rule **GPC-**
 2121 **V-APP1**. Suppose we have $\Psi \vdash^G \text{plus} : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$, then we can apply it to 1 to get

$$\frac{\Psi \vdash^G \text{plus} : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad \Psi \vdash^G 1 : \text{Int} \quad \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \sim \text{Int} \rightarrow ? \rightarrow \text{Int}}{\Psi \vdash \text{plus } 1 : ? \rightarrow \text{Int}} \text{GPC-V-APP1}$$

2122 Further applying it to true we get

$$\frac{\Psi \vdash^G \text{plus } 1 : ? \rightarrow \text{Int} \quad \Psi \vdash^G \text{true} : \text{Bool} \quad ? \rightarrow \text{Int} \sim \text{Bool} \rightarrow \text{Int}}{\Psi \vdash \text{plus } 1 \text{ true} : \text{Int}} \text{GPC-V-APP1}$$

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:

$$\begin{array}{c}
 \text{GPC-V-SUB} \\
 \frac{\Psi \vdash^G e : \sigma \quad \sigma <: \sigma_2}{\Psi \vdash^G e : \sigma_2} \\
 \\
 \text{GPC-V-APP2} \\
 \frac{\Psi \vdash^G e : \sigma \quad \Psi \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \quad \Psi \vdash^G e_2 : \sigma_3 \quad \sigma_1 \sim \sigma_3}{\Psi \vdash^G e_1 e_2 : \sigma_2} \\
 \\
 \frac{}{\Psi \vdash^G \sigma_1 \rightarrow \sigma_2 \triangleright \sigma_1 \rightarrow \sigma_2} \qquad \frac{}{\Psi \vdash^G ? \triangleright ? \rightarrow ?}
 \end{array}$$

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 $\sigma_1 \rightarrow \sigma_2$. 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_m^G to indicate the more declarative version):

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

Lemma 4.25 (Soundness of \vdash_m^G). *If $\Psi \vdash_m^G e : \sigma_1$, 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	$\sigma ::= \text{Int} \mid a \mid \sigma_1 \rightarrow \sigma_2 \mid \forall a. \sigma \mid ? \mid \mathcal{S} \mid \mathcal{G}$
Monotypes	$\tau ::= \text{Int} \mid a \mid \tau_1 \rightarrow \tau_2 \mid \mathcal{S} \mid \mathcal{G}$
Castable Types	$\mathbb{C} ::= \text{Int} \mid a \mid \mathbb{C}_1 \rightarrow \mathbb{C}_2 \mid \forall a. \mathbb{C} \mid ? \mid \mathcal{G}$
Castable Monotypes	$t ::= \text{Int} \mid a \mid t_1 \rightarrow t_2 \mid \mathcal{G}$

$\Psi \vdash^G \sigma_1 \lesssim \sigma_2$

(Consistent Subtyping)

$\frac{\text{GPC-CS-TVAR} \quad a \in \Psi}{\Psi \vdash^G a \lesssim a}$	$\frac{\text{GPC-CS-INT}}{\Psi \vdash^G \text{Int} \lesssim \text{Int}}$	$\frac{\text{GPC-CS-ARROW} \quad \Psi \vdash^G \sigma_3 \lesssim \sigma_1 \quad \Psi \vdash^G \sigma_2 \lesssim \sigma_4}{\Psi \vdash^G \sigma_1 \rightarrow \sigma_2 \lesssim \sigma_3 \rightarrow \sigma_4}$
$\frac{\text{GPC-CS-FORALLR} \quad \Psi, a \vdash^G \sigma_1 \lesssim \sigma_2}{\Psi \vdash^G \sigma_1 \lesssim \forall a. \sigma_2}$	$\frac{\text{GPC-CS-FORALLL} \quad \Psi \vdash^G \tau \quad \Psi \vdash^G \sigma_1[a \mapsto \tau] \lesssim \sigma_2}{\Psi \vdash^G \forall a. \sigma_1 \lesssim \sigma_2}$	$\frac{\text{GPC-CS-UNKNOWNLL}}{\Psi \vdash^G ? \lesssim \mathbb{C}}$
$\frac{\text{GPC-CS-UNKNOWNRR}}{\Psi \vdash^G \mathbb{C} \lesssim ?}$	$\frac{\text{GPC-CS-SPAR}}{\Psi \vdash^G \mathcal{S} \lesssim \mathcal{S}}$	$\frac{\text{GPC-CS-GPAR}}{\Psi \vdash^G \mathcal{G} \lesssim \mathcal{G}}$

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

2173 sistent subtype of all castable types, rather than of all types (rule [GPC-CS-UNKNOWNLL](#)), and
 2174 vice versa (rule [GPC-CS-UNKNOWNRR](#)). Moreover, the static type parameter \mathcal{S} is a consistent
 2175 subtype of itself (rule [GPC-CS-SPAR](#)), and similarly for the gradual type parameter (rule [GPC-](#)
 2176 [CS-GPAR](#)). From this definition it follows immediately that $?$ is incomparable with types that
 2177 contain static type parameters \mathcal{S} , such as $\mathcal{S} \rightarrow \text{Int}$.

2178 **TYPING AND TRANSLATION.** Given these extensions to types and consistent subtyping, the
 2179 typing process remains the same as in Figure 4.7. To account for the changes in the transla-
 2180 tion, if we extend λB with type parameters as in Garcia and Cimini [2015], then the transla-
 2181 tion remains the same as well.

2182 5.2 SUBSTITUTIONS AND REPRESENTATIVE TRANSLATIONS

2183 As we mentioned, type parameters serve as placeholders for monotypes. As a consequence,
 2184 wherever a type parameter is used, any *suitable* monotype could appear just as well. To for-
 2185 malize this observation, we define substitutions for type parameters as follows:

2186 **Definition 12** (Substitution). Substitutions for type parameters are defined as:

- 2187 1. Let $S^S : \mathcal{S} \rightarrow \tau$ be a total function mapping static type parameters to monotypes.
- 2188 2. Let $S^G : \mathcal{G} \rightarrow t$ be a total function mapping gradual type parameters to castable
2189 monotypes.
- 2190 3. Let $S^P = S^G \cup S^S$ be a union of S^S and S^G mapping static and gradual type param-
2191 eters accordingly.

2192 Note that since \mathcal{G} might be compared with $?$, only castable monotypes are suitable substitutes,
2193 whereas \mathcal{S} can be replaced by any monotypes. Therefore, we can substitute \mathcal{G} for \mathcal{S} , but not
2194 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 $\text{Int} \rightarrow \text{Int}$ nor $\text{Bool} \rightarrow \text{Int}$ is general enough. With type parameters, however, we can state a more *general* translation that covers both through substitution:

$$f : \forall a. a \rightarrow \text{Int} \vdash^G (\lambda x : ?.fx) : ? \rightarrow \text{Int} \\ \rightsquigarrow (\lambda x : ?. (\forall a. a \rightarrow \text{Int} \hookrightarrow \mathcal{G} \rightarrow \text{Int})f)(\langle ? \hookrightarrow \mathcal{G} \rangle x)$$

2195 The advantage of type parameters is that they help reasoning about the dynamic semantics.
2196 Now we are not limited to a particular choice, such as $\text{Int} \rightarrow \text{Int}$ or $\text{Bool} \rightarrow \text{Int}$, which might
2197 or might not emit a cast error at runtime. Instead we have a general choice $\mathcal{G} \rightarrow \text{Int}$.

2198 What does the more general choice with type parameters tell us? First, we know that in
2199 this case, there is no concrete constraint on a , so we can instantiate it with a type parameter.
2200 Second, the fact that the general choice uses \mathcal{G} rather than \mathcal{S} indicates that any chosen
2201 instantiation needs to be a castable type. It follows that any concrete instantiation will have
2202 an impact on the runtime behavior; therefore it is best to instantiate a with $?$. However, type
2203 inference cannot instantiate a with $?$, and substitution cannot replace \mathcal{G} with $?$ either. This
2204 means that we need a syntactic refinement process of the translated programs in order to
2205 replace type parameters with allowed gradual types.

2206 **SYNTACTIC REFINEMENT.** We define syntactic refinement of the translated expressions as
2207 follows. As \mathcal{S} denotes no constraints at all, substituting it with any monotype would work.
2208 Here we arbitrarily use Int . We interpret \mathcal{G} as $?$ since any monotype could possibly lead to a
2209 cast error.

2210 **Definition 13** (Syntactic Refinement). The syntactic refinement of a translated expression s
2211 is denoted by $\lceil s \rceil$, and defined as follows:

	$\boxed{\text{Int}}$	=	Int
	\boxed{a}	=	a
	$\boxed{\sigma_1 \rightarrow \sigma_2}$	=	$\boxed{\sigma_1} \rightarrow \boxed{\sigma_2}$
2212	$\boxed{\forall a. \sigma}$	=	$\forall a. \boxed{\sigma}$
	$\boxed{?}$	=	$?$
	$\boxed{\mathcal{S}}$	=	Int
	$\boxed{\mathcal{G}}$	=	$?$

2213 Applying the syntactic refinement to the translated expression, we get

$$(\lambda x : ?. (\langle \forall a. a \rightarrow \text{Int} \hookrightarrow \boxed{?} \rightarrow \text{Int} \rangle f) (\langle ? \hookrightarrow \boxed{?} \rangle x))$$

2214 where two \mathcal{G} are refined by $?$ as highlighted. It is easy to verify that both applying this expres-
2215 sion to 3 and to *true* now results in a translation that evaluates to a value.

2216 **REPRESENTATIVE TRANSLATIONS.** To decide whether one translation is more general than
2217 the other, we define a preorder between translations.

2218 **Definition 14** (Translation Pre-order). Suppose $\Psi \vdash^G e : \sigma \rightsquigarrow \boxed{s_1}$ and $\Psi \vdash^G e : \sigma \rightsquigarrow \boxed{s_2}$,
2219 we define $s_1 \leq s_2$ to mean $s_2 \equiv_\alpha S^P(s_1)$ for some S^P .

2220 **Proposition 5.1.** If $s_1 \leq s_2$ and $s_2 \leq s_1$, then s_1 and s_2 are α -equivalent (i.e., equivalent up
2221 to renaming of type parameters).

2222 The preorder between translations gives rise to a notion of what we call *representative*
2223 *translations*:

2224 **Definition 15** (Representative Translation). A translation s is said to be a representative
2225 translation of a typing derivation $\Psi \vdash^G e : \sigma \rightsquigarrow \boxed{s}$ if and only if for any other translation
2226 $\Psi \vdash^G e : \sigma \rightsquigarrow \boxed{s'}$ such that $s' \leq s$, we have $s \leq s'$. From now on we use r to denote a
2227 representative translation.

2228 An important property of representative translations, which we conjecture for the lack
2229 of rigorous proof, is that if there exists any translation of an expression that (after syntac-
2230 tic refinement) can reduce to a value, so can a representative translation of that expression.
2231 Conversely, if a representative translation runs into a blame, then no translation of that ex-
2232 pression can reduce to a value.

2233 **Conjecture 5.2** (Property of Representative Translations). For any expression e such that
2234 $\Psi \vdash^G e : \sigma \rightsquigarrow \boxed{s}$ and $\Psi \vdash^G e : \sigma \rightsquigarrow \boxed{r}$ and $\forall C. C : (\Psi \vdash^B \sigma) \rightsquigarrow (\bullet \vdash^B \text{Int})$, we have

2235 • If $\mathcal{C}\{\lceil s \rceil\} \Downarrow n$, then $\mathcal{C}\{\lceil r \rceil\} \Downarrow n$.

2236 • If $\mathcal{C}\{\lceil r \rceil\} \Downarrow \text{blame}$, then $\mathcal{C}\{\lceil s \rceil\} \Downarrow \text{blame}$.

2237 Given this conjecture, we can state a stricter coherence property (without the “up to casts”
2238 part) between any two representative translations. We first strengthen Definition 8 following
2239 Ahmed et al. [2009]:

2240 **Definition 16** (Contextual Approximation à la Ahmed et al. [2009]).

$$\begin{aligned} \Psi \vdash s_1 \preceq_{ctx} s_2 : \sigma &\triangleq \Psi \vdash^B s_1 : \sigma \wedge \Psi \vdash^B s_2 : \sigma \wedge \\ &\text{for all } \mathcal{C}. \mathcal{C} : (\Psi \vdash^B \sigma) \rightsquigarrow (\bullet \vdash^B \text{Int}) \implies \\ 2241 &(\mathcal{C}\{\lceil s_1 \rceil\} \Downarrow n \implies \mathcal{C}\{\lceil s_2 \rceil\} \Downarrow n) \wedge \\ &(\mathcal{C}\{\lceil s_1 \rceil\} \Downarrow \text{blame} \implies \mathcal{C}\{\lceil s_2 \rceil\} \Downarrow \text{blame}) \end{aligned}$$

2242 The only difference is that now when a program containing s_1 reduces to a value, so does
2243 one containing s_2 .

2244 From Conjecture 5.2, it follows that coherence holds between two representative transla-
2245 tions of the same expression.

2246 **Corollary 5.3** (Coherence for Representative Translations). *For any expression e such that*
2247 *$\Psi \vdash^G e : \sigma \rightsquigarrow r_1$ and $\Psi \vdash^G e : \sigma \rightsquigarrow r_2$, we have $\Psi \vdash r_1 \preceq_{ctx} r_2 : \sigma$.*

2248 We have proved that for every typing derivation, at least one representative translation
2249 exists.

2250 **Lemma 5.4** (Representative Translation for Typing). *For any typing derivation $\Psi \vdash^G e : \sigma$*
2251 *there exists at least one representative translation r such that $\Psi \vdash^G e : \sigma \rightsquigarrow r$.*

2252 For our example, $(\lambda x : ?. (\forall a. a \rightarrow \text{Int} \hookrightarrow \mathcal{G} \rightarrow \text{Int})f) (\langle ? \hookrightarrow \mathcal{G} \rangle x)$ is a representative
2253 translation, while the other two are not.

2254 5.3 DYNAMIC GRADUAL GUARANTEE, RELOADED

2255 Given the above propositions, we are ready to revisit the dynamic gradual guarantee. The
2256 nice thing about representative translations is that the dynamic gradual guarantee of our
2257 source language is essentially that of λB , our target language. However, the dynamic gradual
2258 guarantee for λB is still an open question. According to Igarashi et al. [2017], the difficulty
2259 lies in the definition of term precision that preserves the semantics. We leave it here as a
2260 conjecture as well. From a declarative point of view, we cannot prevent the system from
2261 picking undesirable instantiations, but we know that some choices are better than the others,
2262 so we can restrict the discussion of dynamic gradual guarantee to representative translations.

Types	$\sigma ::= \text{Int} \mid a \mid \hat{\alpha} \mid \sigma_1 \rightarrow \sigma_2 \mid \forall a. \sigma \mid ? \mid \mathcal{S} \mid \mathcal{G}$
Monotypes	$\tau ::= \text{Int} \mid a \mid \hat{\alpha} \mid \tau_1 \rightarrow \tau_2 \mid \mathcal{S} \mid \mathcal{G}$
Existential variables	$\hat{\alpha} ::= \hat{\alpha}_S \mid \hat{\alpha}_G$
Castable Types	$\mathbb{C} ::= \text{Int} \mid a \mid \hat{\alpha} \mid \mathbb{C}_1 \rightarrow \mathbb{C}_2 \mid \forall a. \mathbb{C} \mid ? \mid \mathcal{G}$
Castable Monotypes	$t ::= \text{Int} \mid a \mid \hat{\alpha} \mid t_1 \rightarrow t_2 \mid \mathcal{G}$
Algorithmic Contexts	$\Gamma, \Delta, \Theta ::= \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \hat{\alpha} \mid \Gamma, \hat{\alpha}_S = \tau \mid \Gamma, \hat{\alpha}_G = t \mid \Gamma, \blacktriangleright_{\hat{\alpha}}$
Complete Contexts	$\Omega ::= \bullet \mid \Omega, x : \sigma \mid \Omega, a \mid \Omega, \hat{\alpha}_S = \tau \mid \Omega, \hat{\alpha}_G = t \mid \Omega, \blacktriangleright_{\hat{\alpha}}$

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

2263 **Conjecture 5.5** (Dynamic Gradual Guarantee in terms of Representative Translations). *Sup-*
 2264 *pose* $e' \sqsubseteq e$,

- 2265 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
 2266 $\sigma_2 \sqsubseteq \sigma$, and $\lceil r' \rceil \Downarrow v'$, and $v' \sqsubseteq v$.
- 2267 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
 2268 $\sigma_2 \sqsubseteq \sigma$. Moreover, $\lceil r \rceil \Downarrow v$ and $v' \sqsubseteq v$, or $\lceil r \rceil \Downarrow \text{blame}$.

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.

$$(\lambda f : \forall a. a \rightarrow \text{Int}. \lambda x : \text{Int}. fx) (\lambda x : \text{Int}. 1) 3 \quad (\lambda f : \forall a. a \rightarrow \text{Int}. \lambda x : \text{Int}. fx) (\lambda x : ?. 1) 3$$

2269 More importantly, in what follows, we show that our extended algorithm is able to find
 2270 those representative translations.

2271 5.4 EXTENDED ALGORITHMIC TYPE SYSTEM

2272 To understand the design choices involved in the new algorithmic system, we consider the
 2273 following algorithmic typing example:

$$f : \forall a. a \rightarrow \text{Int}, x : ? \vdash^G fx : \text{Int} \dashv f : \forall a. a \rightarrow \text{Int}, x : ?, \hat{\alpha}$$

2274 Compared with declarative typing, where we have many choices (e.g., $\text{Int} \rightarrow \text{Int}$, $\text{Bool} \rightarrow \text{Int}$,
 2275 and so on) to instantiate $\forall a. a \rightarrow \text{Int}$, the algorithm computes the instantiation $\hat{\alpha} \rightarrow \text{Int}$ with
 2276 $\hat{\alpha}$ unsolved in the output context. What can we know from the algorithmic typing? First we
 2277 know that, here $\hat{\alpha}$ is *not constrained* by the typing problem. Second, and more importantly,

2278 $\hat{\alpha}$ has been compared with an unknown type (when typing (fx)). Therefore, it is possible to
 2279 make a more refined distinction between different kinds of existential variables:

- 2280 • the first kind of existential variables are those that indeed have no constraints at all, as
 2281 they do not affect the dynamic semantics;
- 2282 • the second kind (as in this example) are those where the only constraint is that *the*
 2283 *variable was once compared with an unknown type* [Garcia and Cimini 2015].

2284 The syntax of types is shown in Figure 5.2. A notable difference, apart from the addition
 2285 of static and gradual parameters, is that we further split existential variables $\hat{\alpha}$ into static
 2286 existential variables $\hat{\alpha}_S$ and gradual existential variables $\hat{\alpha}_G$. Depending on whether an ex-
 2287 istential variable has been compared with $?$ or not, its solution space changes. More specifi-
 2288 cally, static existential variables can be solved to a monotype τ , whereas gradual existential
 2289 variables can only be solved to a castable monotype t , as can be seen in the changes of algo-
 2290 rithmic contexts and complete contexts. As a result, the typing result for the above example
 2291 now becomes

$$f : \forall a. a \rightarrow \text{Int}, x : ? \vdash^G fx : \text{Int} \dashv f : \forall a. a \rightarrow \text{Int}, x : ?, \hat{\alpha}_G$$

2292 since we can solve any unconstrained $\hat{\alpha}_G$ to \mathcal{G} , it is easy to verify that the resulting translation
 2293 is indeed a representative translation.

2294 Our extended algorithm is novel in the following aspects. We naturally extend the concept
 2295 of existential variables [Dunfield and Krishnaswami 2013] to deal with comparisons between
 2296 existential variables and unknown types. Unlike Garcia and Cimini [2015], where they use
 2297 an extra set to store types that have been compared with unknown types, our two kinds of
 2298 existential variables emphasize the type distinction better, and correspond more closely to
 2299 the two kinds of type parameters, as we can solve $\hat{\alpha}_S$ to \mathcal{S} and $\hat{\alpha}_G$ to \mathcal{G} .

2300 5.4.1 EXTENDED ALGORITHMIC CONSISTENT SUBTYPING

2301 While the changes in the syntax seem negligible, the addition of static and gradual type pa-
 2302 rameters changes the algorithmic judgments in a significant way. We first discuss the al-
 2303 gorithmic consistent subtyping, which is shown in Figure 5.3. For notational convenience,
 2304 when static and gradual existential variables have the same rule form, we compress them into
 2305 one rule. For example, rule **GPC-AS-EVAR** is really two rules $\Gamma[\hat{\alpha}_S] \vdash^G \hat{\alpha}_S \lesssim \hat{\alpha}_S \dashv \Gamma[\hat{\alpha}_S]$
 2306 and $\Gamma[\hat{\alpha}_G] \vdash^G \hat{\alpha}_G \lesssim \hat{\alpha}_G \dashv \Gamma[\hat{\alpha}_G]$; same for rules **GPC-AS-INSTL** and **GPC-AS-INSTR**.

$\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$

(Algorithmic Consistent Subtyping)

GPC-AS-TVAR $\frac{}{\Gamma[a] \vdash^G a \lesssim a \dashv \Gamma[a]}$	GPC-AS-INT $\frac{}{\Gamma \vdash^G \text{Int} \lesssim \text{Int} \dashv \Gamma}$	GPC-AS-EVAR $\frac{}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \hat{\alpha} \dashv \Gamma[\hat{\alpha}]}$		
<div style="background-color: #f0f0f0; padding: 5px; border: 1px solid #ccc;"> GPC-AS-SPAR $\frac{}{\Gamma \vdash^G \mathcal{S} \lesssim \mathcal{S} \dashv \Gamma}$ </div>	<div style="background-color: #f0f0f0; padding: 5px; border: 1px solid #ccc;"> GPC-AS-GPAR $\frac{}{\Gamma \vdash^G \mathcal{G} \lesssim \mathcal{G} \dashv \Gamma}$ </div>			
<div style="background-color: #f0f0f0; padding: 5px; border: 1px solid #ccc; display: inline-block; width: 100%;"> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%; text-align: center; padding: 5px;"> GPC-AS-UNKNOWNLL $\frac{}{\Gamma \vdash^G ? \lesssim \mathbb{C} \dashv \text{contaminate}(\Gamma, \mathbb{C})}$ </td> <td style="width: 50%; text-align: center; padding: 5px;"> GPC-AS-UNKNOWNRR $\frac{}{\Gamma \vdash^G \mathbb{C} \lesssim ? \dashv \text{contaminate}(\Gamma, \mathbb{C})}$ </td> </tr> </table> </div>			GPC-AS-UNKNOWNLL $\frac{}{\Gamma \vdash^G ? \lesssim \mathbb{C} \dashv \text{contaminate}(\Gamma, \mathbb{C})}$	GPC-AS-UNKNOWNRR $\frac{}{\Gamma \vdash^G \mathbb{C} \lesssim ? \dashv \text{contaminate}(\Gamma, \mathbb{C})}$
GPC-AS-UNKNOWNLL $\frac{}{\Gamma \vdash^G ? \lesssim \mathbb{C} \dashv \text{contaminate}(\Gamma, \mathbb{C})}$	GPC-AS-UNKNOWNRR $\frac{}{\Gamma \vdash^G \mathbb{C} \lesssim ? \dashv \text{contaminate}(\Gamma, \mathbb{C})}$			
GPC-AS-ARROW $\frac{\Gamma \vdash^G \sigma_3 \lesssim \sigma_1 \dashv \Theta \quad \Theta \vdash^G [\Theta]\sigma_2 \lesssim [\Theta]\sigma_4 \dashv \Delta}{\Gamma \vdash^G \sigma_1 \rightarrow \sigma_2 \lesssim \sigma_3 \rightarrow \sigma_4 \dashv \Delta}$	GPC-AS-FORALLR $\frac{\Gamma, a \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta, a, \Theta}{\Gamma \vdash^G \sigma_1 \lesssim \forall a. \sigma_2 \dashv \Delta}$			
<div style="background-color: #f0f0f0; padding: 5px; border: 1px solid #ccc;"> GPC-AS-FORALLL $\frac{\Gamma, \blacktriangleright_{\hat{a}_S}, \hat{\alpha}_S \vdash^G \sigma_1[a \mapsto \hat{\alpha}_S] \lesssim \sigma_2 \dashv \Delta, \blacktriangleright_{\hat{a}_S}, \Theta}{\Gamma \vdash^G \forall a. \sigma_1 \lesssim \sigma_2 \dashv \Delta}$ </div>				
GPC-AS-INSTL $\frac{\hat{\alpha} \notin \text{FV}(\sigma) \quad \Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta}$	GPC-AS-INSTR $\frac{\hat{\alpha} \notin \text{FV}(\sigma) \quad \Gamma[\hat{\alpha}] \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta}$			

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 $\text{contaminate}(\Gamma, \mathbb{C})$ 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 $\hat{\alpha}_S \vdash^G ? \lesssim \hat{\alpha}_S \rightarrow \hat{\alpha}_S \dashv \hat{\alpha}_S$, which seems fine. If this holds, then – since $\hat{\alpha}_S$ is unsolved in the output context – we can solve it to \mathcal{S} for example (recall that $\hat{\alpha}_S$ can be solved to some monotype), resulting in $? \lesssim \mathcal{S} \rightarrow \mathcal{S}$. However, this is in direct conflict with rule **GPC-CS-UNKNOWNLL** in the declarative system precisely because $\mathcal{S} \rightarrow \mathcal{S}$ 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 $\hat{\alpha}_S \vdash^G ? \lesssim \hat{\alpha}_S \rightarrow \hat{\alpha}_S \dashv \hat{\alpha}_S$ does not hold, $\hat{\alpha}_G \vdash^G ? \lesssim \hat{\alpha}_G \rightarrow \hat{\alpha}_G \dashv \hat{\alpha}_G$ 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 $\hat{\alpha}_S \vdash^G ? \lesssim \hat{\alpha}_S \rightarrow \hat{\alpha}_S \dashv \hat{\alpha}_G, \hat{\alpha}_S = \hat{\alpha}_G$.

Based on the above discussion, the following defines $\text{contaminate}(\Gamma, \sigma)$:

Definition 17. $\text{contaminate}(\Gamma, \sigma)$ is defined inductively as follows

$\text{contaminate}(\bullet, \sigma)$	$=$	\bullet
$\text{contaminate}((\Gamma, x : \sigma), \sigma)$	$=$	$\text{contaminate}(\Gamma, \sigma), x : \sigma$
$\text{contaminate}((\Gamma, a), \sigma)$	$=$	$\text{contaminate}(\Gamma, \sigma), a$
$\text{contaminate}((\Gamma, \hat{\alpha}_S), \sigma)$	$=$	$\text{contaminate}(\Gamma, \hat{\alpha}_G, \sigma), \hat{\alpha}_S = \hat{\alpha}_G$ if $\hat{\alpha}_S$ occurs in σ
$\text{contaminate}((\Gamma, \hat{\alpha}_S), \sigma)$	$=$	$\text{contaminate}(\Gamma, \sigma), \hat{\alpha}_S$ if $\hat{\alpha}_S$ does not occur in σ
$\text{contaminate}((\Gamma, \hat{\alpha}_G), \sigma)$	$=$	$\text{contaminate}(\Gamma, \sigma), \hat{\alpha}_G$
$\text{contaminate}((\Gamma, \hat{\alpha} = \tau), \sigma)$	$=$	$\text{contaminate}(\Gamma, \sigma), \hat{\alpha} = \tau$
$\text{contaminate}((\Gamma, \blacktriangleright_{\hat{\alpha}}), \sigma)$	$=$	$\text{contaminate}(\Gamma, \sigma), \blacktriangleright_{\hat{\alpha}}$

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

2333 Rule `GPC-AS-FORALLL` is slightly different from rule `GPC-AS-FORALL` in the original al-
 2334 gorithmic system in that we replace a with a new static existential variable $\hat{\alpha}_S$. Note that $\hat{\alpha}_S$
 2335 might be solved to a gradual existential variable later. The rest of the rules are the same as
 2336 those in the original system.

2337 5.4.2 EXTENDED INSTANTIATION

2338 The instantiation judgments shown in Figure 5.4 also change significantly. The complication
 2339 comes from the fact that now we have two different kinds of existential variables, and the
 2340 relative order that they appear in the context affects their solutions.

2341 Rules `GPC-INSTL-SOLVES` and `GPC-INSTL-SOLVEG` are the refinement to rule `GPC-INSTL-`
 2342 `SOLVE` in the original system. The next two rules deal with situations where one side is an
 2343 existential variable and the other side is an unknown type. Rule `GPC-INSTL-SOLVEUS` is a
 2344 special case of rule `GPC-AS-UNKNOWNRR` where we create a new gradual existential variable
 2345 $\hat{\alpha}_G$ and set the solution of $\hat{\alpha}_S$ to be $\hat{\alpha}_G$ in the output context. Rule `GPC-INSTL-SOLVEUG` is
 2346 the same as rule `GPC-INSTL-SOLVEU` in the original system and simply propagates the input
 2347 context. The next two rules `GPC-INSTL-REACHSG1` and `GPC-INSTL-REACHSG2` are a bit in-
 2348 volved, but they both answer to the same question: how to solve a gradual existential variable
 2349 when it is declared after some static existential variable. More concretely, in rule `GPC-INSTL-`
 2350 `REACHSG1`, we feel that we need to solve $\hat{\beta}_G$ to another existential variable. However, simply
 2351 setting $\hat{\beta}_G = \hat{\alpha}_S$ and leaving $\hat{\alpha}_S$ untouched in the output context is wrong. The reason is
 2352 that $\hat{\beta}_G$ could be a gradual existential variable created by rule `GPC-AS-UNKNOWNLL`/`GPC-AS-`
 2353 `UNKNOWNRR` and solving $\hat{\beta}_G$ to a static existential variable would result in the same problem
 2354 as we have discussed. Instead, we create another new gradual existential variable $\hat{\alpha}_G$ and set
 2355 the solutions of both $\hat{\alpha}_S$ and $\hat{\beta}_G$ to it; similarly in rule `GPC-INSTL-REACHSG2`. Rule `GPC-`
 2356 `INSTL-REACHOTHER` deals with the other cases (e.g., $\hat{\alpha}_S \lesssim \hat{\beta}_S$, $\hat{\alpha}_G \lesssim \hat{\beta}_G$ and so on). In
 2357 those cases, we employ the same strategy as in the original system.

2358 As for the other instantiation judgment, most of the rules are symmetric and thus omit-
 2359 ted. The only interesting rule is `GPC-ISTR-FORALLL`, which is similar to what we did for
 2360 rule `GPC-AS-FORALLL`.

2361 5.4.3 ALGORITHMIC TYPING AND METATHEORY

2362 Fortunately, the changes in the algorithmic bidirectional system are minimal: we replace
 2363 every existential variable with a static existential variable. Furthermore, we proved that the
 2364 extended algorithmic system is sound and complete with respect to the extended declarative
 2365 system. The full extended algorithmic system can be found in Appendix B.

$$\boxed{\Gamma \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta}$$

(Instantiation I)

$$\frac{\text{GPC-INSTL-SOLVE S} \quad \Gamma \vdash^G \tau}{\Gamma, \hat{\alpha}_S, \Gamma' \vdash^G \hat{\alpha}_S \lesssim \tau \dashv \Gamma, \hat{\alpha}_S = \tau, \Gamma'}$$

$$\frac{\text{GPC-INSTL-SOLVE G} \quad \Gamma \vdash^G t}{\Gamma, \hat{\alpha}_G, \Gamma' \vdash^G \hat{\alpha}_G \lesssim t \dashv \Gamma, \hat{\alpha}_G = t, \Gamma'}$$

$$\frac{\text{GPC-INSTL-SOLVE US}}{\Gamma[\hat{\alpha}_S] \vdash^G \hat{\alpha}_S \lesssim ? \dashv \Gamma[\hat{\alpha}_G, \hat{\alpha}_S = \hat{\alpha}_G]}$$

$$\frac{\text{GPC-INSTL-SOLVE UG}}{\Gamma[\hat{\alpha}_G] \vdash^G \hat{\alpha}_G \lesssim ? \dashv \Gamma[\hat{\alpha}_G]}$$

$$\frac{\text{GPC-INSTL-REACH S G1}}{\Gamma[\hat{\alpha}_S][\hat{\beta}_G] \vdash^G \hat{\alpha}_S \lesssim \hat{\beta}_G \dashv \Gamma[\hat{\alpha}_G, \hat{\alpha}_S = \hat{\alpha}_G][\hat{\beta}_G = \hat{\alpha}_G]}$$

$$\frac{\text{GPC-INSTL-REACH S G2}}{\Gamma[\hat{\beta}_S][\hat{\alpha}_G] \vdash^G \hat{\alpha}_G \lesssim \hat{\beta}_S \dashv \Gamma[\hat{\beta}_G, \hat{\beta}_S = \hat{\beta}_G][\hat{\alpha}_G = \hat{\beta}_G]}$$

$$\frac{\text{GPC-INSTL-REACH OTHER}}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash^G \hat{\alpha} \lesssim \hat{\beta} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]}$$

$$\frac{\text{GPC-INSTL-FORALL R} \quad \Gamma[\hat{\alpha}], b \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta, b, \Theta}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \forall b. \sigma \dashv \Delta}$$

$$\frac{\text{GPC-INSTL-ARR} \quad \Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \vdash^G \sigma_1 \lesssim \hat{\alpha}_1 \dashv \Theta \quad \Theta \vdash^G \hat{\alpha}_2 \lesssim [\Theta]\sigma_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \sigma_1 \rightarrow \sigma_2 \dashv \Delta}$$

$$\boxed{\Gamma \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta}$$

(Instantiation II, excerpt)

$$\frac{\text{GPC-INSTL-FORALL L} \quad \Gamma[\hat{\alpha}], \blacktriangleright_{\hat{b}_S}, \hat{\beta}_S \vdash^G \sigma[b \mapsto \hat{\beta}_S] \lesssim \hat{\alpha} \dashv \Delta, \blacktriangleright_{\hat{b}_S}, \Theta}{\Gamma[\hat{\alpha}] \vdash^G \forall b. \sigma \lesssim \hat{\alpha} \dashv \Delta}$$

Figure 5.4: Instantiation in the extended algorithmic system

2366 5.4.4 DISCUSSION

2367 DO WE REALLY NEED TYPE PARAMETERS IN THE ALGORITHMIC SYSTEM? As we mentioned
 2368 earlier, type parameters in the declarative system are merely an analysis tool, and in practice,
 2369 type parameters are inaccessible to programmers. For the sake of proving soundness and
 2370 completeness, we have to endow the algorithmic system with type parameters. However,
 2371 the algorithmic system already has static and gradual existential variables, which can serve
 2372 the same purpose. In that regard, we could directly solve every *unsolved* static and gradual
 2373 existential variable in the output context to `Int` and `?`, respectively.

