

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	24
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 ALGORITHM 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	122
6.1.3	Polymorphic Promotion	123
6.2	Unification for the Simply Typed Lambda Calculus	125
6.2.1	Declarative System	126
6.2.2	Algorithmic System	126
6.2.3	Soundness and Completeness	130
6.3	Subtyping for Higher-Rank Polymorphism	131
6.3.1	Declarative System	132
6.3.2	Algorithmic System	132
6.3.3	Soundness and Completeness	137
6.4	Discussion	138
6.4.1	Promoting Dependent Types	139
6.4.2	Promoting Gradual Types	140
6.5	Promotion in Action	140
6.6	Levels and Generalization	143
7	KIND INFERENCE FOR DATATYPES	145
7.1	Introduction and Motivation	145
7.2	Overview	147
7.2.1	Kind Inference in Haskell98	147
7.2.2	Kind Inference in Modern GHC Haskell	148
7.2.3	Desirable Properties for Kind Inference	152
7.3	Datatypes in Haskell98	152
7.3.1	Groups and Dependency Analysis	152
7.3.2	Declarative Typing Rules	152
7.4	Kind Inference for Haskell98	154
7.4.1	Syntax	154
7.4.2	Algorithmic Typing Rules	155
7.4.3	Defaulting	157
7.4.4	Checking Datatype Declarations	158
7.4.5	Kinding	160
7.4.6	Unification	160

7.4.7	Soundness and Completeness	161
7.5	Type Parameters, Principal Kinds and Completeness in Haskell98	162
7.5.1	Type Parameters	162
7.5.2	Principal Kinds and Defaulting	163
7.5.3	Completeness	163
7.6	Declarative Syntax and Semantics of PolyKinds	164
7.6.1	Groups and Dependency Analysis	164
7.6.2	Checking Programs	165
7.6.3	Checking Kinds	170
7.7	Kind Inference for PolyKinds	171
7.7.1	Algorithmic Program Typing	172
7.7.2	The Quantification Check	174
7.7.3	Kinding	175
7.7.4	Unification	175
7.7.5	Termination	179
7.7.6	Soundness, Completeness and Principality	181
7.8	Language Extensions	182
7.8.1	Higher-Rank Polymorphism	182
7.8.2	Generalized Algebraic Datatypes (GADTs)	183
7.8.3	Type Families	184
V	EPILOGUE	187
8	RELATED WORK	189
8.1	Type Inference for Higher-Rank Types	189
8.2	Bidirectional Type Checking	190
8.3	Gradual Typing	191
8.4	Gradual Type Systems with Explicit Polymorphism	192
8.5	Gradual Type Inference	193
8.6	Haskell and GHC	193
8.7	Unification with dependent types	194
9	SUMMARY AND FUTURE DIRECTIONS	197
9.1	Dependent Type Systems with Application mode	197
9.2	Type Inference for Intersection Type Systems	198
9.3	Gradualizing Type Classes	199

9.4	Generalized Algebraic Datatypes (GADTs)	200
	BIBLIOGRAPHY	203
VI	TECHNICAL APPENDIX	219
A	FULL RULES FOR ALGORITHMIC AP	221
B	THE EXTENDED ALGORITHMIC GPC	225
B.1	Syntax	225
B.2	Type System	225
C	KIND INFERENCE FOR DATATYPES	229
C.1	Other Language Extensions	229
C.1.1	Visible Dependent Quantification	229
C.1.2	Datatype Promotion	230
C.1.3	Partial Type Signatures	230
C.2	Today’s GHC	231
C.2.1	Constraint-Based Type Inference	231
C.2.2	Contexts	231
C.2.3	Unification	232
C.2.4	Complete User-Supplied Kinds	233
C.2.5	Dependency Analysis	234
C.2.6	Approach to Kind-Checking Datatypes	234
C.2.7	Polymorphic Recursion	235
C.2.8	The Quantification Check	236
C.2.9	ScopedSort	237
C.2.10	The “Forall-or-Nothing” Rule	238
C.3	Complete Set of Rules	238
C.3.1	Declarative Haskell98	238
C.3.2	Algorithmic Haskell98	239
C.3.3	Declarative PolyKinds	240
C.3.4	Algorithmic PolyKinds	240
C.3.5	Context Application in PolyKinds	244
C.3.6	Context Extension in PolyKinds	245

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	25
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	Syntax and declarative typing of the simply typed lambda calculus.	126
6.2	Syntax of algorithmic STLC	127
6.3	Algorithmic typing, promotion and unification of algorithmic STLC	128
6.4	Context extension	130
6.5	Syntax and subtyping for higher-rank polymorphism	132
6.6	Types, contexts, and well-formedness of the algorithmic system	133
6.7	Unification, subtyping and polymorphic promotion of the algorithmic system)	135
7.1	Declarative specification of Haskell98 datatype declarations	153
7.2	Algorithmic program typing in Haskell98	156
7.3	Defaulting in Haskell98.	157
7.4	Algorithmic kinding, unification and promotion in Haskell98.	159
7.5	Context extension in Haskell98	161
7.6	Syntax of PolyKinds	165
7.7	Declarative specification of PolyKinds	166
7.8	Auxiliary functions of declarative specification in PolyKinds	167
7.9	Declarative kind-checking in PolyKinds	169
7.10	Declarative elaborated kinding in PolyKinds	170
7.11	Algorithmic syntax in PolyKinds	171
7.12	Algorithmic program typing in PolyKinds	173
7.13	Unsolved set of unification variables and quantification check in PolyKinds	174
7.14	Selected rules for algorithmic kinding in PolyKinds	176
7.15	Selected rules for unification, promotion, and moving in PolyKinds	177
7.16	Example of dependency graph	180

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

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

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 $\overline{a_i}^i$ 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 :  $\forall a. a \rightarrow a$ . (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 :  $\forall a. a \rightarrow a$ . (f 1, f 'a')) : ( $\forall a. a \rightarrow a$ )  $\rightarrow$  (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-FORALL** 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. This is one of the reasons that the *Glasgow Haskell Compiler* (GHC), the state-of-the-art Haskell compiler, has recently removed co/contravariant arrow from the language [Peyton Jones 2020].

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

2 Background

could easily deal with polymorphic languages with subtyping. The key idea in local type inference is simple. The "local" in local type inference comes from the fact that:

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

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

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

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

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

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

```
((\f. (f 1, f 'c')) : ( $\forall a. a \rightarrow a$ )  $\rightarrow$  (Int, Char)) (\x . x)
```

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

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

2.3.2 DECLARATIVE SYSTEM

SYNTAX. The syntax of the DK is given in Figure 2.6. Comparing to OL, only the definition of expressions slightly differs. First, the expressions e in DK have no let expressions.

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

Dunfield and Krishnaswami [2013] omitted let-bindings from the formal development, but argued that restoring let-bindings is easy, as long as they get no special treatment incompatible with substitution (e.g., a syntax-directed HM does polymorphic generalization only at let-bindings). Second, DK has annotated expressions $e : \sigma$ (instead of annotated lambda expressions $\lambda x : \sigma. e$), in which the type annotation can be propagated into the expression, as we will see shortly.

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

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

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

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

Rule **DK-CHK-INT** checks literals against the integer type `Int`. Rule **DK-CHK-LAM** is where the system benefits from bidirectional type checking: the type information gets pushed inside an lambda. For an unannotated lambda abstraction $\lambda x. e$, recall that in the inference

2 Background

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

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

mode, we can only guess a monotype for x . With the checking mode, when $\lambda x. e$ is checked against $\sigma_1 \rightarrow \sigma_2$, we do not need to guess any type. Instead, x gets directly the (possibly polymorphic) argument type σ_1 . Then the rule proceeds by checking e with σ_2 , allowing the type information to be pushed further inside. Note how rule **DK-CHK-LAM** improves over HM and OL, by allowing lambda abstractions to have a polymorphic argument type without requiring type annotations.

Rule **DK-CHK-GEN** deals with a polymorphic type $\forall a. \sigma$, by putting the (fresh) type variable a into the context to check e against σ . Rule **DK-CHK-SUB** switches the mode from checking to inference: an expression e can be checked against σ_2 , if e infers the type σ_1 and σ_1 is a subtype of σ_2 .

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 monotype, until the type becomes a function type (rule **DK-APP-ARR**). In the function type case, since the function expects an argument of type σ_1 , the rule proceeds by checking e_2 against σ_1 .

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

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

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.

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

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

2 Background

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

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

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

621 2.3.3 ALGORITHMIC TYPE SYSTEM

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

627 2.3.4 DISCUSSION: LAZY AND EAGER INSTANTIATION

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

633 The differences between lazy and eager instantiation have pervasive consequences. The
634 first and direct consequence is that for the same expression lazy instantiation can derive a
635 more polymorphic type. Consider that given $\text{id}: \forall a. a \rightarrow a$, we want to infer the type of
636 id . With lazy instantiation, the algorithm can directly return $\forall a. a \rightarrow a$ unchanged, while
637 eager instantiation can only return $a \rightarrow a$ (among others). In this case, we can make two
638 types match by generalizing the final result from eager instantiation, which gives us $\forall a.$
639 $a \rightarrow a$. However, when the system features higher-rank types, after generalization eager
640 instantiation may still fail to derive the same type. For example, consider the expression

641 $\backslash x: \mathbf{Int}. \text{id}$

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

647 $\text{let } g = \backslash x: \mathbf{Int}. \text{id} \text{ in } (\backslash f: \mathbf{Int} \rightarrow \forall a. a \rightarrow a. f) g$

648 The problem can be avoided by, for example, featuring *deep skolemisation* as in Peyton Jones
 649 et al. [2007]. Deep skolemisation floats out all its universal quantifiers that appear to the
 650 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
 651 isomorphic.

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

657 $\text{if } \text{flag} \text{ then } \text{id} \text{ else } \backslash x. x + 1$

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

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

667 $\text{let } f = \backslash x. x$
 668 $\text{in let } g = \backslash y. f$
 669 $\text{in } e$

670 If we generalize the type of `f` as in HM, then we get $f: \forall a. a \rightarrow a$. Now we continue type-
 671 checking `g`, which requires another step of generalization. With lazy instantiation, we get
 672 $g: \forall b. b \rightarrow \forall a. a \rightarrow a$, which is again, more polymorphic than the type $g: \forall b a. b \rightarrow$
 673 $a \rightarrow a$ we get from eager instantiation. However, if we inline the definition of `f`, then the
 674 program becomes

2 Background

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

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

682 In this thesis, the choice does not matter so much for the key contributions of the work.
683 As we do not feature deep skolemisation, we mostly follow the idea of lazy instantiation as
684 in DK. Nevertheless, we believe that deep skolemisation is compatible with our subtyping
685 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-

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

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

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

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

806 3.1.3 BENEFITS OF INFORMATION FLOWING FROM ARGUMENTS TO FUNCTIONS

807 **LOCAL CONSTRAINT SOLVER FOR FUNCTION VARIABLES.** Many type systems, including
 808 type systems with *implicit polymorphism* and/or *static overloading*, need information about
 809 the types of the arguments when type checking function variables. For example, in conven-
 810 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.

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

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

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

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

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

3.1.4 TYPE INFERENCE OF HIGHER-RANK TYPES

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

840 Consider again the motivation example used in Section 2.2:

841 $(\lambda f. (f \ 1, f \ \text{'a'})) (\lambda x. x)$

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:

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

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.