2374 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 $(\lambda x : \text{Int}. x) 1$, the following shows two possible translations:

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

2375 The difference comes from the fact that in the second translation, we apply rule `GEN` while
 2376 typing `1` to get $\bullet \vdash 1 : \forall a. \text{Int}$. As a consequence, the translation of `1` is accompanied by a
 2377 cast from $\forall a. \text{Int}$ to `Int` since the former is a consistent subtype of the latter. This difference is
 2378 harmless, because obviously these two expressions will reduce to the same value in λB , thus
 2379 preserving coherence (up to cast error). While it is not going to break coherence, it does result
 2380 in multiple representative translations for one expression (e.g., the above two translations are
 2381 both the representative translations).

2382 There are several ways to make the translation process more deterministic. For example,
 2383 we can restrict generalization to happen only in `let` expressions and require `let` expressions
 2384 to include annotations, as `let $x : \sigma = e_1$ in e_2` . Another feasible option would be to give a
 2385 declarative, bidirectional system as the specification (instead of the type assignment one), in
 2386 the same spirit of DK [Dunfield and Krishnaswami 2013]. Then we can restrict generalization
 2387 to be performed through annotations in checking mode.

2388 With restricted generalization, we hypothesize that now each expression has exactly one
 2389 representative translation (up to renaming of fresh type parameters). Instead of calling it a

2390 *representative* translation, we can say it is a *principal* translation. Of course the above is only
2391 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 $\widehat{\beta} \rightarrow \text{Int}$ under the context $\widehat{\alpha}, \widehat{\beta}$. Here we cannot simply set $\widehat{\alpha}$ to $\widehat{\beta} \rightarrow \text{Int}$, 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 pro-

cess finishes when the existential variable being unified is found. In this case, Gundry et al. [2010] would return a solution context $\widehat{\beta}, \widehat{\alpha} = \widehat{\beta} \rightarrow \text{Int}$, 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 $\widehat{\alpha}, \widehat{\beta} \vdash^G \widehat{\alpha} \lesssim \widehat{\beta} \rightarrow \text{Int}$ in GPC:

$$\begin{array}{c}
 \Delta = \widehat{\alpha}_1, \widehat{\alpha}_2, \widehat{\alpha} = \widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2, \widehat{\beta} \\
 \\
 \frac{\text{GPC-INSTL-REACH} \quad \frac{\Delta \vdash^G \widehat{\beta} \lesssim \widehat{\alpha}_1 \vdash \Delta[\widehat{\beta} = \widehat{\alpha}_1]}{\widehat{\alpha}, \widehat{\beta} \vdash^G \widehat{\alpha} \lesssim \widehat{\beta} \rightarrow \text{Int} \vdash \Delta[\widehat{\alpha}_2 = \text{Int}][\widehat{\beta} = \widehat{\alpha}_1]} \quad \text{GPC-INSTL-SOLVE} \quad \frac{\Delta[\widehat{\beta} = \widehat{\alpha}_1] \vdash^G \widehat{\alpha}_2 \lesssim \text{Int} \vdash \Delta[\widehat{\alpha}_2 = \text{Int}][\widehat{\beta} = \widehat{\alpha}_1]}{\widehat{\alpha}, \widehat{\beta} \vdash^G \widehat{\alpha} \lesssim \widehat{\beta} \rightarrow \text{Int} \vdash \Delta[\widehat{\alpha}_2 = \text{Int}][\widehat{\beta} = \widehat{\alpha}_1]} \text{GPC-INSTL-ARR}
 \end{array}$$

By rule **GPC-INSTL-ARR**, the variable $\widehat{\alpha}$ is solved by an arrow type $\widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2$ 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-INSTL-REACH** to set $\widehat{\alpha}_1$ to $\widehat{\beta}$ instead. The final solution context is $\Delta[\widehat{\alpha}_2 = \text{Int}][\widehat{\beta} = \widehat{\alpha}_1] = \widehat{\alpha}_1, \widehat{\alpha}_2 = \text{Int}, \widehat{\alpha} = \widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2, \widehat{\beta} = \widehat{\alpha}_1$.

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 ($\Gamma \vdash^G \widehat{\alpha} \lesssim \sigma \vdash \Delta$ in Figure 4.11), and when it appears on the right

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

$$\begin{array}{c}
 \text{GPC-INSTL-ARR} \\
 \frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \vdash^G \sigma_1 \lesssim \hat{\alpha}_1 \dashv \Theta \quad \Theta \vdash^G \hat{\alpha}_2 \lesssim [\Theta]\sigma_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \sigma_1 \rightarrow \sigma_2 \dashv \Delta} \\
 \\
 \text{GPC-INSTR-ARR} \\
 \frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \vdash^G \hat{\alpha}_1 \lesssim \sigma_1 \dashv \Theta \quad \Theta \vdash^G [\Theta]\sigma_2 \lesssim \hat{\alpha}_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \sigma_1 \rightarrow \sigma_2 \lesssim \hat{\alpha} \dashv \Delta}
 \end{array}$$

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 $\hat{\alpha}, \hat{\beta}$, we want to instantiate $\hat{\alpha}$ with a dependent type $\Pi a : \hat{\beta}.a$. Here because $\hat{\beta}$ appears after $\hat{\alpha}$, we cannot directly set $\hat{\alpha} = \Pi a : \hat{\beta}.a$, which is ill-typed. However, if we try to decompose the type $\Pi a : \hat{\beta}.a$ like in rule **GPC-INSTL-ARR**, in which case we have $\hat{\alpha} = \Pi a : \hat{\alpha}_1.\hat{\alpha}_2$, it is obvious that $\hat{\alpha}_2$ should be solved by a . Then, in order to make the solution well typed, we need to put a in the front of $\hat{\alpha}_2$ 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 $\hat{\alpha}, \hat{\beta} \vdash^G \hat{\alpha} \lesssim \hat{\beta} \rightarrow \text{Int}$. The problem here is that $\hat{\beta}$ is out of the scope of $\hat{\alpha}$ so we cannot directly set $\hat{\alpha} = \hat{\beta} \rightarrow \text{Int}$. Therefore, we first *promote* the type $\hat{\beta} \rightarrow \text{Int}$. 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 $\hat{\alpha}$, such that existential variables in the promoted type are all in the scope of $\hat{\alpha}$. In this case, we will solve $\hat{\beta}$ with a fresh variable $\hat{\alpha}_1$, producing the context $\hat{\alpha}_1, \hat{\alpha}, \hat{\beta} = \hat{\alpha}_1$. Notice that $\hat{\alpha}_1$ is inserted right before $\hat{\alpha}$. Now the instantiation example becomes $\hat{\alpha}_1, \hat{\alpha}, \hat{\beta} = \hat{\alpha}_1 \vdash^G \hat{\alpha} \lesssim \hat{\alpha}_1 \rightarrow \text{Int}$, and $\hat{\alpha}_1 \rightarrow \text{Int}$ is a valid solution for

2474 $\hat{\alpha}$. Therefore, we get a final solution context $\hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \text{Int}, \hat{\beta} = \hat{\alpha}_1$. Comparing the
 2475 result with the solution context we get from DK ($\hat{\alpha}_1, \hat{\alpha}_2 = \text{Int}, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2, \hat{\beta} = \hat{\alpha}_1$), it is
 2476 obvious that these two solutions are equivalent up to substitution.

2477 **INTERPRETATION OF PROMOTION.** The approach taken by Gundry et al. [2010] and the ap-
 2478 proach used by DK are based on the same observation: *the relative order between existential*
 2479 *variables does not matter for solving a constraint*. The promotion process captures precisely
 2480 this observation. Its task is to “move” existential variables to suitable positions *indirectly*, by
 2481 solving those out-of-scope existential variables with fresh in-scope ones.

2482 This seems to go against the design principle that the contexts are ordered. However, or-
 2483 dering is still important for variables whose order matters. For instance, for polymorphic
 2484 types, the order between existential variables $\hat{\alpha}$ and type variables a is important, so we can-
 2485 not set $\hat{\alpha}$ to a under the context $(\hat{\alpha}, a)$ as a is not in the scope of $\hat{\alpha}$. Moreover, ordering
 2486 prevents invalid cyclic contexts, e.g., $\hat{\alpha} = \hat{\beta} \rightarrow \text{Int}, \hat{\beta} = \hat{\alpha} \rightarrow \text{Int}$.

2487 **UNIFICATION FOR THE SIMPLY TYPED LAMBDA CALCULUS.** As a first illustration of the pro-
 2488 motion process, Section 6.2 recasts the unification process for the simply typed lambda cal-
 2489 culus (STLC) using the promotion process. This system illustrates the key idea of promotion.

2490 6.1.3 POLYMORPHIC PROMOTION

2491 Instead of unification, the instantiation relation in DK actually deals with the polymorphic
 2492 subtyping relation between existential variables and other types. The promotion process we
 2493 described above only works for unification. In this section, we discuss promotion for poly-
 2494 morphic subtyping.

2495 The difficulty of subtyping is that it needs to take unification into account at the same time.
 2496 For example, given that $\hat{\alpha}$ is a subtype of Int , the only possible solution is $\hat{\alpha} = \text{Int}$. Now
 2497 consider $\hat{\alpha} \vdash \forall a. a \rightarrow a <: \hat{\alpha}$. How can we promote the polymorphic type $\forall a. a \rightarrow a$ into a
 2498 monotype which can serve as a valid solution for $\hat{\alpha}$? One possible answer is to set $\hat{\alpha} = \text{Int} \rightarrow$
 2499 Int , or $\hat{\alpha} = \text{Bool} \rightarrow \text{Bool}$. In fact, the most general solution for this subtyping problem is
 2500 $\hat{\alpha} = \hat{\beta} \rightarrow \hat{\beta}$ with fresh $\hat{\beta}$. Namely, we remove the universal quantifier in $\forall a. a \rightarrow a$ and
 2501 replace the variable a with a fresh existential variable $\hat{\beta}$ added to the front of $\hat{\alpha}$, resulting in
 2502 the solution context $\hat{\beta}, \hat{\alpha} = \hat{\beta} \rightarrow \hat{\beta}$.

2503 On the other hand, how can we promote the type $\forall a. a \rightarrow a$ in $\hat{\alpha} \vdash \hat{\alpha} <: \forall a. a \rightarrow a$?
 2504 It turns out that this subtyping is actually unsolvable, as there is no monotype that can be a
 2505 subtype of $\forall a. a \rightarrow a$. Therefore, in this case, promoting $\forall a. a \rightarrow a$ will directly add the
 2506 type variable a to the tail of the context to promote $a \rightarrow a$. Since a is added to the tail, it

means that a is out of the scope of $\hat{\alpha}$ and promoting $a \rightarrow a$ 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, $\forall a. \text{Int} \rightarrow \text{Int}$ can be promoted to $\text{Int} \rightarrow \text{Int}$, which is a valid solution for $\hat{\alpha}$ in $\hat{\alpha} \vdash \hat{\alpha} <: \forall a. \text{Int} \rightarrow \text{Int}$.

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 $\forall a. a \rightarrow a$ in the case of $\hat{\alpha} \vdash \forall a. a \rightarrow a <: \hat{\alpha}$, 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-INTR-FORALLL](#), except that with promotion, the new existential variable $\hat{\beta}$ (in rule [GPC-INTR-FORALLL](#)) will be added directly before $\hat{\alpha}$ and there is no need to create a marker or to discard the context after $\hat{\beta}$ anymore.

The covariant mode promotes types as $\forall a. a \rightarrow a$ in the case of $\hat{\alpha} \vdash \hat{\alpha} <: \forall a. a \rightarrow a$. In this case, promoting $\forall a. a \rightarrow a$ 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 $\hat{\alpha} <: \text{Int}$ and $\text{Int} <: \hat{\alpha}$, $\hat{\alpha} = \text{Int}$ 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, $\hat{\alpha} = (\hat{\beta} \rightarrow \hat{\beta}) \rightarrow (\text{Int} \rightarrow \text{Int})$ is a solution for $\hat{\alpha} \vdash \hat{\alpha} <: (\forall a. a \rightarrow a) \rightarrow (\forall a. \text{Int} \rightarrow \text{Int})$, where $(\forall a. a \rightarrow a) \rightarrow (\forall a. \text{Int} \rightarrow \text{Int})$ is promoted under the covariant mode, which means $\forall a. a \rightarrow a$ is promoted under the *contravariant* mode and $\forall a. \text{Int} \rightarrow \text{Int}$ 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.

2539 6.2 UNIFICATION FOR THE SIMPLY TYPED LAMBDA CALCULUS

2540 This section first introduces the simply typed lambda calculus, and then presents a unification
 2541 algorithm that uses the novel promotion mechanism.

2542 6.2.1 DECLARATIVE SYSTEM

2543 The definition of declarative types in STLC is given below. We have only monotypes τ , which
 2544 includes the integer type Int and function types $\tau_1 \rightarrow \tau_2$. In this section, we focus on the
 2545 unification process. Hence, we do not elaborate the details of expressions' syntax or typing
 2546 rules.

$$2547 \quad \text{Monotypes } \tau ::= \text{Int} \mid \tau_1 \rightarrow \tau_2$$

2548 6.2.2 ALGORITHMIC SYSTEM

2549 The syntax of the algorithmic system is given in Figure 6.1. Following DK [Dunfield and
 2550 Krishnaswami 2013] and GPC, algorithmic monotypes include existential type variables $\hat{\alpha}$.
 2551 Algorithmic contexts also contain declarations of existential type variables, either unsolved
 2552 ($\hat{\alpha}$) or solved ($\hat{\alpha} = \tau$). Complete contexts Ω contain only solved variables. We use the
 2553 judgment $\Gamma \vdash^{\text{wf}} \tau$ to indicate that all existential variables in τ are well-scoped. Its definition
 2554 is standard and thus omitted. We also use $\Gamma \longrightarrow \Delta$ for context extension, whose definition
 2555 is essentially a simplified version of the one in GPC (Section 4.4.5).

2556 UNIFICATION. Figure 6.1 defines the unification process. The judgment $\Gamma \Vdash \tau_1 \approx \tau_2 \dashv \Delta$
 2557 reads that under the input context Γ , unifying τ_1 with τ_2 results in the output context Δ .
 2558 Rule **U-REFL** is our base case, and rule **U-ARROW** unifies the components of the arrow types.
 2559 When unifying $\hat{\alpha} \approx \tau_1$ (rule **U-EVARL**), we cannot simply set $\hat{\alpha}$ to τ_1 , as τ_1 might include
 2560 variables bound to the right of $\hat{\alpha}$. Instead, we need to *promote* (\Vdash^{pr}) τ_1 . After promoting τ_1 to
 2561 τ_2 , we can directly set $\hat{\alpha} = \tau_2$. Rule **U-EVARR** is symmetric to rule **U-EVARL**. Note that when
 2562 unifying $\hat{\alpha} \approx \hat{\beta}$, either rule **U-EVARL** and rule **U-EVARR** could be tried; an implementation
 2563 can arbitrarily choose between them.

2564 PROMOTION. The promotion relation $\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta$ given at the bottom of Figure 6.1
 2565 reads that under the input context Δ , promoting type τ_1 yields type τ_2 , so that τ_2 is well-
 2566 formed in the prefix context of $\hat{\alpha}$, while retaining $[\Delta]\tau_1 = [\Delta]\tau_2$. At a high-level, \Vdash^{pr} looks
 2567 for free variables in τ_1 . Integers are always well-formed (rule **PR-INT**). Promoting a function
 2568 recursively promotes its components (rule **PR-ARROW**). Variables bound to the left of $\hat{\alpha}$ in Γ

Monotypes	$\tau ::= \text{Int} \mid \tau_1 \rightarrow \tau_2 \mid \hat{\alpha}$
Algorithmic Contexts	$\Gamma, \Delta, \Theta ::= \bullet \mid \Gamma, \hat{\alpha} \mid \Gamma, \hat{\alpha} = \tau$
Complete Contexts	$\Omega ::= \bullet \mid \Omega, \hat{\alpha} = \tau$

$\Gamma \vdash^u \tau_1 \approx \tau_2 \dashv \Delta$

(Unification)

U-REFL

$$\frac{}{\Gamma \vdash^u \tau \approx \tau \dashv \Gamma}$$

U-ARROW

$$\frac{\Gamma \vdash^u \tau_1 \approx \tau_3 \dashv \Theta \quad \Theta \vdash^u [\Theta]\tau_2 \approx [\Theta]\tau_4 \dashv \Delta}{\Gamma \vdash^u \tau_1 \rightarrow \tau_2 \approx \tau_3 \rightarrow \tau_4 \dashv \Delta}$$

U-EVARL

$$\frac{\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta[\hat{\alpha}]}{\Gamma \vdash^u \hat{\alpha} \approx \tau_1 \dashv \Delta[\hat{\alpha} = \tau_2]}$$

U-EVARR

$$\frac{\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta[\hat{\alpha}]}{\Gamma \vdash^u \tau_1 \approx \hat{\alpha} \dashv \Delta[\hat{\alpha} = \tau_2]}$$

$\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta$

(Promotion)

PR-INT

$$\frac{}{\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \text{Int} \rightsquigarrow \text{Int} \dashv \Gamma}$$

PR-ARROW

$$\frac{\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_3 \dashv \Theta \quad \Theta \vdash_{\hat{\alpha}}^{\text{pr}} [\Theta]\tau_2 \rightsquigarrow \tau_4 \dashv \Delta}{\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightarrow \tau_2 \rightsquigarrow \tau_3 \rightarrow \tau_4 \dashv \Delta}$$

PR-EVARL

$$\frac{}{\Gamma[\hat{\beta}][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \Gamma[\hat{\beta}][\hat{\alpha}]}$$

PR-EVARR

$$\frac{}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta}_1 \dashv \Gamma[\hat{\beta}_1, \hat{\alpha}][\hat{\beta} = \hat{\beta}_1]}$$

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 $\widehat{\alpha}, \widehat{\beta} \vdash^u \widehat{\alpha} \approx \widehat{\beta} \rightarrow \text{Int}$ discussed in Section 6.1.1.

$$\begin{array}{c}
 \text{PR-EVARL} \qquad \qquad \qquad \text{PR-INT} \\
 \hline
 \frac{\widehat{\alpha}, \widehat{\beta} \vdash_{\widehat{\alpha}}^{\text{pr}} \widehat{\beta} \rightsquigarrow \widehat{\alpha} \dashv \widehat{\alpha}_1, \widehat{\alpha}, \widehat{\beta} = \widehat{\alpha}_1 \quad \widehat{\alpha}_1, \widehat{\alpha}, \widehat{\beta} = \widehat{\alpha}_1 \vdash_{\widehat{\alpha}}^{\text{pr}} \text{Int} \rightsquigarrow \text{Int} \dashv \widehat{\alpha}_1, \widehat{\alpha}, \widehat{\beta} = \widehat{\alpha}_1}{\widehat{\alpha}, \widehat{\beta} \vdash_{\widehat{\alpha}}^{\text{pr}} \widehat{\beta} \rightarrow \text{Int} \rightsquigarrow \widehat{\alpha}_1 \rightarrow \text{Int} \dashv \widehat{\alpha}_1, \widehat{\alpha}, \widehat{\beta} = \widehat{\alpha}_1} \text{PR-ARROW} \\
 \hline
 \widehat{\alpha}, \widehat{\beta} \vdash^u \widehat{\alpha} \approx \widehat{\beta} \rightarrow \text{Int} \dashv \widehat{\alpha}_1, \widehat{\alpha} = \widehat{\alpha}_1 \rightarrow \text{Int}, \widehat{\beta} = \widehat{\alpha}_1 \quad \text{U-EVAL-R}
 \end{array}$$

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 $\Gamma \vdash_{\widehat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta$, then $[\Delta]\tau_1 = [\Delta]\tau_2$. Moreover, given $\Delta = \Delta_1, \widehat{\alpha}, \Delta_2$, we have $\Delta_1 \vdash^{\text{wf}} \tau_2$,*

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

Theorem 6.2 (Soundness of Unification). *If $\Gamma \vdash^u \tau_1 \approx \tau_2 \dashv \Delta$, 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 $\widehat{\alpha} \notin \text{FV}(\tau_1)$, or otherwise promotion would fail.

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

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

Theorem 6.4 (Completeness of Unification). *Given $\Gamma \longrightarrow \Omega$, and $\Gamma \vdash^{\text{wf}} \tau_1$, and $\Gamma \vdash^{\text{wf}} \tau_2$, 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 $\Gamma \vdash \tau_1 \approx \tau_2 \dashv \Delta$.*

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	$::=$	$\text{Int} \mid a \mid \sigma_1 \rightarrow \sigma_2 \mid \forall a. \sigma$
Monotypes	τ	$::=$	$\text{Int} \mid a \mid \tau_1 \rightarrow \tau_2$
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,

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

SUBTYPING. Figure 6.2 also includes the subtyping judgment $\Gamma \vdash^{\text{sub}} \sigma_1 <: \sigma_2 \dashv \Delta$, 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 $\hat{\alpha}$ and σ relies on the instantiation rules, which are duplicated for the case when $\hat{\alpha}$ is on the left and the case when $\hat{\alpha}$ is on the right. Here, instead

Types	$\sigma ::= \text{Int} \mid a \mid \sigma_1 \rightarrow \sigma_2 \mid \forall a. \sigma \mid \hat{\alpha}$
Monotypes	$\tau ::= \text{Int} \mid a \mid \hat{\alpha} \mid \tau_1 \rightarrow \tau_2$
Algorithmic Contexts	$\Gamma, \Delta, \Theta ::= \bullet \mid \Gamma, a \mid \Gamma, \hat{\alpha} \mid \Gamma, \hat{\alpha} = \tau$
Complete Contexts	$\Omega ::= \bullet \mid \Omega, a \mid \Omega, \hat{\alpha} = \tau$
Promotion Modes	$\otimes ::= + \mid -$

$\Gamma \vdash^{\text{sub}} \sigma_1 <: \sigma_2 \dashv \Delta$

(Subtyping)

<p>S-TVAR</p> $\frac{}{\Gamma[a] \vdash^{\text{sub}} a <: a \dashv \Gamma[a]}$	<p>S-INT</p> $\frac{}{\Gamma \vdash^{\text{sub}} \text{Int} <: \text{Int} \dashv \Gamma}$	<p>S-EVAR</p> $\frac{}{\Gamma[\hat{\alpha}] \vdash^{\text{sub}} \hat{\alpha} <: \hat{\alpha} \dashv \Gamma[\hat{\alpha}]}$
<p>S-ARROW</p> $\frac{\Gamma \vdash^{\text{sub}} \sigma_3 <: \sigma_1 \dashv \Theta \quad \Theta \vdash^{\text{sub}} [\Theta]\sigma_2 <: [\Theta]\sigma_4 \dashv \Delta}{\Gamma \vdash^{\text{sub}} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4 \dashv \Delta}$	<p>S-FORALLR</p> $\frac{\Gamma, a \vdash^{\text{sub}} \sigma_1 <: \sigma_2 \dashv \Delta, a, \Theta}{\Gamma \vdash^{\text{sub}} \sigma_1 <: \forall a. \sigma_2 \dashv \Delta}$	
<p>S-FORALLL</p> $\frac{\Gamma, \blacktriangleright_{\hat{\alpha}}, \hat{\alpha} \vdash^{\text{sub}} \sigma_1[a \mapsto \hat{\alpha}] <: \sigma_2 \dashv \Delta, \blacktriangleright_{\hat{\alpha}}, \Theta}{\Gamma \vdash^{\text{sub}} \forall a. \sigma_1 <: \sigma_2 \dashv \Delta}$	<p>S-INSTL</p> $\frac{\Gamma[\hat{\alpha}] \vdash_{\hat{\alpha}}^+ \sigma \rightsquigarrow \tau \dashv \Delta[\hat{\alpha}]}{\Gamma[\hat{\alpha}] \vdash^{\text{sub}} \hat{\alpha} <: \sigma \dashv \Delta[\hat{\alpha} = \tau]}$	
<p>S-INSTR</p> $\frac{\Gamma[\hat{\alpha}] \vdash_{\hat{\alpha}}^- \sigma \rightsquigarrow \tau \dashv \Delta[\hat{\alpha}]}{\Gamma[\hat{\alpha}] \vdash^{\text{sub}} \sigma <: \hat{\alpha} \dashv \Delta[\hat{\alpha} = \tau]}$		

$\Gamma \vdash_{\hat{\alpha}}^{\otimes} \sigma \rightsquigarrow \tau \dashv \Delta$

(Polymorphic Promotion)

<p>P-PR-FORALLL</p> $\frac{\Gamma[\hat{\beta}, \hat{\alpha}] \vdash_{\hat{\alpha}}^- \sigma[a \mapsto \hat{\beta}] \rightsquigarrow \tau \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash_{\hat{\alpha}}^- \forall a. \sigma \rightsquigarrow \tau \dashv \Delta}$	<p>P-PR-FORALLR</p> $\frac{\Gamma, a \vdash_{\hat{\alpha}}^+ \sigma \rightsquigarrow \tau \dashv \Delta, a}{\Gamma \vdash_{\hat{\alpha}}^+ \forall a. \sigma \rightsquigarrow \tau \dashv \Delta}$
<p>P-PR-ARROW</p> $\frac{\Gamma \vdash_{\hat{\alpha}}^{\otimes} \sigma_1 \rightsquigarrow \tau_1 \dashv \Theta \quad \Theta \vdash_{\hat{\alpha}}^{\otimes} [\Theta]\sigma_2 \rightsquigarrow \tau_2 \dashv \Delta}{\Gamma \vdash_{\hat{\alpha}}^{\otimes} \sigma_1 \rightarrow \sigma_2 \rightsquigarrow \tau_1 \rightarrow \tau_2 \dashv \Delta}$	<p>P-PR-MONO</p> $\frac{\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta}{\Gamma \vdash_{\hat{\alpha}}^{\otimes} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta}$

$\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta$

(Promotion)

PR-TVAR

$$\frac{}{\Gamma[a][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\text{pr}} a \rightsquigarrow a \dashv \Gamma[a][\hat{\alpha}]}$$

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 $\hat{\alpha}$ to τ .

POLYMORPHIC PROMOTION. The judgment $\Gamma \vdash_{\hat{\alpha}}^{\otimes} \sigma \rightsquigarrow \tau \dashv \Delta$ 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 $\hat{\alpha}$.

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 $\hat{\alpha}$ and put it before $\hat{\alpha}$. 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 $\hat{\alpha}$ and can actually never be used in σ or otherwise promotion would fail. This makes sense, as for a subtyping relation $\Gamma \vdash^{\text{sub}} \hat{\alpha} <: \forall a. \sigma$ to hold, a must not be used in σ . That means $\forall a. \sigma$ can only be types like $\forall a. \text{Int}$ or $\forall a. \text{Int} \rightarrow \text{Int}$, in which case $\hat{\alpha}$ can be promoted to Int or $\text{Int} \rightarrow \text{Int}$ respectively. In the conclusion of the rule, we discard a in the return context. Note that we can simplify the rule by directly requiring $a \notin \text{FV}(\sigma)$, 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.

$$\frac{\text{P-PR-FORALLRR} \quad a \notin \text{FV}(\sigma) \quad \Gamma \vdash_{\hat{\alpha}}^+ \sigma \rightsquigarrow \tau \dashv \Delta}{\Gamma \vdash_{\hat{\alpha}}^+ \forall a. \sigma \rightsquigarrow \tau \dashv \Delta}$$

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^{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 $\hat{\alpha}$, 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 $\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta$, and $\Delta = \Delta_1, \hat{\alpha}, \Delta_2$, then $\Delta_1 \vdash^{\text{wf}} \tau_2$, 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 $\hat{\alpha}$. 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 $\Gamma \vdash_{\hat{\alpha}}^{\otimes} \sigma \rightsquigarrow \tau \dashv \Delta$, and $\Delta = \Delta_1, \hat{\alpha}, \Delta_2$, then $\Delta_1 \vdash^{\text{wf}} \tau_2$. Moreover, 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 $\Gamma \vdash^{\text{sub}} \sigma_1 <: \sigma_2 \dashv \Delta$, 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 $\Gamma \vdash^{\text{wf}} \hat{\alpha}$, and $\Gamma \vdash^{\text{wf}} \tau$, and $[\Gamma]\hat{\alpha} = \hat{\alpha}$, and $[\Gamma]\tau = \tau$, if $\hat{\alpha} \notin \text{FV}(\tau)$, there exist τ_2, Δ and Ω' such that $\Gamma \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$ and $\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \tau \rightsquigarrow \tau_2 \dashv \Delta$.*

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 $\hat{\alpha} \notin \text{FV}(\sigma)$.

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

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

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 $\Gamma \vdash^{\text{wf}} \sigma_1$, and $\Gamma \vdash^{\text{wf}} \sigma_2$, 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 $\Gamma \vdash^{\text{sub}} [\Gamma]\sigma_1 <: [\Gamma]\sigma_2 \dashv \Delta$.*

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 $\Pi a : \tau_1. \tau_2$.

$$\frac{\text{PR-PI} \quad \Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_3 \dashv \Theta \quad \Theta, a \vdash_{\hat{\alpha}}^{\text{pr}} [\Theta]\tau_2 \rightsquigarrow \tau_4 \dashv \Delta, a}{\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \Pi a : \tau_1. \tau_2 \rightarrow \text{Int} \rightsquigarrow \Pi a : \tau_3. \tau_4 \dashv \Delta}$$

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 $\Pi a : \hat{\beta}. a$.

$$\frac{\frac{\text{PR-EVARL} \quad \widehat{\beta}, \hat{\alpha} \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \hat{\beta}, \hat{\alpha}}{\widehat{\beta}, \hat{\alpha} \vdash_{\hat{\alpha}}^{\text{pr}} \Pi a : \hat{\beta}. a \rightarrow \text{Int} \rightsquigarrow} \quad \frac{\text{PR-EVARL} \quad \widehat{\beta}, \hat{\alpha}, a \vdash_{\hat{\alpha}}^{\text{pr}} a \rightsquigarrow ???}{\widehat{\beta}, \hat{\alpha}, a \vdash_{\hat{\alpha}}^{\text{pr}} a \rightsquigarrow ???}}{\widehat{\beta}, \hat{\alpha} \vdash_{\hat{\alpha}}^{\text{pr}} \Pi a : \hat{\beta}. a \rightarrow \text{Int} \rightsquigarrow} \text{PR-PI}$$

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

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

$$\frac{\text{PR-TVARR}}{\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} a \rightsquigarrow a \dashv \Gamma}$$

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 $\hat{\alpha}$. Therefore, we need to adjust the rule of subtyping to check explicitly that the result is well-formed, i.e.,

$$\begin{array}{c}
 \text{S-INSTLL} \\
 \frac{\Gamma[\hat{\alpha}] \vdash_{\hat{\alpha}}^+ \sigma \rightsquigarrow \tau \dashv \Delta_1, \hat{\alpha}, \Delta_2 \quad \Delta_1 \vdash^{\text{wf}} \tau}{\Gamma[\hat{\alpha}] \vdash^{\text{sub}} \hat{\alpha} <: \sigma \dashv \Delta_1, \hat{\alpha} = \tau, \Delta_2} \\
 \\
 \text{S-INSTRR} \\
 \frac{\Gamma[\hat{\alpha}] \vdash_{\hat{\alpha}}^- \sigma \rightsquigarrow \tau \dashv \Delta_1, \hat{\alpha}, \Delta_2 \quad \Delta_1 \vdash^{\text{wf}} \tau}{\Gamma[\hat{\alpha}] \vdash^{\text{sub}} \sigma <: \hat{\alpha} \dashv \Delta_1, \hat{\alpha} = \tau, \Delta_2}
 \end{array}$$

2680 Xie and Oliveira [2017] include a more detailed discussion and formalization of applying
 2681 promotion to a dependently typed lambda calculus.

2682 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:

$$\begin{array}{c}
 \text{P-PR-UNKNOWN} \\
 \hline
 \Gamma[\hat{\alpha}] \vdash_{\hat{\alpha}}^{\otimes} ? \rightsquigarrow \hat{\beta} \dashv \Gamma[\hat{\beta}, \hat{\alpha}]
 \end{array}$$

2683 For example, we have $\hat{\alpha} \vdash_{\hat{\alpha}}^{\text{pr}} \text{Int} \rightarrow ? \rightsquigarrow \text{Int} \rightarrow \hat{\beta} \dashv \hat{\beta}, \hat{\alpha}$.

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

$$\begin{array}{c}
 \text{P-PR-UNKNOWNG} \\
 \hline
 \Gamma[\hat{\alpha}] \vdash_{\hat{\alpha}}^{\otimes} ? \rightsquigarrow \hat{\beta}_G \dashv \Gamma[\hat{\beta}_G, \hat{\alpha}]
 \end{array}$$

2684 With these rules it would be possible to apply polymorphic promotion to GPC. Note this
 2685 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

```
data Maybe a = Nothing | 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 Applnt f = Mk (f Int)
```

The type of constructor `Mk` applies the type variable `f` to an argument `Int`. Accordingly, `Applnt Bool` would not work, as the type `Bool Int` (in the instantiated type of `Mk`) is invalid. Instead, we must write something like `Applnt Maybe`: the argument to `Applnt` must be suitable for applying to `Int`. In Haskell98, `Applnt` has kind $(\star \rightarrow \star) \rightarrow \star$. 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 Haskell¹ 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 $(\star \rightarrow \star) \rightarrow \star \rightarrow \star$. Therefore `T` should be rejected, because in `MkT2` the datatype `App` is applied to `Fix :: (\star \rightarrow \star) \rightarrow \star` and `Maybe :: \star \rightarrow \star`, which do not match the expected kinds of `App`. However, with kind polymorphism, `T` is accepted, because `App` has the more general kind $\forall k. (k \rightarrow \star) \rightarrow k \rightarrow \star$. 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

¹We 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 ($k_1 \rightarrow k_2$). Kind inference for Haskell98 datatypes is thus closely related to type inference for the simply typed λ -calculus (STLC). For example, consider a term $+$ of type $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ and a type constructor $(+ :: \star \rightarrow \star \rightarrow \star)$. At the term level, we infer that $\text{add } a \ b = a + b$ yields $\text{add} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$. Similarly, we can create a datatype

```
data Add a b = Add (a : + : b)
```

and infer $\text{Add} :: \star \rightarrow \star \rightarrow \star$.

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 $k :: \text{Int} \rightarrow \text{Int}$ and $k :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$. Assuming that there are no type variables, the STLC accordingly has no *principal types*. An analogous datatype declaration is

```
data K a = K Int
```

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

2771 **DEFAULTING.** Definitions like k (in STLC) or K (in Haskell98) do not have a principal
 2772 type/kind, which raises the immediate question of what type/kind to infer. Haskell98 solves
 2773 this problem by using a *defaulting* strategy: *if the kind of a type variable cannot be inferred,*
 2774 *then it is defaulted to \star .* Therefore the kind of K in Haskell98 is $\star \rightarrow \star$. From the perspec-
 2775 tive of type inference, such defaulting strategy may seem somewhat ad-hoc, but due to the
 2776 role that \star plays at the type level it seems a defensible design for kind inference. Defaulting
 2777 brings complications in writing a declarative specification. We discuss this point further in
 2778 Section 7.4.3.

2779 7.2.2 KIND INFERENCE IN MODERN GHC HASKELL

2780 The type and kind languages for modern GHC are *unified* (i.e., types and kinds are indistin-
 2781 guishable), *dependently typed*, and the kind system includes the $\star :: \star$ axiom Cardelli [1986];
 2782 Weirich et al. [2013]. We informally use the word *type* or *kind* where we find it appropriate.
 2783 Unlike Haskell98’s datatypes, whose inference problem is quite closely related to the well-
 2784 studied inference problem for STLC, type inference for various features in modern Haskell
 2785 is not well-studied. While we are motivated concretely by Haskell, many of the challenges we
 2786 face would be present in any dependently typed language seeking principled type inference.
 2787 We use the term PolyKinds to refer to the fragment of modern Haskell that we model.² We
 2788 enumerate the key features of this fragment below.

2789 **KIND POLYMORPHISM AND DEPENDENT TYPES** Global type inference, in the style of Damas
 2790 and Milner [1982], allows polymorphic kinds to be assigned to datatype definitions. For
 2791 instance, reconsider

2792 `data K a = K Int`

2793 In PolyKinds, K can be given the kind $K :: \forall \{k\}. k \rightarrow \star$. This example shows one of the
 2794 interesting new features of PolyKinds over Haskell98: *kind polymorphism* [Yorgey et al. 2012].
 2795 The polymorphic kind is obtained via *generalization*, which is a standard feature in Damas-
 2796 Milner algorithms. Polymorphic types are helpful for recovering principal types, since they
 2797 generalize many otherwise incomparable monomorphic types.

2798 System-F-based languages do not have dependent types. In contrast, PolyKinds supports
 2799 dependent kinds such as

2800 `data D :: $\forall (k :: \star) (a :: k). K a \rightarrow \star$`

²Some 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:

```
data X :: ∀a (b :: ★ → ★). a b → ★      -- accepted
data Y :: ∀(c :: Maybe Bool). X c → ★    -- rejected
```

In X 's kind, we discover $a :: (\star \rightarrow \star) \rightarrow \star$. 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 $Maybe\ Bool$, which is c 's kind. It is tempting to solve this with $a \mapsto Maybe$ and $b \mapsto Bool$, 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 $\forall(k :: \star). k \rightarrow \star$.

```

2835   data P a = MkP (Q a)
2836   data Q a = MkQ (P a)

```

2837 The recursion is simple here: all recursive occurrences are at the same type. In existing
 2838 type-inference algorithms, such recursive definitions are well understood and do not bring
 2839 considerable complexity to type inference. However, we must also consider *polymorphic re-*
 2840 *cursion* as in *Poly*:

```

2841   data Poly :: ∀k. k → ★
2842   data Poly k = C1 (Poly Int) | C2 (Poly Maybe)

```

2843 This example includes a *kind signature*, meaning that we must *check* the kind of the datatype,
 2844 not *infer* it. In the definition of *Poly*, the type *Poly Int* requires an instantiation $k \mapsto \star$,
 2845 while the type *Poly Maybe* requires an instantiation of $k \mapsto (\star \rightarrow \star)$. These differing
 2846 instantiations mean that the declaration employs polymorphic recursion.

2847 PolyKinds deals with such cases of polymorphic recursion, which also appear at the term
 2848 level—for example, when writing recursive functions over GADTs or nested datatypes [Bird
 2849 and Meertens 1998]. Polymorphic recursion is known to render type-inference undecid-
 2850 able [Henglein 1993]. Furthermore, most existing formalizations of type inference avoid
 2851 the question entirely, either by not modeling recursion at all or not allowing polymorphic
 2852 recursion. Our PolyKinds system has full support for polymorphic recursion, implemented
 2853 directly without the use of a *fix* operator. Polymorphic recursion is allowed only on datatypes
 2854 with a kind signature; other datatypes are treated as monomorphic during inference.

2855 **VISIBLE KIND APPLICATION** PolyKinds lifts *visible type application* (VTA) [Eisenberg et al.
 2856 2016], whereby we can explicitly instantiate a function call, as in *id @Bool True*, to kinds,
 2857 giving us *visible kind application* (VKA). Following the design of VTA, we distinguish *spec-*
 2858 *ified variables* from *inferred variables*. As described by Eisenberg et al. [2016, Section 3.1],
 2859 only specified variables can be instantiated via VKA. Instantiation of variables is inferred
 2860 when no explicit kind application is given. To illustrate, consider

```

2861   data T :: ∀a b. a b → ★

```

2862 Here, *a* and *b* are specified variables. Because their order is given, explicit instantiation of
 2863 *a* must happen before *b*. For example, *T @Maybe* instantiates *a* to *Maybe*. On the other
 2864 hand, the kind of *a* and *b* can be generalized to $a :: k \rightarrow \star$ and $b :: k$. Elaborating the kind
 2865 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
 2866 not available for instantiation with VKA. This split between specified and inferred variables
 2867 supports predictable type inference: if the variables generated by the compiler (e.g., *k*) were
 2868 available for instantiation, then we have no way of knowing what order to instantiate them.

2869 OPEN KIND SIGNATURES AND GENERALIZATION ORDER Echoing the design of Haskell, Poly-
 2870 Kinds supports *open kind signatures*. We say a signature is *closed* if it contains no free vari-
 2871 ables, e.g.,

2872 $\text{data } T :: \forall a. a \rightarrow \star$

2873 Otherwise, it is *open*, e.g.,

2874 $\text{data } Q :: \forall (a :: (f\ b))\ (c :: k). f\ c \rightarrow \star$

2875 Free variables (in this case, f, b, k) will be generalized over. We have a decision to make: in
 2876 which order do we generalize the free variables? This question is non-trivial, as there can be
 2877 dependency between the variables. We infer $k :: \star, f :: k \rightarrow \star, b :: k$. Even though f and
 2878 b appear before k , their kinds end up depending on k and we must quantify k before f and
 2879 b . Inferring this order is a challenge: we cannot know the correct order before completing
 2880 inference. We thus introduce *local scopes*, which are sets of variables that may be reordered.
 2881 Since the ordering is not fixed by the programmer, these variables are considered *inferred*,
 2882 not *specified*, with respect to VKA.

2883 EXISTENTIAL QUANTIFICATION. PolyKinds supports existentially quantified variables on
 2884 datatype constructors. This is useful, for example, to model GADTs. Given

2885 $\text{data } T1 = \forall a. MkT1\ a$

2886 we get $MkT1 :: \forall (a :: \star). a \rightarrow T1$. The type of the data constructor declaration can also be
 2887 generalized. Given

2888 $\text{data } P1 :: \forall (a :: \star). \star$

2889 from $\text{data } T2 = MkT2\ P1$, we infer $MkT2 :: \forall \{a :: \star\}. P1\ @a \rightarrow T2$, where $P1$ is elaborated
 2890 to $P1\ @a$ with a generalized as an inferred variable.

2891 7.2.3 DESIRABLE PROPERTIES FOR KIND INFERENCE

2892 The goal of this work is to provide concrete, principled guidance to implementors of depen-
 2893 dently typed languages, such as GHC/Haskell. It is thus important to be able to describe our
 2894 inference algorithm as sound and complete against a *declarative specification*. This declar-
 2895 ative specification is what we might imagine a programmer to have in her head as she pro-
 2896 grams. This system should be designed with a minimum of low-level detail and a minimum
 2897 of surprises. It is then up to an algorithm to live up to the expectations set by the specification.

2898 The algorithm is sound when all programs it accepts are also accepted by the specification; it
 2899 is complete when all programs accepted by the specification are accepted by the algorithm.

2900 Why choose the particular set of features described here? Because they lead to interesting
 2901 kind inference challenges. We have found that the features above are sufficient in exploring
 2902 kind inference in modern Haskell. We consider unformalized extensions in Section 7.8.

2903 7.3 DATATYPES IN HASKELL98

2904 We begin our formal presentation with Haskell98. The fragment of the syntax of Haskell98
 2905 that concerns us appears at the top of Figure 7.1, including datatype declarations, types,
 2906 kinds, and contexts. The metavariable e refers to expressions, but we do not elaborate the
 2907 details of expressions' syntax or typing rules here. A program pgm is a sequence of groups
 2908 (defined below) of datatype declarations \mathcal{T} , followed by an expression e . We write $\tau_1 \rightarrow \tau_2$
 2909 as an abbreviation for $(\rightarrow)\tau_1 \tau_2$.

2910 7.3.1 GROUPS AND DEPENDENCY ANALYSIS

2911 Users are free to write declarations in any order: earlier declarations can depend on later
 2912 ones in the same compilation unit. However, any kind-checking algorithm must process
 2913 the declarations in dependency order. Complicating this is that type declarations may be
 2914 mutually recursive. A formal analysis of this dependency analysis is not enlightening, so
 2915 we consider it to be a preprocessing step that produces the grammar in Figure 7.1. This
 2916 dependency analysis breaks up the (unordered) raw input into mutually recursive groups
 2917 (potentially containing just one declaration), and puts these in dependency order. We use
 2918 the term *group* to describe a set of mutually recursive declarations.

2919 7.3.2 DECLARATIVE TYPING RULES

2920 The declarative typing rules are in Figure 7.1. There are no surprises here; we review these
 2921 rules briefly. The top judgment is $\Sigma; \Psi \vdash^{pgm} pgm : \sigma$. Its rule **PGM-DT** extends the input
 2922 type context Σ with kinds for the datatype declarations to form Σ' , which is used to check
 2923 both the datatype declarations and the rest of the program. In rule **PGM-DT**, we implicitly
 2924 extract the names \overline{T}^i from the declarations $\overline{\mathcal{T}}^i$ (and use this abuse of notation throughout
 2925 our work, relating T to \mathcal{T} and D to \mathcal{D}). The kinds are *guessed* for an entire group all at once:
 2926 they are added to the context *before* looking at the declarations. This is needed because the
 2927 declarations in the group refer to one another. Guessing the right answer is typical of declar-
 2928 ative type systems. The algorithmic system presented in Section 7.4 provides a mechanism

program	pgm	$::=$	$\mathbf{rec} \overline{\tau}_i^i; pgm \mid e$
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	$::=$	\dots
polytype	σ	$::=$	$\forall \overline{a}_i : \overline{\kappa}_i^i. \tau$
monotype	τ	$::=$	$\mathbf{Int} \mid a \mid T \mid \tau_1 \tau_2 \mid \rightarrow$
kind	κ	$::=$	$\star \mid \kappa_1 \rightarrow \kappa_2$
term context	Ψ	$::=$	$\bullet \mid \Psi, D : \sigma$
type context	Σ	$::=$	$\bullet \mid \Sigma, a : \kappa \mid \Sigma, T : \kappa$

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

$$\frac{\text{PGM-EXPR} \quad \Sigma; \Psi \vdash e : \sigma}{\Sigma; \Psi \vdash^{\text{pgm}} e : \sigma} \quad \frac{\text{PGM-DT} \quad \Sigma' = \Sigma, \overline{\tau}_i : \overline{\kappa}_i^i \quad \overline{\Sigma'} \vdash^{\text{dt}} \overline{\tau}_i \rightsquigarrow \overline{\Psi}_i^i \quad \Sigma'; \Psi, \overline{\Psi}_i^i \vdash^{\text{pgm}} pgm : \sigma}{\Sigma; \Psi \vdash^{\text{pgm}} \mathbf{rec} \overline{\tau}_i^i; pgm : \sigma}$$

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

$$\frac{\text{DT-DECL} \quad (T : \overline{\kappa}_i^i \rightarrow \star) \in \Sigma \quad \overline{\Sigma}, \overline{a}_i : \overline{\kappa}_i^i \vdash_{T \overline{a}_i^i}^{\text{dc}} \overline{\mathcal{D}}_j^j \rightsquigarrow \tau_j^j}{\Sigma \vdash^{\text{dt}} \mathbf{data} T \overline{a}_i^i = \overline{\mathcal{D}}_j^j \rightsquigarrow D_j : \overline{\forall \overline{a}_i : \overline{\kappa}_i^i. \tau_j^j}}$$

$$\boxed{\Sigma \vdash_{\tau}^{\text{dc}} \mathcal{D} \rightsquigarrow \tau'} \quad (\text{Typing Data Constructor Decl.})$$

$$\frac{\text{DC-DECL} \quad \Sigma \vdash^k \overline{\tau}_i^i \rightarrow \tau : \star}{\Sigma \vdash_{\tau}^{\text{dc}} D \overline{\tau}_i^i \rightsquigarrow \overline{\tau}_i^i \rightarrow \tau}$$

$$\boxed{\Sigma \vdash^k \tau : \kappa} \quad (\text{Kinding})$$

$$\begin{array}{llll} \text{K-VAR} & \text{K-TCON} & \text{K-NAT} & \text{K-ARROW} \\ \frac{(a : \kappa) \in \Sigma}{\Sigma \vdash^k a : \kappa} & \frac{(T : \kappa) \in \Sigma}{\Sigma \vdash^k T : \kappa} & \frac{}{\Sigma \vdash^k \mathbf{Int} : \star} & \frac{}{\Sigma \vdash \rightarrow : \star \rightarrow \star \rightarrow \star} \\ & \text{K-APP} & & \\ & \frac{\Sigma \vdash^k \tau_2 : \kappa_1 \quad \Sigma \vdash^k \tau_1 : \kappa_1 \rightarrow \kappa_2}{\Sigma \vdash^k \tau_1 \tau_2 : \kappa_2} & & \end{array}$$

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 $\Sigma \vdash^{\text{grp}} \text{rec } \overline{T}_i^i \rightsquigarrow \overline{\kappa}_i^i; \overline{\Psi}_i^i$ 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 $\Sigma \vdash^{\text{dt}} \mathcal{T} \rightsquigarrow \Psi$. 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^{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^{dc} judgment is the return type of the constructors, whose types are easily checked by rule **DC-DECL**. The kinding judgment $\Sigma \vdash^{\text{k}} \tau : \kappa$ 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^{dt}) each datatype definition (Section 7.4.4), producing the context Θ_{n+1} . Then we default (Sec-

kind	κ	$::=$	$\star \mid \kappa_1 \rightarrow \kappa_2 \mid \widehat{\alpha}$
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	$::=$	$\bullet \mid \Omega, a : \kappa \mid \Omega, T : \kappa \mid \Omega, \widehat{\alpha} = \kappa$

$$\boxed{\Omega; \Gamma \Vdash^{\text{pgm}} \text{pgm} : \sigma}$$

(Typing Program)

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

A-PGM-DT

$$\frac{\Theta_1 = \Omega, \widehat{\alpha}_i^i, \overline{T_i : \widehat{\alpha}_i^i}^i \quad \Theta_{n+1} \longrightarrow \Omega' \quad \Omega'; \Gamma, \overline{\Gamma_i^i} \Vdash^{\text{pgm}} \text{pgm} : \sigma}{\Omega; \Gamma \Vdash^{\text{pgm}} \text{rec } \overline{\mathcal{T}_i^{i \in 1..n}}; \text{pgm} : \sigma}$$

$$\boxed{\Delta \Vdash^{\text{dt}} \mathcal{T} \rightsquigarrow \Gamma \dashv \Theta}$$

(Typing Datatype Decl.)

A-DT-DECL

$$\frac{(T : \kappa) \in \Delta \quad \Delta, \widehat{\alpha}_i^i \Vdash^{\mu} [\Delta]\kappa \approx (\widehat{\alpha}_i^i \rightarrow \star) \dashv \Theta_1, \overline{\widehat{\alpha}_i = \kappa_i^i}^i}{\frac{\Theta_j, \overline{a_i : \kappa_i^i} \Vdash^{\text{dc}}_{T \overline{a_i^i}} \mathcal{D}_j \rightsquigarrow \tau_j \dashv \Theta_{j+1}, \overline{a_i : \kappa_i^i}^{i^j}}{\Delta \Vdash^{\text{dt}} \text{data } T \overline{a_i^i} = \overline{\mathcal{D}_j^{j \in 1..n}} \rightsquigarrow \overline{D_j : \forall a_i : \kappa_i^i. \tau_j^j} \dashv \Theta_{n+1}}}$$

$$\boxed{\Delta \Vdash_{\tau}^{\text{dc}} \mathcal{D} \rightsquigarrow \tau' \dashv \Theta}$$

(Typing Data Constructor Decl.)

A-DC-DECL

$$\frac{\Delta \Vdash^k \overline{\tau_i^i} \rightarrow \tau : \star \dashv \Theta}{\Delta \Vdash_{\tau}^{\text{dc}} \mathcal{D} \overline{\tau_i^i} \rightsquigarrow \overline{\tau_i^i} \rightarrow \tau \dashv \Theta}$$

Figure 7.2: Algorithmic program typing in Haskell98

tion 7.4.3) all unsolved unification variables with \star using $\Theta_{n+1} \longrightarrow \Omega$, 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^{pgm} is always a complete context. Echoing the notation for the declarative system, we write $\Omega \Vdash^{\text{grp}} \text{rec } \overline{\mathcal{T}}_i^i \rightsquigarrow \overline{\kappa}_i^i; \overline{\Gamma}_i^i \dashv \Theta$ 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). An algorithmic context Δ is defaulted to a complete context Ω , written $\Delta \longrightarrow \Omega$ by replacing all unsolved unification variables $\hat{\alpha}$ in Δ with $\hat{\alpha} = \star$.