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

907 3.2.1 SYNTAX

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

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

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

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

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

925 3.2.2 TYPE SYSTEM

926 The top part of Figure 3.2 gives the typing rules for our language. The judgment $\Psi; \Sigma \vdash^{AP}$
 927 $e \Rightarrow \sigma$ 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)
$\frac{\text{AP-INF-INT}}{\Psi \vdash^{AP} n \Rightarrow \text{Int}}$	$\frac{\text{AP-INF-LAM}}{\Psi, x : \tau \vdash^{AP} e \Rightarrow \sigma} \quad \Psi \vdash^{AP} \lambda x. e \Rightarrow \tau \rightarrow \sigma$	$\frac{\text{AP-INF-LAMANN}}{\Psi \vdash^{AP} \lambda x : \sigma_1. e \Rightarrow \sigma_1 \rightarrow \sigma_2} \quad \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_2$	
$\boxed{\Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma}$			(Typing Application Mode)
$\frac{\text{AP-APP-VAR}}{(x : \sigma_1) \in \Psi \quad \Sigma \vdash^{AP} \sigma_1 <: \sigma_2} \quad \Psi; \Sigma \vdash^{AP} x \Rightarrow \sigma_2$	$\frac{\text{AP-APP-LAM}}{\Psi; \Sigma, \sigma_1 \vdash^{AP} \lambda x. e \Rightarrow \sigma_1 \rightarrow \sigma_2} \quad \Psi, x : \sigma_1; \Sigma \vdash^{AP} e \Rightarrow \sigma_2$		
$\frac{\text{AP-APP-LAMANN}}{\Psi; \Sigma, \sigma_2 \vdash^{AP} \lambda x : \sigma_1. e \Rightarrow \sigma_2 \rightarrow \sigma_3} \quad \vdash^{AP} \sigma_2 <: \sigma_1 \quad \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_3$			
$\frac{\text{AP-APP-APP}}{\Psi; \Sigma \vdash^{AP} e_1 e_2 \Rightarrow \sigma_3} \quad \begin{array}{l} \Psi \vdash^{AP} e_2 \Rightarrow \sigma_1 \quad \bar{a}_i^i = \text{FV}(\sigma_1) - \text{FV}(\Psi) \\ \sigma_2 = \forall \bar{a}_i^i. \sigma_1 \quad \Psi; \Sigma, \sigma_2 \vdash^{AP} e_1 \Rightarrow \sigma_2 \rightarrow \sigma_3 \end{array}$			
$\boxed{\vdash^{AP} \sigma_1 <: \sigma_2}$			(Subtyping)
$\frac{\text{AP-S-TVAR}}{\vdash^{AP} a <: a}$	$\frac{\text{AP-S-INT}}{\vdash^{AP} \text{Int} <: \text{Int}}$	$\frac{\text{AP-S-ARROW}}{\vdash^{AP} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4} \quad \vdash^{AP} \sigma_3 <: \sigma_1 \quad \vdash^{AP} \sigma_2 <: \sigma_4$	
$\frac{\text{AP-S-FORALLL}}{\vdash^{AP} \forall a. \sigma_1 <: \sigma_2} \quad \vdash^{AP} \sigma_1[a \mapsto \tau] <: \sigma_2$	$\frac{\text{AP-S-FORALLR}}{\vdash^{AP} \sigma_1 <: \forall a. \sigma_2} \quad a \notin \text{FV}(\sigma_1) \quad \vdash^{AP} \sigma_1 <: \sigma_2$		
$\boxed{\Sigma \vdash^{AP} \sigma_1 <: \sigma_2}$			(Application Subtyping)
$\frac{\text{AP-AS-EMPTY}}{\bullet \vdash^{AP} \sigma <: \sigma}$	$\frac{\text{AP-AS-FORALL}}{\Sigma, \sigma_3 \vdash^{AP} \forall a. \sigma_1 <: \sigma_2} \quad \Sigma, \sigma_3 \vdash^{AP} \sigma_1[a \mapsto \tau] <: \sigma_2$	$\frac{\text{AP-AS-ARROW}}{\Sigma, \sigma_3 \vdash^{AP} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4} \quad \vdash^{AP} \sigma_3 <: \sigma_1 \quad \Sigma \vdash^{AP} \sigma_2 <: \sigma_4$	

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:

988 **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*
 989 *$\Psi; \Sigma_2, \Sigma_1 \vdash^{AP} e \Rightarrow \Sigma_1 \rightarrow \Sigma_2 \rightarrow \sigma$.*

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

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

995 3.2.3 SUBTYPING

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

998 **SUBTYPING.** The first subtyping judgment $\vdash^{AP} \sigma_1 <: \sigma_2$ follows OL, but in HM-style; that
 999 is, without tracking type variables. Rule **AP-S-FORALLR** states σ_1 is subtype of $\forall a. \sigma_2$ only
 1000 if σ_1 is a subtype of σ_2 , with the assumption a is a fresh variable. Rule **AP-S-FORALLL** says
 1001 $\forall a. \sigma_1$ is a subtype of σ_2 if we can instantiate it with some τ and show the result is a subtype
 1002 of σ_2 .

1003 **APPLICATION SUBTYPING.** The typing rule **AP-APP-VAR** uses the second subtyping judg-
 1004 ment $\Sigma \vdash^{AP} \sigma_1 <: \sigma_2$. To motivate this new kind of judgment, consider the expression $\text{id } 1$
 1005 for example, whose derivation is stuck at rule **AP-APP-VAR** (here we assume $\text{id} : \forall a. a \rightarrow a \in \Psi$):
 1006 Ψ):

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

1007 Here we know that $\text{id} : \forall a. a \rightarrow a$ and also, from the application context, that id is applied
 1008 to an argument of type Int . Thus we need a mechanism for solving the instantiation $a = \text{Int}$
 1009 and returning a supertype $\text{Int} \rightarrow \text{Int}$ as the type of id . This is precisely what the application
 1010 subtyping achieves: resolving instantiation constraints according to the application context.
 1011 Notice that unlike existing works (Peyton Jones et al. [2007] or DK), application subtyping
 1012 provides a way to solve instantiation more *locally*, since it does not mutually depend on typ-
 1013 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 $\Sigma_2, \Sigma_1 \vdash^{AP} \sigma_1 <: \Sigma_1 \rightarrow \Sigma_2 \rightarrow \sigma_3$.*

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

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

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

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

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

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

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

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

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

3.3 TYPE-DIRECTED TRANSLATION

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

3.3.1 TARGET LANGUAGE

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

3 Higher-Rank Polymorphism with the Application Mode

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)
$\text{AP-S-TVAR} \quad \frac{}{\vdash^{AP} a <: a \rightsquigarrow \lambda x : a. x}$	$\text{AP-S-INT} \quad \frac{}{\vdash^{AP} \text{Int} <: \text{Int} \rightsquigarrow \lambda x : \text{Int}. x}$	
$\text{AP-S-ARROW} \quad \frac{\vdash^{AP} \sigma_3 <: \sigma_1 \rightsquigarrow f_1 \quad \vdash^{AP} \sigma_2 <: \sigma_4 \rightsquigarrow f_2}{\vdash^{AP} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4 \rightsquigarrow \lambda x : \sigma_1 \rightarrow \sigma_2. \lambda y : \sigma_3. f_2(x(f_1 y))}$		
$\text{AP-S-FORALLL} \quad \frac{\vdash^{AP} \sigma_1[a \mapsto \tau] <: \sigma_2 \rightsquigarrow f}{\vdash^{AP} \forall a. \sigma_1 <: \sigma_2 \rightsquigarrow \lambda x : \forall a. \sigma_1. f(x\tau)}$	$\text{AP-S-FORALLR} \quad \frac{a \notin \text{FV}(\sigma_1) \quad \vdash^{AP} \sigma_1 <: \sigma_2 \rightsquigarrow f}{\vdash^{AP} \sigma_1 <: \forall a. \sigma_2 \rightsquigarrow \lambda x : \sigma_1. \Lambda a. f x}$	
$\boxed{\Sigma \vdash^{AP} \sigma_1 <: \sigma_2 \rightsquigarrow f}$		(Application Subtyping)
$\text{AP-AS-EMPTY} \quad \frac{}{\bullet \vdash^{AP} \sigma <: \sigma \rightsquigarrow \lambda x : \sigma. x}$	$\text{AP-AS-FORALL} \quad \frac{\Sigma, \sigma_3 \vdash^{AP} \sigma_1[a \mapsto \tau] <: \sigma_2 \rightsquigarrow f}{\Sigma, \sigma_3 \vdash^{AP} \forall a. \sigma_1 <: \sigma_2 \rightsquigarrow \lambda x : \forall a. \sigma_1. f(x\tau)}$	
$\text{AP-AS-ARROW} \quad \frac{\vdash^{AP} \sigma_3 <: \sigma_1 \rightsquigarrow f_1 \quad \Sigma \vdash^{AP} \sigma_2 <: \sigma_4 \rightsquigarrow f_2}{\Sigma, \sigma_3 \vdash^{AP} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4 \rightsquigarrow \lambda x : \sigma_1 \rightarrow \sigma_2. \lambda y : \sigma_3. f_2(x(f_1 y))}$		

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.

1115 3.3.5 COHERENCE

1116 One problem with the translation is that there are multiple targets corresponding to one ex-
 1117 pression. This is because in original system there are multiple choices when instantiating a
 1118 polymorphic type, or guessing the annotation for unannotated lambda abstraction (rule [AP-](#)
 1119 [INF-LAM](#)). For each choice, the corresponding target will be different. For example, expres-
 1120 sion $\lambda x. x$ can be type checked with $\text{Int} \rightarrow \text{Int}$, or $a \rightarrow a$, and the corresponding targets are
 1121 $\lambda x : \text{Int}. x$, and $\lambda x : a. x$.

1122 Therefore, in order to prove the translation is coherent, we turn to prove that all the trans-
 1123 lations have the same operational semantics. There are two steps towards the goal: type
 1124 erasure, and considering η expansion and identity functions.

1125 **TYPE ERASURE.** Since type information is useless after type-checking, we erase the type in-
 1126 formation of the targets for comparison. The erasure process is given at the top of Figure 3.6.

1127 The erasure process is standard, where we erase the type annotation in abstractions, and
 1128 remove type abstractions and type applications. The calculus after erasure is the untyped
 1129 lambda calculus.

1130 **ETA-ID EQUALITY.** However, even if we have type erasure, multiple targets for one expres-
 1131 sion can still be syntactically different. The problem is that we can insert more coercion
 1132 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.

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

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

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:

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

1201 Note that another way to model those two rules would be to simply have an initial typing
 1202 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
 1203 elimination of pairs be dealt directly by the rule for variables.

1204 An extended version of the calculus extended with rules for pairs (rule [AP-INF-PAIR](#), rule [AP-](#)
 1205 [S-PAIR](#), rule [AP-APP-FST-VAR](#) and rule [AP-APP-SND-VAR](#)), has been formally studied. All the
 1206 theorems presented before hold with the extension of pairs.

1207 3.5.3 MORE EXPRESSIVE TYPE APPLICATIONS

1208 The design choice of propagating arguments to functions was subject to consideration in
 1209 the original work on local type inference [Pierce and Turner 2000], but was rejected due to
 1210 possible non-determinism introduced by explicit type applications:

1211 *“It is possible, of course, to come up with examples where it would be beneficial to*
 1212 *synthesize the argument types first and then use the resulting information to avoid*

1213 *type annotations in the function part of an application expression....Unfortunately*
 1214 *this refinement does not help infer the type of polymorphic functions. For example,*
 1215 *we cannot uniquely determine the type of x in the expression $(\text{fun}[X](x) e) [\text{Int}] 3$."*

1216 As a response to this challenge, we also present an application of the application mode to a
 1217 variant of System F [Xie and Oliveira 2018]. The development of the calculus shows that the
 1218 application mode can actually work well with calculi with explicit type applications. Here
 1219 we explain the key ideas of the design of the system, but refer to Xie and Oliveira [2018] for
 1220 more details.

1221 To explain the new design, consider the expression:

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

1222 which is not typeable in the traditional type system for System F. In System F the lambda
 1223 abstractions do not account for the context of possible function applications. Therefore when
 1224 type checking the inner body of the lambda abstraction, the expression $x + 1$ is ill-typed,
 1225 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:

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

1226 For type applications, rule **AP-APP-TAPP** stores the type argument σ_1 into the application
 1227 context. Since Ψ tracks type equalities, we apply Ψ as a type substitution to σ_1 (i.e., $[\Psi]\sigma_1$)
 1228 Moreover, to distinguish between type arguments and types of term arguments, we put type
 1229 arguments in brackets (i.e., $[[\Psi]\sigma_1]$). For type abstractions (rule **AP-APP-TLAM**), if the appli-
 1230 cation context is non-empty, we put a new type equality between the type variable a and the
 1231 type argument σ_1 into the context.

1232 Now, back to the problematic expression $(\text{fun}[X](x) e) [\text{Int}] 3$, the type of x can be inferred
 1233 as either X or Int since they are actually equivalent.

1234 **SUGAR FOR TYPE SYNONYMS.** In the same way that we can regard **let** expressions as syntactic
 1235 sugar, in the new type system we further *gain built-in type synonyms*. A *type synonym* is a

1236 new name for an existing type. Type synonyms are common in languages such as Haskell.
 1237 In our calculus a simple form of type synonyms can be desugared as follows:

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

1238 One practical benefit of such syntactic sugar is that it enables a direct encoding of a System F-
 1239 like language with declarations (including type-synonyms). Although declarations are often
 1240 viewed as a routine extension to a calculus, and are not formally studied, they are highly
 1241 relevant in practice. Therefore, a more realistic formalization of a programming language
 1242 should directly account for declarations. By providing a way to encode declarations, our
 1243 new calculus enables a simple way to formalize declarations.