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

```
data Q1 a = MkQ1 -- Q1 :: ( $\star \rightarrow \star$ )
data Q2 = MkQ2 (Q1 Maybe) -- rejected

data P1 a = MkP1 P2 -- P1 :: ( $\star \rightarrow \star$ )  $\rightarrow \star$ 
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 $\Theta_{n+1} \longrightarrow \Omega$ (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.

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

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 $\Delta \Vdash^{\text{dt}} \mathcal{T} \rightsquigarrow \Gamma \dashv \Theta$ 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 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 $\overline{\widehat{\alpha}}^i \rightarrow \star$. The rule then unifies κ with $\overline{\widehat{\alpha}}^i \rightarrow \star$. 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 $\Delta \Vdash^{\text{dc}} \mathcal{D} \rightsquigarrow \tau' \dashv \Theta$ type-checks a data constructor, by simply checking that the expected type $\overline{\tau}_i^i \rightarrow \tau$ is well-kinded.

7.4.5 KINDING

The algorithmic kinding $\Delta \Vdash^{\text{k}} \tau : \kappa \dashv \Theta$ 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 $\Delta \Vdash^{\text{kapp}} \kappa_1 \bullet \kappa_2 : \kappa \dashv \Theta$ 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

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

$$\begin{array}{c}
 \text{A-K-ARROW} \\
 \hline
 \Delta \Vdash^k \rightarrow : \star \rightarrow \star \rightarrow \star \dashv \Delta
 \end{array}
 \quad
 \begin{array}{c}
 \text{A-K-TCON} \\
 (T : \kappa) \in \Delta \\
 \hline
 \Delta \Vdash^k T : \kappa \dashv \Delta
 \end{array}
 \quad
 \begin{array}{c}
 \text{A-K-NAT} \\
 \hline
 \Delta \Vdash^k \text{Int} : \star \dashv \Delta
 \end{array}
 \quad
 \begin{array}{c}
 \text{A-K-VAR} \\
 (a : \kappa) \in \Delta \\
 \hline
 \Delta \Vdash^k a : \kappa \dashv \Delta
 \end{array}$$

$$\begin{array}{c}
 \text{A-K-APP} \\
 \Delta \Vdash^k \tau_1 : \kappa_1 \dashv \Theta_1 \quad \Theta_1 \Vdash^k \tau_2 : \kappa_2 \dashv \Theta_2 \quad \Theta_2 \Vdash^{\text{kapp}} [\Theta_2]\kappa_1 \bullet [\Theta_2]\kappa_2 : \kappa_3 \dashv \Theta \\
 \hline
 \Delta \Vdash^k \tau_1 \tau_2 : \kappa_3 \dashv \Theta
 \end{array}$$

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

$$\begin{array}{c}
 \text{A-KAPP-KUVAR} \\
 \Delta[\hat{\alpha}_1, \hat{\alpha}_2, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \Vdash^{\mu} \hat{\alpha}_1 \approx \kappa \dashv \Theta \\
 \hline
 \Delta[\hat{\alpha}] \Vdash^{\text{kapp}} \hat{\alpha} \bullet \kappa : \hat{\alpha}_2 \dashv \Theta
 \end{array}
 \quad
 \begin{array}{c}
 \text{A-KAPP-ARROW} \\
 \Delta \Vdash^{\mu} \kappa_1 \approx \kappa \dashv \Theta \\
 \hline
 \Delta \Vdash^{\text{kapp}} \kappa_1 \rightarrow \kappa_2 \bullet \kappa : \kappa_2 \dashv \Theta
 \end{array}$$

$$\boxed{\Delta \Vdash^{\mu} \kappa_1 \approx \kappa_2 \dashv \Theta} \quad (\text{Kind Unification})$$

$$\begin{array}{c}
 \text{A-U-REFL} \\
 \hline
 \Delta \Vdash^{\mu} \kappa \approx \kappa \dashv \Delta
 \end{array}
 \quad
 \begin{array}{c}
 \text{A-U-ARROW} \\
 \Delta \Vdash^{\mu} \kappa_1 \approx \kappa_3 \dashv \Theta_1 \quad \Theta_1 \Vdash^{\mu} [\Theta_1]\kappa_2 \approx [\Theta_1]\kappa_4 \dashv \Theta \\
 \hline
 \Delta \Vdash^{\mu} \kappa_1 \rightarrow \kappa_2 \approx \kappa_3 \rightarrow \kappa_4 \dashv \Theta
 \end{array}$$

$$\begin{array}{c}
 \text{A-U-KVARL} \\
 \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \kappa \rightsquigarrow \kappa_2 \dashv \Theta[\hat{\alpha}] \\
 \hline
 \Delta[\hat{\alpha}] \Vdash^{\mu} \hat{\alpha} \approx \kappa \dashv \Theta[\hat{\alpha} = \kappa_2]
 \end{array}
 \quad
 \begin{array}{c}
 \text{A-U-KVARR} \\
 \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \kappa \rightsquigarrow \kappa_2 \dashv \Theta[\hat{\alpha}] \\
 \hline
 \Delta[\hat{\alpha}] \Vdash^{\mu} \kappa \approx \hat{\alpha} \dashv \Theta[\hat{\alpha} = \kappa_2]
 \end{array}$$

$$\boxed{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \kappa_1 \rightsquigarrow \kappa_2 \dashv \Theta} \quad (\text{Promotion})$$

$$\begin{array}{c}
 \text{A-PR-STAR} \\
 \hline
 \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \star \rightsquigarrow \star \dashv \Delta
 \end{array}
 \quad
 \begin{array}{c}
 \text{A-PR-ARROW} \\
 \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \kappa_1 \rightsquigarrow \kappa_3 \dashv \Delta_1 \quad \Delta_1 \vdash_{\hat{\alpha}}^{\text{pr}} [\Delta_1]\kappa_2 \rightsquigarrow \kappa_4 \dashv \Theta \\
 \hline
 \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \kappa_1 \rightarrow \kappa_2 \rightsquigarrow \kappa_3 \rightarrow \kappa_4 \dashv \Theta
 \end{array}$$

$$\begin{array}{c}
 \text{A-PR-KUVARL} \\
 \Delta[\hat{\beta}][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \Delta[\hat{\beta}][\hat{\alpha}] \\
 \hline
 \Delta[\hat{\alpha}][\hat{\beta}] \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \Delta[\hat{\beta}_1, \hat{\alpha}][\hat{\beta} = \hat{\beta}_1]
 \end{array}
 \quad
 \begin{array}{c}
 \text{A-PR-KUVARR} \\
 \Delta[\hat{\alpha}][\hat{\beta}] \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta}_1 \dashv \Delta[\hat{\beta}_1, \hat{\alpha}][\hat{\beta} = \hat{\beta}_1] \\
 \hline
 \Delta[\hat{\alpha}][\hat{\beta}] \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta}_1 \dashv \Delta[\hat{\beta}_1, \hat{\alpha}][\hat{\beta} = \hat{\beta}_1]
 \end{array}$$

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

is a unification variable $\hat{\alpha}$ (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 $\hat{\alpha}$ using $\hat{\alpha}_1 \rightarrow \hat{\alpha}_2$, unify $\hat{\alpha}_1$ with the argument kind κ , and return $\hat{\alpha}_2$. Note that the unification variables $\hat{\alpha}_1$ and $\hat{\alpha}_2$ 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 $\kappa_1 \rightarrow \kappa_2$ (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 $\Delta \Vdash \kappa_1 \approx \kappa_2 \dashv \Theta$ is given in Figure 7.3. The elaborate style of this judgment (with the promotion process \Vdash^{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 $\hat{\alpha} \approx \kappa$ (rule [A-U-KVARL](#)), we cannot simply set $\hat{\alpha}$ to κ , as κ might include variables bound to the *right* of $\hat{\alpha}$. Instead, we need to *promote* (\Vdash^{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 $\hat{\alpha}$. We can then set $\hat{\alpha} = \kappa_2$ 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^{pr} is that *the relative order between unification variables does not matter for solving a constraint*. The promotion judgment $\Delta \Vdash_{\hat{\alpha}}^{\text{pr}} \kappa_1 \rightsquigarrow \kappa_2 \dashv \Theta$ 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 $\hat{\alpha}$, 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 $\hat{\beta}$ bound to the right of $\hat{\alpha}$ in Δ is replaced by a fresh variable introduced to $\hat{\alpha}$'s left. It is this promotion algorithm that guarantees that all the $\hat{\alpha}_i$ 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^{pgm}). *If Ω ok, and $\Omega \Vdash^{\text{ectx}} \Gamma$, and $\Omega; \Gamma \Vdash^{\text{pgm}} \text{pgm} : \sigma$, then $[\Omega]\Omega; [\Omega]\Gamma \Vdash^{\text{pgm}} \text{pgm} : \sigma$.*

This lemma statement refers to judgments Ω ok and $\Omega \Vdash^{\text{ectx}} \Gamma$; these basic well-formedness checks are standard. Because the declarative judgment \Vdash^{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^{grp}). *Given Ω ok, if $[\Omega]\Omega \Vdash^{\text{grp}} \mathbf{rec} \overline{\mathcal{T}}_i^i \rightsquigarrow \overline{\kappa}_i^i; \overline{\Psi}_i^i$, then there exists $\overline{\kappa}'_i, \overline{\Gamma}_i^i, \Theta$, and Ω' , such that $\Omega \Vdash^{\text{grp}} \mathbf{rec} \overline{\mathcal{T}}_i^i \rightsquigarrow \overline{\kappa}'_i; \overline{\Gamma}_i^i \dashv \Theta$, where $\Theta \longrightarrow \Omega'$, and $[\Omega']\overline{\kappa}'_i = \overline{\kappa}_i^i$, and $\overline{\Psi}_i^i = [\Omega']\overline{\Gamma}_i^i$.*

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^{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^{pgm} and \Vdash^{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.

3081 7.5.1 TYPE PARAMETERS

3082 Consider the datatype `data App f a = MkApp (f a)` again. The parameter `a` in this example
 3083 can be of any kind, including \star , $\star \rightarrow \star$, or others. To express this polymorphism without
 3084 introducing first-class polymorphism, we endow the declarative system with a set of *kind*
 3085 *parameters*. Importantly, kind parameters live only in our reasoning; users are not allowed
 3086 to write any kind parameters in the source. We amend the definition of kinds in Figure 7.1
 3087 as follows.

3088	kind parameter	P	\in	KPARAM
	kind	κ	$::=$	$\star \mid \kappa_1 \rightarrow \kappa_2 \mid \boxed{P}$

3089 Kind parameters are uninterpreted kinds: there is no special treatment of kind parameters
 3090 in the type system. Think of them as abstract, opaque kind constants. Kind parameters are
 3091 eliminated by substitutions S , which map kind parameters to kinds, and homomorphically
 3092 work on kinds themselves. For example, `App` can be assigned kind $(P \rightarrow \star) \rightarrow P \rightarrow \star$. By
 3093 substituting for P , we can get, for example, $(\star \rightarrow \star) \rightarrow \star \rightarrow \star$. Indeed, from $(P \rightarrow \star) \rightarrow$
 3094 $P \rightarrow \star$ we can get all other possible kinds of `App`. This leads to the definition of *principal*
 3095 *kinds* for a group; and to the property that for every well-formed group, there exists a list of
 3096 principal kinds.

3097 **Definition 19** (Principal Kind in Haskell98 with Kind Parameters). Given a context Σ , a
 3098 group $\text{rec } \overline{\mathcal{T}}_i^i$, and a list of kinds $\overline{\kappa}_i^i$, we say that the $\overline{\kappa}_i^i$ are *principal kinds* of Σ and $\text{rec } \overline{\mathcal{T}}_i^i$, de-
 3099 noted as $\Sigma \vdash \text{rec } \overline{\mathcal{T}}_i^i \leadsto^p \overline{\kappa}_i^i$, if $\Sigma \vdash^{\text{grp}} \text{rec } \overline{\mathcal{T}}_i^i \leadsto \overline{\kappa}_i^i; \overline{\Psi}_i^i$, and whenever $\Sigma \vdash^{\text{grp}} \text{rec } \overline{\mathcal{T}}_i^i \leadsto$
 3100 $\overline{\kappa}_i^i; \overline{\Psi}_i^i$ holds, there exists some substitution S , such that $S(\kappa_i) = \kappa_i^i$ and $S(\Psi_i) = \Psi_i^i$.

3101 **Theorem 7.3** (Principality of Haskell98 with Kind Parameters). If $\Sigma \vdash^{\text{grp}} \text{rec } \overline{\mathcal{T}}_i^i \leadsto \overline{\kappa}_i^i; \overline{\Psi}_i^i$,
 3102 then there exists some $\overline{\kappa}_i^i$ such that $\Sigma \vdash \text{rec } \overline{\mathcal{T}}_i^i \leadsto^p \overline{\kappa}_i^i$.

3103 7.5.2 PRINCIPAL KINDS AND DEFAULTING

3104 Using the notion of kind parameters, we can now incorporate defaulting into the declarative
 3105 specification of Haskell98. To this end, we define the defaulting kind parameter substitution
 3106 S^* :

3107 **Definition 20** (Defaulting Kind Parameter Substitution). Let $S^* \in \text{KPARAM} \rightarrow \kappa$ denote
 3108 the substitution that substitutes all kind parameters to \star .

Using S^* , 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 $S^*(\kappa_i)$.

$$\frac{\text{PGM-DTP} \quad \Sigma \vdash^{\text{grp}} \text{rec } \overline{T}_i^i \rightsquigarrow \overline{\kappa}_i^i; \overline{\Psi}_i^i \quad \Sigma \vdash \text{rec } \overline{T}_i^i \rightsquigarrow^{\text{p}} \overline{\kappa}_i^i \quad \Sigma, \overline{T}_i : S^*(\kappa_i)^i; \Psi, \overline{S^*(\Psi_i)}^i \vdash^{\text{pgm}} \text{pgm} : \sigma}{\Sigma; \Psi \vdash^{\text{pgm}} \text{rec } \overline{T}_i^i; \text{pgm} : \sigma}$$

7.5.3 COMPLETENESS

The two versions of defaulting (the one above and $\Delta \longrightarrow \Omega$ 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^{pgm} with Kind Parameters). *Given algorithmic contexts Ω , Γ , and a program pgm , if $[\Omega]\Omega; [\Omega]\Gamma \vdash^{\text{pgm}} \text{pgm} : \sigma$, then $\Omega; \Gamma \Vdash^{\text{pgm}} \text{pgm} : \sigma$.*

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 $\forall \{\phi^c\}. \mu$, 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	$::=$	$\mathbf{sig} \mathcal{S}; pgm \mid \mathbf{rec} \overline{\mathcal{T}}_i^i; pgm \mid e$
datatype signature	\mathcal{S}	$::=$	$\mathbf{data} T : \sigma$
datatype decl.	\mathcal{T}	$::=$	$\mathbf{data} T \overline{a}_i^i = \overline{\mathcal{D}}_j^j$
data constructor decl.	\mathcal{D}	$::=$	$\forall \phi. D \overline{\tau}_i^i$
type, kind	σ, K	$::=$	$\forall \phi. \sigma \mid \tau$
monotype, monokind	$\tau, \kappa, \rho, \omega$	$::=$	$\star \mid \mathbf{Int} \mid a \mid T \mid \tau_1 \tau_2 \mid \tau_1 @ \tau_2 \mid \rightarrow$
elaborated type, kind	μ, η	$::=$	$\forall \{\phi^c\}. \mu \mid \forall \phi^c. \mu \mid \rho$
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	$::=$	$\bullet \mid \phi^c, a : \rho$

Figure 7.4: Syntax of PolyKinds

- 3136 (i) If the signature/definition of T_1 mentions T_2 , then:
- 3137 a) if T_2 has a signature, the signature/definition of T_1 depends on the signature of
- 3138 T_2 ;
- 3139 b) otherwise, the signature/definition of T_1 depends on the definition of T_2 .

- 3140 (ii) A definition depends on its signature.

3141 To avoid a type that mentions itself in its own kind, we disallow self-dependency or mutual

3142 dependency involving signatures. For example, a group

3143 $\mathbf{data} \textcolor{violet}{T1} :: \textcolor{teal}{T2} \textcolor{teal}{a} \rightarrow \star$

3144 $\mathbf{data} \textcolor{teal}{T2} :: \textcolor{violet}{T1} \rightarrow \star$

3145 is rejected, lest $\textcolor{violet}{T1}$ be assigned type $\forall(a :: \textcolor{teal}{T1}). \textcolor{teal}{T2} \textcolor{teal}{a} \rightarrow \star$. In other words, signatures do not

3146 form groups: they are always processed individually. Moreover, the definition of a datatype

3147 which has a signature does not join others in a group, as according to Definition 21, there

3148 will be no dependency from datatypes on it. This simplifies the kinding procedure, as we will

3149 see in the coming section.

3150 The declarative typing rules appear in Figure 7.5. The judgment $\Sigma; \Psi \vdash^{pgm} pgm : \sigma$

3151 checks the program. From now on we omit the typing rule for expressions in programs,

3152 which is essentially the same as in Haskell98. Rule **PGM-SIG** processes kind signatures by

3153 elaborating and generalizing the kind, then adding it to the context Σ . The helper judgment

3154 $\Sigma \vdash^{sig} \mathcal{S} \rightsquigarrow T : \eta$ checks a kind signature $\mathbf{data} T : \sigma$. First, it uses $\lceil \sigma \rceil$ to ensure σ returns

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

$$\frac{\text{PGM-SIG} \quad \Sigma \vdash^{\text{sig}} \mathcal{S} \rightsquigarrow T : \eta \quad \Sigma, T : \eta; \Psi \vdash^{\text{pgm}} \text{pgm} : \mu}{\Sigma; \Psi \vdash^{\text{pgm}} \text{sig } \mathcal{S}; \text{pgm} : \mu}$$

$$\frac{\text{PGM-DT-TTS} \quad (T : \eta) \in \Sigma \quad \Sigma \vdash^{\text{dt}} \mathcal{T} \rightsquigarrow \Psi_1 \quad \Sigma; \Psi, \Psi_1 \vdash^{\text{pgm}} \text{pgm} : \mu}{\Sigma; \Psi \vdash^{\text{pgm}} \text{rec } \mathcal{T}; \text{pgm} : \mu}$$

$$\frac{\text{PGM-DT-TT} \quad \overline{\Sigma, \phi_i^c \vdash^{\text{ela}} \omega_i : \star}^i \quad \overline{\phi_i^c \in \mathcal{Q}(\omega_i)}^i \quad \overline{\Sigma, \cup \overline{\phi_i^c}^i, \overline{T_i : \omega_i}^i \vdash^{\text{dt}} \mathcal{T}_i \rightsquigarrow \Psi_i}^i \quad \overline{\Sigma, \cup \overline{\phi_i^c}^i, \overline{T_i : \omega_i}^i \vdash_{\phi_i^c}^{\text{gen}} \Psi_i \rightsquigarrow \Psi'_i}^i}{\Sigma, \overline{T_i : \forall \{\phi_i^c\}. \omega_i}^i; \Psi, \Psi'_i[\overline{T_i \mapsto T_i @ \phi_i^c}^i] \vdash^{\text{pgm}} \text{pgm} : \sigma} \quad \Sigma; \Psi \vdash^{\text{pgm}} \text{rec } \overline{\mathcal{T}_i}^i; \text{pgm} : \sigma$$

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

$$\frac{\text{SIG-TT} \quad \lceil \sigma \rceil \quad \phi \in \mathcal{Q}(\sigma) \quad \phi_1^c \in \mathcal{Q}(\forall \phi^c. \eta) \quad \Sigma, \phi_1^c \vdash^k \forall \phi. \sigma : \star \rightsquigarrow \forall \phi^c. \eta \quad |\phi| = |\phi^c|}{\Sigma \vdash^{\text{sig}} \text{data } T : \sigma \rightsquigarrow T : \forall \{\phi_1^c\}. \forall \{\phi^c\}. \eta}$$

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

$$\frac{\text{DT-TT} \quad (T : \forall \{\phi_1^c\}. \forall \phi_2^c. \overline{\omega_i}^i \rightarrow \star) \in \Sigma \quad \overline{\Sigma, \phi_1^c, \phi_2^c, \overline{a_i : \omega_i}^i \vdash_{(T @ \phi_1^c @ \phi_2^c \overline{a_i}^i)}^{\text{dc}} \mathcal{D}_j \rightsquigarrow \mu_j}^j}{\Sigma \vdash^{\text{dt}} \text{data } T \overline{a_i}^i = \overline{\mathcal{D}_j}^j \rightsquigarrow \overline{D_j : \forall \{\phi_1^c\}. \forall \phi_2^c. \forall \overline{a_i : \omega_i}^i. \mu_j}^j}$$

$$\boxed{\Sigma \vdash_{\rho}^{\text{dc}} \mathcal{D} \rightsquigarrow \mu} \quad (\text{Typing Data Constructor Decl.}) \quad \boxed{\Sigma \vdash_{\phi^c}^{\text{gen}} \Psi_1 \rightsquigarrow \Psi_2} \quad (\text{Generalization})$$

$$\frac{\text{DC-TT} \quad \phi^c \in \mathcal{Q}(\mu \setminus_{\Sigma, \overline{\tau_i}^i}) \quad \Sigma, \phi^c \vdash^k \forall \phi. \overline{\tau_i}^i \rightarrow \rho : \star \rightsquigarrow \mu}{\Sigma \vdash_{\rho}^{\text{dc}} \forall \phi. D \overline{\tau_i}^i \rightsquigarrow \forall \{\phi^c\}. \mu}$$

$$\frac{\text{GEN} \quad \overline{\phi^c, \phi_i^c \in \mathcal{Q}(\mu_i)}^i}{\Sigma \vdash_{\phi^c}^{\text{gen}} \overline{D_i : \mu_i}^i \rightsquigarrow \overline{D_i : \forall \{\phi^c, \phi_i^c\}. \mu_i}^i}$$

Figure 7.5: Declarative specification of PolyKinds

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

$$\frac{\text{INST-REFL}}{\Sigma \vdash^{\text{inst}} \mu : \omega <: \omega \rightsquigarrow \mu} \quad \frac{\text{INST-FORALL} \quad \Sigma \vdash^{\text{ela}} \rho : \omega_1 \quad \Sigma \vdash^{\text{inst}} \mu_1 @ \rho : \eta[a \mapsto \rho] <: \omega_2 \rightsquigarrow \mu_2}{\Sigma \vdash^{\text{inst}} \mu_1 : \forall a : \omega_1. \eta <: \omega_2 \rightsquigarrow \mu_2}$$

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

$$\frac{\text{KC-SUB} \quad \Sigma \vdash^{\text{k}} \sigma : \eta \rightsquigarrow \mu_1 \quad \Sigma \vdash^{\text{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2}{\Sigma \vdash^{\text{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu_2}$$

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

$$\frac{\text{KTT-STAR}}{\Sigma \vdash^{\text{k}} \star : \star \rightsquigarrow \star}$$

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

$$\frac{\text{KTT-KAPP} \quad \Sigma \vdash^{\text{k}} \kappa_1 : \forall a : \omega. \eta \rightsquigarrow \rho_1 \quad \Sigma \vdash^{\text{kc}} \kappa_2 \Leftarrow \omega \rightsquigarrow \rho_2}{\Sigma \vdash^{\text{k}} \kappa_1 @ \kappa_2 : \eta[a \mapsto \rho_2] \rightsquigarrow \rho_1 @ \rho_2}$$

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

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

$$\frac{\text{ELA-APP} \quad \Sigma \vdash^{\text{ela}} \rho_1 : \omega_1 \rightarrow \omega_2 \quad \Sigma \vdash^{\text{ela}} \rho_2 : \omega_1}{\Sigma \vdash^{\text{ela}} \rho_1 \rho_2 : \omega_2} \quad \frac{\text{ELA-KAPP} \quad \Sigma \vdash^{\text{ela}} \rho_1 : \forall a : \omega. \eta \quad \Sigma \vdash^{\text{ela}} \rho_2 : \omega}{\Sigma \vdash^{\text{ela}} \rho_1 @ \rho_2 : \eta[a \mapsto \rho_2]}$$

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

\star : $\lceil \sigma \rceil$ simply traverses over arrows and forall's, 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^k) the kind $\forall \phi. \sigma$, where ϕ and ϕ^c have the same length ($|\phi| = |\phi^c|$). As the elaborated result $\forall \phi^c. \eta$ can be further generalized, we bring the free variables $\phi_1^c \in \mathcal{Q}(\forall \phi^c. \eta)$ into scope when elaborating. The concluding output is $T : \forall \{\phi_1^c\}. \forall \{\phi^c\}. \eta$. As an example, consider a kind signature $\forall a. b \rightarrow \star$. We have $\phi = b$, $\phi^c = b : \star$, and $\phi_1^c = c : \star$, and the final kind is $\forall \{c : \star\}. \forall \{b : \star\}. \forall (a : c). b \rightarrow \star$. 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^{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^{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 $T_i @ \phi_i^c$, where T_i is applied to all the variables bound in ϕ_i^c .

The judgment of checking datatype declarations $\Sigma \vdash^{\text{dt}} \mathcal{T} \rightsquigarrow \Psi$ 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 $\forall \phi. \bar{\tau}_i^i \rightarrow \rho$ 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	$::=$	$\star \mid \text{Int} \mid a \mid T \mid \rho_1 \rho_2 \mid \rho_1 @ \rho_2 \mid \rightarrow \mid \hat{\alpha}$
term context	Γ	$::=$	$\bullet \mid \Gamma, D : \mu$
type context	Δ, Θ	$::=$	$\bullet \mid \Delta, a : \omega \mid \Delta, T : \eta$
			$\mid \Delta, \hat{\alpha} : \omega \mid \Delta, \hat{\alpha} : \omega = \rho \mid \Delta, \{\Delta'\} \mid \Delta, \blacktriangleright_D$
complete type context	Ω	$::=$	$\bullet \mid \Omega, a : \omega \mid \Omega, T : \eta \mid \Omega, \hat{\alpha} : \omega = \rho \mid \Omega, \{\Omega'\} \mid \Omega, \blacktriangleright_D$
kind binder list	$\hat{\phi}^c$	$::=$	$\bullet \mid \hat{\phi}^c, \hat{\alpha} : \kappa$

Figure 7.7: Algorithmic syntax in PolyKinds

3188 7.6.2 CHECKING KINDS

3189 The kinding judgment \vdash^k appears in Figure 7.6. We only highlight selected rules. Kinding
3190 $\Sigma \vdash^k \sigma : \eta \rightsquigarrow \mu$ infers the type σ to have kind η , and it elaborates σ to μ . The kinding rules
3191 are built upon the axiom $\Sigma \vdash^k \star : \star \rightsquigarrow \star$ (rule **KTT-STAR**). While this axiom is known to
3192 violate logical consistency, as Haskell is already logically inconsistent because of its general
3193 recursion, we do not consider it as an issue here. Rule **KTT-APP** concerns applications $\tau_1 \tau_2$.
3194 It first infers the kind of τ_1 to be η_1 . The kind η_1 can be a polymorphic kind headed by a
3195 \forall , though it is expected to be a function kind. Thus the rule uses \vdash^{inst} to instantiate η_1 to
3196 $\omega_1 \rightarrow \omega_2$. The instantiation judgment $\Sigma \vdash^{\text{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2$ instantiates a kind η to
3197 a monokind ω , where if μ_1 has kind η then μ_2 has kind ω . After instantiation, rule **KTT-**
3198 **APP** checks (\vdash^{kc}) the argument τ_2 against the expected argument kind ω_1 . The kind checking
3199 judgment \vdash^{kc} simply delegates the work to kinding and instantiation. Rule **KTT-KAPP** checks
3200 visible kind applications. Note in the return kind η , the variable a is substituted by the elab-
3201 orated argument ρ_2 . Rule **KTT-FORALLI** elaborates an unannotated type $\forall a. \sigma$ to $\forall a : \omega. \mu$,
3202 where ω is an *elaborated* kind (\vdash^{ela}) guessed for a .

3203 The stand-alone elaborated kinding judgment \vdash^{ela} type-checks elaborated types. As all
3204 necessary instantiation has been done, type-checking for elaborated types is easy. For ex-
3205 ample, rule **ELA-APP** concerns applications $\rho_1 \rho_2$. Compared to rule **KTT-APP**, here ρ_1 has an
3206 arrow kind, and takes exactly the kind of ρ_2 . All judgments output well-formed elaborated
3207 types, as the following lemma states:

3208 **Lemma 7.5** (Type Elaboration). *We have: 1. if $\Sigma \vdash^k \sigma : \eta \rightsquigarrow \mu$, then $\Sigma \vdash^{\text{ela}} \mu : \eta$; 2. if*
3209 *$\Sigma \vdash^{\text{kc}} \sigma \Leftarrow \eta \rightsquigarrow \mu$, then $\Sigma \vdash^{\text{ela}} \mu : \eta$; 3. if $\Sigma \vdash^{\text{ela}} \mu_1 : \eta$, and $\Sigma \vdash^{\text{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2$, then*
3210 *$\Sigma \vdash^{\text{ela}} \mu_2 : \omega$.*

3211 7.7 KIND INFERENCE FOR POLYKINDS

3212 We now describe the *algorithmic* counterpart of the PolyKinds system. Figure 7.7 presents the
 3213 syntax of kinds and contexts in the algorithmic system for PolyKinds. Elaborated monotypes
 3214 are extended with unification variables $\hat{\alpha}$. Echoing the algorithm for Haskell98, type contexts
 3215 are extended with unification variables, which now have kinds ($\hat{\alpha} : \omega$ and $\hat{\alpha} : \omega = \rho$). Also
 3216 added to contexts are local scopes $\{\Delta\}$. These are special type contexts, where *variables can*
 3217 *be reordered*. Recall the kind $\forall (a :: (f\ b))\ (c :: k). f\ c \rightarrow \star$ in Section 7.2.2, where f
 3218 and b appear before k , but end up depending on k . In which order should we put f , b and
 3219 k in the algorithmic context to kind-check the signature? We cannot have a correct order
 3220 before completing inference. Therefore, we put them into a local scope, knowing we can
 3221 reorder the variables during kind-checking according to the dependency information. The
 3222 well-formedness judgment for local scopes requires them to be well-scoped, leading to the
 3223 fact that $\Delta, \{\Delta'\}$ is well-formed iff Δ, Δ' is. The marker \blacktriangleright_D , subscripted by the name of a
 3224 data constructor, is used only in and explained with rule A-DC-TT.

3225 7.7.1 ALGORITHMIC PROGRAM TYPING

3226 The algorithmic typing rules appear in Figure 7.8. The judgment $\Omega; \Gamma \Vdash^{\text{pgm}} \text{pgm} : \mu$ checks
 3227 the program. The rule A-PGM-SIG and rule A-PGM-DT-TTS correspond directly to the declar-
 3228 ative rules. Note that as the datatype declaration in rule A-PGM-DT-TTS already has a sig-
 3229 nature, the output type context remains unchanged. Rule A-PGM-DT-TT concerns a group
 3230 (without kind signatures). Like in Haskell98, it first assigns a fresh unification variable $\hat{\alpha}_i : \star$
 3231 as the kind of each type constructor, and then type-checks each datatype declaration, yield-
 3232 ing the output context Θ_{n+1} . Unlike Haskell98 which then uses defaulting, here from each
 3233 $\hat{\alpha}_i$ we get their unsolved unification variables $\hat{\phi}_i^c$ and generalize the kind of each type con-
 3234 structor as well as the type of each data constructor. The **unsolved** (Δ) metafunction simply
 3235 extracts a set of free unification variables in Δ , with their kinds substituted by Δ . Before
 3236 generalization, we apply Θ_{n+1} to the results so all solved unification variables get substituted
 3237 away. We use the notation $\hat{\phi}_i^c \mapsto \phi_i^c$ to mean that all unification variables in $\hat{\phi}_i^c$ are replaced
 3238 by fresh type variables in ϕ_i^c . The algorithmic generalization judgment \Vdash^{gen} corresponds
 3239 straightforwardly to the declarative rule, and thus is omitted. Though they appear daunting,
 3240 the extended contexts used in the last premise to this rule are unsurprising: they just apply
 3241 the relevant substitutions (the solved unification variables in Θ_{n+1} , the replacement of uni-
 3242 fication variables with fresh proper type variables $\hat{\phi}_i^c \mapsto \phi_i^c$, and the generalization of the
 3243 kinds of the group of datatypes $T_i \mapsto T_i @ \phi_i^c$).

$$\boxed{\Omega; \Gamma \Vdash^{\text{pgm}} \text{pgm} : \mu}$$

(Typing Program)

A-PGM-SIG

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

A-PGM-DT-TTS

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

A-PGM-DT-TT

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

$$\boxed{\Omega \Vdash^{\text{sig}} \mathcal{S} \rightsquigarrow T : \eta}$$

(Typing Signature)

A-SIG-TT

$$\frac{\begin{array}{c} \lfloor \sigma \rfloor \quad \overline{a_i}^i = \text{fkV}(\sigma) \quad \Omega, \{\overline{\widehat{\alpha}_i : \star}, a_i : \widehat{\alpha}_i\} \Vdash^k \sigma : \star \rightsquigarrow \eta \dashv \Delta \\ \widehat{\phi}_1^c = \text{scoped_sort}(\overline{a_i : [\Delta]\widehat{\alpha}_i}^i) \quad \widehat{\phi}_2^c = \text{unsolved}(\Delta) \quad \Delta \hookrightarrow \overline{a_i}^i \end{array}}{\Omega \Vdash^{\text{sig}} \text{data } T : \sigma \rightsquigarrow T : \forall\{\phi_2^c\}.((\forall\{\phi_1^c\}.[\Delta]\eta)[\widehat{\phi}_2^c \mapsto \phi_2^c])}$$

$$\boxed{\Delta \Vdash^{\text{dt}} \mathcal{T} \rightsquigarrow \Gamma \dashv \Theta}$$

(Typing Datatype Decl.)

A-DT-TT

$$\frac{\begin{array}{c} (T : \forall\{\phi_1^c\}. \forall\phi_2^c. \omega) \in \Delta \\ \Delta, \phi_1^c, \phi_2^c, \overline{\widehat{\alpha}_i : \star}^i \Vdash^\mu [\Delta]\omega \approx (\overline{\widehat{\alpha}_i}^i \rightarrow \star) \dashv \Theta_1, \phi_1^c, \phi_2^c, \overline{\widehat{\alpha}_i : \star = \omega_i}^i \\ \Theta_j, \phi_1^c, \phi_2^c, \overline{a_i : \omega_i}^i \Vdash_{(T @ \phi_1^c @ \phi_2^c \overline{a_i}^i)}^{\text{dc}} \mathcal{D}_j \rightsquigarrow \mu_j \dashv \Theta_{j+1}, \phi_1^c, \phi_2^c, \overline{a_i : \omega_i}^i \end{array}}{\Delta \Vdash^{\text{dt}} \text{data } T \overline{a_i}^i = \overline{\mathcal{D}_j}^{j \in 1..n} \rightsquigarrow D_j : \forall\{\phi_1^c\}. \forall\phi_2^c. \forall \overline{a_i : \omega_i}^i. \mu_j^j \dashv \Theta_{n+1}}$$

$$\boxed{\Delta \Vdash_{\rho}^{\text{dc}} \mathcal{D} \rightsquigarrow \mu \dashv \Theta}$$

(Typing Data Constructor Decl.)

A-DC-TT

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

Figure 7.8: Algorithmic program typing in PolyKinds

The judgment $\Omega \Vdash^{\text{sig}} \mathcal{S} \rightsquigarrow T : \eta$ type-checks a signature definition. We get all free variables in σ using $\text{fkv}(\sigma)$ 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 $\widehat{\phi}_2^c$ in Δ . As $\widehat{\phi}_2^c$ 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:

```

data Proxy :: ∀k. k → ★
data Relate :: ∀a (b :: a). a → Proxy b → ★
data T :: ∀(a :: ★) (b :: a) (c :: a) d. Relate b d → ★

```

Proxy just gives us a way to write a type whose kind is not \star . The *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 $\widehat{\alpha} = \text{Proxy } \widehat{\beta}$, 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 $\text{fkv}(\text{unsolved}(\Delta)) \# \phi$, ensures that free variables in $\text{unsolved}(\Delta)$ are disjoint ($\#$) with ϕ , so that we can safely generalize $\text{unsolved}(\Delta)$ outside ϕ .³

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

$$\begin{array}{c} \text{A-INST-REFL} \\ \Delta \Vdash^{\text{u}} \omega_1 \approx \omega_2 \dashv \Theta \\ \hline \Delta \Vdash^{\text{inst}} \mu : \omega_1 <: \omega_2 \rightsquigarrow \mu \dashv \Theta \end{array} \quad \begin{array}{c} \text{A-INST-FORALL} \\ \Delta, \hat{\alpha} : \omega_1 \Vdash^{\text{inst}} \mu_1 @ \hat{\alpha} : \eta[a \mapsto \hat{\alpha}] <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta \\ \hline \Delta \Vdash^{\text{inst}} \mu_1 : \forall a : \omega_1. \eta <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta \end{array}$$

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

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

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

$$\text{A-KTT-STAR} \quad \frac{}{\Delta \Vdash^{\text{k}} \star : \star \rightsquigarrow \star \dashv \Delta}$$

$$\frac{\text{A-KTT-APP} \quad \Delta \Vdash^{\text{k}} \tau_1 : \eta_1 \rightsquigarrow \rho_1 \dashv \Delta_1 \quad \Delta_1 \Vdash^{\text{kapp}} (\rho_1 : [\Delta_1]\eta_1) \bullet \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\text{k}} \tau_1 \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}$$

$$\frac{\text{A-KTT-FORALLI} \quad \Delta, \hat{\alpha} : \star, a : \hat{\alpha} \Vdash^{\text{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu \dashv \Delta_2, a : \hat{\alpha}, \Delta_3 \quad \Delta_3 \hookrightarrow a}{\Delta \Vdash^{\text{k}} \forall a. \sigma : \star \rightsquigarrow \forall a : \hat{\alpha}. [\Delta_3]\mu \dashv \Delta_2, \text{unsolved}(\Delta_3)}$$

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

$$\frac{\text{A-KAPP-TT-ARROW} \quad \Delta \Vdash^{\text{kc}} \tau \Leftarrow \omega_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta \Vdash^{\text{kapp}} (\rho_1 : \omega_1 \rightarrow \omega_2) \bullet \tau : \omega_2 \rightsquigarrow \rho_1 \rho_2 \dashv \Theta}$$

$$\frac{\text{A-KAPP-TT-FORALL} \quad \Delta, \hat{\alpha} : \omega_1 \Vdash^{\text{kapp}} (\rho_1 @ \hat{\alpha} : \eta[a \mapsto \hat{\alpha}]) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\text{kapp}} (\rho_1 : \forall a : \omega_1. \eta) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}$$

$$\frac{\text{A-KAPP-TT-KUVAR} \quad \Delta_1, \hat{\alpha}_1 : \star, \hat{\alpha}_2 : \star, \hat{\alpha} : \omega = (\hat{\alpha}_1 \rightarrow \hat{\alpha}_2), \Delta_2 \Vdash^{\text{kc}} \tau \Leftarrow \hat{\alpha}_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta_1, \hat{\alpha} : \omega, \Delta_2 \Vdash^{\text{kapp}} (\rho_1 : \hat{\alpha}) \bullet \tau : \hat{\alpha}_2 \rightsquigarrow \rho_1 \rho_2 \dashv \Theta}$$

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

$$\begin{array}{c} \text{A-ELA-APP} \\ \Delta \Vdash^{\text{ela}} \rho_1 : \omega_1 \rightarrow \omega_2 \quad \Delta \Vdash^{\text{ela}} \rho_2 : \omega_1 \\ \hline \Delta \Vdash^{\text{ela}} \rho_1 \rho_2 : \omega_2 \end{array} \quad \begin{array}{c} \text{A-ELA-KAPP} \\ \Delta \Vdash^{\text{ela}} \rho_1 : \forall a : \omega. \eta \quad \Delta \Vdash^{\text{ela}} \rho_2 : \omega \\ \hline \Delta \Vdash^{\text{ela}} \rho_1 @ \rho_2 : \eta[a \mapsto [\Delta]\rho_2] \end{array}$$

Figure 7.9: Selected rules for algorithmic kinding in PolyKinds

3274 7.7.3 KINDING

3275 Figure 7.9 presents the selected rules for kinding judgment \Vdash^k , along with the auxiliary judg-
 3276 ments. Full rules can be found in Appendix C.3. Most rules correspond directly to their
 3277 declarative counterparts. For applications $\tau_1 \tau_2$, rule **A-KTT-APP** first synthesizes the kind of
 3278 τ_1 to be η_1 , then uses \Vdash^{kapp} to type-check τ_2 . The judgment $\Delta \Vdash^{kapp} (\rho_1 : \eta) \bullet \tau : \omega \rightsquigarrow \rho_2 \dashv \Theta$
 3279 is interpreted as, under context Δ , applying the type ρ_1 of kind η to the type τ returns kind
 3280 ω , the elaboration result ρ_2 , and an output context Θ . When η_1 is polymorphic (rule **A-KAPP-**
 3281 **TT-FORALL**), we instantiate it with a fresh unification variable. Rule **A-KTT-FORALLI** checks
 3282 a polymorphic type. We assign a unification variable as the kind of a , bring $\hat{\alpha} : \star, a : \hat{\alpha}$ into
 3283 scope to check the body against \star , yielding the output context $\Delta_2, a : \hat{\alpha}, \Delta_3$. As a goes out
 3284 of the scope in the conclusion, we need to drop a in the concluding context. To make sure
 3285 that dropping a outputs a well-formed context, we substitute away all solved unification vari-
 3286 ables in Δ_3 for the return kind, and keep only **unsolved** (Δ_3), which are ensured ($\Delta_3 \hookrightarrow a$)
 3287 to have no dependency on a .

3288 In the algorithmic elaborated kinding judgment $\Delta \Vdash^{ela} \mu : \eta$, we keep the invariant:
 3289 $[\Delta]\eta = \eta$. That is why in rule **A-ELA-APP** we substitute a with $[\Delta]\rho_2$.

3290 Instantiation (\Vdash^{inst}) contains the only entry to unification (rule **A-INST-REFL**).

3291 7.7.4 UNIFICATION

3292 The judgments of unification and promotion are excerpted in Figure 7.10. Most rules are
 3293 natural extensions of those in Haskell98.

3294 **PROMOTION** The promotion judgment $\Delta \vdash_{\hat{\alpha}}^{pr} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta$ is extended with kind an-
 3295 notations for unification variables. As our unification variables have kinds now, rule **A-PR-**
 3296 **KUVARR-TT** must also promote the kind of $\hat{\beta}$, so that $\hat{\beta}_1 : \rho_1$ in the context is well-formed.
 3297 Promotion now has a new failure mode: it cannot move proper quantified type variables. In
 3298 rule **A-PR-TVAR**, the variable a must be to the left of $\hat{\alpha}$.

3299 Unfortunately, now we cannot easily tell whether promoting is terminating. In particular,
 3300 the convergence of promotion in Haskell98 is built upon the obvious fact that the size of the
 3301 kind being promoted always gets smaller from the conclusion to the hypothesis. However,
 3302 rule **A-PR-KUVARR-TT** breaks this invariant, as the judgment recurs into the kinds of unifi-
 3303 cation variables, and the size of the kinds may be larger than the unification variables. As
 3304 shown in Section 7.7.5, we prove that promotion is terminating.

³See also the alternative design in Appendix C.2.9.

$\Delta \Vdash \omega_1 \approx \omega_2 \dashv \Theta$

(Unification)

$\frac{\text{A-U-REFL-TT}}{\Delta \Vdash \omega \approx \omega \dashv \Delta}$	$\frac{\text{A-U-APP} \quad \Delta \Vdash \rho_1 \approx \rho_3 \dashv \Delta_1 \quad \Delta_1 \Vdash [\Delta_1]\rho_2 \approx [\Delta_1]\rho_4 \dashv \Theta}{\Delta \Vdash \rho_1 \rho_2 \approx \rho_3 \rho_4 \dashv \Theta}$
$\frac{\text{A-U-KVARL-TT} \quad \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \hat{\alpha} : \omega_1, \Theta_2 \quad \Theta_1 \Vdash^{\text{ela}} \rho_2 : \omega_2 \quad \Theta_1 \Vdash [\Theta_1]\omega_1 \approx \omega_2 \dashv \Theta_3}{\Delta \Vdash \hat{\alpha} \approx \rho_1 \dashv \Theta_3, \hat{\alpha} : \omega_1 = \rho_2, \Theta_2}$	
$\frac{\text{A-U-KVARL-LO-TT} \quad \Delta_1, \Delta_2 \Vdash^{\text{mv}} \hat{\alpha} : \omega_1 \rightsquigarrow \Theta \quad \Delta[\{\Theta\}] \vdash_{\hat{\alpha}}^{\text{pr}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \{\Theta_2, \hat{\alpha} : \omega_1, \Theta_3\}, \Theta_4 \quad \Theta_1, \{\Theta_2\} \Vdash^{\text{ela}} \rho_2 : \omega_2 \quad \Theta_1, \{\Theta_2\} \Vdash [\Theta_1, \Theta_2]\omega_1 \approx \omega_2 \dashv \Theta_5, \{\Theta_6\}}{\Delta[\{\Delta_1, \hat{\alpha} : \omega_1, \Delta_2\}] \Vdash \hat{\alpha} \approx \rho_1 \dashv \Theta_5, \{\Theta_6, \hat{\alpha} : \omega_1 = \rho_2, \Theta_3\}, \Theta_4}$	

$\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta$

(Promotion)

$\frac{\text{A-PR-TVAR}}{\Delta[a][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\text{pr}} a \rightsquigarrow a \dashv \Delta[a][\hat{\alpha}]}$	$\frac{\text{A-PR-KUVARR-TT} \quad \Delta \vdash_{\hat{\alpha}}^{\text{pr}} [\Delta]\rho \rightsquigarrow \rho_1 \dashv \Theta[\hat{\alpha}][\hat{\beta} : \rho]}{\Delta[\hat{\alpha}][\hat{\beta} : \rho] \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta}_1 \dashv \Theta[\hat{\beta}_1 : \rho_1, \hat{\alpha}][\hat{\beta} : \rho = \hat{\beta}_1]}$
--	---

$\Delta_1 \Vdash^{\text{mv}} \Delta_2 \rightsquigarrow \Theta$

(Moving)

$\frac{\text{A-MV-EMPTY}}{\bullet \Vdash^{\text{mv}} \Delta \rightsquigarrow \Delta}$	$\frac{\text{A-MV-KUVAR} \quad \text{var}(\omega) \nmid \text{dom}(\Delta_2) \quad \Delta_1 \Vdash^{\text{mv}} \Delta_2 \rightsquigarrow \Theta}{\hat{\alpha} : \omega, \Delta_1 \Vdash^{\text{mv}} \Delta_2 \rightsquigarrow \hat{\alpha} : \omega, \Theta}$
$\frac{\text{A-MV-KUVARM} \quad \neg(\text{var}(\omega) \nmid \text{dom}(\Delta_2)) \quad \Delta_1 \Vdash^{\text{mv}} \Delta_2, \hat{\alpha} : \omega \rightsquigarrow \Theta}{\hat{\alpha} : \omega, \Delta_1 \Vdash^{\text{mv}} \Delta \rightsquigarrow \Theta}$	

Figure 7.10: Selected rules for unification, promotion, and moving in PolyKinds

UNIFICATION The unification judgment $\Delta \Vdash \omega_1 \approx \omega_2 \dashv \Theta$ for PolyKinds features *heterogeneous constraints*. Recall the definition of X and Y discussed in Section 7.2.2. When unifying $\hat{\alpha} \hat{\beta}$ with *Maybe Bool*, setting $\hat{\alpha} = \text{Maybe}$ and $\hat{\beta} = \text{Bool}$ 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 $\hat{\alpha}$ 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 $\hat{\alpha} : \omega_1 = \rho_2$. Under this rule, the unification $\hat{\alpha} \hat{\beta} \approx \text{Maybe Bool}$ 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* $\hat{\alpha}$ 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 $\{\hat{\alpha} : \star, a : \star, b : \hat{\alpha}, c : \star\} \vdash \hat{\alpha} \approx a$. We see that a is not well-formed under the context before $\hat{\alpha}$, and thus we cannot rewrite $\hat{\alpha} : \star$ with $\hat{\alpha} = a : \star$. However, we *can* reorder the context to put $\hat{\alpha}$ to the right of a . In fact, to maximize the prefix context of $\hat{\alpha}$, we can move $\hat{\alpha}$ to the end of the context, yielding $\{a : \star, c : \star, \hat{\alpha} : \star, b : \hat{\alpha}\}$. As b depends on $\hat{\alpha}$, b is also moved to the end of the context. The final context is now $\{a : \star, c : \star, \hat{\alpha} : \star = a, b : \hat{\alpha}\}$.

The *moving* judgment $\Delta_1 \dashv \Delta_2 \rightsquigarrow \Theta$ 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 $\hat{\alpha}$ 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 $\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta$, then $\langle \Delta \rangle = \langle \Theta \rangle$.*

3337 **Lemma 7.7** (Unification Makes Progress). *If $\Delta \Vdash \omega_1 \approx \omega_2 \dashv \Theta$, then either $\Theta = \Delta$, or*
 3338 *$\langle \Theta \rangle < \langle \Delta \rangle$.*

3339 Now we measure unification $\Delta \Vdash \omega_1 \approx \omega_2 \dashv \Theta$ using the lexicographic order of the
 3340 pair $(\langle \Delta \rangle, |\omega_1|)$, where $|\omega_1|$ computes the standard size of ω_1 . We prove the pair always gets
 3341 smaller from the conclusion to the hypothesis. Formally, assuming promotion terminates,
 3342 we have

3343 **Theorem 7.8** (Unification Terminates). *Given a context Δ ok, and kinds ρ_1 and ρ_2 , where*
 3344 *$[\Delta]\rho_1 = \rho_1$, and $[\Delta]\rho_2 = \rho_2$, it is decidable whether there exists Θ such that $\Delta \Vdash \rho_1 \approx \rho_2 \dashv$*
 3345 *Θ .*

3346 We are not yet done, since Theorem 7.8 depends on the convergence of promotion. As
 3347 observed in rule [A-PR-KUVR](#), the size of the type being promoted increases from the con-
 3348 clusion to the hypothesis. Worse, the context never decreases. How do we prove promotion
 3349 terminates? The crucial observation for rule [A-PR-KUVR](#) is that, when we move from the
 3350 conclusion to the hypothesis, we also move from a unification variable to its kind. Since
 3351 the kind is well-formed under the prefix context of the variable, we are somehow moving
 3352 leftward in the context.

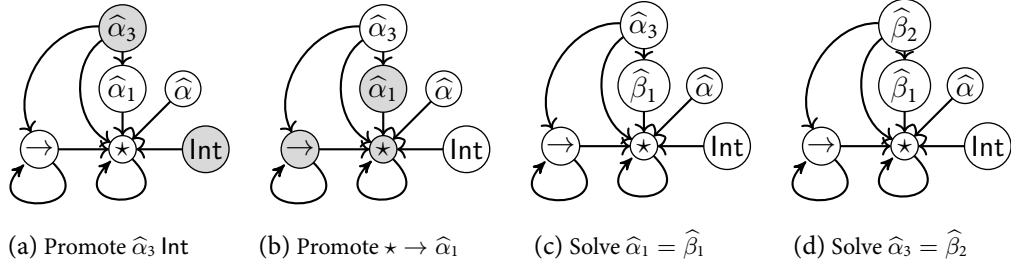
3353 To formalize the observation, we define the *dependency graph* of a context.

3354 **Definition 22** (Dependency Graph). The dependency graph of a context Δ is a *directed* graph
 3355 where:

- 3356 1. Nodes are all type variables and unsolved unification variables of Δ , and the terminal
 3357 symbols \star , \rightarrow and Int .
- 3358 2. Edges indicate the dependency from a type to its substituted kind. For example, if
 3359 $\hat{\alpha} : \omega$, then there is a directed edge from $\hat{\alpha}$ to all the nodes appearing in $[\Delta]\omega$.

3360 As an illustration, consider the context $\Delta = \hat{\alpha} : \star, \hat{\alpha}_1 : \star, \hat{\alpha}_2 : \star = \hat{\alpha}_1, \hat{\alpha}_3 : \star \rightarrow \hat{\alpha}_2$,
 3361 whose dependency graph is given in Figure 7.11a (the reader is advised to ignore the color
 3362 for now). There are several notable properties. First, as long as the context is well-formed,
 3363 the graph is *acyclic* except for the self-loop of \star and \rightarrow . Second, solved unification variables
 3364 never appear in the graph. The kind of $\hat{\alpha}_3$ depends on $\hat{\alpha}_2$, which is already solved by $\hat{\alpha}_1$, so
 3365 the dependency goes from $\hat{\alpha}_3$ to $\hat{\alpha}_1$.

3366 Now let us consider how promotion works in terms of the dependency graph, by trying
 3367 to unify $\Delta \vdash \hat{\alpha} \approx \hat{\alpha}_3 \text{Int}$. We start by promoting $\hat{\alpha}_3 \text{Int}$. The derivation of the promotion is
 3368 given at the bottom of Figure 7.11. We omit some details via (\dots) as promoting constants
 3369 $(\star, \rightarrow$ and $\text{Int})$ is trivial. At the top of Figure 7.11 we give the dependency graph at certain



$$\begin{aligned}
 \Theta_1 &= \hat{\beta}_1 : \star, \hat{\alpha} : \star, \hat{\alpha}_1 : \star = \hat{\beta}_1, \hat{\alpha}_2 : \star = \hat{\alpha}_1, \hat{\alpha}_3 : \star \rightarrow \hat{\alpha}_2 \\
 \Theta_2 &= \hat{\beta}_1 : \star, \hat{\beta}_2 : \star \rightarrow \hat{\beta}_1, \hat{\alpha} : \star, \hat{\alpha}_1 : \star = \hat{\beta}_1, \hat{\alpha}_2 : \star = \hat{\alpha}_1, \hat{\alpha}_3 : \star \rightarrow \hat{\alpha}_2 = \hat{\beta}_2 \\
 &\dots \frac{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \star \rightsquigarrow \star \dashv \Delta}{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\alpha}_1 \rightsquigarrow \hat{\beta}_1 \dashv \Theta_1} \text{A-PR-KUVARR} \quad \boxed{7.11c} \\
 &\frac{\boxed{7.11b} \quad \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \star \rightarrow \hat{\alpha}_1 \rightsquigarrow \star \rightarrow \hat{\beta}_1 \dashv \Theta_1}{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\alpha}_3 \rightsquigarrow \hat{\beta}_2 \dashv \Theta_2} \text{A-PR-APP} \quad \boxed{7.11d} \\
 &\dots \frac{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\alpha}_3 \rightsquigarrow \hat{\beta}_2 \dashv \Theta_2}{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\alpha}_3 \text{ Int} \rightsquigarrow \hat{\beta}_2 \text{ Int} \dashv \Theta_2} \text{A-PR-APP} \quad \boxed{7.11a}
 \end{aligned}$$

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 $\hat{\alpha}_3$, by rule **A-PR-KUVARR**, we first promote the kind of $\hat{\alpha}_3$, which is (after context application) $\star \rightarrow \hat{\alpha}_1$ (Figure 7.11b). As \star and \rightarrow are always well-formed, we then promote $\hat{\alpha}_1$ whose kind is the well-formed \star . Now we create a fresh variable $\hat{\beta}_1 : \star$, and solve $\hat{\alpha}_1$ with $\hat{\beta}_1$ (Figure 7.11c). Note since $\hat{\alpha}_1$ is solved, the dependency from $\hat{\alpha}_3$ goes to $\hat{\beta}_1$. Finally, we create a fresh variable $\hat{\beta}_2$ with kind $\star \rightarrow \hat{\beta}_1$, and solve $\hat{\alpha}_3$ with $\hat{\beta}_2$ (Figure 7.11d). Going back to unification, we solve $\hat{\alpha} = \hat{\beta}_2$ 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 ($\hat{\alpha}_3$) to its adjacent nodes (\star , \rightarrow , and $\hat{\alpha}_1$). 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 $\Delta[\hat{\alpha}] \text{ ok}$, and a kind ω_1 with $[\Delta]\omega_1 = \omega_1$, it is decidable whether there exists Θ such that $\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta$.*

7.7.6 SOUNDNESS, COMPLETENESS AND PRINCIPALITY

We prove our algorithm is sound:

Theorem 7.10 (Soundness of \Vdash^{pgm}). *If $\Omega; \Gamma \Vdash^{\text{pgm}} \text{pgm} : \mu$, then $[\Omega]\Omega; [\Omega]\Gamma \vdash^{\text{pgm}} \text{pgm} : [\Omega]\mu$.*

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 $d :: \text{Proxy } b$ or $d :: \text{Proxy } c$, 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.

```

3401   data Eq ::  $\forall k. k \rightarrow k \rightarrow \star$ 
3402   data P  ::  $\forall k (a :: k) b. \text{Eq } a \ b \rightarrow \star$ 

```

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 $\Sigma \vdash \eta <: \omega$, if $\omega_1[\phi_1 \mapsto \bar{\rho}_1][\phi_2 \mapsto \bar{\rho}_2] = \omega$ for some $\bar{\rho}_1$ and $\bar{\rho}_2$.

The relation is embedded in $\Sigma \vdash^{\text{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2$ (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 $\Sigma \vdash \eta_2 <: \omega$, we have $\Sigma \vdash \eta_1 <: \omega$.

3417 To understand the definition, consider that if the program type-checks under $T : \eta_2$, then
 3418 it must type-check under $T : \eta_1$, as $T : \eta_1$ can be instantiated to all monokinds that $T : \eta_2$
 3419 is used at.

3420 Now we lift the definition of \Vdash^{grp} to be the generalized result of kinds and contexts.

3421 **Theorem 7.11** (Principality of \Vdash^{grp}). *If $\Omega \Vdash^{\text{grp}} \text{rec } \overline{\tau}_i^i \rightsquigarrow \overline{\eta}_i^i; \overline{\Gamma}_i^i$, then whenever $[\Omega]\Omega \vdash^{\text{grp}}$
 3422 $\text{rec } \overline{\tau}_i^i \rightsquigarrow \overline{\eta}'_i^i; \overline{\Psi}_i^i$ holds, we have $[\Omega]\Omega \vdash [\Omega]\eta_i \preceq \eta'_i$.*

3423 This result echoes the result in the term-level type inference algorithm for Haskell ([Vy-
 3424 tiniotis et al. 2011, Section 6.5]): our algorithm does not infer every kind acceptable by the
 3425 declarative system, but the kinds it does infer are always the best (principal) ones.

3426 7.8 LANGUAGE EXTENSIONS

3427 We have seen that the PolyKinds system incorporates many features and enjoys desirable
 3428 properties. In this section, we discuss how the PolyKinds system can be extended with more
 3429 related language features. Appendix C.1 contains a few more, less impactful extensions.

3430 7.8.1 HIGHER-RANK POLYMORPHISM

3431 The system can be extended naturally to support higher-rank polymorphism [Dunfield and
 3432 Krishnaswami 2013; Peyton Jones et al. 2007]. With higher-rank polymorphism, every type
 3433 can have a polymorphic kind. For example, data constructor declarations become $\forall\phi. D \overline{\sigma}_i^i$
 3434 instead of $\forall\phi. D \overline{\tau}_i^i$.

3435 Unfortunately, higher-rank polymorphism breaks principality. Consider:

3436 **data** $Q1 :: \forall k_1 k_2. k_1 \rightarrow \star$
 3437 **data** $Q2 :: (\forall (k_1 : \star) (k_2 : k_1). k_1 \rightarrow \star) \rightarrow \star$

3438 First, we modify the definition of partial order of kinds (Definition 24) to state that one
 3439 kind is more general than another if it can be instantiated to all *polykinds* that the other
 3440 kind can be instantiated to. Now consider the kind of $Q1$, which under the algorithm is
 3441 generalized to $\forall\{k3 : \star\} (k_1 : \star) (k_2 : k3). k_1 \rightarrow \star$. In Theorem 7.11, we guarantee that
 3442 this kind is a principal kind as it can be instantiated to all monokinds that other possible
 3443 kinds for $Q1$, e.g., $\forall (k_1 :: \star) (k_2 :: k_1). k_1 \rightarrow \star$, can be instantiated to. However, under
 3444 the new definition, $\forall\{k3 :: \star\} (k_1 :: \star) (k_2 :: k3). k_1 \rightarrow \star$ is no longer more general than
 3445 $\forall (k_1 :: \star) (k_2 :: k_1). k_1 \rightarrow \star$, as there is no way to instantiate the former to the latter. To see
 3446 why we need to modify the definition at all, consider the rank-2 kind of $Q2$, which expects
 3447 exactly an argument of kind $\forall (k_1 :: \star) (k_2 :: k_1). k_1 \rightarrow \star$.

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 $\forall\{k3::\star\} (k1::\star) (k2::k3). k1 \rightarrow \star$ to $\forall(k1::\star) (k2::k1). k1 \rightarrow \star$ if we could abstract over the $k1$ 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^{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

3480 into scope, which would influence the unification algorithm. However, as pattern matching
 3481 does not happen in the context of datatype declarations, we avoid this wrinkle here.

3482 SYNTAX The implementation of GADTs in GHC has an unusual syntax:

```
3483     data G a where
3484         MkG :: a → G Int
```

3485 The surprising aspect of this syntax is that the two *as* above are *different*: the *a* in the header
 3486 is unrelated to the *a* in the data constructor. This seemingly inconsequential design choice
 3487 makes kind inference for GADTs very challenging, as constructors have no way to refer back
 3488 to the datatype parameters. Given that this aspect of GADTs is a quirk of GHC’s design—and
 3489 is not repeated in other languages that support GADTs—we remark here that it is odd and
 3490 perhaps should be remedied. We will return back to this discussion in Section 9.4.

3491 7.8.3 TYPE FAMILIES

3492 Type families [Chakravarty et al. 2005] are, effectively, type-level functions. Kind inference
 3493 of type families thus can be designed much like type inference for ordinary functions. How-
 3494 ever, as they can have dependency, the complications we describe in this paper would arise
 3495 here, too. In particular, unification would have to be kind-directed, as we have described.
 3496 The current syntax for closed type families [Eisenberg et al. 2014] shares the same scoping
 3497 problem as the syntax for GADTs, so our arguments above apply to closed type families
 3498 equally.

3499 The challenge with type families is that they indeed do pattern-matching, and thus (in
 3500 concert with GADTs) can bring local equalities into scope. A full analysis of the ramifications
 3501 here is beyond the scope of this paper, but we believe the literature on type inference in the
 3502 presence of local equalities would be helpful. Principal among these is the work of Vytiniotis
 3503 et al. [2011], but Gundry [2013] and Eisenberg [2016] also approach this problem in the
 3504 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 $\forall a. \text{Int} \rightarrow \text{Int} <: \text{Int} \rightarrow \forall a. a$ 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 relies 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.

3531 The FPH system [Vytiniotis et al. 2008] introduces boxy monotypes into System F types.
 3532 One critique of boxy type inference is that the impredicativity is deeply hidden in the algo-
 3533 rithmic type inference rules, which makes it hard to understand the interaction between its
 3534 predicative constraints and impredicative instantiations [Rémy 2005].

3535 Recently, Serrano et al. [2020, 2018] exploit impredicative instantiations of type variables
 3536 that appears under a type constructor (i.e., type variables are *guarded*). Serrano et al. [2018]
 3537 distinguish variables using three *sorts*, so that certain sorts of variables can be instantiated
 3538 with higher-rank polymorphic types. Serrano et al. [2020] inspect the function arguments
 3539 and assign impredicative instantiations before monomorphic ones.

3540 8.2 BIDIRECTIONAL TYPE CHECKING

3541 Bidirectional type checking was popularized by the work of Pierce and Turner [2000]. It has
 3542 since been applied to many type systems with advanced features. The alternative application
 3543 mode introduced in Chapter 3 enables a variant of bidirectional type checking. There are
 3544 many other efforts to refine bidirectional type checking.

3545 Colored local type inference [Odersky et al. 2001] allows partial type information to be
 3546 propagated, by distinguishing inherited types (known from the context) and synthesized
 3547 types (inferred from terms). A similar distinction is achieved in Dunfield and Krishnaswami
 3548 [2013] by manipulating type variables.

3549 *Tridirectional* type checking [Dunfield and Pfenning 2004] is based on bidirectional type
 3550 checking and has a rich set of property types including intersections, unions and quantified
 3551 dependent types, but without parametric polymorphism. Tridirectional type checking has a
 3552 new direction for supporting type checking unions and existential quantification.

3553 Greedy bidirectional polymorphism [Dunfield 2009] adopts a greedy idea from Cardelli
 3554 [1993] on bidirectional type checking with higher-rank types, where type variables in in-
 3555 stantiations are determined by their first constraint. In this way, they support some uses of
 3556 impredicative polymorphism. However, the greediness also makes many obvious programs
 3557 rejected.

3558 A detailed survey of the development of bidirectional type checking is given by Dunfield
 3559 and Krishnaswami [2020], which collect and explain the design principles of bidirectional
 3560 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.

3596 8.4 GRADUAL TYPE SYSTEMS WITH EXPLICIT POLYMORPHISM

3597 Morris [1973] dynamically enforces parametric polymorphism and uses *sealing* functions as
 3598 the dynamic type mechanism. More recent works on integrating gradual typing with para-
 3599 metric polymorphism include the dynamic type of Abadi et al. [1995] and the *Sage* language
 3600 of Gronski et al. [2006]. None of these has carefully studied the interaction between statically
 3601 and dynamically typed code.

3602 Ahmed et al. [2009] proposed λB that extends the blame calculus [Wadler and Findler
 3603 2009] to incorporate polymorphism. The key novelty of their work is to use dynamic seal-
 3604 ing to enforce parametricity. As such, they end up with a sophisticated dynamic seman-
 3605 tics. Later, Ahmed et al. [2017] prove that with more restrictions, λB satisfies parametricity.
 3606 Compared to their work, our GPC type system can catch more errors earlier since, as we
 3607 argued, their notion of *compatibility* is too permissive. For example, the following is rejected
 3608 (more precisely, the corresponding source program never gets elaborated) by our type sys-
 3609 tem:

$$(\lambda x : ?. x + 1) : \forall a. a \rightarrow a \rightsquigarrow \langle ? \rightarrow \text{Int} \hookrightarrow \forall a. a \rightarrow a \rangle (\lambda x : ?. x + 1)$$

3610 while the type system of λB would accept the translation, though at runtime, the program
 3611 would result in a cast error as it violates parametricity. We emphasize that it is the combina-
 3612 tion of our powerful type system together with the powerful dynamic semantics of λB that
 3613 makes it possible to have implicit higher-rank polymorphism in a gradually typed setting.
 3614 Devriese et al. [2017] proved that embedding of System F terms into λB is not fully abstract.
 3615 Igarashi et al. [2017] also studied integrating gradual typing with parametric polymorphism.
 3616 They proposed System F_G , a gradually typed extension of System F, and System F_C , a new
 3617 polymorphic blame calculus. As has been discussed extensively, their definition of type con-
 3618 sistency does not apply to our setting (implicit polymorphism). All of these approaches mix
 3619 consistency with subtyping to some extent, which we argue should be orthogonal. On a side
 3620 note, it seems that our calculus can also be safely translated to System F_C . However we do
 3621 not understand all the tradeoffs involved in the choice between λB and System F_C as a target.

3622 Recently, Toro et al. [2019] applied AGT to designing a gradual language with explicit
 3623 parametric polymorphism, claiming that graduality and parametricity are inherently incom-
 3624 patible. However, later New et al. [2019] show that by modifying System F's syntax to make
 3625 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.

3690 Ziliani and Sozeau [2015] present the higher-order unification algorithm for CIC, the base
 3691 logic of Coq. They favor syntactic equality by trying first-order unification, as they argue
 3692 the first-order solution gives the most *natural* solution. However, they omit a correctness
 3693 proof for their algorithm. Coen [2004] also considers first-order unification, but only the
 3694 soundness lemma is proved. Different from their systems, our system is based on the novel
 3695 promotion judgment, and correctness including soundness and termination is proved.

3696 The technique of *suspended substitutions* [Eisenberg 2016; Gundry and McBride 2013]
 3697 is widely adopted in unification algorithms. Our system provides a design alternative, our
 3698 *quantification check*. Choosing between suspended substitutions and the quantification check
 3699 is a user-facing language design decision, as suspended substitutions can accept some more
 3700 programs. The quantification check means that the kind of a locally quantified variable a
 3701 must be fully determined in a 's scope; it may *not* be influenced by usage sites of the con-
 3702 struct that depends on a . Suspended substitutions relax this restriction. We conjecture that
 3703 suspended substitutions can yield a complete algorithm. However, that mechanism is com-
 3704 plex. Moreover, unification based on suspended substitutions is only decidable for the pat-
 3705 tern fragment. Our system, in contrast, avoids all the complication introduced by suspended
 3706 substitutions through its quantification check. Our unification terminates for all inputs, pre-
 3707 serving backward compatibility to Hindley-Milner-style inference. Although we reject the
 3708 definition of T (Section 7.7.2), we can solve more constraints outside the pattern fragment.
 3709 We conjecture that those constraints are much more common than definitions like T . Sus-
 3710 pended substitutions often come with a *pruning* process [Abel and Pientka 2011], which
 3711 produces a valid solution before solving a unification variable. Our promotion process has a
 3712 similar effect.

3713 HOMOGENEOUS KIND-PRESERVING UNIFICATION. Jones [1995] proposed a homogeneous
 3714 kind-preserving unification between two types. Kinds κ are defined only as \star or $\kappa_1 \rightarrow \kappa_2$.
 3715 As the kind system is much simpler, kind-preserving unification \sim_κ is simply subscripted
 3716 by the kind, and working out the kinds is straightforward. Our unification subsumes Jones's
 3717 algorithm.

3718 CONTEXT EXTENSION. Our approach of recording unification variables and their solutions
 3719 in the contexts is inspired by Gundry et al. [2010] and Dunfield and Krishnaswami [2013].
 3720 Gundry and McBride [2013] applied the approach to unification in dependent types, where
 3721 the context also records constraints; constraints also appear in context in Eisenberg [2016].
 3722 Further, in PolyKinds, we extend the context extension approach with local scopes, support-
 3723 ing 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 $a = \text{Int}$ in $\lambda x : a. x + 1$` into $(\lambda a. \lambda x : a. x + 1) \text{Int}$, 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](#).

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

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

3749 With such rules it would be possible to handle declarations easily in dependent type sys-
3750 tems.

3751 9.2 TYPE INFERENCE FOR INTERSECTION TYPE SYSTEMS

3752 Another type system that could possibly benefit from the application mode is intersection
3753 type systems [Coppo et al. 1979; Pottinger 1980; Salle 1978]. In particular, we consider in-
3754 tersection type systems with an explicit *merge operator* [Dunfield 2014]. In such a system,
3755 we can construct terms of an intersection type, like $1, , \text{true}$ of type $\text{Int} \& \text{Bool}$. Thanks to
3756 *subtyping*, a term of type $\text{Int} \& \text{Bool}$ can also be used as if it had type Int , or as if it had type
3757 Bool . Calculi with *disjoint intersection types* [Alpuim et al. 2017; Bi et al. 2019; Oliveira et al.
3758 2016] incorporate a *coherent* merge operator. In such calculi the merge operator can merge
3759 two terms with *arbitrary* types as long as their types are disjoint; disjointness conflicts are
3760 reported as type-errors. As illustrated by Xie et al. [2020], the expressive power of disjoint
3761 intersection types can encode diverse programming language features, promising an econ-
3762 omy of theory and implementation.

3763 Disjoint intersection types also pose challenges to type inference. Supposing that we have
3764 $\text{succ} : \text{Int} \rightarrow \text{Int}$ and $\text{not} : \text{Bool} \rightarrow \text{Bool}$, consider the following term:

3765 $(\text{succ} , , \text{not}) 3$

3766 We expect the expression to type-check, as according to subtyping, the term $(\text{succ} , , \text{not})$
3767 of type $(\text{Int} \rightarrow \text{Int} \& \text{Bool} \rightarrow \text{Bool})$ can also be used as type $\text{Int} \rightarrow \text{Int}$. Thus we expect
3768 typing to automatically pick succ and apply it to 3. To this end, we need to push the type
3769 information of the argument (3) into the function $(\text{succ} , , \text{not})$.

3770 Future work is required to explore how well the application mode can be used for type
3771 inference in intersection type systems, and whether it can be integrated with the distributivity
3772 subtyping rules of intersection types [Bi et al. 2019].

3773 9.3 GRADUALIZING TYPE CLASSES

3774 In Section 4.1.2, we discussed about gradualizing modern functional programming languages
 3775 like Haskell. One of its core abstraction features in Haskell is *type classes*. Type classes
 3776 [Wadler and Blott 1989] were initially introduced in Haskell to make ad-hoc overloading less
 3777 ad-hoc, and since then have been adopted in many languages including Mercury [Henderson
 3778 et al. 1996], Coq [Sozeau and Oury 2008], PureScript [Freeman 2017], and Lean [de Moura
 3779 et al. 2015]. An interesting future direction then is to gradualizing type classes.

3780 Consider again the example used in Section 4.1.2:

```
3781 (\f. (f 1, f 'a')) (\x. x)
```

3782 While $f : \forall a. a \rightarrow a$ is of course a valid type annotation, it unfortunately rules out many
 3783 valid arguments that may have type class constraints in their types, e.g.,

```
3784 show      :: Show a => a -> String
```

```
3785
```

```
3786 (\f :: ∀a. a -> a. (f 1, f 'a')) show      -- rejected
```

3787 With gradual typing, if we annotation f with the the unknown type $?$, we expect that the
 3788 following expression can type-check.

```
3789 (\f :: ?. (f 1, f 'a')) show
```

3790 However, a nontrivial challenge in gradualizing type classes is that the dynamic seman-
 3791 tics of type classes is not expressed directly but rather by type-directed elaboration into a
 3792 simpler language without type classes. Thus the dynamic semantics of type classes is given
 3793 indirectly as the dynamic semantics of their elaborated forms. Consider `show` as an exam-
 3794 ple. The *dictionary-passing* elaboration of type-classes translates the type of `show` into the
 3795 following one, supposing `ShowD` is the dictionary type of the type class `show`.

```
3796 show :: ShowD a -> a -> String
```

3797 Now with the unknown type, we cannot predict how to elaborate the original expression.
 3798 In particular, if f is applied to `show`, it means that f needs to be elaborated into a function
 3799 that actually takes two arguments, first the dictionary and then the argument.