1244 **TYPE ABSTRACTION.** The type equalities introduced by type applications may seem like
 1245 we are breaking System F type abstraction. However, we argue that *type abstraction* is still
 1246 supported by our System F variant. For example:

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

1247 (after desugaring) does *not* type-check, as in a System-F like language. In our type system
 1248 lambda abstractions that are immediately applied to an argument, and unapplied lambda
 1249 abstractions behave differently. Unapplied lambda abstractions are just like System F ab-
 1250 stractions and retain type abstraction. The example above illustrates this. In contrast the
 1251 typeable example $(\Lambda a. \lambda x : a. x + 1) \ \mathbf{Int}$, which uses a lambda abstraction directly applied to
 1252 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.

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

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

1280 4.1 INTRODUCTION AND MOTIVATION

1281 4.1.1 BACKGROUND: GRADUAL TYPING

1282 As is well known, a gradually typed language supports both fully static and fully dynamic
 1283 checking of program properties, as well as the continuum between these two extremes. It
 1284 also offers programmers fine-grained control over the static-to-dynamic spectrum, i.e., a
 1285 program can be evolved by introducing more or less precise types as needed [Garcia et al.
 1286 2016]. The untyped part of a program is dynamically checked, while the typed part is stati-
 1287 cally checked. Therefore, an expression like $(\lambda x : ?. x + 1) \text{ true}$ is dynamically rejected, while
 1288 an expression like $(\lambda x : \text{Int}. x + 1) \text{ true}$ is statically rejected. Here in the example we see the
 1289 use of the *unknown type* $?$, which indicates missing of static type information and thus the
 1290 use of dynamic typing.

1291 Gradual type calculi formalize the interaction between static types and dynamic type by
 1292 featuring the unknown type $?$, as well as a key concept, *consistency*, which weakens type
 1293 equality to allow for the presence of the unknown type $?$. Essentially, if σ_1 is consistent with
 1294 σ_2 (written $\sigma_1 \sim \sigma_2$), then a function e_1 expecting an argument of type σ_1 can also accept
 1295 an argument e_2 of type σ_2 . Importantly, the unknown type $?$ is consistent with any type.
 1296 Therefore, the expression $(\lambda x : ?. x + 1) \text{ true}$ type checks and only later gets rejected at
 1297 runtime.

1298 Siek and Taha [2007] developed a gradual type system for object-oriented languages that
 1299 they call $\mathbf{FOb}_{<}^?$. The definition of type consistency, $\sigma_1 \sim \sigma_2$, is given at the top of Figure 4.1.

1300 The intuition is that consistency relaxes the structure of a type system to tolerate unknown
 1301 positions in a gradual type. Consider some examples:

$$\begin{array}{lcl}
 & \text{Int} & \sim \text{Int} \\
 & \text{Int} & \sim ? \\
 1302 & ? & \sim \text{Int} \\
 & \text{Int} \rightarrow ? & \sim ? \rightarrow \text{Int}
 \end{array}$$

1303 $\text{FOb}_{\lesssim}^?$ also features subtyping. Siek and Taha defined the subtyping relation in a way that
 1304 static type safety is preserved. Their key insight is that the unknown type $?$ is neutral to sub-
 1305 typing, with only $? <: ?$. The subtyping relation $\sigma_1 <: \sigma_2$ relations are defined in Figure 4.1.
 1306 With subtyping, if $\sigma_1 <: \sigma_2$, then a function e_1 expecting an argument of type σ_1 can also
 1307 accept an argument e_2 of type σ_2 . For example, a function expecting `Float` can accept an
 1308 integer.

1309 However, the orthogonality of consistency and subtyping does not lead to a deterministic
 1310 relation. Now the question is that if we know a function e_1 expecting an argument of type
 1311 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
 1312 end, Siek and Taha defined *consistent subtyping*, such that $e_1 e_2$ is well-typed only if σ_2 is a
 1313 consistent subtype of σ_1 (written $\sigma_2 \lesssim \sigma_1$). They defined algorithmic consistent subtyping
 1314 based on a *restriction operator*, written $\sigma_1|_{\sigma_2}$ that “masks off” the parts of type σ_1 that are
 1315 unknown in type σ_2 . For example,

$$\begin{array}{lcl}
 1316 & \text{Int} \rightarrow \text{Int}|_{\text{Bool} \rightarrow \text{Bool}} & = \text{Int} \rightarrow ? \\
 & \text{Bool} \rightarrow ?|_{\text{Int} \rightarrow \text{Int}} & = \text{Bool} \rightarrow ?
 \end{array}$$

1317 The definition of the restriction operator is given below:

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

1318

1319 With the restriction operator, consistent subtyping is simply defined as:

1320 **Definition 3** (Algorithmic Consistent Subtyping of Siek and Taha [2007]). $\sigma_1 \lesssim \sigma_2 \equiv$
 1321 $\sigma_1|_{\sigma_2} <: \sigma_2|_{\sigma_1}$.

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.

1355 4.1.3 APPLICATION: EFFICIENT (PARTLY) TYPED ENCODINGS OF ADTs

1356 We illustrate two concrete applications of gradually typed higher-rank polymorphism related
 1357 to algebraic datatypes. The first application shows how gradual typing helps in defining Scott
 1358 encodings of algebraic datatypes [Curry et al. 1958; Parigot 1992], which are impossible to
 1359 encode in plain System F. The second application shows how gradual typing makes it easy to
 1360 model and use heterogeneous containers.

1361 Our calculus does not provide built-in support for algebraic datatypes (ADTs). Neverthe-
 1362 less, the calculus is expressive enough to support efficient function-based encodings of (op-
 1363 tionally polymorphic) ADTs¹. This offers an immediate way to model algebraic datatypes
 1364 in our calculus without requiring extensions to our calculus or, more importantly, to its
 1365 target—the polymorphic blame calculus. While we believe that such extensions are possible,
 1366 they would likely require non-trivial extensions to the polymorphic blame calculus. Thus
 1367 the alternative of being able to model algebraic datatypes without extending the target cal-
 1368 culus is appealing. The encoding also paves the way to provide built-in support for algebraic
 1369 datatypes in the source language, while elaborating them via the encoding into λB .

1370 CHURCH AND SCOTT ENCODINGS. It is well-known that polymorphic calculi such as System
 1371 F can encode datatypes via Church encodings. However these encodings have well-known
 1372 drawbacks [Koopman et al. 2014]. In particular, some operations are hard to define, and
 1373 they can have a time complexity that is greater than that of the corresponding functions
 1374 for built-in algebraic datatypes. A well-known example is the definition of the predecessor
 1375 function for Church numerals [Church 1941]. Its definition requires significant ingenuity
 1376 (while it is trivial with built-in algebraic datatypes), and it has *linear* time complexity (versus
 1377 the *constant* time complexity for a definition using built-in algebraic datatypes).

1378 An alternative to Church encodings are the so-called Scott encodings [Curry et al. 1958].
 1379 They address the two drawbacks of Church encodings: they allow simple definitions that
 1380 directly correspond to programs implemented with built-in algebraic datatypes, and those
 1381 definitions have the same time complexity to programs using algebraic datatypes.

1382 Unfortunately, Scott encodings, or more precisely, their typed variant [Parigot 1992], can-
 1383 not be expressed in System F: in the general case they require recursive types, which System
 1384 F does not support. However, with gradual typing, we can remove the need for recursive
 1385 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:

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

1386 With the reinterpretation of **fold** and **unfold** as functions instead of built-in primitives, we
1387 have exactly the same definitions of $\text{nil}_?$ and $\text{cons}_?$.

1388 Note that when we elaborate our calculus into the polymorphic blame calculus, the above
1389 type annotations give rise to explicit casts. For instance, after elaboration $\mathbf{fold}_{\text{List}_? a} e$ results
1390 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.

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

4.2 REVISITING CONSISTENT SUBTYPING

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

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

4.2.1 CONSISTENCY AND SUBTYPING

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

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

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

1452 consistent with any functions $\sigma_1 \rightarrow \sigma_2$, for instance, $\text{Int} \rightarrow \text{Bool}$, this means that we could
 1453 provide an expression of type $\forall a. a \rightarrow a$ to a function where the input type is supposed to be
 1454 $\text{Int} \rightarrow \text{Bool}$. However, as we know, $\forall a. a \rightarrow a$ is definitely not compatible with $\text{Int} \rightarrow \text{Bool}$.
 1455 Indeed, this does not hold in any polymorphic type systems without gradual typing. So the
 1456 gradual type system should not accept it either. (This is the *conservative extension* property
 1457 that will be made precise in Section 4.3.3.)

1458 Importantly there is a subtle distinction between a type variable and the unknown type,
 1459 although they both represent a kind of “arbitrary” type. The unknown type stands for the
 1460 absence of type information: it could be *any type at any instance*. Therefore, the unknown
 1461 type is consistent with any type, and additional type-checks have to be performed at runtime.
 1462 On the other hand, a type variable indicates *parametricity*. In other words, a type variable can
 1463 only be instantiated to a single type. For example, in the type $\forall a. a \rightarrow a$, the two occurrences
 1464 of a represent an arbitrary but single type (e.g., $\text{Int} \rightarrow \text{Int}$, $\text{Bool} \rightarrow \text{Bool}$), while $? \rightarrow ?$ could
 1465 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:

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

1466 Notice that, in rule **COMP-ALLL**, the universal quantifier is *always* instantiated to $?$. How-
 1467 ever, this way, λB allows $\forall a. a \rightarrow a \prec \text{Int} \rightarrow \text{Bool}$, which as we discussed before might
 1468 not be what we expect. Indeed λB relies on sophisticated runtime checks to rule out such
 1469 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*.

1470 Compared with our consistency definition in Figure 4.2, their first rule is the same as ours.
 1471 The second rule says that a non \forall -type can be consistent with a \forall -type only if it contains $?$.
 1472 In this way, their type system is able to reject $\forall a. a \rightarrow a \sim \text{Int} \rightarrow \text{Bool}$. However, in order
 1473 to keep conservativity, they also reject $\forall a. a \rightarrow a \sim \text{Int} \rightarrow \text{Int}$, which is perfectly sensible
 1474 in their setting of explicit polymorphism. However with implicit polymorphism, we would
 1475 expect $\forall a. a \rightarrow a$ to be related with $\text{Int} \rightarrow \text{Int}$, since a can be instantiated to Int .

1476 Nonetheless, when it comes to interactions between dynamically typed and polymorphi-
 1477 cally typed terms, both relations allow $\forall a. a \rightarrow \text{Int}$ to be related with $? \rightarrow \text{Int}$ for example
 1478 (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]),
 1479 which in our view, is a kind of (implicit) polymorphic subtyping combined with type consis-
 1480 tency, and that should be derivable by the more primitive notions in the type system (instead
 1481 of inventing new relations). One of our design principles is that subtyping and consistency
 1482 are *orthogonal*, and can be naturally superimposed, echoing the opinion of Siek and Taha
 1483 [2007]. In this case, we can for example derive the relation between $\forall a. a \rightarrow \text{Int}$ and $? \rightarrow \text{Int}$
 1484 via an intermediate type $\text{Int} \rightarrow \text{Int}$, and by the subtyping relation $\forall a. a \rightarrow \text{Int} \prec: \text{Int} \rightarrow \text{Int}$
 1485 and the consistency relation $\text{Int} \rightarrow \text{Int} \sim ? \rightarrow \text{Int}$.

1486 4.2.2 TOWARDS CONSISTENT SUBTYPING

1487 With the definitions of consistency and subtyping, the question now is how to compose the
 1488 two relations so that two types can be compared in a way that takes both relations into ac-
 1489 count.

1490 Unfortunately, the strawman version of consistent subtyping (Definition 4) does not work
 1491 well with our definitions of consistency and subtyping for polymorphic types. Consider two
 1492 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
 1493 type in one way (first by applying consistency, then subtyping), but not the other way, as
 1494 shown in Figure 4.3a. We use $????$ to mean that we cannot find such a type. Similarly, there
 1495 are situations where the first type can only reach the second type by the other way (first
 1496 applying subtyping, and then consistency), as shown in Figure 4.3b.