```
3800 (\f. (f showInt 1, f showChar 'a')) show
```

3801 This kind of uncertainty in elaboration brings extra complexity and may interact with
 3802 explicit casts in the target blame calculi.

3803 9.4 GENERALIZED ALGEBRAIC DATATYPES (GADTs)

3804 A natural extension of PolyKinds is to include GADTs. We have briefly discussed GADTs in
 3805 Section 7.8.2. In particular, we are interested in finding the right formalization of GADTs.

3806 Haskell’s *syntax* for GADT declarations is quite troublesome. Consider these examples:

3807 `data R a where`
 3808 `MkR :: b → R b`

3809 `data S a where`
 3810 `MkS :: S b`

3811 `data T a where`
 3812 `MkT :: ∀(k :: ★) (b :: k). T b`

3813 In GHC’s implementation of GADTs, any variables declared in the header (between `data` and
 3814 `where`) *do not scope*. In all the examples above, the type variable `a` does not scope over the
 3815 constructor declarations. This is why we have written the variable `b` in those types, to make
 3816 it clear that `b` is distinct from `a`. We could have written `a`—it would still be a distinct `a` from
 3817 that in the header—but it would be more confusing.

3818 The question is: how do we determine the kind of the parameter to the datatype? One
 3819 possibility is to look only in the header. In all cases above, we would infer no constraints and
 3820 would give each type a kind of $\forall(k :: \star). k \rightarrow \star$. This is unfortunate, as it would make `R`
 3821 a kind-indexed GADT: the `MkR` constructor would carry a proof that the kind of its type
 3822 parameter is \star . This, in turn, wreaks havoc with type inference, as it is hard to infer the result
 3823 type of a pattern-match against a GADT Vytiniotis et al. [2011].

3824 Furthermore, this approach might accept *more* programs than the user wants. Consider
 3825 this definition:

3826 `data P a where`
 3827 `MkP1 :: b → P b`
 3828 `MkP2 :: f a → P f`

3829 Does the user want a kind-indexed GADT, noting that `b` and `f` have different kinds? Or
 3830 would the user want this rejected? If we make the fully general kind $\forall k. k \rightarrow \star$ for `P`, this
 3831 would be accepted, perhaps surprising users.

3832 It thus seems we wish to look at the data constructors when inferring the kind of the
 3833 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 $\star \rightarrow \star$. 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 $S :: \forall (k :: \star). k \rightarrow \star$. 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 $\hat{\alpha}$ —with k , because k would be bound to the right of $\hat{\alpha}$ 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, 80, 82, 85, 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 61 and 173]

- 3872 Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive dis-
3873 joint polymorphism for compositional programming. In *European Symposium on Pro-*
3874 *gramming (ESOP)*. [cited on page 180]
- 3875 Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In
3876 *Proceedings of the 28th European Conference on Object-Oriented Programming*. [cited on
3877 pages 8, 61, and 173]
- 3878 Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C#.
3879 In *Proceedings of the European Conference on Object-Oriented Programming*. [cited on
3880 page 61]
- 3881 Richard S. Bird and Lambert Meertens. 1998. Nested datatypes. In *LNCS 1422: Pro-*
3882 *ceedings of Mathematics of Program Construction*, Johan Jeuring (Ed.). Springer-Verlag,
3883 Marstrand, Sweden, 52–67. [http://www.cs.ox.ac.uk/people/richard.bird/](http://www.cs.ox.ac.uk/people/richard.bird/online/BirdMeertens98Nested.pdf)
3884 [online/BirdMeertens98Nested.pdf](http://www.cs.ox.ac.uk/people/richard.bird/online/BirdMeertens98Nested.pdf) [cited on page 138]
- 3885 Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Op-
3886 tional Types for Clojure. In *Programming Languages and Systems*. [cited on pages 8 and 61]
- 3887 Joachim Breitner, Richard A Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016.
3888 Safe zero-cost coercions for Haskell. *Journal of Functional Programming* 26 (2016). [cited
3889 on page 176]
- 3890 L. Cardelli. 1986. *A polymorphic lambda-calculus with Type:Type*. Technical Report 10. SRC.
3891 [cited on page 136]
- 3892 Luca Cardelli. 1993. *An implementation of FSub*. Technical Report. Research Report 97,
3893 Digital Equipment Corporation Systems Research Center. [cited on pages 89 and 172]
- 3894 Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection
3895 Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 41 (Aug. 2017), 28 pages. [cited on
3896 pages 61, 75, 99, and 173]
- 3897 Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated type
3898 synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Func-*
3899 *tional Programming* (Tallinn, Estonia) (ICFP '05). ACM, New York, NY, USA, 241–253.
3900 <https://doi.org/10.1145/1086365.1086397> [cited on page 168]
- 3901 Gang Chen. 2003. Coercive Subtyping for the Calculus of Constructions (POPL '03). 10.
3902 [cited on page 53]

- 3903 Alonzo Church. 1941. *The calculi of lambda-conversion*. Number 6. Princeton University
3904 Press. [cited on page 65]
- 3905 Matteo Cimini and Jeremy G. Siek. 2016. The Gradualizer: A Methodology and Algorithm
3906 for Generating Gradual Type Systems. In *Proceedings of the 43rd Symposium on Principles
3907 of Programming Languages*. [cited on pages 61, 75, 83, and 173]
- 3908 Matteo Cimini and Jeremy G. Siek. 2017. Automatically Generating the Dynamic Seman-
3909 tics of Gradually Typed Languages. In *Proceedings of the 44th Symposium on Principles of
3910 Programming Languages*. [cited on pages 61 and 173]
- 3911 Jesper Cockx, Dominique Devriese, and Frank Piessens. 2016. Unifiers as equivalences:
3912 proof-relevant unification of dependently typed data. In *Proceedings of the 21st ACM
3913 SIGPLAN International Conference on Functional Programming (Nara, Japan) (ICFP
3914 2016)*. ACM, New York, NY, USA, 270–283. [https://doi.org/10.1145/2951913.](https://doi.org/10.1145/2951913.2951917)
3915 2951917 [cited on page 176]
- 3916 Claudio Sacerdoti Coen. 2004. *Mathematical knowledge management and interactive theorem
3917 proving*. Ph.D. Dissertation. University of Bologna, 2004. Technical Report UBLCS 2004-
3918 5. [cited on page 177]
- 3919 Marco Comini, Ferruccio Damiani, and Samuel Vrech. 2008. On polymorphic recursion,
3920 type systems, and abstract interpretation. In *International Static Analysis Symposium*.
3921 Springer, 144–158. [cited on page 176]
- 3922 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Patrick Sallé. 1979. Functional charac-
3923 terization of some semantic equalities inside λ -calculus. In *International Colloquium on
3924 Automata, Languages, and Programming*. Springer, 133–146. [cited on page 180]
- 3925 Thierry Coquand. 1996. An algorithm for type-checking dependent types. *Science of Com-
3926 puter Programming* 26, 1-3 (1996), 167–177. [cited on page 24]
- 3927 Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin.
3928 1958. *Combinatory logic*. Vol. 1. North-Holland Amsterdam. [cited on page 65]
- 3929 Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs.
3930 In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Program-
3931 ming Languages (Albuquerque, New Mexico) (POPL '82)*. Association for Computing Ma-
3932 chinery, New York, NY, USA, 207–212. <https://doi.org/10.1145/582153.582176>
3933 [cited on pages 3, 4, 15, 18, 133, 136, and 175]

- 3934 Ferruccio Damiani. 2003. Rank 2 intersection types for local definitions and conditional
 3935 expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 4
 3936 (2003), 401–451. [cited on page 176]
- 3937 Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In
 3938 *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Program-*
 3939 *ming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 198–208.
 3940 <https://doi.org/10.1145/351240.351259> [cited on pages 24 and 99]
- 3941 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von
 3942 Raumer. 2015. The Lean theorem prover. (2015). [cited on page 181]
- 3943 Dominique Devriese, Marco Patrignani, and Frank Piessens. 2017. Parametricity versus the
 3944 universal type. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 38.
 3945 [cited on page 174]
- 3946 Joshua Dunfield. 2009. Greedy Bidirectional Polymorphism. In *Workshop on ML*. [cited on
 3947 page 172]
- 3948 Joshua Dunfield. 2014. Elaborating intersection and union types. *Journal of Functional Pro-*
 3949 *gramming (JFP)* 24, 2-3 (2014), 133–165. [cited on page 180]
- 3950 Jana Dunfield and Neel Krishnaswami. 2020. Bidirectional Typing. arXiv:1908.05839 [cs.PL]
 3951 [cited on page 172]
- 3952 Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional
 3953 Typechecking for Higher-Rank Polymorphism. In *Proceedings of the 18th ACM SIGPLAN*
 3954 *International Conference on Functional Programming (Boston, Massachusetts, USA) (ICFP*
 3955 *'13)*. Association for Computing Machinery, New York, NY, USA, 429–442. [https://](https://doi.org/10.1145/2500365.2500582)
 3956 doi.org/10.1145/2500365.2500582 [cited on pages 3, 6, 7, 12, 23, 25, 28, 30, 61, 81,
 3957 85, 95, 109, 114, 119, 120, 124, 127, 142, 148, 166, 171, 172, and 177]
- 3958 Joshua Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and Complete Bidi-
 3959 rectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed
 3960 Types. *Proc. ACM Program. Lang.* 3, POPL, Article 9 (Jan. 2019), 28 pages. [https:](https://doi.org/10.1145/3290322)
 3961 [//doi.org/10.1145/3290322](https://doi.org/10.1145/3290322) [cited on page 171]
- 3962 Joshua Dunfield and Frank Pfenning. 2004. Tridirectional Typechecking. *SIGPLAN Not.*
 3963 39, 1 (Jan. 2004), 281–292. <https://doi.org/10.1145/982962.964025> [cited on
 3964 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, 91, 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, 62, 68, 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]

- 3995 Roberta Gori and Giorgio Levi. 2002. An experiment in type inference and verification by
3996 abstract interpretation. In *International Workshop on Verification, Model Checking, and*
3997 *Abstract Interpretation*. Springer, 225–239. [cited on page 176]
- 3998 Roberta Gori and Giorgio Levi. 2003. Properties of a type abstract interpreter. In *Interna-*
3999 *tional Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer,
4000 132–145. [cited on page 176]
- 4001 Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N Freund, and Cormac Flanagan.
4002 2006. Sage: Hybrid Checking for Flexible Specifications. In *Scheme and Functional Pro-*
4003 *gramming Workshop*. [cited on page 174]
- 4004 Adam Gundry and Conor McBride. 2013. A tutorial implementation of dynamic pattern
4005 unification. *Unpublished draft* (2013). [cited on pages 176 and 177]
- 4006 Adam Gundry, Conor McBride, and James McKinna. 2010. Type inference in context. In
4007 *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional*
4008 *programming*. ACM, 43–54. [cited on pages 12, 119, 120, 121, 122, and 177]
- 4009 Adam Michael Gundry. 2013. *Type inference, Haskell and dependent types*. Ph.D. Disserta-
4010 tion. University of Strathclyde. [cited on pages 137, 168, and 176]
- 4011 Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon
4012 Taylor, and Chris Speirs. 1996. *The Mercury Language Reference Manual*. Technical
4013 Report. [cited on page 181]
- 4014 Fritz Henglein. 1993. Type inference with polymorphic recursion. *ACM Trans. Program.*
4015 *Lang. Syst.* 15, 2 (April 1993), 253–289. <https://doi.org/10.1145/169701.169692>
4016 [cited on pages 138 and 216]
- 4017 J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic.
4018 *Trans. Amer. Math. Soc.* 146 (1969), 29–60. [cited on pages 3, 4, 15, and 133]
- 4019 G. Huet. 1973. A unification algorithm for typed lambda calculus. *Theoretical Computer*
4020 *Science* 1, 1 (1973), 27–57. [cited on page 137]
- 4021 Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing.
4022 In *Proceedings of the 22nd International Conference on Functional Programming*. [cited on
4023 pages 68, 70, 71, 75, 103, 107, and 174]

- 4024 Khurram A. Jafery and Joshua Dunfield. 2017. Sums of Uncertainty: Refinements Go Grad-
4025 ual. In *Proceedings of the 44th Symposium on Principles of Programming Languages*. 14.
4026 [cited on pages 61, 75, and 173]
- 4027 Trevor Jim. 1996. What are principal typings and what are they good for?. In *Proceedings*
4028 *of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.
4029 ACM, 42–53. [cited on page 176]
- 4030 Mark P Jones. 1995. A system of constructor classes: overloading and implicit higher-order
4031 polymorphism. *Journal of functional programming* 5, 1 (1995), 1–35. [cited on pages 134,
4032 137, 167, and 177]
- 4033 Mark P. Jones. 1996. Using Parameterized Signatures to Express Modular Structure. In
4034 *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Program-*
4035 *ming Languages* (St. Petersburg Beach, Florida, USA) (POPL '96). 68–78. <https://doi.org/10.1145/237721.237731> [cited on page 5]
- 4037 Mark P. Jones. 1999. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop*
4038 *(Haskell '99)*, Erik Meijer (Ed.). Paris, France, pp. 9–22. University of Utrecht Technical
4039 Report UU-CS-1999-28. [cited on page 150]
- 4040 Mark P Jones. 2000. Type classes with functional dependencies. In *European Symposium on*
4041 *Programming*. Springer, 230–244. [cited on page 67]
- 4042 Assaf J Kfoury and Jerzy Tiuryn. 1992. Type reconstruction in finite rank fragments of the
4043 second-order λ -calculus. *Information and computation* 98, 2 (1992), 228–257. [cited on
4044 page 19]
- 4045 Oleg Kiselyov, Ralf Lämmel, and Kean Schupke. 2004. Strongly typed heterogeneous col-
4046 lections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. ACM, 96–107.
4047 [cited on page 67]
- 4048 Csongor Kiss, Susan Eisenbach, Tony Field, and Simon Peyton Jones. 2019. Higher-order
4049 type-level programming in Haskell. In *Proceedings of the 24th ACM SIGPLAN Interna-*
4050 *tional Conference on Functional Programming (ICFP 2019)*. ACM. [cited on page 167]
- 4051 Pieter Koopman, Rinus Plasmeijer, and Jan Martin Jansen. 2014. Church Encoding of Data
4052 Types Considered Harmful for Implementations: Functional Pearl. In *Proceedings of the*
4053 *26nd 2014 International Symposium on Implementation and Application of Functional Lan-*
4054 *guages* (Boston, MA, USA) (IFL '14). Association for Computing Machinery, New York,

- 4055 NY, USA, Article 4, 12 pages. <https://doi.org/10.1145/2746325.2746330> [cited
4056 on page 65]
- 4057 Didier Le Botlan and Didier Rémy. 2003. MLF: Raising ML to the Power of System F (*ICFP*
4058 '03). 12. [cited on pages 6 and 171]
- 4059 Didier Le Botlan and Didier Rémy. 2009. Recasting MLF. *Information and Computation* 207,
4060 6 (2009), 726–785. [cited on page 171]
- 4061 Jukka Lehtosalo et al. 2006. Mypy. <http://www.mypy-lang.org/> [cited on pages 8
4062 and 61]
- 4063 Daan Leijen. 2009. Flexible Types: Robust Type Inference for First-class Polymorphism
4064 (*POPL '09*). 12. [cited on pages 6 and 171]
- 4065 Andres Löb, Conor McBride, and Wouter Swierstra. 2010. A tutorial implementation of a
4066 dependently typed lambda calculus. *Fundamenta informaticae* 102, 2 (2010), 177–207.
4067 [cited on page 24]
- 4068 Jacob Matthews and Amal Ahmed. 2008. Parametric polymorphism through run-time seal-
4069 ing or, theorems for low, low prices!. In *European Symposium on Programming*. Springer,
4070 16–31. [cited on page 80]
- 4071 Conor McBride. 2002. Faking it Simulating dependent types in Haskell. *Journal of functional*
4072 *programming* 12, 4-5 (2002), 375–392. [cited on page 67]
- 4073 McCracken. 1984. The typechecking of programs with implicit type structure. In *Lecture*
4074 *Notes in Computer Science (Semantics of Data Types)*, Vol. 173. [cited on page 5]
- 4075 Dale Miller. 1991. Unification of simply typed lambda-terms as logic programming. (1991).
4076 [cited on page 176]
- 4077 Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer*
4078 *and system sciences* 17, 3 (1978), 348–375. [cited on page 15]
- 4079 James H. Morris, Jr. 1973. Types Are Not Sets. In *Proceedings of the 1st Annual ACM*
4080 *SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Mas-
4081 sachusetts) (*POPL '73*). ACM, New York, NY, USA, 120–124. [https://doi.org/10.](https://doi.org/10.1145/512927.512938)
4082 [1145/512927.512938](https://doi.org/10.1145/512927.512938) [cited on page 174]
- 4083 James Hiram Morris Jr. 1969. *Lambda-calculus models of programming languages*. Ph.D.
4084 Dissertation. Massachusetts Institute of Technology. [cited on page 82]

- 4085 Alan Mycroft. 1984. Polymorphic type schemes and recursive definitions. In *International*
 4086 *Symposium on Programming*. Springer, 217–228. [cited on page 176]
- 4087 Georg Neis, Derek Dreyer, and Andreas Rossberg. 2009. Non-parametric Parametric-
 4088 ity. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional*
 4089 *Programming* (Edinburgh, Scotland) (ICFP '09). ACM, New York, NY, USA, 135–148.
 4090 <https://doi.org/10.1145/1596550.1596572> [cited on page 80]
- 4091 Max S. New, Dustin Jamner, and Amal Ahmed. 2019. Graduality and Parametricity: Together
 4092 Again for the First Time. *Proc. ACM Program. Lang.* 4, POPL, Article 46 (Dec. 2019),
 4093 32 pages. <https://doi.org/10.1145/3371114> [cited on page 174]
- 4094 Martin Odersky and Konstantin Läuffer. 1996. Putting Type Annotations to Work. In *Pro-*
 4095 *ceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming*
 4096 *Languages* (St. Petersburg Beach, Florida, USA) (POPL '96). Association for Comput-
 4097 ing Machinery, New York, NY, USA, 54–67. [https://doi.org/10.1145/237721.](https://doi.org/10.1145/237721.237729)
 4098 237729 [cited on pages 3, 7, 15, 18, 19, 20, 68, 133, and 171]
- 4099 Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored Local Type Infer-
 4100 ence (POPL '01). 13. [cited on pages 37 and 172]
- 4101 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In
 4102 *International Conference on Functional Programming* (ICFP). [cited on page 180]
- 4103 Michel Parigot. 1992. Recursive programming with proofs. *Theoretical Computer Science* 94,
 4104 2 (1992), 335–356. [cited on page 65]
- 4105 Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge
 4106 University Press. [cited on page 135]
- 4107 Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: exploring the design
 4108 space. In *Haskell workshop*, Vol. 1997. [cited on page 67]
- 4109 Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Prac-
 4110 tical type inference for arbitrary-rank types. *Journal of functional programming* 17, 1
 4111 (2007), 1–82. [cited on pages 3, 6, 7, 15, 18, 24, 28, 29, 35, 38, 39, 45, 46, 47, 53, 81,
 4112 133, 137, 166, 167, 171, and 175]
- 4113 Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006.
 4114 Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM*
 4115 *SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA)

- 4116 (ICFP '06). ACM, New York, NY, USA, 50–61. [https://doi.org/10.1145/1159803.](https://doi.org/10.1145/1159803.1159811)
4117 1159811 [cited on pages 133 and 167]
- 4118 Benjamin C Pierce. 2002. *Types and programming languages*. [cited on page 81]
- 4119 Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Pro-*
4120 *gram. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. <https://doi.org/10.1145/345099.345100>
4121 [cited on pages 7, 11, 23, 56, and 172]
- 4122 François Pottier and Didier Rémy. 2005. The essence of ML type inference. *Advanced Topics*
4123 *in Types and Programming Languages* (2005). [cited on pages 175, 176, and 211]
- 4124 Garrel Pottinger. 1980. A type assignment for the strongly normalizable λ -terms. *To HB*
4125 *Curry: essays on combinatory logic, lambda calculus and formalism* (1980), 561–577. [cited
4126 on page 180]
- 4127 Jason Reed. 2009. Higher-order constraint simplification in dependent type theory. In *Pro-*
4128 *ceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages:*
4129 *Theory and Practice*. ACM, 49–56. [cited on page 176]
- 4130 Didier Rémy. 2005. Simple, Partial Type-inference for System F Based on Type-containment
4131 (ICFP '05). 14. [cited on page 172]
- 4132 Didier Rémy and Boris Yakobowski. 2008. From ML to MLF: Graphic Type Constraints with
4133 Efficient Type Inference (ICFP '08). 12. [cited on page 171]
- 4134 John C Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium*.
4135 Springer, 408–425. [cited on page 5]
- 4136 John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Proceedings*
4137 *of the IFIP 9th World Computer Congress*. [cited on page 80]
- 4138 John C. Reynolds. 1991. The Coherence of Languages with Intersection Types. In *Proceedings*
4139 *of the International Conference on Theoretical Aspects of Computer Software*. [cited on
4140 page 82]
- 4141 Patrick Salle. 1978. Une Extension De La Theorie Des Types En lambda-Calcul. In *Proceed-*
4142 *ings of the Fifth Colloquium on Automata, Languages and Programming*. Springer-Verlag,
4143 London, UK, UK, 398–410. [cited on page 180]

- 4144 Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009.
 4145 Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIG-*
 4146 *PLAN International Conference on Functional Programming* (Edinburgh, Scotland) (ICFP
 4147 '09). ACM, New York, NY, USA, 341–352. [https://doi.org/10.1145/1596550.](https://doi.org/10.1145/1596550.1596599)
 4148 1596599 [cited on page 175]
- 4149 Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A
 4150 Quick Look at Impredicativity. *Proc. ACM Program. Lang.* 4, ICFP, Article 89 (Aug. 2020),
 4151 29 pages. <https://doi.org/10.1145/3408971> [cited on pages 6, 22, and 172]
- 4152 Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018.
 4153 Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Con-*
 4154 *ference on Programming Language Design and Implementation* (Philadelphia, PA, USA)
 4155 (PLDI 2018). ACM, New York, NY, USA, 783–796. [https://doi.org/10.1145/](https://doi.org/10.1145/3192366.3192389)
 4156 3192366.3192389 [cited on pages 6, 22, and 172]
- 4157 Paula Severi and Erik Poll. 1994. Pure Type Systems with Definitions. *Logical Foundations*
 4158 *of Computer Science* (1994), 316–328. [cited on page 179]
- 4159 Jeremy Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the design space of higher-
 4160 order casts. In *European Symposium on Programming*. 17–31. [cited on page 173]
- 4161 Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceed-*
 4162 *ings of the 2006 Scheme and Functional Programming Workshop*. [cited on pages 8, 11, 61,
 4163 101, and 173]
- 4164 Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *European Conference*
 4165 *on Object-Oriented Programming*. [cited on pages 6, 9, 11, 61, 62, 63, 64, 68, 71, 73, 98,
 4166 99, 101, and 173]
- 4167 Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-based In-
 4168 ference. In *Proceedings of the 2008 Symposium on Dynamic Languages*. [cited on page 175]
- 4169 Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined
 4170 criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL*
 4171 *2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. [cited on pages 9, 11, 12, 75,
 4172 78, 83, 85, and 173]
- 4173 Vincent Simonet and François Pottier. 2007. A constraint-based approach to guarded alge-
 4174 braic data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*
 4175 29, 1 (2007), 1. [cited on page 175]

- 4176 Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *TPHOLs '08* (Montreal,
4177 P.Q., Canada). Springer-Verlag, 278–293. [cited on page 181]
- 4178 Sam Tobin-Hochstadt. 2019. Gradual Typing from Theory to Practice. [https://blog.
4179 sigplan.org/2019/07/12/gradual-typing-theory-practice/](https://blog.sigplan.org/2019/07/12/gradual-typing-theory-practice/) [cited on page 8]
- 4180 Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual Parametricity, Revisited.
4181 *Proc. ACM Program. Lang.* 3, POPL, Article 17 (Jan. 2019), 30 pages. [https://doi.
4182 org/10.1145/3290330](https://doi.org/10.1145/3290330) [cited on page 174]
- 4183 Julien Verlauguet. 2013. Facebook: Analyzing PHP statically. In *Proceedings of Commercial
4184 Users of Functional Programming*. [cited on page 61]
- 4185 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and
4186 Evaluation of Gradual Typing for Python. In *Proceedings of the 10th Symposium on Dy-
4187 namic languages*. [cited on pages 8 and 61]
- 4188 Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. Out-
4189 sideIn (X) Modular type inference with local assumptions. *Journal of functional program-
4190 ming* 21, 4-5 (2011), 333–412. [cited on pages 137, 166, 167, 168, 175, 182, 211, and 212]
- 4191 Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2006. Boxy types: in-
4192 ference for higher-rank types and impredicativity. In *Proceedings of the Eleventh ACM
4193 SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA)
4194 (*ICFP '06*). ACM, New York, NY, USA. [https://doi.org/10.1145/1159803.
4195 1159811](https://doi.org/10.1145/1159803.1159811) [cited on page 37]
- 4196 Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: First-class
4197 Polymorphism for Haskell (*ICFP '08*). 12. [cited on page 172]
- 4198 P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *POPL
4199 '89*. ACM. [cited on page 181]
- 4200 Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed.
4201 In *Proceedings of the 18th European Symposium on Programming Languages and Systems*.
4202 [cited on page 174]
- 4203 Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. 2019. A
4204 Role for dependent types in Haskell. *Proc. ACM Program. Lang.* 3, ICFP, Article 101 (July
4205 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]

- 4238 Ningning Xie, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers. 2020. Row and
4239 Bounded Polymorphism via Disjoint Polymorphism. In *34th European Conference on*
4240 *Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Infor-*
4241 *matics (LIPIcs))*, Robert Hirschfeld and Tobias Pape (Eds.), Vol. 166. Schloss Dagstuhl–
4242 Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 27:1–27:30. [https://doi.org/](https://doi.org/10.4230/LIPIcs.ECOOP.2020.27)
4243 [10.4230/LIPIcs.ECOOP.2020.27](https://doi.org/10.4230/LIPIcs.ECOOP.2020.27) [cited on page 180]
- 4244 Ningnign Xie and Richard A Eisenberg. 2018. Coercion Quantification. In *Haskell Imple-*
4245 *mentors' Workshop*. [cited on page 176]
- 4246 Ningning Xie, Richard A Eisenberg, and Bruno CDS Oliveira. 2019b. Kind Inference for
4247 Datatypes: Technical Supplement. *arXiv preprint arXiv:1911.06153* (2019). [https://](https://arxiv.org/abs/1911.06153)
4248 arxiv.org/abs/1911.06153 [cited on pages 136 and 176]
- 4249 Ningning Xie, Richard A. Eisenberg, and Bruno C. d. S. Oliveira. 2019c. Kind Inference for
4250 Datatypes. *Proc. ACM Program. Lang.* 4, POPL, Article 53 (Dec. 2019), 28 pages. [https:](https://doi.org/10.1145/3371121)
4251 [//doi.org/10.1145/3371121](https://doi.org/10.1145/3371121) [cited on page 13]
- 4252 Ningning Xie and Bruno C d S Oliveira. 2017. Towards Unification for Dependent Types. In
4253 *Draft Proceedings of the 18th Symposium on Trends in Functional Programming (TFP '18)*.
4254 Extended abstract. [cited on pages 13 and 132]
- 4255 Ningning Xie and Bruno C d S Oliveira. 2018. Let Arguments Go First. In *European Sympo-*
4256 *sium on Programming*. Springer, 272–299. [cited on pages 13, 38, and 57]
- 4257 Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis,
4258 and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th*
4259 *ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Philadelphia,
4260 Pennsylvania, USA) (*TLDI '12*). ACM, New York, NY, USA, 53–66. [https://doi.org/](https://doi.org/10.1145/2103786.2103795)
4261 [10.1145/2103786.2103795](https://doi.org/10.1145/2103786.2103795) [cited on pages 134, 136, and 210]
- 4262 Beta Ziliani and Matthieu Sozeau. 2015. A Unification Algorithm for Coq Featuring Universe
4263 Polymorphism and Overloading. In *Proceedings of the 20th ACM SIGPLAN International*
4264 *Conference on Functional Programming* (Vancouver, BC, Canada) (*ICFP 2015*). ACM, New
4265 York, NY, USA, 179–191. <https://doi.org/10.1145/2784731.2784751> [cited on
4266 page 176]

PART VI

TECHNICAL APPENDIX

4267

A

FULL RULES FOR ALGORITHMIC AP

4268

$$\boxed{(S_1, N_1) \vdash^{AP} \sigma <: \sigma_2 \hookrightarrow (S_2, N_2)}$$

(Algorithmic Subtyping)

4269

$$\frac{\text{AP-A-S-MONO} \quad S_0 \vdash^{AP} \tau_1 \approx \tau_2 \hookrightarrow S_1}{(S_0, N_0) \vdash^{AP} \tau_1 <: \tau_2 \hookrightarrow (S_1, N_0)}$$

4270

$$\frac{\text{AP-A-S-ARROWL} \quad \begin{array}{l} (S_0, N_0) \vdash^{AP} \sigma \triangleright \sigma_3 \rightarrow \sigma_4 \hookrightarrow (S_1, N_1) \\ (S_1, N_1) \vdash^{AP} \sigma_3 <: \sigma_1 \hookrightarrow (S_2, N_2) \quad (S_2, N_2) \vdash^{AP} \sigma_2 <: \sigma_4 \hookrightarrow (S_3, N_3) \end{array}}{(S_0, N_0) \vdash^{AP} \sigma_1 \rightarrow \sigma_2 <: \sigma \hookrightarrow (S_3, N_3)}$$

4271

$$\frac{\text{AP-A-S-ARROWR} \quad \begin{array}{l} (S_0, N_0) \vdash^{AP} \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \hookrightarrow (S_1, N_1) \\ (S_1, N_1) \vdash^{AP} \sigma_3 <: \sigma_1 \hookrightarrow (S_2, N_2) \quad (S_2, N_2) \vdash^{AP} \sigma_2 <: \sigma_4 \hookrightarrow (S_3, N_3) \end{array}}{(S_0, N_0) \vdash^{AP} \sigma <: \sigma_3 \rightarrow \sigma_4 \hookrightarrow (S_3, N_3)}$$

4272

$$\frac{\text{AP-A-S-FORALLL} \quad (S_0, N_0) \vdash^{AP} \sigma_1[a \mapsto \widehat{\beta}] <: \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0 \widehat{\beta}) \vdash^{AP} \forall a. \sigma_1 <: \sigma_2 \hookrightarrow (S_1, N_1)}$$

4273

$$\frac{\text{AP-A-S-FORALLR} \quad (S_0, N_0) \vdash^{AP} \sigma_1 <: \sigma_2[a \mapsto b] \hookrightarrow (S_1, N_1) \quad b \notin \text{FV}(S(\sigma_1)) \quad b \notin \text{FV}(S(\forall a. \sigma_2))}{(S_0, N_0 b) \vdash^{AP} \sigma_1 <: \forall a. \sigma_2 \hookrightarrow (S_1, N_1)}$$

4274

$$\boxed{(S_1, N_1); \Sigma \vdash^{AP} \sigma <: \sigma_2 \hookrightarrow (S_2, N_2)}$$

(Algorithmic Application Subtyping)

4275

$$\frac{\text{AP-A-AS-EMPTY}}{(S_0, N_0); \bullet \vdash^{AP} \sigma <: \sigma \hookrightarrow (S_0, N_0)}$$

4276

$$\frac{\text{AP-A-AS-FORALL} \quad (S_0, N_0); \Sigma, \sigma_3 \vdash^{AP} \sigma_1[a \mapsto \widehat{\beta}] <: \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0 \widehat{\beta}); \Sigma, \sigma_3 \vdash^{AP} \forall a. \sigma_1 <: \sigma_2 \hookrightarrow (S_1, N_1)}$$

A Full Rules for Algorithmic AP

4277	$\frac{\text{AP-A-AS-ARROW} \quad (S_0, N_0) \vdash^{AP} \sigma_3 <: \sigma_1 \hookrightarrow (S_1, N_1) \quad (S_1, N_1); \Sigma \vdash^{AP} \sigma_2 <: \sigma_4 \hookrightarrow (S_2, N_2)}{(S_0, N_0); \Sigma, \sigma_3 \vdash^{AP} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4 \hookrightarrow (S_2, N_2)}$	
4278	$\frac{\text{AP-A-AS-MONO} \quad (S_0, N_0) \vdash^{AP} \tau \triangleright \tau_1 \rightarrow \tau_2 \hookrightarrow (S_1, N_1) \quad (S_1, N_1); \Sigma, \sigma_3 \vdash^{AP} \tau_1 \rightarrow \tau_2 <: \sigma \hookrightarrow (S_2, N_2)}{(S_0, N_0 \hat{\beta}); \Sigma, \sigma_3 \vdash^{AP} \tau <: \sigma \hookrightarrow (S_2, N_2)}$	
4279	$\boxed{(S_1, N_1) \vdash^{AP} \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \hookrightarrow (S_2, N_2)}$	(Matching)
4280	$\frac{\text{AP-A-M-TVAR} \quad S_0 \vdash^{AP} \hat{\alpha} \approx \hat{\alpha}_1 \rightarrow \hat{\alpha}_2 \hookrightarrow S_1}{(S_0, N_0 \hat{\alpha}_1 \hat{\alpha}_2) \vdash^{AP} \hat{\alpha} \triangleright \hat{\alpha}_1 \rightarrow \hat{\alpha}_2 \hookrightarrow (S_1, N_0)}$	
4281	$\frac{\text{AP-A-M-ARROW}}{(S_0, N_0) \vdash^{AP} \sigma_1 \rightarrow \sigma_2 \triangleright \sigma_1 \rightarrow \sigma_2 \hookrightarrow (S_0, N_0)}$	
4282	$\boxed{S_1 \vdash^{AP} \tau_1 \approx \tau_2 \hookrightarrow S_2}$	(Unification)
4283	$\frac{\text{AP-A-U-REFL}}{S_0 \vdash^{AP} \tau \approx \tau \hookrightarrow S_0} \quad \frac{\text{AP-A-U-SOLVED\texttt{EVARL}} \quad \hat{\alpha} \in S_0 \quad S_0 \vdash^{AP} S_0(\hat{\alpha}) \approx \tau \hookrightarrow S_1}{S_0 \vdash^{AP} \hat{\alpha} \approx \tau \hookrightarrow S_1}$	
4284	$\frac{\text{AP-A-U-EVARL} \quad \hat{\alpha} \notin S_0 \quad \hat{\alpha} \notin \text{FV}(S_0(\tau))}{S_0 \vdash^{AP} \hat{\alpha} \approx \tau \hookrightarrow [\hat{\alpha} \mapsto S_0(\tau)] \cdot S_1} \quad \frac{\text{AP-A-U-SOLVED\texttt{EVARR}} \quad \hat{\alpha} \in S_0 \quad S_0 \vdash^{AP} \tau \approx S_0(\hat{\alpha}) \hookrightarrow S_1}{S_0 \vdash^{AP} \tau \approx \hat{\alpha} \hookrightarrow S_1}$	
4285	$\frac{\text{AP-A-U-EVARR} \quad \hat{\alpha} \notin S_0 \quad \hat{\alpha} \notin \text{FV}(S_0(\tau))}{S_0 \vdash^{AP} \tau \approx \hat{\alpha} \hookrightarrow [\hat{\alpha} \mapsto S_0(\tau)] \cdot S_1} \quad \frac{\text{AP-A-U-ARROW} \quad S_0 \vdash^{AP} \tau_1 \approx \tau_3 \hookrightarrow S_1 \quad S_1 \vdash^{AP} \tau_2 \approx \tau_4 \hookrightarrow S_2}{S_0 \vdash^{AP} \tau_1 \rightarrow \tau_2 \approx \tau_3 \rightarrow \tau_4 \hookrightarrow S_2}$	
4286	$\boxed{(S_1, N_1); \Psi \vdash^{AP} e \Rightarrow \sigma \hookrightarrow (S_2, N_2)}$	(Algorithmic Typing Inference)

AP-A-INF-INT

$$\frac{}{(S_0, N_0); \Psi \vdash^{AP} n \Rightarrow \text{Int} \hookrightarrow (S_0, N_0)}$$

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

$$\frac{\text{AP-A-INF-LAMANN} \quad (S_0, N_0); \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0); \Psi \vdash^{AP} \lambda x : \sigma_1. e \Rightarrow \sigma_1 \rightarrow \sigma_2 \hookrightarrow (S_1, N_1)}$$

$$\boxed{(S_1, N_1); \Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma \hookrightarrow (S_2, N_2)} \quad (\text{Algorithmic Typing Application Mode})$$

$$\frac{\text{AP-A-APP-VAR} \quad (x : \sigma_1) \in \Psi \quad (S_0, N_0); \Sigma \vdash^{AP} \sigma_1 <: \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0); \Psi; \Sigma \vdash^{AP} x \Rightarrow \sigma_2 \hookrightarrow (S_1, N_1)}$$

$$\frac{\text{AP-A-APP-LAM} \quad (S_0, N_0); \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0); \Psi; \Sigma, \sigma_1 \vdash^{AP} \lambda x. e \Rightarrow \sigma_1 \rightarrow \sigma_2 \hookrightarrow (S_1, N_1)}$$

$$\frac{\text{AP-A-APP-LAMANN} \quad (S_0, N_0) \vdash^{AP} \sigma_2 <: \sigma_1 \hookrightarrow (S_1, N_1) \quad (S_1, N_1); \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_3 \hookrightarrow (S_2, N_2)}{(S_0, N_0); \Psi; \Sigma, \sigma_2 \vdash^{AP} \lambda x : \sigma_1. e \Rightarrow \sigma_2 \rightarrow \sigma_3 \hookrightarrow (S_2, N_2)}$$

$$\frac{\text{AP-A-APP-APP} \quad (S_0, N_0); \Psi \vdash^{AP} e_2 \Rightarrow \sigma_1 \hookrightarrow (S_1, N_1 \overline{a_i}^i) \quad \overline{\alpha_i}^i = \text{FV}(S_1(\sigma_1)) - \text{FV}(S_1(\Psi)) \quad \sigma_2 = \forall \overline{a_i}^i. \sigma_1[\overline{\alpha_i}^i \mapsto \overline{a_i}^i] \quad (S_1, N_1); \Psi; \Sigma, \sigma_2 \vdash^{AP} e_1 \Rightarrow \sigma_2 \rightarrow \sigma_3 \hookrightarrow (S_2, N_2)}{(S_0, N_0); \Psi; \Sigma \vdash^{AP} e_1 e_2 \Rightarrow \sigma_3 \hookrightarrow (S_2, N_2)}$$

4295

B THE EXTENDED ALGORITHMIC GPC

4296

B.1 SYNTAX

4297

4298

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	$\sigma ::= \mathbf{Int} \mid a \mid \hat{\alpha} \mid \sigma_1 \rightarrow \sigma_2 \mid \forall a. \sigma \mid ? \mid \mathcal{S} \mid \mathcal{G}$
Monotypes	$\tau ::= \mathbf{Int} \mid a \mid \hat{\alpha} \mid \tau_1 \rightarrow \tau_2 \mid \mathcal{S} \mid \mathcal{G}$
Existential variables	$\hat{\alpha} ::= \hat{\alpha}_S \mid \hat{\alpha}_G$
Castable Types	$\mathbb{C} ::= \mathbf{Int} \mid a \mid \hat{\alpha} \mid \mathbb{C}_1 \rightarrow \mathbb{C}_2 \mid \forall a. \mathbb{C} \mid ? \mid \mathcal{G}$
Castable Monotypes	$t ::= \mathbf{Int} \mid a \mid \hat{\alpha} \mid t_1 \rightarrow t_2 \mid \mathcal{G}$
Algorithmic Contexts	$\Gamma, \Delta, \Theta ::= \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \hat{\alpha} \mid \Gamma, \hat{\alpha}_S = \tau \mid \Gamma, \hat{\alpha}_G = t \mid \Gamma, \blacktriangleright_{\hat{\alpha}}$
Complete Contexts	$\Omega ::= \bullet \mid \Omega, x : \sigma \mid \Omega, a \mid \Omega, \hat{\alpha}_S = \tau \mid \Omega, \hat{\alpha}_G = t \mid \Omega, \blacktriangleright_{\hat{\alpha}}$

4299

B.2 TYPE SYSTEM

4300

$$\boxed{\Gamma \vdash^G \sigma \lesssim aB \dashv \Delta}$$

(Algorithmic Consistent Subtyping)

4301

GPC-AS-TVAR

$$\frac{}{\Gamma[a] \vdash^G a \lesssim a \dashv \Gamma[a]}$$

GPC-AS-EVAR

$$\frac{}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \hat{\alpha} \dashv \Gamma[\hat{\alpha}]}$$

GPC-AS-INT

$$\frac{}{\Gamma \vdash^G \mathbf{Int} \lesssim \mathbf{Int} \dashv \Gamma}$$

4302

GPC-AS-ARROW

$$\frac{\Gamma \vdash^G \sigma_3 \lesssim \sigma_1 \dashv \Theta \quad \Theta \vdash^G [\Theta] \sigma_2 \lesssim [\Theta] \sigma_4 \dashv \Delta}{\Gamma \vdash^G \sigma_1 \rightarrow \sigma_2 \lesssim \sigma_3 \rightarrow \sigma_4 \dashv \Delta}$$

GPC-AS-FORALLR

$$\frac{\Gamma, a \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta, a, \Theta}{\Gamma \vdash^G \sigma_1 \lesssim \forall a. \sigma_2 \dashv \Delta}$$

4303

GPC-AS-FORALLLL

$$\frac{\Gamma, \blacktriangleright_{\hat{\alpha}_S}, \hat{\alpha}_S \vdash^G \sigma_1[a \mapsto \hat{\alpha}_S] \lesssim \sigma_2 \dashv \Delta, \blacktriangleright_{\hat{\alpha}_S}, \Theta}{\Gamma \vdash^G \forall a. \sigma_1 \lesssim \sigma_2 \dashv \Delta}$$

GPC-AS-SPAR

$$\frac{}{\Gamma \vdash^G \mathcal{S} \lesssim \mathcal{S} \dashv \Gamma}$$

4304

GPC-AS-GPAR

$$\frac{}{\Gamma \vdash^G \mathcal{G} \lesssim \mathcal{G} \dashv \Gamma}$$

GPC-AS-UNKNOWNLL

$$\frac{}{\Gamma \vdash^G ? \lesssim \mathbb{C} \dashv \mathbf{contaminate}(\Gamma, \mathbb{C})}$$

4305	$\frac{\text{GPC-AS-UNKNOWNRR}}{\Gamma \vdash^G \mathbb{C} \lesssim ? \dashv \text{contaminate}(\Gamma, \mathbb{C})}$	$\frac{\text{GPC-AS-INSTL} \quad \hat{\alpha} \notin \text{FV}(\sigma) \quad \Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta}$
4306	$\frac{\text{GPC-AS-INSTR} \quad \hat{\alpha} \notin \text{FV}(\sigma) \quad \Gamma[\hat{\alpha}] \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta}$	
4307	$\boxed{\Gamma \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta}$	(Instantiation I)
4308	$\frac{\text{GPC-INSTL-SOLVE S} \quad \Gamma \vdash^G \tau}{\Gamma, \hat{\alpha}_S, \Gamma' \vdash^G \hat{\alpha}_S \lesssim \tau \dashv \Gamma, \hat{\alpha}_S = \tau, \Gamma'}$	$\frac{\text{GPC-INSTL-SOLVE G} \quad \Gamma \vdash^G t}{\Gamma, \hat{\alpha}_G, \Gamma' \vdash^G \hat{\alpha}_G \lesssim t \dashv \Gamma, \hat{\alpha}_G = t, \Gamma'}$
4309	$\frac{\text{GPC-INSTL-SOLVE US}}{\Gamma[\hat{\alpha}_S] \vdash^G \hat{\alpha}_S \lesssim ? \dashv \Gamma[\hat{\alpha}_G, \hat{\alpha}_S = \hat{\alpha}_G]}$	$\frac{\text{GPC-INSTL-SOLVE UG}}{\Gamma[\hat{\alpha}_G] \vdash^G \hat{\alpha}_G \lesssim ? \dashv \Gamma[\hat{\alpha}_G]}$
4310	$\frac{\text{GPC-INSTL-REACH SG1}}{\Gamma[\hat{\alpha}_S][\hat{\beta}_G] \vdash^G \hat{\alpha}_S \lesssim \hat{\beta}_G \dashv \Gamma[\hat{\alpha}_G, \hat{\alpha}_S = \hat{\alpha}_G][\hat{\beta}_G = \hat{\alpha}_G]}$	
4311	$\frac{\text{GPC-INSTL-REACH SG2}}{\Gamma[\hat{\beta}_S][\hat{\alpha}_G] \vdash^G \hat{\alpha}_G \lesssim \hat{\beta}_S \dashv \Gamma[\hat{\beta}_G, \hat{\beta}_S = \hat{\beta}_G][\hat{\alpha}_G = \hat{\beta}_G]}$	
4312	$\frac{\text{GPC-INSTL-REACH OTHER}}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash^G \hat{\alpha} \lesssim \hat{\beta} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]}$	
4313	$\frac{\text{GPC-INSTL-ARR} \quad \Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \vdash^G \sigma_1 \lesssim \hat{\alpha}_1 \dashv \Theta \quad \Theta \vdash^G \hat{\alpha}_2 \lesssim [\Theta]\sigma_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \sigma_1 \rightarrow \sigma_2 \dashv \Delta}$	
4314	$\frac{\text{GPC-INSTL-FORALL R} \quad \Gamma[\hat{\alpha}], b \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta, b, \Theta}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \forall b. \sigma \dashv \Delta}$	
4315	$\boxed{\Gamma \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta}$	(Instantiation II)

4316	$\frac{\text{GPC-INSTR-SOLVES} \quad \Gamma \vdash^G \tau}{\Gamma, \hat{\alpha}_S, \Gamma' \vdash^G \tau \lesssim \hat{\alpha}_S \dashv \Gamma, \hat{\alpha}_S = \tau, \Gamma'}$	$\frac{\text{GPC-INSTR-SOLVEG} \quad \Gamma \vdash^G t}{\Gamma, \hat{\alpha}_G, \Gamma' \vdash^G t \lesssim \hat{\alpha}_G \dashv \Gamma, \hat{\alpha}_G = t, \Gamma'}$
4317	$\frac{\text{GPC-INSTR-SOLVEUS}}{\Gamma[\hat{\alpha}_S] \vdash^G ? \lesssim \hat{\alpha}_S \dashv \Gamma[\hat{\alpha}_G, \hat{\alpha}_S = \hat{\alpha}_G]}$	$\frac{\text{GPC-INSTR-SOLVEUG}}{\Gamma[\hat{\alpha}_G] \vdash^G ? \lesssim \hat{\alpha}_G \dashv \Gamma[\hat{\alpha}_G]}$
4318	$\frac{\text{GPC-INSTR-REACHSG1}}{\Gamma[\hat{\alpha}_S][\hat{\beta}_G] \vdash^G \hat{\beta}_G \lesssim \hat{\alpha}_S \dashv \Gamma[\hat{\alpha}_G, \hat{\alpha}_S = \hat{\alpha}_G][\hat{\beta}_G = \hat{\alpha}_G]}$	
4319	$\frac{\text{GPC-INSTR-REACHSG2}}{\Gamma[\hat{\beta}_S][\hat{\alpha}_G] \vdash^G \hat{\beta}_S \lesssim \hat{\alpha}_G \dashv \Gamma[\hat{\beta}_G, \hat{\beta}_S = \hat{\beta}_G][\hat{\alpha}_G = \hat{\beta}_G]}$	
4320	$\frac{\text{GPC-INSTR-REACHOTHER}}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash^G \hat{\beta} \lesssim \hat{\alpha} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]}$	
4321	$\frac{\text{GPC-INSTR-ARR} \quad \Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \vdash^G \hat{\alpha}_1 \lesssim \sigma_1 \dashv \Theta \quad \Theta \vdash^G [\Theta]\sigma_2 \lesssim \hat{\alpha}_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \sigma_1 \rightarrow \sigma_2 \lesssim \hat{\alpha} \dashv \Delta}$	
4322	$\frac{\text{GPC-INSTR-FORALLLL} \quad \Gamma[\hat{\alpha}], \blacktriangleright_{\hat{b}_S}, \hat{\beta}_S \vdash^G \sigma[b \mapsto \hat{\beta}_S] \lesssim \hat{\alpha} \dashv \Delta, \blacktriangleright_{\hat{b}_S}, \Theta}{\Gamma[\hat{\alpha}] \vdash^G \forall b. \sigma \lesssim \hat{\alpha} \dashv \Delta}$	
4323	$\boxed{\Gamma \vdash^G e \Rightarrow \sigma \dashv \Delta}$	(Inference)
4324	$\frac{\text{GPC-INF-VAR} \quad (x : \sigma) \in \Gamma}{\Gamma \vdash^G x \Rightarrow \sigma \dashv \Gamma}$	$\frac{\text{GPC-INF-INT}}{\Gamma \vdash^G n \Rightarrow \text{Int} \dashv \Gamma}$
4325	$\frac{\text{GPC-INF-LAMANN2} \quad \Gamma \vdash^G \sigma \quad \Gamma, \hat{\beta}_S, x : \sigma \vdash^G e \Leftarrow \hat{\beta}_S \dashv \Delta, x : \sigma, \Theta}{\Gamma \vdash^G \lambda x : \sigma. e \Rightarrow \sigma \rightarrow \hat{\beta}_S \dashv \Delta}$	
4326	$\frac{\text{GPC-INF-LAM2} \quad \Gamma, \hat{\alpha}_S, \hat{\beta}_S, x : \hat{\alpha}_S \vdash^G e \Leftarrow \hat{\beta}_S \dashv \Delta, x : \hat{\alpha}_S, \Theta}{\Gamma \vdash^G \lambda x. e \Rightarrow \hat{\alpha}_S \rightarrow \hat{\beta}_S \dashv \Delta}$	$\frac{\text{GPC-INF-ANNO} \quad \Gamma \vdash^G \sigma \quad \Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta}{\Gamma \vdash^G e : \sigma \Rightarrow \sigma \dashv \Delta}$

B The Extended Algorithmic GPC

4327	$\frac{\text{GPC-INF-APP} \quad \Gamma \vdash^G e_1 \Rightarrow \sigma \dashv \Theta_1 \quad \Theta_1 \vdash^G [\Theta_1] \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Theta_2 \quad \Theta_2 \vdash^G e_2 \Leftarrow [\Theta_2] \sigma_1 \dashv \Delta}{\Gamma \vdash^G e_1 e_2 \Rightarrow \sigma_2 \dashv \Delta}$	
4328	$\frac{\text{GPC-INF-LET2} \quad \Gamma \vdash^G e_1 \Rightarrow \sigma \dashv \Theta_1 \quad \Theta_1, \hat{\alpha}_S, x : \sigma \vdash^G e_2 \Leftarrow \hat{\alpha}_S \dashv \Delta, x : \sigma, \Theta_2}{\Gamma \vdash^G \text{let } x = e_1 \text{ in } e_2 \Rightarrow \hat{\alpha}_S \dashv \Delta}$	
4329	$\boxed{\Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta}$	(Checking)
4330	$\frac{\text{GPC-CHK-LAM} \quad \Gamma, x : \sigma_1 \vdash^G e \Leftarrow \sigma_2 \dashv \Delta, x : \sigma_1, \Theta}{\Gamma \vdash^G \lambda x. e \Leftarrow \sigma_1 \rightarrow \sigma_2 \dashv \Delta} \quad \frac{\text{GPC-CHK-GEN} \quad \Gamma, a \vdash^G e \Leftarrow \sigma \dashv \Delta, a, \Theta}{\Gamma \vdash^G e \Leftarrow \forall a. \sigma \dashv \Delta}$	
4331	$\frac{\text{GPC-CHK-SUB} \quad \Gamma \vdash^G e \Rightarrow \sigma_1 \dashv \Theta \quad \Theta \vdash^G [\Theta] \sigma_1 \lesssim [\Theta] \sigma_2 \dashv \Delta}{\Gamma \vdash^G e \Leftarrow \sigma_2 \dashv \Delta}$	
4332	$\boxed{\Gamma \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta}$	(Algorithmic Matching)
4333	$\frac{\text{GPC-AM-FORALL} \quad \Gamma, \hat{\alpha}_S \vdash^G \sigma[a \mapsto \hat{\alpha}_S] \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta}{\Gamma \vdash^G \forall a. \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta} \quad \frac{\text{GPC-AM-ARR}}{\Gamma \vdash^G \sigma_1 \rightarrow \sigma_2 \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Gamma}$	
4334	$\frac{\text{GPC-AM-UNKNOWN}}{\Gamma \vdash^G ? \triangleright ? \rightarrow ? \dashv \Gamma} \quad \frac{\text{GPC-AM-VAR}}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \triangleright \hat{\alpha}_1 \rightarrow \hat{\alpha}_2 \dashv \Gamma[\hat{\alpha}_1, \hat{\alpha}_2, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2]}$	

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 :: ∀(k :: ★) → k → ★`, with which users are forced to provide type arguments to `T`. That is, one would use `T` with, e.g., `T ★ Int` and `T (★ → ★) 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