1497 What is worse, if those two examples are composed in a way that those types all appear
 1498 co-variantly, then the resulting types cannot reach each other in either way. For example,
 1499 Figure 4.3c shows two such types by putting a Bool type in the middle, and neither definition
 1500 of consistent subtyping in Definition 4 works.

1501 OBSERVATIONS ON CONSISTENT SUBTYPING BASED ON INFORMATION PROPAGATION. In
 1502 order to develop a correct definition of consistent subtyping for polymorphic types, we need
 1503 to understand how consistent subtyping works. We first review two important properties of

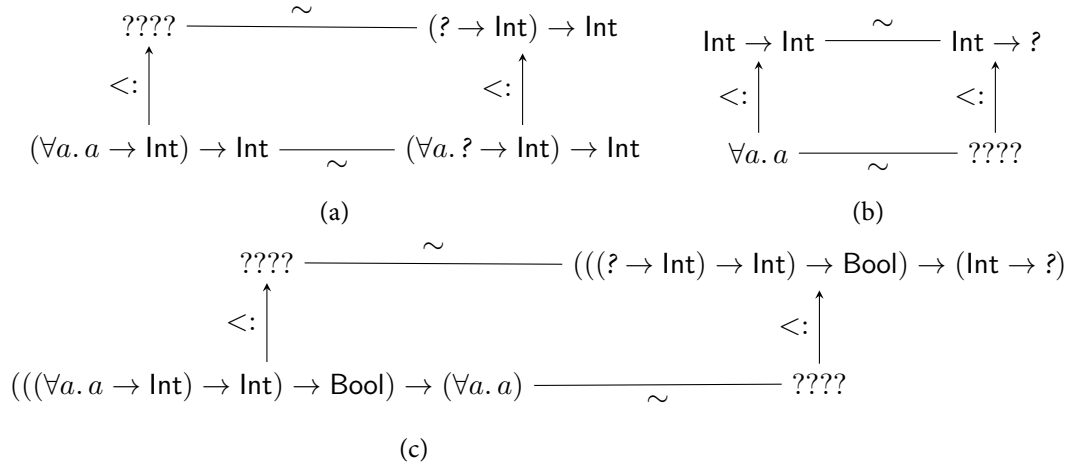


Figure 4.3: Examples that break the original definition of consistent subtyping.

1504 subtyping: (1) subtyping induces the subsumption rule used in typing: if $\sigma_1 <: \sigma_2$, then
 1505 an expression of type σ_1 can be used where σ_2 is expected; (2) subtyping is transitive: if
 1506 $\sigma_1 <: \sigma_2$, and $\sigma_2 <: \sigma_3$, then $\sigma_1 <: \sigma_3$. Though consistent subtyping takes the unknown
 1507 type into consideration, the subsumption rule should also apply: if $\sigma_1 \lesssim \sigma_2$, then an ex-
 1508 pression of type σ_1 can also be used where σ_2 is expected, given that there might be some
 1509 information lost by consistency. A crucial difference from subtyping is that consistent sub-
 1510 typing is *not* transitive because information can only be lost once; otherwise, any two types
 1511 are a consistent subtype of each other, e.g., we could derive $\text{Int} \sim \text{Bool}$ from $\text{Int} \sim ?$ and
 1512 $? \sim \text{Bool}$. Now consider a situation where we have both $\sigma_1 <: \sigma_2$, and $\sigma_2 \lesssim \sigma_3$, this means
 1513 that σ_1 can be used where σ_2 is expected, and σ_2 can be used where σ_3 is expected, with
 1514 possibly some loss of information. In other words, we should expect that σ_1 can be used
 1515 where σ_3 is expected, since there is at most one-time loss of information. We thus have the
 1516 following desirable property:

1517 **Observation 1.** If $\sigma <: \sigma_2$, and $\sigma_2 \lesssim \sigma_3$, then $\sigma \lesssim \sigma_3$.

1518 This is reflected in Figure 4.4a. A symmetrical observation is given in Figure 4.4b:

1519 **Observation 2.** If $\sigma_3 \lesssim \sigma_2$, and $\sigma_2 <: \sigma_1$, then $\sigma_3 \lesssim \sigma_1$.

1520 From the above observations, we see what the problem is with the original definition.
 1521 In Figure 4.4a, if σ_2 can reach σ_3 by σ' , then by subtyping transitivity, σ_1 can reach σ_3 by
 1522 σ' . However, if σ_2 can only reach σ_3 by σ'' , then σ cannot reach σ_3 through the original
 1523 definition. A similar problem is shown in Figure 4.4b.

1524 It turns out that these two problems can be fixed using the same strategy: instead of taking
 1525 one-step subtyping and one-step consistency, our definition of consistent subtyping allows

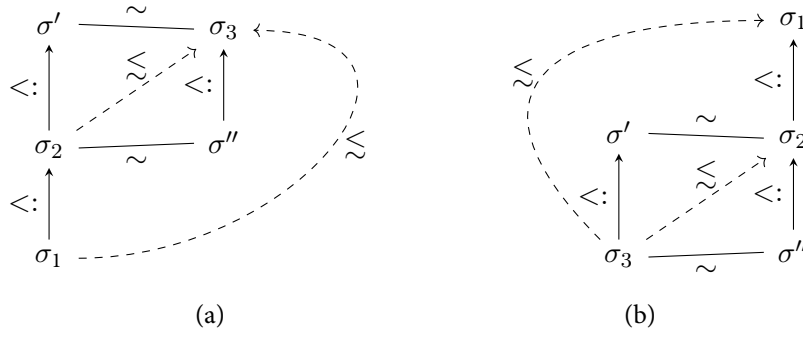


Figure 4.4: Observations of consistent subtyping

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:

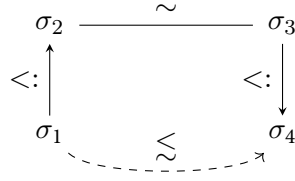
Definition 5 (Consistent Subtyping). $\Psi \vdash^G \sigma_1 \lesssim \sigma_2$ if and only if $\Psi \vdash^G \sigma_1 <: \sigma', \sigma' \sim \sigma''$ and $\Psi \vdash^G \sigma'' <: \sigma_2$ for some σ' and σ'' .

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.

1547 4.2.3 ABSTRACTING GRADUAL TYPING

1548 Garcia et al. [2016] presented a new foundation for gradual typing that they call the *Abstract-*
 1549 *ing Gradual Typing* (AGT) approach. In the AGT approach, gradual types are interpreted as
 1550 sets of static types, where static types refer to types containing no unknown types. In this
 1551 interpretation, predicates and functions on static types can then be lifted to apply to gradual
 1552 types. Central to their approach is the so-called *concretization* function. For simple types, a
 1553 concretization γ from gradual types to a set of static types is defined as follows:

1554 **Definition 6** (Concretization).

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

1556 Based on the concretization function, subtyping between static types can be lifted to grad-
 1557 ual types, resulting in the consistent subtyping relation:

1558 **Definition 7** (Consistent Subtyping in AGT). $\sigma_1 \widetilde{<} \sigma_2$ if and only if $\sigma'_1 <: \sigma'_2$ for some *static*
 1559 *types* σ'_1 and σ'_2 such that $\sigma'_1 \in \gamma(\sigma_1)$ and $\sigma'_2 \in \gamma(\sigma_2)$.

1560 Later they proved that this definition of consistent subtyping coincides with that of Def-
 1561 inition 4. By Proposition 4.1, we can directly conclude that our definition coincides with
 1562 AGT:

1563 **Corollary 4.2** (Equivalence to AGT on Simple Types). $\sigma_1 \lesssim \sigma_2$ if and only if $\sigma_1 \widetilde{<} \sigma_2$.

1564 However, AGT does not show how to deal with polymorphism (e.g. the interpretation of
 1565 type variables) yet. Still, as noted by Garcia et al. [2016], this is a promising line of future
 1566 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 $\text{Int} \rightarrow ? \rightarrow ? <: \text{Int} \rightarrow ? \rightarrow ?$, we conclude that $\text{Int} \rightarrow ? \rightarrow \text{Bool} \lesssim \text{Int} \rightarrow \text{Int} \rightarrow ?$.

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

$\boxed{\Psi \vdash^G \sigma_1 \lesssim \sigma_2}$				(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

1648 system for predicative implicit polymorphism, GPC, that employs the consistent subtyping
 1649 relation. The declarative system itself is already quite interesting as it is equipped with both
 1650 higher-rank polymorphism and the unknown type.

1651 The syntax of expressions in the declarative system is given at the top of Figure 4.7. The
 1652 definition of expressions are the same as of OL in Figure 2.3. Meta-variable e ranges over
 1653 expressions. Expressions include variables x , integers n , annotated lambda abstractions $\lambda x : \sigma. e$,
 1654 un-annotated lambda abstractions $\lambda x. e$, applications $e_1 e_2$, and let expressions **let** $x = e_1$ **in** e_2 .
 1655

1656 4.3.1 TYPING IN DETAIL

1657 Figure 4.7 gives the typing rules for our declarative system (the reader is advised to ignore
 1658 the gray-shaded parts for now). Rule **GPC-VAR** extracts the type of the variable from the
 1659 typing context. Rule **GPC-INT** always infers integer types. Rule **GPC-LAMANN** puts x with type
 1660 annotation σ into the context, and continues type checking the body e . Rule **GPC-LAM** assigns
 1661 a monotype τ to x , and continues type checking the body e . Gradual types and polymorphic
 1662 types are introduced via explicit annotations. Rule **GPC-GEN** puts a fresh type variable a into
 1663 the type context and generalizes the typing result σ to $\forall a. \sigma$. Rule **GPC-LET** infers the type
 1664 σ of e_1 , then puts $x : \sigma$ in the context to infer the type of e_2 . Rule **GPC-APP** first infers the
 1665 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
 1666 the codomain type σ_2 from type σ . The type σ_3 of the argument e_2 is then compared with σ_1
 1667 using the consistent subtyping judgment.

1668 **MATCHING.** The matching judgment of Siek et al. [2015] is extended to polymorphic types
 1669 naturally, resulting in $\Psi \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2$. Note that the matching rules generalize that of DK
 1670 in Section 2.3.2 with the unknown type. In rule **GPC-M-FORALL**, a monotype τ is guessed
 1671 to instantiate the universal quantifier a . If σ is a polymorphic type, the judgment works by
 1672 guessing instantiations until it reaches an arrow type. Rule **GPC-M-ARR** returns the domain
 1673 type σ_1 and range type σ_2 as expected. If the input is $?$, then rule **GPC-M-UNKNOWN** returns
 1674 $?$ as both the type for the domain and the range.

1675 Note that in GPC, matching saves us from having a subsumption rule (rule **OL-SUB** in
 1676 Figure 2.5). The subsumption rule is incompatible with consistent subtyping, since the latter
 1677 is not transitive. A discussion of a subsumption rule based on normal subtyping can be found
 1678 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

1679 4.3.2 TYPE-DIRECTED TRANSLATION

1680 We give the dynamic semantics of our language by translating it to λB [Ahmed et al. 2009].
 1681 Below we show a subset of the terms in λB that are used in the translation:

$$1682 \quad \lambda B \text{ Terms } \quad s ::= x \mid n \mid \lambda x : \sigma. s \mid \Lambda a. s \mid s_1 s_2 \mid \langle \sigma_1 \hookrightarrow \sigma_2 \rangle s$$

1683 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
 1684 to σ_2 is permitted only if the types are *compatible*, written $\sigma_1 \prec \sigma_2$, as briefly mentioned in
 1685 Section 4.2.1. The syntax of types in λB is the same as ours.

1686 The translation is given in the gray-shaded parts in Figure 4.7. The only interesting case
 1687 here is to insert explicit casts in the application rule. Note that there is no need to translate
 1688 matching or consistent subtyping. Instead we insert the source and target types of a cast
 1689 directly in the translated expressions, thanks to the following two lemmas:

1690 **Lemma 4.5** (\triangleright to \prec). *If $\Psi \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2$, then $\sigma \prec \sigma_1 \rightarrow \sigma_2$.*

1691 **Lemma 4.6** (\lesssim to \prec). *If $\Psi \vdash^G \sigma_1 \lesssim \sigma_2$, then $\sigma_1 \prec \sigma_2$.*

1692 In order to show the correctness of the translation, we prove that our translation always
 1693 produces well-typed expressions in λB . By Lemmas 4.5 and 4.6, we have the following the-
 1694 orem:

1695 **Theorem 4.7** (Type Safety). *If $\Psi \vdash^G e : \sigma \rightsquigarrow s$, then $\Psi \vdash^B s : \sigma$.*

1696 **PARAMETRICITY.** An important semantic property of polymorphic types is *relational para-*
 1697 *metricity* [Reynolds 1983]. The parametricity property says that all instances of a poly-
 1698 morphic function should behave *uniformly*. A classic example is a function with the type
 1699 $\forall a. a \rightarrow a$. The parametricity property guarantees that a value of this type must be either
 1700 the identity function (i.e., $\lambda x. x$) or the undefined function (one which never returns a value).
 1701 However, with the addition of the unknown type $?$, careful measures are to be taken to ensure
 1702 parametricity. Our translation target λB is taken from Ahmed et al. [2009], where relational
 1703 parametricity is enforced by dynamic sealing [Matthews and Ahmed 2008; Neis et al. 2009],
 1704 but there is no rigorous proof. Later, Ahmed et al. [2009] imposed a syntactic restriction on
 1705 terms of λB , where all type abstractions must have *values* as their body. With this invari-
 1706 ant, they proved that the restricted λB satisfies relational parametricity. It remains to see if
 1707 our translation process can be adjusted to target restricted λB . One possibility is to impose
 1708 similar restriction to the rule **GPC-GEN**:

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

1709 where we only generate type abstractions if the inner body is a value. However, the type
 1710 system with this rule is a weaker calculus, which is not a conservative extension of the OL
 1711 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:

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

1712 If we apply $\lambda x : ?.fx$ to 3, which is fine since the function can take any input, the first
 1713 translation runs smoothly in λB , while the second one will raise a cast error (Int cannot be
 1714 cast to Bool). Similarly, if we apply it to `true`, then the second succeeds while the first fails.
 1715 The culprit lies in the highlighted parts where the instantiation of a appears in the explicit
 1716 cast. More generally, any choice introduces an explicit cast to that type in the translation,
 1717 which causes a runtime cast error if the function is applied to a value whose type does not
 1718 match the guessed type. Note that this does not compromise the type safety of the translated
 1719 expressions, since cast errors are part of the type safety guarantees.

1720 The semantic discrepancy is due to the guessing nature of the *declarative* system. As far
 1721 as the static semantics is concerned, both $\text{Int} \rightarrow \text{Int}$ and $\text{Bool} \rightarrow \text{Int}$ are equally acceptable.
 1722 But this is not the case at runtime. The astute reader may have found that the *only* appropriate
 1723 choice is to instantiate the type of f to $? \rightarrow \text{Int}$ in the matching judgment. However,
 1724 as specified by rule [GPC-M-FORALL](#) in Figure 4.7, we can only instantiate type variables to
 1725 monotypes, but $?$ is *not* a monotype! We will get back to this issue in Chapter 5.

1726 COHERENCE. The ambiguity of translation seems to imply that the declarative system is
 1727 *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 $\Psi \vdash s_1 \simeq_{ctx} s_2 : \sigma$.*

1756 4.3.3 CORRECTNESS CRITERIA

1757 Siek et al. [2015] present a set of properties, the *refined criteria*, that a well-designed gradual
 1758 typing calculus must have. Among all the criteria, those related to the static aspects of gradual
 1759 typing are well summarized by Cimini and Siek [2016]. Here we review those criteria and
 1760 adapt them to our notation. We have proved in Coq that our type system satisfies all these
 1761 criteria.

1762 **Lemma 4.10** (Correctness Criteria).

- 1763 • **Conservative extension:** for all static Ψ , e , and σ_1 ,
 - 1764 – if $\Psi \vdash^{OL} e : \sigma_1$, then there exists σ_2 , such that $\Psi \vdash^G e : \sigma_2$, and $\Psi \vdash^G \sigma_2 <: \sigma_1$.
 - 1765 – if $\Psi \vdash^G e : \sigma$, then $\Psi \vdash^{OL} e : \sigma$
- 1766 • **Monotonicity w.r.t. precision:** for all Ψ , e , e' , σ_1 , if $\Psi \vdash^G e : \sigma_1$, and $e' \sqsubseteq e$, then
 1767 $\Psi \vdash^G e' : \sigma_2$, and $\sigma_2 \sqsubseteq \sigma_1$ for some σ_2 .
- 1768 • **Type Preservation of cast insertion:** for all Ψ , e , σ , if $\Psi \vdash^G e : \sigma$, then $\Psi \vdash^G e : \sigma \rightsquigarrow s$,
 1769 and $\Psi \vdash^B s : \sigma$ for some s .
- 1770 • **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
 1771 $\Psi \vdash^G e_2 : \sigma \rightsquigarrow s_2$, and $e_1 \sqsubseteq e_2$, then $\Psi \vdash s_1 \sqsubseteq^B s_2$.

1772 The first criterion states that the gradual type system should be a conservative extension
 1773 of the original system. In other words, a *static* program is typeable in the OL type system if
 1774 and only if it is typeable in the gradual type system. A static program is one that does not
 1775 contain any type ⁶. However since our gradual type system does not have the subsumption
 1776 rule, it produces more general types.

1777 The second criterion states that if a typeable expression loses some type information, it
 1778 remains typeable. This criterion depends on the definition of the precision relation, written
 1779 $\sigma_1 \sqsubseteq \sigma_2$, which is given in Figure 4.8. The relation intuitively captures a notion of types con-
 1780 taining more or less unknown types (?). The precision relation over types lifts to programs,
 1781 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
 1782 types.

1783 The first two criteria are fundamental to gradual typing. They explain for example why
 1784 these two programs $\lambda x : \text{Int}. x + 1$ and $\lambda x : ?. x + 1$ are typeable, as the former is typeable in
 1785 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	$\Gamma, \Delta, \Theta ::= \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \hat{\alpha} \mid \Gamma, \hat{\alpha} = \tau \mid \Gamma, \blacktriangleright_{\hat{\alpha}}$
Complete Contexts	$\Omega ::= \bullet \mid \Omega, x : \sigma \mid \Omega, a \mid \Omega, \hat{\alpha} = \tau \mid \Omega, \blacktriangleright_{\hat{\alpha}}$

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

1811 ALGORITHMIC CONTEXTS. The top of Figure 4.9 shows the syntax of the algorithmic sys-
 1812 tem. A noticeable difference are the algorithmic contexts Γ , which are represented as an *or-*
 1813 *dered* list containing declarations of type variables a and term variables $x : \sigma$. Unlike declar-
 1814 ative contexts, algorithmic contexts also contain declarations of unification type variables
 1815 $\hat{\alpha}$, which can be either unsolved (written $\hat{\alpha}$) or solved to some monotype (written $\hat{\alpha} = \tau$).
 1816 Finally, algorithmic contexts include a *marker* $\blacktriangleright_{\hat{\alpha}}$ (read “marker $\hat{\alpha}$ ”), which is used to delin-
 1817 eate unification variables created by the algorithm. We will have more to say about markers
 1818 when we examine the rules. Complete contexts Ω are the same as contexts, except that they
 1819 contain no unsolved variables.

1820 Apart from expressions in the declarative system, we add annotated expressions $e : \sigma$. The
 1821 well-formedness judgments for types and contexts are shown in Figure 4.9.

1822 NOTATIONAL CONVENIENCE. Following DK's system, we use contexts as substitutions on
 1823 types. We write $[\Gamma]\sigma$ to mean Γ applied as a substitution to type σ , whose definition is given
 1824 below. We also use a hole notation, which is useful when manipulating contexts by inserting
 1825 and replacing declarations in the middle. The hole notation is used extensively in proving
 1826 soundness and completeness. For example, $\Gamma[\Theta]$ means Γ has the form $\Gamma_L, \Theta, \Gamma_R$; if we have
 1827 $\Gamma[\hat{\alpha}] = (\Gamma_L, \hat{\alpha}, \Gamma_R)$, then $\Gamma[\hat{\alpha} = \tau] = (\Gamma_L, \hat{\alpha} = \tau, \Gamma_R)$. Occasionally, we will see a context
 1828 with two *ordered* holes, e.g., $\Gamma = \Gamma_0[\Theta_1][\Theta_2]$ means Γ has the form $\Gamma_L, \Theta_1, \Gamma_M, \Theta_2, \Gamma_R$.

	$[\Gamma]\text{Int}$	$=$	Int
	$[\Gamma]a$	$=$	a
1829	$[\Gamma[\hat{\alpha}]]\hat{\alpha}$	$=$	$\hat{\alpha}$
	$[\Gamma[\hat{\alpha} = \tau]]\hat{\alpha}$	$=$	$[\Gamma[\hat{\alpha} = \tau]]\tau$
	$[\Gamma]\sigma_1 \rightarrow \sigma_2$	$=$	$[\Gamma]\sigma_1 \rightarrow [\Delta]\sigma_2$
	$[\Gamma](\forall a. \sigma)$	$=$	$\forall a. [\Gamma]\sigma$

1830 INPUT AND OUTPUT CONTEXTS. The algorithmic system, compared with the declarative
 1831 system, includes similar judgment forms, except that we replace the declarative context Ψ
 1832 with an algorithmic context Γ (the *input context*), and add an *output context* Δ after a back-
 1833 ward turnstile, e.g., $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$ is the judgment form for the algorithmic consistent
 1834 subtyping. All algorithmic rules manipulate input and output contexts in a way that is con-
 1835 sistent with the notion of *context extension*, which will be described in Section 4.4.5.

1836 We start with the explanation of the algorithmic consistent subtyping as it involves ma-
 1837 nipulating unification type variables explicitly (and solving them if possible).

1838 4.4.1 ALGORITHMIC CONSISTENT SUBTYPING

1839 Figure 4.10 presents the rules of algorithmic consistent subtyping $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$, which
 1840 says that under input context Γ , σ_1 is a consistent subtype of σ_2 , with output context Δ . The
 1841 first five rules do not manipulate contexts, but illustrate how contexts are propagated.

1842 Rule [GPC-AS-TVAR](#) and rule [GPC-AS-INT](#) do not involve unification variables, so the out-
 1843 put contexts remain unchanged. Rule [GPC-AS-EVAR](#) says that any unsolved unification vari-
 1844 able is a consistent subtype of itself. The output is still the same as the input context as
 1845 the rule gives no clue as to what is the solution of that unification variable. Rules [GPC-](#)
 1846 [AS-UNKNOWNL](#) and [AS-UNKNOWNR](#) are the counterparts of rule [GPC-CS-UNKNOWNL](#) and
 1847 rule [GPC-CS-UNKNOWNR](#).

1848 Rule [GPC-AS-ARROW](#) is a natural extension of its declarative counterpart. The output con-
 1849 text of the first premise is used by the second premise, and the output context of the second

<div style="border: 1px solid black; display: inline-block; padding: 2px 10px;"> $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$ </div> (Under input context Γ, σ_1 is a consistent subtype of σ_2, with output context Δ)		
GPC-AS-TVAR $\overline{\Gamma[a] \vdash^G a \lesssim a \dashv \Gamma[a]}$	GPC-AS-INT $\overline{\Gamma \vdash^G \text{Int} \lesssim \text{Int} \dashv \Gamma}$	GPC-AS-EVAR $\overline{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \hat{\alpha} \dashv \Gamma[\hat{\alpha}]}$
GPC-AS-UNKNOWNL $\overline{\Gamma \vdash^G ? \lesssim \sigma \dashv \Gamma}$		GPC-AS-UNKNOWNR $\overline{\Gamma \vdash^G \sigma \lesssim ? \dashv \Gamma}$
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 $\frac{\Gamma, a \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta, a, \Theta}{\Gamma \vdash^G \sigma_1 \lesssim \forall a. \sigma_2 \dashv \Delta}$
GPC-AS-FORALLL $\frac{\Gamma, \blacktriangleright_{\hat{\alpha}}, \hat{\alpha} \vdash^G \sigma_1[a \mapsto \hat{\alpha}] \lesssim \sigma_2 \dashv \Delta, \blacktriangleright_{\hat{\alpha}}, \Theta}{\Gamma \vdash^G \forall a. \sigma_1 \lesssim \sigma_2 \dashv \Delta}$	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 4.10: Algorithmic consistent subtyping

premise is the output context of the conclusion. Note that we do not simply check $\sigma_2 \lesssim \sigma_4$, but apply Θ (the input context of the second premise) to both types (e.g., $[\Theta]\sigma_2$). This is to maintain an important invariant: whenever $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$ holds, the types σ_1 and σ_2 are fully applied under input context Γ (they contain no unification 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 unification variables to appear after the type variable a , in a trailing context Θ . These unification 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 unification variable $\hat{\alpha}$, and replaces a with $\hat{\alpha}$ in the body σ . The new unification 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 unification 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 unification variable on one side and an arbitrary type on the other side. Apart from checking that the unification variable does not occur in the type σ , both rules do not directly solve the unification 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 unification 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 unification 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

1884 Rule **GPC-INSTL-SOLVEU** is similar to rule **GPC-AS-UNKNOWNR** in that we put no con-
 1885 straint⁷ on $\hat{\alpha}$ when it meets the unknown type $?$. This design decision reflects the point
 1886 that type inference only produces static types [Garcia and Cimini 2015].