```
data T :: ∀(k :: ★) → k → ★
data T k a = MkT
```

We should get a context $k :: ★, a :: k$ 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 $P :: ∀(f :: ★) → f → ★$.

```
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., $\hat{\alpha} \rightarrow \hat{\beta} \rightarrow ★$ or $\forall(f : \hat{\alpha}) \rightarrow \hat{\beta} \rightarrow ★$, 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

¹VDQ 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 .

```
data App (f :: * → *) 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.

$$\text{datatype decl. } \mathcal{T} ::= \text{data } T \phi = \overline{\mathcal{D}}_j^j$$

Kind annotations can also contain free variables, which need to be generalized in a similar way as signatures. For example, $T2$ has kind $\forall\{k :: *\}. \forall(f :: k). *$.

```
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 be 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^{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

4419 is that the compiler must maintain an extra invariant that all occurrences of a variable store
 4420 the same kind; this is straightforward to maintain in practice.

4421 Beyond just storing variables' kinds, the typing context in this work also critically stores
 4422 variables' ordering. Lacking contexts, GHC uses a different mechanism: *level numbers*, orig-
 4423 inally invented to implement untouchability [Vytiniotis et al. 2011, Section 5.1]. Every type
 4424 variable in GHC is assigned a level number during inference. Type variables contain a struc-
 4425 ture that includes level numbers. Roughly, the level number of a type variable *a* corresponds
 4426 to the number of type variables in scope before *a*. Accordingly, we can tell the relative order
 4427 (in a hypothetical context, according to the systems in this work) of two variables simply by
 4428 comparing their level numbers. One of GHC's invariants is that a unification variable at level
 4429 *n* is never unified with a type that mentions a variable with a level number *m* > *n*; this is
 4430 much like the extra checks in the unification judgments in our work.

4431 The *local scopes* of this work are also tracked by GHC. All the variables in the same local
 4432 scope are assigned the same level number, and they are flagged as reorderable. After inference
 4433 is complete, GHC does a topological sort to get the final order.

4434 A final role that contexts play in our formalism is that they store solutions for unification
 4435 variables; we apply contexts as a substitution. In GHC, unification variables store mutable
 4436 cells that get filled in. It has a process called *zonking*², which is exactly analogous to our
 4437 use of contexts as substitutions. Zonking a unification variable replaces the variable with its
 4438 solution, if any.

4439 C.2.3 UNIFICATION

4440 The solver in GHC still has to carry out unification, much along the lines of the unification
 4441 judgment we present here. This algorithm has to deal with the heterogeneous unification
 4442 problems we consider, as well. Indeed, GHC's unification algorithm recurs into the kinds of a
 4443 unification variable and the type it is unifying with, just as ours does. As implied by our focus
 4444 on decidability of unification, there have been a number of bugs in GHC's implementation
 4445 that led to loops in the type checker; the most recent is #16902.

4446 GHC actually uses several unification algorithms internally. It has an eager unifier, much
 4447 like the one we describe. When that unifier fails, it generates the constraint that is sent to the
 4448 solver. (The eager unifier is meant solely to be an optimization.) There is also a unifier meant
 4449 to work after type inference is complete; it checks for instance overlap, for example. All the
 4450 unifiers recur into kinds:

²There 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.

- 4451 • The eager unifier recurs into kinds.
- 4452 • The unifier in the solver recurs into kinds.
- 4453 • The pure unifier uses an invariant that the kinds are related before looking at the types.
- 4454 It must recur when decomposing applications.

4455 In addition, GHC also has an overlap problem within unification, as exhibited in our work
 4456 by the overlap between rules `A-U-KVARL` and `A-U-KVARR` in Figure 7.3. Both the eager unifier
 4457 and the constraint-solver unifier deal with this ambiguity by using heuristics to choose which
 4458 variable might be more suitable for unification. This particular issue—which variable to unify
 4459 when there is a choice—has been the subject of some amount of churn over the years.

4460 C.2.4 PROMOTION

4461 The promotion operation, too, is present in GHC, though its form is quite different than what
 4462 we have presented. Instead of promoting during unification, GHC simply refuses to solve
 4463 a unification variable if any of the free variables of its supposed solution lives to the right
 4464 of the variable in the context. Because GHC is working with constraints, it just leaves the
 4465 unification problem as an unsolved constraint. If there remain unsolved constraints, GHC
 4466 then promotes the variables it can: some cannot be promoted because they depend on locally
 4467 bound quantified (not unification) type variables.

4468 C.2.5 COMPLETE USER-SUPPLIED KINDS

4469 Along with stand-alone kind signatures, as described in this work, GHC supports *complete*
 4470 *user-supplied kinds*, or CUSKs. A datatype has a CUSK when certain syntactic conditions are
 4471 satisfied; GHC detects these conditions *before* doing any kind inference. These CUSKs are a
 4472 poor substitute for proper kind signatures, as the syntactic cues are fragile and unexpected:
 4473 users sometimes write a CUSK without meaning to, and also sometimes leave out a necessary
 4474 part of a CUSK when they intend to specify the kind. Stand-alone kind signatures are a new
 4475 feature; they begin with the keyword `type` instead of `data`, as we have used in our work.

4476 Interestingly, it would be wrong to support CUSKs in a system without polymorphic kinds.
 4477 Consider this example:

```
4478     data S1 a = MkT1 S2
4479     data S2 = MkS2 (S1 Maybe)
```

4480 The types `S1` and `S2` form a group. We put `S2` (which has a CUSK) into the context with
 4481 kind `*`. When we check `S1`, we find no constraints on `a` (in the constraint-generation pass;

4482 see the general approach below). The kind of *S1* is then defaulted to $\star \rightarrow \star$. Checking
 4483 *S2* fails. Instead, we wish to pretend that *S2* does not have a CUSK. This would mean that
 4484 constraint-generation happens for all the constructors in both *S1* and *S2*, and *S1* would get
 4485 its correct kind $(\star \rightarrow \star) \rightarrow \star$.

4486 With kind-polymorphism, we have no problem because the kind of *T1* will be generalized
 4487 to $\forall(k :: \star). k \rightarrow \star$.

4488 This was reported as bug #16609.

4489 C.2.6 DEPENDENCY ANALYSIS

4490 The algorithm implemented in GHC for processing datatype declarations starts with depen-
 4491 dency analysis, as ours does. The dependency analysis is less fine-grained than what we have
 4492 proposed in this work: signatures are ignored in the dependency analysis, and so datatypes
 4493 with signatures are processed alongside all the others. This means that the kinds in the ex-
 4494 ample below have more restrictive kinds in GHC than they do in our system:

```
4495     data S1 :: ∀k. k → ★
4496     data S1 a = MkS1 (S2 Int)
4497     data S2 a = MkS2 (S3 Int)
4498     data S3 a = MkS3 (S1 Int)
```

4499 A naïve dependency analysis would put all three definitions in the same group. The kind for
 4500 *S1* is given; it would indeed have that kind. The parameters of *S2* and *S3* would initially have
 4501 an unknown kind, but when occurrences of *S2* and *S3* are processed (in the definitions of
 4502 *S1* and *S2*, respectively), this unknown kind would become \star . Neither *S2* nor *S3* would be
 4503 generalized.

4504 There is a ticket to improve the dependency analysis: #9427.

4505 C.2.7 APPROACH TO KIND-CHECKING DATATYPES

4506 In GHC’s approach, after dependency analysis, so-called *initial kinds* are produced for all the
 4507 datatypes in the group. These either come from a datatype’s CUSK or from a simple analysis
 4508 of the header of the datatype (without looking at constructors). This step corresponds to our
 4509 algorithm’s placing a binding for the datatype in the context, either with the kind signature
 4510 or with a unification variable (rules [A-PGM-DT-TTS](#) and [A-PGM-DT-TT](#)).

4511 If there is no CUSK, GHC then passes over all the datatype’s constructors, collecting
 4512 constraints on unification variables. After solving these constraints, GHC generalizes the
 4513 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:

```

data T1 (a :: k) b = MkT1 (T2 a b)
data T2 (c :: j) d = MkT2 (T1 c d)

data T3 a where
  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 $\hat{\alpha}$. When checking the *MkT3* constructor, we will need to unify $\hat{\alpha}$ with the locally bound *k*. We cannot set $\hat{\alpha} := k$, as that will fill $\hat{\alpha}$ with a *k*, bound to $\hat{\alpha}$'s *right* in the context. Instead, we must set $k := \hat{\alpha}$. 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:

```
4546      data T4 a =  $\forall(k :: \star) (b :: k). MkT4 (T4 b)$ 
```

4547 The definition of *T4* is polymorphically recursive: the occurrence *T4 b* is specialized
 4548 to a kind other than the kind of *a*. Yet this definition is accepted. The two kinds unify
 4549 (as *k* becomes a unification variable, set to the guessed kind of *a*) during the constraint-
 4550 generation pass. Then, *T4* is generalized to get the kind $\forall k. k \rightarrow \star$, at which point the
 4551 last pass goes through without a hitch.

4552 The reason this acceptance is troublesome is not that *T4* is somehow dangerous or
 4553 unsafe. It is that we know that polymorphic recursion cannot be inferred Henglein
 4554 [1993], and yet GHC does it. Invariably, this must mean that GHC’s algorithm will be
 4555 hard to specify beyond its implementation.

4556 2. In rare cases, the constraint-generation pass will succeed, while the final pass—meant
 4557 to be redundant—will fail. Here is an example:

```
4558      data SameKind :: k  $\rightarrow$  k  $\rightarrow$  Type
4559      data Bad a where
4560          MkBad ::  $\forall k_1 k_2 (a :: k_1) (b :: k_2). Bad (SameKind a b)$ 
```

4561 During the constraint-generation pass, the kinds *k*₁ and *k*₂ are allowed to unify, ac-
 4562 cepting the definition of *Bad*. During the final pass, however, *k*₁ and *k*₂ are proper
 4563 quantified type variables, always distinct. Thus the *SameKind a b* type is ill-kinded
 4564 and rejected.

4565 The fact that this final pass can fail means that we cannot implement it via a simple
 4566 substitution, as we do in rule A-PGM-DT-TT. One possible solution is our suggestion
 4567 to change the scoping of type parameters to GADT-syntax datatype declarations. With
 4568 that change, our second motivation above for TyVarTvs would disappear. GHC could
 4569 then use TyVarTvs only for kind variables in the head of a datatype declaration, using
 4570 proper quantified type variables in constructors. Of course, this change would break
 4571 much code in the wild, and we do not truly expect it to ever be adopted.

4572 C.2.9 THE QUANTIFICATION CHECK

4573 Our quantification check (Section 7.7.2) also has a parallel in GHC, but GHC’s solution to the
 4574 problem differed from ours. Instead of rejecting programs that fail the quantification check,
 4575 GHC accepted them, replacing the variables that would be (but cannot be) quantified with
 4576 its constant *Any* :: $\forall k. k$. The *Any* type is uninhabited, but exists at all kinds. As such, it is
 4577 an appropriate replacement for unquantifiable, unconstrained unification variables. Yet this

4578 decision in GHC had unfortunate consequences: the *Any* type can appear in error messages,
 4579 and its introduction induces hard-to-understand type errors.

4580 We have later implemented our quantification check in GHC; see #16775.

4581 Another design alternative is to generalize the variable to the leftmost position where it is
 4582 still well-formed. Recall the example in Section 7.7.2:

```
4583 data Proxy :: ∀k. k → ★
4584 data Relate :: ∀a (b :: a). a → Proxy b → ★
4585 data T :: ∀(a :: ★) (b :: a) (c :: a) d. Relate b d → ★
```

4586 We have $d :: \hat{\alpha}$, and $\hat{\alpha} = \text{Proxy } \hat{\beta}$, with $\hat{\beta} :: a$. As there are no further constraints on $\hat{\beta}$, the
 4587 definition of T is rejected by the quantification check.

4588 Instead of rejecting the program, or solving $\hat{\beta}$ using *Any*, we can generalize over $\hat{\beta}$ as a
 4589 fresh variable f , which is put after a to make it well-kinded. Namely, we get

```
4590 data T :: ∀(a :: ★) {f :: a} (b :: a) (c :: a) (d :: Proxy f). Relate @a @f b d → ★
```

4591 However, this ordering of the variables violates our declarative specification. Moreover,
 4592 this type requires an inferred variable to be between specified variables. With higher-rank
 4593 polymorphism, due to the fact that GHC does not support first-class type-level abstraction
 4594 (i.e., Λ in types), this type cannot be instantiated to

```
4595 ∀(a :: ★) (b :: a) (c :: a) (d :: Proxy f). Relate @a @b b d → ★
```

4596 or

```
4597 ∀(a :: ★) (b :: a) (c :: a) (d :: Proxy f). Relate @a @c b d → ★
```

4598 which makes the generalization less useful.

4599 C.2.10 SCOPEDSORT

4600 When GHC deals with a local scope—a set of variables that may be reordered—it does a
 4601 topological sort on the variables at the end. However, not any topological sort will do: it must
 4602 use one that preserves the left-to-right ordering of the variables as much as possible. This is
 4603 because GHC considers these implicitly bound variables to be *specified*: they are available
 4604 for visible type application. For example, recall the example from Section 7.2.2, modified
 4605 slightly:

```
4606 data Q (a :: (f b)) (c :: k) (x :: f c)
```

4607 Inference will tell us that k must come before f and b , but the order of f and b is immaterial.
 4608 Our approach here is to make f , b , and k *inferred* variables: users of Q will not be able to
 4609 instantiate these parameters with visible type application. However, GHC takes a different
 4610 view: because the user has written the names of f , b , and k , they will be *specified*. This choice
 4611 means that the precise sorting algorithm GHC uses to fix the order of local scopes becomes
 4612 part of the *specification* of the language. Indeed, GHC documents the precise algorithm in
 4613 its manual. If we followed suit, the algorithm would have to appear in our declarative speci-
 4614 fication, which goes against the philosophy of a declarative system.

4615 Some recent debate led to a conclusion (see #16726) that we would change the interpre-
 4616 tation of the Q example from the main work, meaning that its kind variables would indeed
 4617 become *inferred*. However, the problem with ScopedSort still exists in type signatures, where
 4618 type variables may be implicitly bound.

4619 C.2.11 THE “FORALL-OR-NOTHING” RULE

4620 GHC implements the so-called *forall-or-nothing* rule, which states that either *all* variables
 4621 are quantified by a user-written \forall or none are. These examples illustrate the effect:

```
4622      ex1 :: a → b → a
4623      ex2 :: ∀a b. a → b → a
4624      ex3 :: ∀a. a → b → a
4625      ex4 :: (∀a. a → b → a)
```

4626 The signatures for both $ex1$ and $ex2$ are accepted: $ex1$ quantifies none, while $ex2$ quantifies
 4627 all. The signature for $ex3$ is rejected, as GHC rejects a mixed economy. However, and perhaps
 4628 surprisingly, $ex4$ is accepted. The only difference between $ex3$ and $ex4$ is the seemingly-
 4629 redundant parentheses. However, because the forall-or-nothing rule applies only at the top
 4630 level of a signature, the rule is not in effect for the \forall in $ex4$.

4631 This rule interacts with the main work only in that our formalism (and some of our exam-
 4632 ples) does not respect it. This may be the cause of differing behavior between GHC and the
 4633 examples we present.

4634 C.3 COMPLETE SET OF RULES

4635 In this section we include the complete set of rules for Chapter 7. Some of the rules are
 4636 repeated from those in the chapter.

4637 C.3.1 DECLARATIVE HASKELL98

4638 $\boxed{\Sigma \vdash^k \sigma : \kappa}$ (Kinding for Polymorphic Types)

$$\frac{\text{K-FORALL} \quad \Sigma, a : \kappa \vdash^k \sigma : \star}{\Sigma \vdash^k \forall a : \kappa. \sigma : \star}$$

4639

4640 $\boxed{\Sigma \vdash \Psi}$ (Well-formed Term Contexts)

$$\frac{\text{ECTX-EMPTY}}{\Sigma \vdash \bullet} \quad \frac{\text{ECTX-DCON} \quad \Sigma \vdash \Psi \quad \Sigma \vdash^k \sigma : \star}{\Sigma \vdash \Psi, D : \sigma}$$

4641

4642 C.3.2 ALGORITHMIC HASKELL98

4643 $\boxed{\Delta \Vdash^k \sigma : \kappa \dashv \Theta}$ (Kinding for Polymorphic Types)

$$\frac{\text{A-K-FORALL} \quad \Delta \Vdash^{kv} \kappa \quad \Delta, a : \kappa \Vdash^k \sigma : \kappa_2 \dashv \Theta, a : \kappa \quad [\Theta]\kappa_2 = \star}{\Delta \Vdash^k \forall a : \kappa. \sigma : \star \dashv \Theta}$$

4644

4645 $\boxed{\Delta \Vdash^{kc} \sigma \Leftarrow \kappa}$ (Checking)

$$\frac{\text{A-KC-EQ} \quad \Delta \Vdash^k \sigma : \kappa_1 \dashv \Delta \quad [\Delta]\kappa_1 = [\Delta]\kappa_2}{\Delta \Vdash^{kc} \sigma \Leftarrow \kappa_2}$$

4646

4647 $\boxed{\Delta \Vdash^{kv} \kappa}$ (Well-formed Kinds)

$$\frac{\text{A-KV-STAR}}{\Delta \Vdash^{kv} \star} \quad \frac{\text{A-KV-ARROW} \quad \Delta \Vdash^{kv} \kappa_1 \quad \Delta \Vdash^{kv} \kappa_2}{\Delta \Vdash^{kv} \kappa_1 \rightarrow \kappa_2} \quad \frac{\text{A-KV-KUVAR} \quad \widehat{\alpha} \in \Delta}{\Delta \Vdash^{kv} \widehat{\alpha}}$$

4648

4649 $\boxed{\Delta \text{ ok}}$ (Well-formed Type Contexts)

$$\frac{\text{A-TCTX-EMPTY}}{\bullet \text{ ok}} \quad \frac{\text{A-TCTX-TVAR} \quad \Delta \text{ ok} \quad \Delta \Vdash^{kv} \kappa}{\Delta, a : \kappa \text{ ok}} \quad \frac{\text{A-TCTX-TCON} \quad \Delta \text{ ok} \quad \Delta \Vdash^{kv} \kappa}{\Delta, T : \kappa \text{ ok}} \quad \frac{\text{A-TCTX-KUVAR} \quad \Delta \text{ ok}}{\Delta, \widehat{\alpha} \text{ ok}}$$

4650

$$\frac{\text{A-TCTX-KUVAR SOLVED} \quad \Delta \text{ ok} \quad \Delta \Vdash^{kv} \kappa}{\Delta, \widehat{\alpha} = \kappa \text{ ok}}$$

4651

4652 $\boxed{\Delta \Vdash^{\text{ectx}} \Gamma}$ (Well-formed Term Contexts)

$$\begin{array}{c}
 \text{A-ECTX-EMPTY} \\
 \hline
 \Delta \Vdash^{\text{ectx}} \bullet
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-ECTX-DCON} \\
 \Delta \Vdash^{\text{ectx}} \Gamma \quad \Delta \Vdash^{\text{kc}} \sigma \Leftarrow \star \\
 \hline
 \Delta \Vdash^{\text{ectx}} \Gamma, D : \sigma
 \end{array}$$

4653

4654 $\boxed{\Delta \longrightarrow \Omega}$ (Defaulting)

$$\begin{array}{c}
 \text{A-CTXDE-EMPTY} \\
 \hline
 \bullet \longrightarrow \bullet
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-CTXDE-TVAR} \\
 \Delta \longrightarrow \Omega \\
 \hline
 \Delta, a : \kappa \longrightarrow \Omega, a : \kappa
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-CTXDE-TCON} \\
 \Delta \longrightarrow \Omega \\
 \hline
 \Delta, T : \kappa \longrightarrow \Omega, T : \kappa
 \end{array}$$

4655

$$\begin{array}{c}
 \text{A-CTXDE-KUVAR SOLVED} \\
 \Delta \longrightarrow \Omega \\
 \hline
 \Delta, \hat{\alpha} = \kappa \longrightarrow \Omega, \hat{\alpha} = \kappa
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-CTXDE-SOLVE} \\
 \Delta \longrightarrow \Omega \\
 \hline
 \Delta, \hat{\alpha} \longrightarrow \Omega, \hat{\alpha} = \star
 \end{array}$$

4656

4657 C.3.3 CONTEXT APPLICATION IN HASKELL98

$$\begin{array}{l}
 \hline
 [\Delta]\kappa \text{ applies } \Delta \text{ as a substitution to } \kappa. \\
 [\Delta]\star = \star \\
 [\Delta]\kappa_1 \rightarrow \kappa_2 = [\Delta]\kappa_1 \rightarrow [\Delta]\kappa_2 \\
 [\Delta][\hat{\alpha}]\hat{\alpha} = \hat{\alpha} \\
 [\Delta][\hat{\alpha} = \kappa]\hat{\alpha} = [\Delta][\hat{\alpha} = \kappa]\kappa \\
 \hline
 \end{array}$$

4658

$$\begin{array}{l}
 \hline
 [\Delta]\Gamma \text{ applies } \Delta \text{ as a substitution to } \Gamma. \\
 [\Delta]\bullet = \bullet \\
 [\Delta](\Gamma, D : \sigma) = [\Delta]\Gamma, D : [\Delta]\sigma \\
 \hline
 \end{array}$$

4659

$$\begin{array}{l}
 \hline
 [\Omega]\Delta \text{ applies } \Omega \text{ as a substitution to } \Delta. \\
 [\bullet]\bullet = \bullet \\
 [\Omega, a : \kappa](\Delta, a : \kappa) = [\Omega]\Delta, a : [\Omega]\kappa \\
 [\Omega, T : \kappa](\Delta, T : \kappa) = [\Omega]\Delta, T : [\Omega]\kappa \\
 [\Omega, \hat{\alpha} = \kappa](\Delta, \hat{\alpha}) = [\Omega]\Delta \\
 [\Omega, \hat{\alpha} = \kappa](\Delta, \hat{\alpha} = \kappa') = [\Omega]\Delta \quad \text{if } [\Omega]\kappa = [\Omega]\kappa' \\
 [\Omega, \hat{\alpha} = \kappa]\Delta = [\Omega]\Delta \quad \text{if } \hat{\alpha} \notin \Delta \\
 \hline
 \end{array}$$

4660

4661 C.3.4 CONTEXT EXTENSION IN HASKELL98

4662 $\boxed{\Delta \longrightarrow \Theta}$ (Context Extension)

$$\begin{array}{c}
\text{A-CTXE-EMPTY} \quad \text{A-CTXE-TVAR} \quad \text{A-CTXE-TCON} \\
\frac{}{\bullet \longrightarrow \bullet} \quad \frac{\Delta \longrightarrow \Theta}{\Delta, a : \kappa \longrightarrow \Theta, a : \kappa} \quad \frac{\Delta \longrightarrow \Theta}{\Delta, T : \kappa \longrightarrow \Theta, T : \kappa} \\
4663 \\
\text{A-CTXE-KUVAR} \quad \text{A-CTXE-KUVAR SOLVED} \quad \text{A-CTXE-SOLVE} \\
\frac{\Delta \longrightarrow \Theta}{\Delta, \hat{\alpha} \longrightarrow \Theta, \hat{\alpha}} \quad \frac{\Delta \longrightarrow \Theta \quad [\Theta]\kappa_1 = [\Theta]\kappa_2}{\Delta, \hat{\alpha} = \kappa_1 \longrightarrow \Theta, \hat{\alpha} = \kappa_2} \quad \frac{\Delta \longrightarrow \Theta \quad \Theta \Vdash^{\text{kv}} \kappa}{\Delta, \hat{\alpha} \longrightarrow \Theta, \hat{\alpha} = \kappa} \\
4664 \\
\text{A-CTXE-ADD} \quad \text{A-CTXE-ADD SOLVED} \\
\frac{\Delta \longrightarrow \Theta}{\Delta \longrightarrow \Theta, \hat{\alpha}} \quad \frac{\Delta \longrightarrow \Theta \quad \Theta \Vdash^{\text{kv}} \kappa}{\Delta \longrightarrow \Theta, \hat{\alpha} = \kappa} \\
4665
\end{array}$$

4666 C.3.5 DECLARATIVE POLYKINDS

4667 $\boxed{\ulcorner \sigma \urcorner}$ (Kind results in \star)

$$\begin{array}{c}
\text{SR-STAR} \quad \text{SR-ARROW} \quad \text{SR-FORALL} \\
\frac{}{\ulcorner \star \urcorner} \quad \frac{\ulcorner \kappa_2 \urcorner}{\ulcorner \kappa_1 \rightarrow \kappa_2 \urcorner} \quad \frac{\ulcorner \sigma \urcorner}{\ulcorner \forall \phi. \sigma \urcorner} \\
4668
\end{array}$$

4669 $\boxed{\Sigma \vdash^{\text{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2}$ (Instantiation)

$$\begin{array}{c}
\text{INST-REFL} \quad \text{INST-FORALL} \\
\frac{}{\Sigma \vdash^{\text{inst}} \mu : \omega <: \omega \rightsquigarrow \mu} \quad \frac{\Sigma \vdash^{\text{ela}} \rho : \omega_1 \quad \Sigma \vdash^{\text{inst}} \mu_1 @ \rho : \eta[a \mapsto \rho] <: \omega_2 \rightsquigarrow \mu_2}{\Sigma \vdash^{\text{inst}} \mu_1 : \forall a : \omega_1. \eta <: \omega_2 \rightsquigarrow \mu_2} \\
4670 \\
\text{INST-FORALL-INFER} \\
\frac{\Sigma \vdash^{\text{ela}} \rho : \omega_1 \quad \Sigma \vdash^{\text{inst}} \mu_1 @ \rho : \eta[a \mapsto \rho] <: \omega_2 \rightsquigarrow \mu_2}{\Sigma \vdash^{\text{inst}} \mu_1 : \forall \{a : \omega_1\}. \eta <: \omega_2 \rightsquigarrow \mu_2} \\
4671
\end{array}$$

4672 $\boxed{\Sigma \vdash^{\text{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu}$ (Kind Checking)

$$\begin{array}{c}
\text{KC-SUB} \\
\frac{\Sigma \vdash^{\text{k}} \sigma : \eta \rightsquigarrow \mu_1 \quad \Sigma \vdash^{\text{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2}{\Sigma \vdash^{\text{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu_2} \\
4673
\end{array}$$

4674 $\boxed{\Sigma \vdash^k \sigma : \eta \rightsquigarrow \mu}$ (Kinding)

$$\begin{array}{c}
 \text{KTT-STAR} \qquad \text{KTT-NAT} \qquad \text{KTT-VAR} \\
 \hline
 \Sigma \vdash^k \star : \star \rightsquigarrow \star \qquad \Sigma \vdash^k \text{Int} : \star \rightsquigarrow \text{Int} \qquad \Sigma \vdash^k a : \omega \rightsquigarrow a \\
 \text{KTT-ARROW} \qquad \text{KTT-TCON} \\
 \hline
 \Sigma \vdash^k \rightarrow : \star \rightarrow \star \rightarrow \star \rightsquigarrow \rightarrow \qquad \Sigma \vdash^k T : \eta \rightsquigarrow T \\
 \text{KTT-APP} \\
 \hline
 \Sigma \vdash^k \tau_1 : \eta_1 \rightsquigarrow \rho_1 \quad \Sigma \vdash^{\text{inst}} \rho_1 : \eta_1 <: (\omega_1 \rightarrow \omega_2) \rightsquigarrow \rho_2 \quad \Sigma \vdash^{\text{kc}} \tau_2 \Leftarrow \omega_1 \rightsquigarrow \rho_3 \\
 \hline
 \Sigma \vdash^k \tau_1 \tau_2 : \omega_2 \rightsquigarrow \rho_2 \rho_3 \\
 \text{KTT-KAPP} \\
 \hline
 \Sigma \vdash^k \kappa_1 : \forall a : \omega. \eta \rightsquigarrow \rho_1 \quad \Sigma \vdash^{\text{kc}} \kappa_2 \Leftarrow \omega \rightsquigarrow \rho_2 \\
 \hline
 \Sigma \vdash^k \kappa_1 @ \kappa_2 : \eta[a \mapsto \rho_2] \rightsquigarrow \rho_1 @ \rho_2 \\
 \text{KTT-KAPP-INFER} \\
 \hline
 \Sigma \vdash^k \kappa_1 : \forall \{\overline{a_i} : \overline{\omega_i^i}\}. \forall a : \omega. \eta \rightsquigarrow \rho'_1 \\
 \hline
 \Sigma \vdash^{\text{ela}} \rho_i : \omega_i[\overline{a_i} \mapsto \overline{\rho_i^i}]^i \quad \Sigma \vdash^{\text{kc}} \kappa_2 \Leftarrow \omega[\overline{a_i} \mapsto \overline{\rho_i^i}] \rightsquigarrow \rho'_2 \\
 \hline
 \Sigma \vdash^k \kappa_1 @ \kappa_2 : \eta[\overline{a_i} \mapsto \overline{\rho_i^i}][a \mapsto \rho_2] \rightsquigarrow \rho'_1 @ \overline{\rho_i^i} @ \rho'_2 \\
 \text{KTT-FORALL} \\
 \hline
 \Sigma \vdash^{\text{kc}} \kappa \Leftarrow \star \rightsquigarrow \omega \quad \Sigma, a : \omega \vdash^{\text{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu \\
 \hline
 \Sigma \vdash^k \forall a : \kappa. \sigma : \star \rightsquigarrow \forall a : \omega. \mu \\
 \text{KTT-FORALLI} \\
 \hline
 \Sigma \vdash^{\text{ela}} \omega : \star \quad \Sigma, a : \omega \vdash^{\text{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu \\
 \hline
 \Sigma \vdash^k \forall a. \sigma : \star \rightsquigarrow \forall a : \omega. \mu
 \end{array}$$

4682 $\boxed{\Sigma \vdash^{\text{ela}} \mu : \eta}$ (Elaborated Kinding)

$$\begin{array}{c}
 \text{ELA-STAR} \qquad \text{ELA-NAT} \qquad \text{ELA-VAR} \qquad \text{ELA-TCON} \\
 \hline
 \Sigma \vdash^{\text{ela}} \star : \star \qquad \Sigma \vdash^{\text{ela}} \text{Int} : \star \qquad \Sigma \vdash^{\text{ela}} a : \omega \qquad \Sigma \vdash^{\text{ela}} T : \eta \\
 \text{ELA-ARROW} \qquad \text{ELA-APP} \\
 \hline
 \Sigma \vdash^{\text{ela}} \rightarrow : \star \rightarrow \star \rightarrow \star \qquad \Sigma \vdash^{\text{ela}} \rho_1 : \omega_1 \rightarrow \omega_2 \quad \Sigma \vdash^{\text{ela}} \rho_2 : \omega_1 \\
 \hline
 \Sigma \vdash^{\text{ela}} \rho_1 \rho_2 : \omega_2
 \end{array}$$

4685	$\frac{\text{ELA-KAPP} \quad \Sigma \vdash^{\text{ela}} \rho_1 : \forall a : \omega. \eta \quad \Sigma \vdash^{\text{ela}} \rho_2 : \omega}{\Sigma \vdash^{\text{ela}} \rho_1 @ \rho_2 : \eta[a \mapsto \rho_2]}$	$\frac{\text{ELA-KAPP-INFER} \quad \Sigma \vdash^{\text{ela}} \rho_1 : \forall \{a : \omega\}. \eta \quad \Sigma \vdash^{\text{ela}} \rho_2 : \omega}{\Sigma \vdash^{\text{ela}} \rho_1 @ \rho_2 : \eta[a \mapsto \rho_2]}$
4686	$\frac{\text{ELA-FORALL} \quad \Sigma \vdash^{\text{ela}} \omega : \star \quad \Sigma, a : \omega \vdash^{\text{ela}} \mu : \star}{\Sigma \vdash^{\text{ela}} \forall a : \omega. \mu : \star}$	$\frac{\text{ELA-FORALL-INFER} \quad \Sigma \vdash^{\text{ela}} \omega : \star \quad \Sigma, a : \omega \vdash^{\text{ela}} \mu : \star}{\Sigma \vdash^{\text{ela}} \forall \{a : \omega\}. \mu : \star}$
4687	$\boxed{\Sigma \text{ ok}}$	(Well-formed Type Contexts)
4688	$\frac{\text{TCTX-EMPTY}}{\bullet \text{ ok}}$ $\frac{\text{TCTX-TVAR-TT} \quad \Sigma \text{ ok} \quad \Sigma \vdash^{\text{ela}} \rho : \star}{\Sigma, a : \rho \text{ ok}}$ $\frac{\text{TCTX-TCON-TT} \quad \Sigma \text{ ok} \quad \Sigma \vdash^{\text{ela}} \eta : \star}{\Sigma, T : \eta \text{ ok}}$	
4689	$\boxed{\Sigma \vdash \Psi}$	(Well-formed Term Contexts)
4690	$\frac{\text{ECTX-EMPTY}}{\Sigma \vdash \bullet}$ $\frac{\text{ECTX-DCON-TT} \quad \Sigma \vdash \Psi \quad \Sigma \vdash^{\text{ela}} \mu : \star}{\Sigma \vdash \Psi, D : \mu}$	
4691	C.3.6 ALGORITHMIC POLYKINDS	
4692	$\boxed{\Delta \Vdash^{\text{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2 \dashv \Theta}$	(Instantiation)
4693	$\frac{\text{A-INST-REFL} \quad \Delta \Vdash^{\text{u}} \omega_1 \approx \omega_2 \dashv \Theta}{\Delta \Vdash^{\text{inst}} \mu : \omega_1 <: \omega_2 \rightsquigarrow \mu \dashv \Theta}$ $\frac{\text{A-INST-FORALL} \quad \Delta, \hat{\alpha} : \omega_1 \Vdash^{\text{inst}} \mu_1 @ \hat{\alpha} : \eta[a \mapsto \hat{\alpha}] <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta}{\Delta \Vdash^{\text{inst}} \mu_1 : \forall a : \omega_1. \eta <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta}$	
4694	$\frac{\text{A-INST-FORALL-INFER} \quad \Delta, \hat{\alpha} : \omega_1 \Vdash^{\text{inst}} \mu_1 @ \hat{\alpha} : \eta[a \mapsto \hat{\alpha}] <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta}{\Delta \Vdash^{\text{inst}} \mu_1 : \forall \{a : \omega_1\}. \eta <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta}$	
4695	$\boxed{\Delta \Vdash^{\text{kc}} \sigma \Leftarrow \eta \rightsquigarrow \mu \dashv \Theta}$	(Kind Checking)
4696	$\frac{\text{A-KC-SUB} \quad \Delta \Vdash^{\text{k}} \sigma : \eta \rightsquigarrow \mu_1 \dashv \Delta_1 \quad \Delta_1 \Vdash^{\text{inst}} \mu_1 : [\Delta_1] \eta <: [\Delta_1] \omega \rightsquigarrow \mu_2 \dashv \Delta_2}{\Delta \Vdash^{\text{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu_2 \dashv \Delta_2}$	

$$4697 \quad \boxed{\Delta \Vdash^k \sigma : \eta \rightsquigarrow \mu \dashv \Theta} \quad (\text{Kinding})$$

$$4698 \quad \begin{array}{ccc} \text{A-KTT-STAR} & \text{A-KTT-NAT} & \text{A-KTT-VAR} \\ \hline \Delta \Vdash^k \star : \star \rightsquigarrow \star \dashv \Delta & \Delta \Vdash^k \text{Int} : \star \rightsquigarrow \text{Int} \dashv \Delta & \frac{(a : \omega) \in \Delta}{\Delta \Vdash^k a : \omega \rightsquigarrow a \dashv \Delta} \end{array}$$

$$4699 \quad \begin{array}{cc} \text{A-KTT-TCON} & \text{A-KTT-ARROW} \\ \hline \frac{(T : \eta) \in \Delta}{\Delta \Vdash^k T : \eta \rightsquigarrow T \dashv \Delta} & \frac{}{\Delta \Vdash^k \rightarrow : \star \rightarrow \star \rightarrow \star \rightsquigarrow \rightarrow \dashv \Delta} \end{array}$$

$$4700 \quad \frac{\text{A-KTT-FORALL} \quad \Delta \Vdash^{\text{kc}} \kappa \Leftarrow \star \rightsquigarrow \omega \dashv \Delta_1 \quad \Delta_1, a : \omega \Vdash^{\text{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu \dashv \Delta_2, a : \omega, \Delta_3 \quad \Delta_3 \hookrightarrow a}{\Delta \Vdash^k \forall a : \kappa. \sigma : \star \rightsquigarrow \forall a : \omega. [\Delta_3]\mu \dashv \Delta_2, \text{unsolved}(\Delta_3)}$$

$$4701 \quad \frac{\text{A-KTT-APP} \quad \Delta \Vdash^k \tau_1 : \eta_1 \rightsquigarrow \rho_1 \dashv \Delta_1 \quad \Delta_1 \Vdash^{\text{kapp}} (\rho_1 : [\Delta_1]\eta_1) \bullet \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^k \tau_1 \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}$$

$$4702 \quad \frac{\text{A-KTT-FORALLI} \quad \Delta, \hat{\alpha} : \star, a : \hat{\alpha} \Vdash^{\text{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu \dashv \Delta_2, a : \hat{\alpha}, \Delta_3 \quad \Delta_3 \hookrightarrow a}{\Delta \Vdash^k \forall a. \sigma : \star \rightsquigarrow \forall a : \hat{\alpha}. [\Delta_3]\mu \dashv \Delta_2, \text{unsolved}(\Delta_3)}$$

$$4703 \quad \frac{\text{A-KTT-KAPP} \quad \Delta \Vdash^k \tau_1 : \eta \rightsquigarrow \rho_1 \dashv \Delta_1 \quad [\Delta_1]\eta = \forall a : \omega. \eta_2 \quad \Delta_1 \Vdash^{\text{kc}} \tau_2 \Leftarrow \omega \rightsquigarrow \rho_2 \dashv \Delta_2}{\Delta \Vdash^k \tau_1 @ \tau_2 : \eta_2[a \mapsto \rho_2] \rightsquigarrow \rho_1 @ \rho_2 \dashv \Delta_2}$$

$$4704 \quad \frac{\text{A-KTT-KAPP-INFER} \quad \Delta \Vdash^k \tau_1 : \eta \rightsquigarrow \rho_1 \dashv \Delta_1 \quad [\Delta_1]\eta = \forall \{\overline{a_i : \omega_i^i}\}. \forall a : \omega. \eta_2 \quad \frac{\Delta_1, \hat{\alpha}_i : \omega_i[\overline{a_i \mapsto \hat{\alpha}_i^i}] \Vdash^{\text{kc}} \tau_2 \Leftarrow \omega[\overline{a_i \mapsto \hat{\alpha}_i^i}] \rightsquigarrow \rho_2 \dashv \Delta_2}{\Delta \Vdash^k \tau_1 @ \tau_2 : \eta_2[\overline{a_i \mapsto \hat{\alpha}_i^i}][a \mapsto \rho_2] \rightsquigarrow \rho_1 @ \overline{\hat{\alpha}_i^i} @ \rho_2 \dashv \Delta_2}$$

$$4705 \quad \boxed{\Delta \Vdash^{\text{kapp}} (\rho_1 : \eta) \bullet \tau : \omega \rightsquigarrow \rho_2 \dashv \Theta} \quad (\text{Application Kinding})$$

$$4706 \quad \frac{\text{A-KAPP-TT-ARROW} \quad \Delta \Vdash^{\text{kc}} \tau \Leftarrow \omega_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta \Vdash^{\text{kapp}} (\rho_1 : \omega_1 \rightarrow \omega_2) \bullet \tau : \omega_2 \rightsquigarrow \rho_1 \rho_2 \dashv \Theta}$$

$$4707 \quad \frac{\text{A-KAPP-TT-FORALL} \quad \Delta, \hat{\alpha} : \omega_1 \Vdash^{\text{kapp}} (\rho_1 @ \hat{\alpha} : \eta[a \mapsto \hat{\alpha}]) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\text{kapp}} (\rho_1 : \forall a : \omega_1. \eta) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}$$

A-KAPP-TT-FORALL-INFER

$$\frac{\Delta, \hat{\alpha} : \omega_1 \Vdash^{\text{kapp}} (\rho_1 @ \hat{\alpha} : \eta[a \mapsto \hat{\alpha}]) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\text{kapp}} (\rho_1 : \forall\{a : \omega_1\}.\eta) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}$$

4708

A-KAPP-TT-KUVAR

$$\frac{\Delta_1, \hat{\alpha}_1 : \star, \hat{\alpha}_2 : \star, \hat{\alpha} : \omega = (\hat{\alpha}_1 \rightarrow \hat{\alpha}_2), \Delta_2 \Vdash^{\text{kc}} \tau \Leftarrow \hat{\alpha}_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta_1, \hat{\alpha} : \omega, \Delta_2 \Vdash^{\text{kapp}} (\rho_1 : \hat{\alpha}) \bullet \tau : \hat{\alpha}_2 \rightsquigarrow \rho_1 \rho_2 \dashv \Theta}$$

4709

$$4710 \quad \boxed{\Delta \Vdash^{\text{ela}} \mu : \eta} \quad (\text{Elaborated Kinding})$$

A-ELA-STAR

$$\frac{}{\Delta \Vdash^{\text{ela}} \star : \star}$$

4711

A-ELA-KUVAR

$$\frac{(\hat{\alpha} : \omega) \in \Delta}{\Delta \Vdash^{\text{ela}} \hat{\alpha} : [\Delta]\omega}$$

A-ELA-NAT

$$\frac{}{\Delta \Vdash^{\text{ela}} \text{Int} : \star}$$

A-ELA-VAR

$$\frac{(a : \omega) \in \Delta}{\Delta \Vdash^{\text{ela}} a : [\Delta]\omega}$$

A-ELA-TCON

$$\frac{(T : \eta) \in \Delta}{\Delta \Vdash^{\text{ela}} T : [\Delta]\eta}$$

4712

ELA-ARROW

$$\frac{}{\Delta \Vdash^{\text{ela}} \rightarrow : \star \rightarrow \star \rightarrow \star}$$

A-ELA-FORALL

$$\frac{\Delta \Vdash^{\text{ela}} \omega : \star \quad \Delta, a : \omega \Vdash^{\text{ela}} \mu : \star}{\Delta \Vdash^{\text{ela}} \forall a : \omega. \mu : \star}$$

A-ELA-FORALL-INFER

$$\frac{\Delta \Vdash^{\text{ela}} \omega : \star \quad \Delta, a : \omega \Vdash^{\text{ela}} \mu : \star}{\Delta \Vdash^{\text{ela}} \forall\{a : \omega\}.\mu : \star}$$

4713

A-ELA-APP

$$\frac{\Delta \Vdash^{\text{ela}} \rho_1 : \omega_1 \rightarrow \omega_2 \quad \Delta \Vdash^{\text{ela}} \rho_2 : \omega_1}{\Delta \Vdash^{\text{ela}} \rho_1 \rho_2 : \omega_2}$$

A-ELA-KAPP

$$\frac{\Delta \Vdash^{\text{ela}} \rho_1 : \forall a : \omega. \eta \quad \Delta \Vdash^{\text{ela}} \rho_2 : \omega}{\Delta \Vdash^{\text{ela}} \rho_1 @ \rho_2 : \eta[a \mapsto [\Delta]\rho_2]}$$

4714

A-ELA-KAPP-INFER

$$\frac{\Delta \Vdash^{\text{ela}} \rho_1 : \forall\{a : \omega\}.\eta \quad \Delta \Vdash^{\text{ela}} \rho_2 : \omega}{\Delta \Vdash^{\text{ela}} \rho_1 @ \rho_2 : \eta[a \mapsto [\Delta]\rho_2]}$$

$$4715 \quad \boxed{\Delta \Vdash_{\phi^c}^{\text{gen}} \Gamma_1 \rightsquigarrow \Gamma_2} \quad (\text{Generalization})$$

A-GEN

$$\frac{\overline{\widehat{\phi}_i^c = \text{unsolved}(\mu_i)}^i}{\Delta \Vdash_{\phi^c}^{\text{gen}} \overline{D_i : \mu_i}^i \rightsquigarrow \overline{D : \forall\{\phi^c\}.\forall\{\phi_i^c\}.\mu[\widehat{\phi}_i^c \mapsto \phi_i^c]}^i}$$

4716

$$4717 \quad \boxed{\Delta \text{ ok}} \quad (\text{Well-formed Type Contexts})$$

4718	$\frac{\text{A-TCTX-EMPTY}}{\bullet \text{ ok}}$	$\frac{\text{A-TCTX-TVAR-TT} \quad \Delta \text{ ok} \quad \Delta \Vdash^{\text{ela}} \omega : \star}{\Delta, a : \omega \text{ ok}}$	$\frac{\text{A-TCTX-TCON-TT} \quad \Delta \text{ ok} \quad \Delta \Vdash^{\text{ela}} \eta : \star}{\Delta, T : \eta \text{ ok}}$
4719	$\frac{\text{A-TCTX-KUVAR-TT} \quad \Delta \text{ ok} \quad \Delta \Vdash^{\text{ela}} \omega : \star}{\Delta, \hat{\alpha} : \omega \text{ ok}}$	$\frac{\text{A-TCTX-KUVAR}^{\text{SOLVED}}\text{-TT} \quad \Delta \text{ ok} \quad \Delta \Vdash^{\text{ela}} \omega_2 : [\Delta]\omega_1}{\Delta, \hat{\alpha} : \omega_1 = \omega_2 \text{ ok}}$	$\frac{\text{A-TCTX-LO} \quad \Delta, \Delta^{\text{lo}} \text{ ok}}{\Delta, \{\Delta^{\text{lo}}\} \text{ ok}}$
4720	$\frac{\text{A-TCTX-MARKER} \quad \Delta \text{ ok}}{\Delta, \blacktriangleright_D \text{ ok}}$		
4721	$\boxed{\Delta \Vdash^{\text{ectx}} \Gamma}$	(Well-formed Term Contexts)	
4722	$\frac{\text{A-ECTX-EMPTY}}{\Delta \Vdash^{\text{ectx}} \bullet}$	$\frac{\text{A-ECTX-DCON-TT} \quad \Delta \Vdash^{\text{ectx}} \Gamma \quad \Delta \Vdash^{\text{ela}} \mu : \star}{\Delta \Vdash^{\text{ectx}} \Gamma, D : \mu}$	
4723	$\boxed{\Delta \Vdash^{\mu} \omega_1 \approx \omega_2 \dashv \Theta}$	(Unification)	
4724	$\frac{\text{A-U-REFL-TT}}{\Delta \Vdash^{\mu} \omega \approx \omega \dashv \Delta}$	$\frac{\text{A-U-APP} \quad \Delta \Vdash^{\mu} \rho_1 \approx \rho_3 \dashv \Delta_1 \quad \Delta_1 \Vdash^{\mu} [\Delta_1]\rho_2 \approx [\Delta_1]\rho_4 \dashv \Theta}{\Delta \Vdash^{\mu} \rho_1 \rho_2 \approx \rho_3 \rho_4 \dashv \Theta}$	
4725	$\frac{\text{A-U-KAPP} \quad \Delta \Vdash^{\mu} \rho_1 \approx \rho_3 \dashv \Delta_1 \quad \Delta_1 \Vdash^{\mu} [\Delta_1]\rho_2 \approx [\Delta_1]\rho_4 \dashv \Theta}{\Delta \Vdash^{\mu} \rho_1 @ \rho_2 \approx \rho_3 @ \rho_4 \dashv \Theta}$		
4726	$\frac{\text{A-U-KVARL-TT} \quad \Delta \Vdash_{\hat{\alpha}}^{\text{pr}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \hat{\alpha} : \omega_1, \Theta_2 \quad \Theta_1 \Vdash^{\text{ela}} \rho_2 : \omega_2 \quad \Theta_1 \Vdash^{\mu} [\Theta_1]\omega_1 \approx \omega_2 \dashv \Theta_3}{\Delta \Vdash^{\mu} \hat{\alpha} \approx \rho_1 \dashv \Theta_3, \hat{\alpha} : \omega_1 = \rho_2, \Theta_2}$		
4727	$\frac{\text{A-U-KVARR-TT} \quad \Delta \Vdash_{\hat{\alpha}}^{\text{pr}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \hat{\alpha} : \omega_1, \Theta_2 \quad \Theta_1 \Vdash^{\text{ela}} \rho_2 : \omega_2 \quad \Theta_1 \Vdash^{\mu} [\Theta_1]\omega_1 \approx \omega_2 \dashv \Theta_3}{\Delta \Vdash^{\mu} \rho_1 \approx \hat{\alpha} \dashv \Theta_3, \hat{\alpha} : \omega_1 = \rho_2, \Theta_2}$		

4728	$\frac{\text{A-U-KVARL-LO-TT} \quad \begin{array}{c} \Delta_1, \Delta_2 \vdash^{\text{mv}} \hat{\alpha} : \omega_1 \rightsquigarrow \Theta \quad \Delta[\{\Theta\}] \vdash_{\hat{\alpha}}^{\text{pr}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \{\Theta_2, \hat{\alpha} : \omega_1, \Theta_3\}, \Theta_4 \\ \Theta_1, \{\Theta_2\} \Vdash^{\text{ela}} \rho_2 : \omega_2 \quad \Theta_1, \{\Theta_2\} \Vdash^{\text{u}} [\Theta_1, \Theta_2] \omega_1 \approx \omega_2 \dashv \Theta_5, \{\Theta_6\} \end{array}}{\Delta[\{\Delta_1, \hat{\alpha} : \omega_1, \Delta_2\}] \Vdash^{\text{u}} \hat{\alpha} \approx \rho_1 \dashv \Theta_5, \{\Theta_6, \hat{\alpha} : \omega_1 = \rho_2, \Theta_3\}, \Theta_4}$	
4729	$\frac{\text{A-U-KVARR-LO-TT} \quad \begin{array}{c} \Delta_1, \Delta_2 \vdash^{\text{mv}} \hat{\alpha} : \omega_1 \rightsquigarrow \Theta \quad \Delta[\{\Theta\}] \vdash_{\hat{\alpha}}^{\text{pr}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \{\Theta_2, \hat{\alpha} : \omega_1, \Theta_3\}, \Theta_4 \\ \Theta_1, \{\Theta_2\} \Vdash^{\text{ela}} \rho_2 : \omega_2 \quad \Theta_1, \{\Theta_2\} \Vdash^{\text{u}} [\Theta_1, \Theta_2] \omega_1 \approx \omega_2 \dashv \Theta_5, \{\Theta_6\} \end{array}}{\Delta[\{\Delta_1, \hat{\alpha} : \omega_1, \Delta_2\}] \Vdash^{\text{u}} \rho_1 \approx \hat{\alpha} \dashv \Theta_5, \{\Theta_6, \hat{\alpha} : \omega_1 = \rho_2, \Theta_3\}, \Theta_4}$	
4730	$\boxed{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta} \quad (\text{Promotion})$	
4731	$\begin{array}{ccc} \text{A-PR-STAR} & \text{A-PR-ARROW} & \text{A-PR-TCON} \\ \hline \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \star \rightsquigarrow \star \dashv \Delta & \Delta \Vdash_{\hat{\alpha}}^{\text{pr}} \rightarrow \rightsquigarrow \rightarrow \dashv \Delta & \Delta[T][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\text{pr}} T \rightsquigarrow T \dashv \Delta[T][\hat{\alpha}] \end{array}$	
4732	$\begin{array}{ccc} \text{A-PR-NAT} & \text{A-PR-APP} & \\ \hline \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \text{Int} \rightsquigarrow \text{Int} \dashv \Delta & \frac{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 \rightsquigarrow \rho_1 \dashv \Delta_1 \quad \Delta_1 \vdash_{\hat{\alpha}}^{\text{pr}} [\Delta_1] \omega_2 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 \omega_2 \rightsquigarrow \rho_1 \rho_2 \dashv \Theta} & \end{array}$	
4733	$\begin{array}{ccc} \text{A-PR-KAPP} & & \text{A-PR-TVAR} \\ \hline \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 \rightsquigarrow \rho_1 \dashv \Delta_1 \quad \Delta_1 \vdash_{\hat{\alpha}}^{\text{pr}} [\Delta_1] \omega_2 \rightsquigarrow \rho_2 \dashv \Theta & & \Delta[a][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\text{pr}} a \rightsquigarrow a \dashv \Delta[a][\hat{\alpha}] \\ \hline \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 @ \omega_2 \rightsquigarrow \rho_1 @ \rho_2 \dashv \Theta & & \end{array}$	
4734	$\begin{array}{ccc} \text{A-PR-KUVARL} & \text{A-PR-KUVARR-TT} & \\ \hline \Delta[\hat{\beta}][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \Delta[\hat{\beta}][\hat{\alpha}] & \frac{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} [\Delta] \rho \rightsquigarrow \rho_1 \dashv \Theta[\hat{\alpha}][\hat{\beta} : \rho]}{\Delta[\hat{\alpha}][\hat{\beta} : \rho] \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta}_1 \dashv \Theta[\hat{\beta}_1 : \rho_1, \hat{\alpha}][\hat{\beta} : \rho = \hat{\beta}_1]} & \end{array}$	
4735	$\boxed{\Delta_1 \vdash^{\text{mv}} \Delta_2 \rightsquigarrow \Theta} \quad (\text{Moving})$	
4736	$\begin{array}{ccc} \text{A-MV-EMPTY} & \text{A-MV-KUVAR} & \\ \hline \bullet \vdash^{\text{mv}} \Delta \rightsquigarrow \Delta & \frac{\text{var}(\omega) \# \text{dom}(\Delta_2) \quad \Delta_1 \vdash^{\text{mv}} \Delta_2 \rightsquigarrow \Theta}{\hat{\alpha} : \omega, \Delta_1 \vdash^{\text{mv}} \Delta_2 \rightsquigarrow \hat{\alpha} : \omega, \Theta} & \end{array}$	
4737	$\frac{\text{A-MV-KUVARM} \quad \neg(\text{var}(\omega) \# \text{dom}(\Delta_2)) \quad \Delta_1 \vdash^{\text{mv}} \Delta_2, \hat{\alpha} : \omega \rightsquigarrow \Theta}{\hat{\alpha} : \omega, \Delta_1 \vdash^{\text{mv}} \Delta \rightsquigarrow \Theta}$	

$$\begin{array}{c}
 \text{A-MV-TVAR} \\
 \frac{\text{var}(\omega) \# \text{dom}(\Delta_2) \quad \Delta_1 ++^{\text{mv}} \Delta_2 \leadsto \Theta}{a : \omega, \Delta_1 ++^{\text{mv}} \Delta_2 \leadsto a : \omega, \Theta} \\
 4738 \\
 \\
 \text{A-MV-TVARM} \\
 \frac{\neg(\text{var}(\omega) \# \text{dom}(\Delta_2)) \quad \Delta_1 ++^{\text{mv}} \Delta_2, a : \omega \leadsto \Theta}{a : \omega, \Delta_1 ++^{\text{mv}} \Delta_2 \leadsto \Theta} \\
 4739
 \end{array}$$

4740 C.3.7 CONTEXT APPLICATION IN POLYKINDS

$$\begin{array}{c}
 \hline
 [\Delta]\eta \text{ applies } \Delta \text{ as a substitution to } \eta. \\
 \begin{array}{lcl}
 [\Delta]\star & = & \star \\
 [\Delta]\text{Int} & = & \text{Int} \\
 [\Delta]a & = & a \\
 [\Delta]T & = & T \\
 [\Delta] \rightarrow & = & \rightarrow \\
 4741 \quad [\Delta]\forall a : \omega. \eta & = & \forall a : [\Delta]\omega. [\Delta]\eta \\
 [\Delta]\forall \{a : \omega\}. \eta & = & \forall \{a : [\Delta]\omega\}. [\Delta]\eta \\
 [\Delta](\rho_1 \rho_2) & = & ([\Delta]\rho_1) ([\Delta]\rho_2) \\
 [\Delta](\rho_1 @ \rho_2) & = & ([\Delta]\rho_1) @ ([\Delta]\rho_2) \\
 [\Delta][\hat{\alpha}]\hat{\alpha} & = & \hat{\alpha} \\
 [\Delta][\hat{\alpha} : \omega = \rho]\hat{\alpha} & = & [\Delta][\hat{\alpha} : \omega = \rho]\rho
 \end{array} \\
 \hline
 \\
 [\Delta]\Gamma \text{ applies } \Delta \text{ as a substitution to } \Gamma. \\
 4742 \quad \begin{array}{lcl}
 [\Omega]\bullet & = & \bullet \\
 [\Omega](\Gamma, D : \mu) & = & [\Omega]\Gamma, D : [\Omega]\mu
 \end{array} \\
 \hline
 \\
 [\Omega]\Delta \text{ applies } \Omega \text{ as a substitution to } \Delta. \\
 \begin{array}{lcl}
 [\Omega]\bullet & = & \bullet \\
 [\Omega, a : \omega](\Delta, a : \omega) & = & [\Omega]\Delta, a : [\Omega]\omega \\
 [\Omega, T : \omega](\Delta, T : \omega) & = & [\Omega]\Delta, T : [\Omega]\omega \\
 4743 \quad [\Omega, \hat{\alpha} : \omega = \rho](\Delta, \hat{\alpha} : \omega) & = & [\Omega]\Delta \\
 [\Omega, \hat{\alpha} : \omega = \rho_1](\Delta, \hat{\alpha} : \omega = \rho_2) & = & [\Omega]\Delta \quad \text{if } [\Omega]\rho_1 = [\Omega]\rho_2 \\
 [\Omega, \hat{\alpha} : \omega = \rho]\Delta & = & [\Omega]\Delta \quad \text{if } \hat{\alpha} \notin \Delta \\
 [\Omega, \blacktriangleright_D](\Delta, \blacktriangleright_D) & = & [\Omega]\Delta \\
 [\Omega, \{\Omega_1\}](\Delta, \{\Delta_1\}) & = & [\Omega, \Omega_1](\Delta, \Delta') \\
 & & \text{where } \Delta' = \text{topo}(\Delta_1)
 \end{array} \\
 \hline
 \end{array}$$

4744 C.3.8 CONTEXT EXTENSION IN POLYKINDS

4745 $\boxed{\Delta \longrightarrow \Theta}$ (Context Extension)

$$\begin{array}{c}
\text{A-CTXE-EMPTY} \qquad \text{A-CTXE-TVAR-TT} \qquad \text{A-CTXE-TCON-TT} \\
\hline
\bullet \longrightarrow \bullet \qquad \Delta \longrightarrow \Theta \qquad \Delta \longrightarrow \Theta \\
\hline
\bullet \longrightarrow \bullet \qquad \Delta, a : \omega \longrightarrow \Theta, a : \omega \qquad \Delta, T : \eta \longrightarrow \Theta, T : \eta
\end{array}$$

4746

$$\begin{array}{c}
\text{A-CTXE-KUVAR-TT} \qquad \text{A-CTXE-KUVARsOLVED-TT} \\
\hline
\Delta \longrightarrow \Theta \qquad \Delta \longrightarrow \Theta \quad [\Theta]\rho_1 = [\Theta]\rho_2 \\
\hline
\Delta, \hat{\alpha} : \omega \longrightarrow \Theta, \hat{\alpha} : \omega \qquad \Delta, \hat{\alpha} : \omega = \rho_1 \longrightarrow \Theta, \hat{\alpha} : \omega = \rho_2
\end{array}$$

4747

$$\begin{array}{c}
\text{A-CTXE-SOLVE-TT} \qquad \text{A-CTXE-ADD-TT} \\
\hline
\Delta \longrightarrow \Theta \quad \Theta \Vdash^{\text{ela}} \rho : [\Theta]\omega \qquad \Delta \longrightarrow \Theta \quad \Theta \Vdash^{\text{ela}} \omega : \star \\
\hline
\Delta, \hat{\alpha} : \omega \longrightarrow \Theta, \hat{\alpha} : \omega = \rho \qquad \Delta \longrightarrow \Theta, \hat{\alpha} : \omega
\end{array}$$

4748

$$\begin{array}{c}
\text{A-CTXE-ADDsOLVED-TT} \qquad \text{A-CTXE-MARKER} \\
\hline
\Delta \longrightarrow \Theta \quad \Theta \Vdash^{\text{ela}} \rho : [\Theta]\omega \qquad \Delta \longrightarrow \Theta \\
\hline
\Delta \longrightarrow \Theta, \hat{\alpha} : \omega = \rho \qquad \Delta, \blacktriangleright_D \longrightarrow \Theta, \blacktriangleright_D
\end{array}$$

4749

$$\begin{array}{c}
\text{A-CTXE-LO} \\
\hline
\Delta \longrightarrow \Theta \quad \Delta, \text{topo}(\Delta_1) \longrightarrow \Theta, \Theta_1 \\
\hline
\Delta, \{\Delta_1\} \longrightarrow \Theta, \{\Theta_1\}
\end{array}$$

4750