1887 Rule **GPC-INSTL-REACH** deals with the situation where two unification variables meet. Re-
 1888 call that $\Gamma[\hat{\alpha}][\hat{\beta}]$ denotes a context where some unsolved unification variable $\hat{\alpha}$ is declared
 1889 before $\hat{\beta}$. In this situation, the only logical thing we can do is to set the solution of one unifi-
 1890 cation variable to the other one, depending on which one is declared before. For example, in
 1891 the output context of rule **GPC-INSTL-REACH**, we have $\hat{\beta} = \hat{\alpha}$ because in the input context, $\hat{\alpha}$
 1892 is declared before $\hat{\beta}$.

1893 Rule **GPC-INSTL-FORALLR** is the instantiation version of rule **GPC-AS-FORALLR**. Since our
 1894 system is predicative, $\hat{\alpha}$ cannot be instantiated to $\forall b. \sigma$, but we can decompose $\forall b. \sigma$ in the
 1895 same way as in rule **GPC-AS-FORALLR**. Rule **GPC-INSTR-FORALLL** is the instantiation version
 1896 of rule **GPC-AS-FORALLL**.

1897 Rule **GPC-INSTL-ARR** applies when $\hat{\alpha}$ meets an arrow type. It follows that the solution must
 1898 also be an arrow type. This is why, in the first premise, we generate two fresh unification
 1899 variables $\hat{\alpha}_1$ and $\hat{\alpha}_2$, and insert them just before $\hat{\alpha}$ in the input context, so that we can solve
 1900 $\hat{\alpha}$ to $\hat{\alpha}_1 \rightarrow \hat{\alpha}_2$. Note that the first premise $\sigma_1 \approx \hat{\alpha}_1$ switches to the other instantiation
 1901 judgment.

1902 4.4.3 ALGORITHMIC TYPING

1903 We now turn to the algorithmic typing rules in Figure 4.12. Because general type infer-
 1904 ence for System F is undecidable [Wells 1999], our algorithmic system uses bidirectional
 1905 type checking to accommodate (first-class) polymorphism. Traditionally, two modes are
 1906 employed in bidirectional systems: the checking mode $\Gamma \vdash^G e \Leftarrow \sigma \dashv \Theta$, which takes a
 1907 term e and a type σ as input, and ensures that the term e checks against σ ; the inference
 1908 mode $\Gamma \vdash^G e \Rightarrow \sigma \dashv \Theta$, which takes a term e and produces a type σ . We first discuss rules
 1909 in the inference mode.

1910 Rule **GPC-INF-VAR** and rule **GPC-INF-INT** do not generate any new information and sim-
 1911 ply propagate the input context. Rule **GPC-INF-ANNO** is standard, switching to the checking
 1912 mode in the premise.

1913 In rule **GPC-INF-LAMANN**, we generate a fresh unification variable $\hat{\beta}$ for the function codomain,
 1914 and check the function body against $\hat{\beta}$. Note that it is tempting to write $\Gamma, x : \sigma \vdash^G e \Rightarrow$
 1915 $\sigma_2 \dashv \Delta, x : \sigma, \Theta$ as the premise (in the hope of better matching its declarative counterpart
 1916 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 unification 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 unification 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 $\hat{\alpha} \rightarrow \hat{\alpha}_1 \rightarrow \hat{\alpha}_2$ 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 unification 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 $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$.*

1976 DECIDABILITY OF ALGORITHMIC TYPING. Similar to proving decidability of algorithmic
 1977 consistent subtyping, the key is to come up with a correct measure. Since the typing rules
 1978 depend on the matching judgment, we first show decidability of the algorithmic matching.

1979 **Lemma 4.13** (Decidability of Algorithmic Matching). *Given a context Γ and a type σ it is*
 1980 *decidable whether there exist types σ_1, σ_2 and a context Δ such that $\Gamma \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta$.*

1981 Now we are ready to show decidability of typing. The proof is obtained by induction on
 1982 the lexicographically ordered triple: size of e , typing judgment (where the inference mode
 1983 \Rightarrow is considered smaller than the checking mode \Leftarrow) and contextual size.

$$1984 \quad \left\langle e, \begin{array}{c} \Rightarrow \\ \Leftarrow \end{array}, |\Gamma \vdash^G \sigma| \right\rangle$$

1985 The above measure is much simpler than the corresponding one in the DK system, where
 1986 they also need to consider the application judgment together with the inference and checking
 1987 judgments. This shows another benefit (besides the independence from typing) of adopting
 1988 the matching judgment.

1989 **Theorem 4.14** (Decidability of Algorithmic Typing).

- 1990 1. *Inference: Given a context Γ and a term e , it is decidable whether there exist a type σ*
 1991 *and a context Δ such that $\Gamma \vdash^G e \Rightarrow \sigma \dashv \Delta$.*
- 1992 2. *Checking: Given a context Γ , a term e and a type σ such that $\Gamma \vdash^G \sigma$, it is decidable*
 1993 *whether there exists a context Δ such that $\Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta$.*

1994 4.4.5 CONTEXT EXTENSION

1995 To be confident that our algorithmic type system and the declarative type system agree with
 1996 each other, we need to prove that the algorithmic rules are sound and complete with respect
 1997 to the declarative specification. Before we give the formal statements of the soundness and
 1998 completeness theorems, we need a meta-theoretical device, called *context extension* [Dun-
 1999 field and Krishnaswami 2013], to capture a notion of information increase from input con-
 2000 texts to output contexts.

2001 A context extension judgment $\Gamma \longrightarrow \Delta$ reads “ Γ is extended by Δ ”. Intuitively, this judg-
 2002 ment says that Δ has at least as much information as Γ : some unsolved unification variables
 2003 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

4.4.6 SOUNDNESS

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

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

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

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

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

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

2023 2. If $\Gamma \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta$ then $[\Omega]\Delta \vdash^G [\Omega]\sigma \lesssim [\Omega]\hat{\alpha}$.

2024 Notice that the declarative judgment uses $[\Omega]\Delta$, an operation that applies a complete con-
 2025 text Ω to the algorithmic context Δ , essentially plugging in all known solutions and removing
 2026 all declarations of unification variables (both solved and unsolved), resulting in a declarative
 2027 context.

2028 With instantiation soundness, next we show that the algorithmic consistent subtyping is
 2029 sound:

2030 **Theorem 4.16** (Soundness of Algorithmic Consistent Subtyping). *If $\Gamma \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta$
 2031 where $[\Gamma]\sigma_1 = \sigma_1$ and $[\Gamma]\sigma_2 = \sigma_2$ and $\Delta \longrightarrow \Omega$ then $[\Omega]\Delta \vdash^G [\Omega]\sigma_1 \lesssim [\Omega]\sigma_2$.*

2032 Finally the soundness theorem of algorithmic typing is:

2033 **Theorem 4.17** (Soundness of Algorithmic Typing). *Given $\Delta \longrightarrow \Omega$:*

2034 1. If $\Gamma \vdash^G e \Rightarrow \sigma \dashv \Delta$ then $\exists e'$ such that $[\Omega]\Delta \vdash^G e' : [\Omega]\sigma$ and $|e| = |e'|$.

2035 2. If $\Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta$ then $\exists e'$ such that $[\Omega]\Delta \vdash^G e' : [\Omega]\sigma$ and $|e| = |e'|$.

2036 4.4.7 COMPLETENESS

2037 Completeness of the algorithmic system is the reverse of soundness: given a declarative judg-
 2038 ment of the form $[\Omega]\Gamma \vdash^G [\Omega] \dots$, we want to get an algorithmic derivation of $\Gamma \vdash^G \dots \dashv \Delta$.
 2039 It turns out that completeness is a bit trickier to state in that the algorithmic rules generate
 2040 unification variables on the fly, so Δ could contain unsolved unification variables that are
 2041 not found in Γ , nor in Ω . Therefore the completeness proof must produce another complete
 2042 context Ω' that extends both the output context Δ , and the given complete context Ω . As
 2043 with soundness, we need erasure to relate both expressions.

2044 **Theorem 4.18** (Instantiation Completeness). *Given $\Gamma \longrightarrow \Omega$ and $\sigma = [\Gamma]\sigma$ and $\hat{\alpha} \in$
 2045 $UNSOLVED(\Gamma)$ and $\hat{\alpha} \notin FV(\sigma)$:*

2046 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
 2047 $\Gamma \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta$.

2048 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
 2049 $\Gamma \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta$.

2050 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.

2105 4.5.2 A MORE DECLARATIVE TYPE SYSTEM

2106 In Section 4.3 we present our declarative system in terms of the matching and consistent
 2107 subtyping judgments. The rationale behind this design choice is that the resulting declara-
 2108 tive system combines subtyping and type consistency in the application rule, thus making
 2109 it easier to design an algorithmic system accordingly. Still, one may wonder if it is possible
 2110 to design a more declarative specification. For example, even though we mentioned that the
 2111 subsumption rule is incompatible with consistent subtyping, it might be possible to accom-
 2112 modate a subsumption rule for normal subtyping (instead of consistent subtyping). In this
 2113 section, we discuss an alternative for the design of the declarative system.

2114 **WRONG DESIGN.** A naive design that does not work is to replace rule **GPC-APP** in Figure 4.7
 2115 with the following two rules:

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

2116 Rule **GPC-V-SUB** is the standard subsumption rule: if an expression e has type σ , then it
 2117 can be assigned some type σ_2 that is a supertype of σ . Rule **GPC-V-APP1** first infers that e_1
 2118 has type σ , and e_2 has type σ_1 , then it finds some σ_2 so that σ is consistent with $\sigma_1 \rightarrow \sigma_2$.

2119 There would be two obvious benefits of this variant if it did work: firstly this approach
 2120 closely resembles the traditional declarative type systems for calculi with subtyping; secondly
 2121 it saves us from discussing various forms of σ in rule **GPC-V-APP1**, leaving the job to the
 2122 consistency judgment.

2123 The design is wrong because of the information loss caused by the choice of σ_2 in rule **GPC-**
 2124 **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

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

2125 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.

2176 sistent subtype of all castable types, rather than of all types (rule [GPC-CS-UNKNOWNLL](#)), and
 2177 vice versa (rule [GPC-CS-UNKNOWNRR](#)). Moreover, the static type parameter \mathcal{S} is a consistent
 2178 subtype of itself (rule [GPC-CS-SPAR](#)), and similarly for the gradual type parameter (rule [GPC-](#)
 2179 [CS-GPAR](#)). From this definition it follows immediately that $?$ is incomparable with types that
 2180 contain static type parameters \mathcal{S} , such as $\mathcal{S} \rightarrow \text{Int}$.

2181 **TYPING AND TRANSLATION.** Given these extensions to types and consistent subtyping, the
 2182 typing process remains the same as in Figure 4.7. To account for the changes in the transla-
 2183 tion, if we extend λB with type parameters as in Garcia and Cimini [2015], then the transla-
 2184 tion remains the same as well.

2185 5.2 SUBSTITUTIONS AND REPRESENTATIVE TRANSLATIONS

2186 As we mentioned, type parameters serve as placeholders for monotypes. As a consequence,
 2187 wherever a type parameter is used, any *suitable* monotype could appear just as well. To for-
 2188 malize this observation, we define substitutions for type parameters as follows:

2189 **Definition 12** (Substitution). Substitutions for type parameters are defined as:

- 2190 1. Let $S^S : \mathcal{S} \rightarrow \tau$ be a total function mapping static type parameters to monotypes.
- 2191 2. Let $S^G : \mathcal{G} \rightarrow t$ be a total function mapping gradual type parameters to castable
2192 monotypes.
- 2193 3. Let $S^P = S^G \cup S^S$ be a union of S^S and S^G mapping static and gradual type param-
2194 eters accordingly.

2195 Note that since \mathcal{G} might be compared with $?$, only castable monotypes are suitable substitutes,
2196 whereas \mathcal{S} can be replaced by any monotypes. Therefore, we can substitute \mathcal{G} for \mathcal{S} , but not
2197 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:

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

2198 The advantage of type parameters is that they help reasoning about the dynamic semantics.
2199 Now we are not limited to a particular choice, such as $\text{Int} \rightarrow \text{Int}$ or $\text{Bool} \rightarrow \text{Int}$, which might
2200 or might not emit a cast error at runtime. Instead we have a general choice $\mathcal{G} \rightarrow \text{Int}$.

2201 What does the more general choice with type parameters tell us? First, we know that in
2202 this case, there is no concrete constraint on a , so we can instantiate it with a type parameter.
2203 Second, the fact that the general choice uses \mathcal{G} rather than \mathcal{S} indicates that any chosen
2204 instantiation needs to be a castable type. It follows that any concrete instantiation will have
2205 an impact on the runtime behavior; therefore it is best to instantiate a with $?$. However, type
2206 inference cannot instantiate a with $?$, and substitution cannot replace \mathcal{G} with $?$ either. This
2207 means that we need a syntactic refinement process of the translated programs in order to
2208 replace type parameters with allowed gradual types.

2209 **SYNTACTIC REFINEMENT.** We define syntactic refinement of the translated expressions as
2210 follows. As \mathcal{S} denotes no constraints at all, substituting it with any monotype would work.
2211 Here we arbitrarily use Int . We interpret \mathcal{G} as $?$ since any monotype could possibly lead to a
2212 cast error.

2213 **Definition 13** (Syntactic Refinement). The syntactic refinement of a translated expression s
2214 is denoted by $\lceil s \rceil$, and defined as follows:

	$\boxed{\text{Int}}$	=	Int
	\boxed{a}	=	a
	$\boxed{\sigma_1 \rightarrow \sigma_2}$	=	$\boxed{\sigma_1} \rightarrow \boxed{\sigma_2}$
2215	$\boxed{\forall a. \sigma}$	=	$\forall a. \boxed{\sigma}$
	$\boxed{?}$	=	$?$
	$\boxed{\mathcal{S}}$	=	Int
	$\boxed{\mathcal{G}}$	=	$?$

2216 Applying the syntactic refinement to the translated expression, we get

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

2217 where two \mathcal{G} are refined by $?$ as highlighted. It is easy to verify that both applying this expres-
2218 sion to 3 and to *true* now results in a translation that evaluates to a value.

2219 **REPRESENTATIVE TRANSLATIONS.** To decide whether one translation is more general than
2220 the other, we define a preorder between translations.

2221 **Definition 14** (Translation Pre-order). Suppose $\Psi \vdash^G e : \sigma \rightsquigarrow \boxed{s_1}$ and $\Psi \vdash^G e : \sigma \rightsquigarrow \boxed{s_2}$,
2222 we define $s_1 \leq s_2$ to mean $s_2 \equiv_\alpha S^P(s_1)$ for some S^P .

2223 **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
2224 to renaming of type parameters).

2225 The preorder between translations gives rise to a notion of what we call *representative*
2226 *translations*:

2227 **Definition 15** (Representative Translation). A translation s is said to be a representative
2228 translation of a typing derivation $\Psi \vdash^G e : \sigma \rightsquigarrow \boxed{s}$ if and only if for any other translation
2229 $\Psi \vdash^G e : \sigma \rightsquigarrow \boxed{s'}$ such that $s' \leq s$, we have $s \leq s'$. From now on we use r to denote a
2230 representative translation.

2231 An important property of representative translations, which we conjecture for the lack
2232 of rigorous proof, is that if there exists any translation of an expression that (after syntac-
2233 tic refinement) can reduce to a value, so can a representative translation of that expression.
2234 Conversely, if a representative translation runs into a blame, then no translation of that ex-
2235 pression can reduce to a value.

2236 **Conjecture 5.2** (Property of Representative Translations). For any expression e such that
2237 $\Psi \vdash^G e : \sigma \rightsquigarrow \boxed{s}$ 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

2238 • If $\mathcal{C}\{\lceil s \rceil\} \Downarrow n$, then $\mathcal{C}\{\lceil r \rceil\} \Downarrow n$.

2239 • If $\mathcal{C}\{\lceil r \rceil\} \Downarrow \text{blame}$, then $\mathcal{C}\{\lceil s \rceil\} \Downarrow \text{blame}$.

2240 Given this conjecture, we can state a stricter coherence property (without the “up to casts”
2241 part) between any two representative translations. We first strengthen Definition 8 following
2242 Ahmed et al. [2009]:

2243 **Definition 16** (Contextual Approximation à la Ahmed et al. [2009]).

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

2245 The only difference is that now when a program containing s_1 reduces to a value, so does
2246 one containing s_2 .

2247 From Conjecture 5.2, it follows that coherence holds between two representative transla-
2248 tions of the same expression.

2249 **Corollary 5.3** (Coherence for Representative Translations). *For any expression e such that*
2250 *$\Psi \vdash^G e : \sigma \rightsquigarrow r_1$ and $\Psi \vdash^G e : \sigma \rightsquigarrow r_2$, we have $\Psi \vdash r_1 \preceq_{ctx} r_2 : \sigma$.*

2251 We have proved that for every typing derivation, at least one representative translation
2252 exists.

2253 **Lemma 5.4** (Representative Translation for Typing). *For any typing derivation $\Psi \vdash^G e : \sigma$*
2254 *there exists at least one representative translation r such that $\Psi \vdash^G e : \sigma \rightsquigarrow r$.*

2255 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
2256 translation, while the other two are not.

2257 5.3 DYNAMIC GRADUAL GUARANTEE, RELOADED

2258 Given the above propositions, we are ready to revisit the dynamic gradual guarantee. The
2259 nice thing about representative translations is that the dynamic gradual guarantee of our
2260 source language is essentially that of λB , our target language. However, the dynamic gradual
2261 guarantee for λB is still an open question. According to Igarashi et al. [2017], the difficulty
2262 lies in the definition of term precision that preserves the semantics. We leave it here as a
2263 conjecture as well. From a declarative point of view, we cannot prevent the system from
2264 picking undesirable instantiations, but we know that some choices are better than the others,
2265 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.

2266 **Conjecture 5.5** (Dynamic Gradual Guarantee in terms of Representative Translations). *Sup-*
 2267 *pose* $e' \sqsubseteq e$,

- 2268 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
 2269 $\sigma_2 \sqsubseteq \sigma$, and $\lceil r' \rceil \Downarrow v'$, and $v' \sqsubseteq v$.
- 2270 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
 2271 $\sigma_2 \sqsubseteq \sigma$. 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.

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

2272 More importantly, in what follows, we show that our extended algorithm is able to find
 2273 those representative translations.

2274 5.4 EXTENDED ALGORITHMIC TYPE SYSTEM

2275 To understand the design choices involved in the new algorithmic system, we consider the
 2276 following algorithmic typing example:

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

2277 Compared with declarative typing, where we have many choices (e.g., $\text{Int} \rightarrow \text{Int}$, $\text{Bool} \rightarrow \text{Int}$,
 2278 and so on) to instantiate $\forall a. a \rightarrow \text{Int}$, the algorithm computes the instantiation $\hat{\alpha} \rightarrow \text{Int}$ with
 2279 $\hat{\alpha}$ unsolved in the output context. What can we know from the algorithmic typing? First we
 2280 know that, here $\hat{\alpha}$ is *not constrained* by the typing problem. Second, and more importantly,

2281 $\hat{\alpha}$ has been compared with an unknown type (when typing (fx)). Therefore, it is possible to
 2282 make a more refined distinction between different kinds of existential variables:

- 2283 • the first kind of existential variables are those that indeed have no constraints at all, as
 2284 they do not affect the dynamic semantics;
- 2285 • the second kind (as in this example) are those where the only constraint is that *the*
 2286 *variable was once compared with an unknown type* [Garcia and Cimini 2015].

2287 The syntax of types is shown in Figure 5.2. A notable difference, apart from the addition
 2288 of static and gradual parameters, is that we further split existential variables $\hat{\alpha}$ into static
 2289 existential variables $\hat{\alpha}_S$ and gradual existential variables $\hat{\alpha}_G$. Depending on whether an ex-
 2290 istential variable has been compared with $?$ or not, its solution space changes. More specifi-
 2291 cally, static existential variables can be solved to a monotype τ , whereas gradual existential
 2292 variables can only be solved to a castable monotype t , as can be seen in the changes of algo-
 2293 rithmic contexts and complete contexts. As a result, the typing result for the above example
 2294 now becomes

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

2295 since we can solve any unconstrained $\hat{\alpha}_G$ to \mathcal{G} , it is easy to verify that the resulting translation
 2296 is indeed a representative translation.

2297 Our extended algorithm is novel in the following aspects. We naturally extend the concept
 2298 of existential variables [Dunfield and Krishnaswami 2013] to deal with comparisons between
 2299 existential variables and unknown types. Unlike Garcia and Cimini [2015], where they use
 2300 an extra set to store types that have been compared with unknown types, our two kinds of
 2301 existential variables emphasize the type distinction better, and correspond more closely to
 2302 the two kinds of type parameters, as we can solve $\hat{\alpha}_S$ to \mathcal{S} and $\hat{\alpha}_G$ to \mathcal{G} .

2303 5.4.1 EXTENDED ALGORITHMIC CONSISTENT SUBTYPING

2304 While the changes in the syntax seem negligible, the addition of static and gradual type pa-
 2305 rameters changes the algorithmic judgments in a significant way. We first discuss the al-
 2306 gorithmic consistent subtyping, which is shown in Figure 5.3. For notational convenience,
 2307 when static and gradual existential variables have the same rule form, we compress them into
 2308 one rule. For example, rule **GPC-AS-EVAR** is really two rules $\Gamma[\hat{\alpha}_S] \vdash^G \hat{\alpha}_S \lesssim \hat{\alpha}_S \dashv \Gamma[\hat{\alpha}_S]$
 2309 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%;"> <div style="display: flex; justify-content: space-between;"> <div style="width: 48%;"> GPC-AS-UNKNOWNLL $\frac{}{\Gamma \vdash^G ? \lesssim \mathbb{C} \dashv \text{contaminate}(\Gamma, \mathbb{C})}$ </div> <div style="width: 48%;"> GPC-AS-UNKNOWNRR $\frac{}{\Gamma \vdash^G \mathbb{C} \lesssim ? \dashv \text{contaminate}(\Gamma, \mathbb{C})}$ </div> </div> </div>		
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
2332	$\text{contaminate}((\Gamma, \hat{\alpha}_S), \sigma)$	$=$	$\text{contaminate}(\Gamma, \sigma), \hat{\alpha}_S$ 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.

2336 Rule `GPC-AS-FORALLL` is slightly different from rule `GPC-AS-FORALL` in the original al-
 2337 gorithmic system in that we replace a with a new static existential variable $\hat{\alpha}_S$. Note that $\hat{\alpha}_S$
 2338 might be solved to a gradual existential variable later. The rest of the rules are the same as
 2339 those in the original system.

2340 5.4.2 EXTENDED INSTANTIATION

2341 The instantiation judgments shown in Figure 5.4 also change significantly. The complication
 2342 comes from the fact that now we have two different kinds of existential variables, and the
 2343 relative order that they appear in the context affects their solutions.

2344 Rules `GPC-INSTL-SOLVES` and `GPC-INSTL-SOLVEG` are the refinement to rule `GPC-INSTL-`
 2345 `SOLVE` in the original system. The next two rules deal with situations where one side is an
 2346 existential variable and the other side is an unknown type. Rule `GPC-INSTL-SOLVEUS` is a
 2347 special case of rule `GPC-AS-UNKNOWNRR` where we create a new gradual existential variable
 2348 $\hat{\alpha}_G$ and set the solution of $\hat{\alpha}_S$ to be $\hat{\alpha}_G$ in the output context. Rule `GPC-INSTL-SOLVEUG` is
 2349 the same as rule `GPC-INSTL-SOLVEU` in the original system and simply propagates the input
 2350 context. The next two rules `GPC-INSTL-REACHSG1` and `GPC-INSTL-REACHSG2` are a bit in-
 2351 volved, but they both answer to the same question: how to solve a gradual existential variable
 2352 when it is declared after some static existential variable. More concretely, in rule `GPC-INSTL-`
 2353 `REACHSG1`, we feel that we need to solve $\hat{\beta}_G$ to another existential variable. However, simply
 2354 setting $\hat{\beta}_G = \hat{\alpha}_S$ and leaving $\hat{\alpha}_S$ untouched in the output context is wrong. The reason is
 2355 that $\hat{\beta}_G$ could be a gradual existential variable created by rule `GPC-AS-UNKNOWNLL`/`GPC-AS-`
 2356 `UNKNOWNRR` and solving $\hat{\beta}_G$ to a static existential variable would result in the same problem
 2357 as we have discussed. Instead, we create another new gradual existential variable $\hat{\alpha}_G$ and set
 2358 the solutions of both $\hat{\alpha}_S$ and $\hat{\beta}_G$ to it; similarly in rule `GPC-INSTL-REACHSG2`. Rule `GPC-`
 2359 `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
 2360 those cases, we employ the same strategy as in the original system.

2361 As for the other instantiation judgment, most of the rules are symmetric and thus omit-
 2362 ted. The only interesting rule is `GPC-ISTR-FORALLL`, which is similar to what we did for
 2363 rule `GPC-AS-FORALLL`.

2364 5.4.3 ALGORITHMIC TYPING AND METATHEORY

2365 Fortunately, the changes in the algorithmic bidirectional system are minimal: we replace
 2366 every existential variable with a static existential variable. Furthermore, we proved that the
 2367 extended algorithmic system is sound and complete with respect to the extended declarative
 2368 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)

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

Figure 5.4: Instantiation in the extended algorithmic system

2369 5.4.4 DISCUSSION

2370 DO WE REALLY NEED TYPE PARAMETERS IN THE ALGORITHMIC SYSTEM? As we mentioned
 2371 earlier, type parameters in the declarative system are merely an analysis tool, and in practice,
 2372 type parameters are inaccessible to programmers. For the sake of proving soundness and
 2373 completeness, we have to endow the algorithmic system with type parameters. However,
 2374 the algorithmic system already has static and gradual existential variables, which can serve
 2375 the same purpose. In that regard, we could directly solve every *unsolved* static and gradual
 2376 existential variable in the output context to `Int` and `?`, respectively.

2377 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:

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

2378 The difference comes from the fact that in the second translation, we apply rule `GEN` while
 2379 typing `1` to get $\bullet \vdash 1 : \forall a. \text{Int}$. As a consequence, the translation of `1` is accompanied by a
 2380 cast from $\forall a. \text{Int}$ to `Int` since the former is a consistent subtype of the latter. This difference is
 2381 harmless, because obviously these two expressions will reduce to the same value in λB , thus
 2382 preserving coherence (up to cast error). While it is not going to break coherence, it does result
 2383 in multiple representative translations for one expression (e.g., the above two translations are
 2384 both the representative translations).

2385 There are several ways to make the translation process more deterministic. For example,
 2386 we can restrict generalization to happen only in `let` expressions and require `let` expressions
 2387 to include annotations, as `let $x : \sigma = e_1$ in e_2` . Another feasible option would be to give a
 2388 declarative, bidirectional system as the specification (instead of the type assignment one), in
 2389 the same spirit of DK [Dunfield and Krishnaswami 2013]. Then we can restrict generalization
 2390 to be performed through annotations in checking mode.

2391 With restricted generalization, we hypothesize that now each expression has exactly one
 2392 representative translation (up to renaming of fresh type parameters). Instead of calling it a

2393 *representative* translation, we can say it is a *principal* translation. Of course the above is only
2394 a sketch; we have not defined the corresponding rules, nor studied metatheory.

PART IV

TYPE INFERENCE WITH PROMOTION

6

HIGHER-RANK TYPE INFERENCE

ALGORITHM WITH TYPE PROMOTION

Designing type inference algorithms is challenging. In particular, while a declarative type system can *guess* a type (e.g., the type of x in $\lambda x. e$) and can even be *non-deterministic* in the guessing (e.g., multiple possible types for x in $\lambda x. x$), an type inference algorithm needs to be deterministic, and thus has to deal with many low-level details including *unification*. At the same time we expect an type inference algorithm to retain desirable properties like *inference of principal types*, as well as *soundness* and *completeness* with respect to the declarative type system. An algorithm is sound and complete, if it accepts and only accepts programs that are well-typed in the declarative type system.

In this chapter, we focus on the design of type inference algorithms in the presence of higher-rank polymorphism. Compared to type inference for simple types, type inference for higher-rank polymorphism needs to further take care of scoping and dependency issues between different kinds of variables. We propose a strategy called *promotion* that helps resolve the dependency of unification variables in the framework of *type inference in context* [Gundry et al. 2010]. To illustrate the key idea, Section 6.2 applies promotion to the unification algorithm for the simply typed lambda calculus. Section 6.3 then proposes *polymorphic promotion* to deal with subtyping for higher-rank polymorphism, which leads to an arguably simpler type inference algorithm for higher-rank polymorphism. Finally, we briefly discuss how promotion can be further applied to other advanced features like 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

Given a declarative type system, the goal of type inference is to reconstruct missing type annotations for the system in a deterministic way that is also sound and complete with respect to the declarative specification. In its standard form, a type inference algorithm represents

2422 missing type annotations using *unification variables* that will be solved during the process of
 2423 *unification* (or more generally, *constraint solving*).

Gundry et al. [2010] model unification and type inference for an ML-style polymorphic system from a general perspective of *information increase* in the problem context. Specifically, they formalize the unification process as:

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

The judgment reads that under the input context Γ , unifying τ_1 with τ_2 yields the output context Δ . The type contexts (Γ and Δ) are used to keep track of the solutions of unification variables. We say that the output context Δ is more *informative* than the original one Γ . For example, the following derivation unifies $\hat{\alpha}$ with Int , which leads to the solution $\hat{\alpha} = \text{Int}$.

$$\hat{\alpha}, \hat{\beta} \vdash^u \hat{\alpha} \approx \text{Int} \dashv \hat{\alpha} = \text{Int}, \hat{\beta}$$

2424 To ensure that the algorithm produces only solutions, the algorithm depends on the notion
 2425 of *well-founded* contexts; that is, unification variable solutions may depend only on earlier
 2426 bindings. For example, $\hat{\alpha}, \hat{\beta} = \hat{\alpha} \rightarrow \text{Int}$ is well-founded, while $\hat{\alpha} = \hat{\beta} \rightarrow \text{Int}, \hat{\beta}$ is not, as the
 2427 binding $\hat{\alpha} = \hat{\beta} \rightarrow \text{Int}$ refers to the variable $\hat{\beta}$ bound later in the type context. Well-founded
 2428 contexts effectively rule out invalid cyclic solutions like $\hat{\alpha} = \hat{\beta} \rightarrow \text{Int}, \hat{\beta} = \hat{\alpha} \rightarrow \text{Int}$.

Besides contexts being ordered, a key insight of their approach lies in how to resolve the dependency between unification variables. Consider the following unification problem

$$\hat{\alpha}, \hat{\beta} \vdash^u \hat{\alpha} \approx \hat{\beta} \rightarrow \text{Int}$$

2429 Here we cannot simply set $\hat{\alpha}$ to $\hat{\beta} \rightarrow \text{Int}$, as $\hat{\beta}$ is out of the scope of $\hat{\alpha}$. Thus, Gundry et al.
 2430 [2010] solve this problem by examining variables in the context from the tail to the head, and
 2431 *moving segments of the context to the left if necessary*. In this case, Gundry et al. [2010] would
 2432 return a solution context $\hat{\beta}, \hat{\alpha} = \hat{\beta} \rightarrow \text{Int}$, where $\hat{\beta}$ is moved to the front of $\hat{\alpha}$. However,
 2433 while moving type variables around is a feasible way to resolve the dependency between
 2434 unification variables, the unpredictable context movements make the information increase
 2435 hard to formalize and reason about. In their system, the information increase of contexts is
 2436 defined in a semantic way: a context Γ_1 is more informative than another context Γ_2 , if there
 2437 exists a substitution such that every item in Γ_2 is, after context substitution, well-formed
 2438 under Γ_1 . This semantic definition makes it hard to prove metatheory formally, especially
 2439 when advanced features are involved.

2440 The Dunfield-Krishnaswami type system (DK) [Dunfield and Krishnaswami 2013] also
 2441 uses ordered contexts as the input and output of the type inference algorithm for a higher-
 2442 rank polymorphic type system (Section 2.3). Unlike Gundry et al. [2010], DK does it in
 2443 a more syntactic way. In their type system, instantiation rules decompose type constructs
 2444 so that unification between unification variables can only happen between single variables,
 2445 which can then be solved by setting the one that appears later to the one that appears earlier.
 2446 This way, the information increase of contexts is modeled as the intuitive and syntactic def-
 2447 inition of *context extension* ($\Gamma \longrightarrow \Delta$), which allows for inductive reasoning and proofs. As
 2448 an example, consider we want to unify (\approx) $\hat{\alpha}$ with $\hat{\beta} \rightarrow \text{Int}$ under the context $\hat{\alpha}, \hat{\beta}$ using DK's
 2449 approach. Recall that we use the hole notation for manipulating contexts by inserting and
 2450 replacing declarations in the middle. For example, $\Gamma[\Theta]$ means Γ has the form $\Gamma_L, \Theta, \Gamma_R$; if
 2451 we have $\Gamma[\hat{\alpha}] = (\Gamma_L, \hat{\alpha}, \Gamma_R)$, then $\Gamma[\hat{\alpha} = \tau] = (\Gamma_L, \hat{\alpha} = \tau, \Gamma_R)$.

$$\Delta = \hat{\alpha}_1, \hat{\alpha}_2, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2, \hat{\beta}$$

$$\frac{\frac{\Delta \vdash \hat{\beta} \approx \hat{\alpha}_1 \dashv \Delta[\hat{\beta} = \hat{\alpha}_1]}{\hat{\alpha}, \hat{\beta} \vdash \hat{\alpha} \approx \hat{\beta} \rightarrow \text{Int} \dashv \Delta[\hat{\alpha}_2 = \text{Int}][\hat{\beta} = \hat{\alpha}_1]} \quad \frac{\Delta[\hat{\beta} = \hat{\alpha}_1] \vdash \hat{\alpha}_2 \approx \text{Int} \dashv \Delta[\hat{\alpha}_2 = \text{Int}][\hat{\beta} = \hat{\alpha}_1]}{\hat{\alpha}, \hat{\beta} \vdash \hat{\alpha} \approx \hat{\beta} \rightarrow \text{Int} \dashv \Delta[\hat{\alpha}_2 = \text{Int}][\hat{\beta} = \hat{\alpha}_1]}}$$

2452 Let us step through the derivation. As $\hat{\alpha}$ is unified with an arrow type, we know that $\hat{\alpha}$
 2453 must be an arrow. Thus the variable $\hat{\alpha}$ is solved by an arrow type $\hat{\alpha}_1 \rightarrow \hat{\alpha}_2$ consisting of
 2454 two fresh unification variables. The two variables $\hat{\alpha}_1$ and $\hat{\alpha}_2$ are then unified with $\hat{\beta}$ and
 2455 Int , respectively. Unifying $\hat{\alpha}_1$ with $\hat{\beta}$ solves $\hat{\beta}$ by $\hat{\alpha}_1$, as $\hat{\alpha}_1$ appears in the context before $\hat{\beta}$;
 2456 or otherwise we can solve $\hat{\alpha}_1$ by $\hat{\beta}$. The final solution context is $\Delta[\hat{\alpha}_2 = \text{Int}][\hat{\beta} = \hat{\alpha}_1] =$
 2457 $\hat{\alpha}_1, \hat{\alpha}_2 = \text{Int}, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2, \hat{\beta} = \hat{\alpha}_1$.

CHALLENGES. However, while the approach of decomposing type constructs works perfectly for this example so far, it has two drawbacks. First, it produces duplication: in order to deal with both cases, the unification rules are duplicated for when the unification variable appears on the left (i.e., $\Gamma \vdash \hat{\alpha} \approx \tau \dashv \Delta$), and when it appears on the right (i.e., $\Gamma \vdash \tau \approx \hat{\alpha} \dashv \Delta$). For example, the following two rules are symmetric to each other:

$$\frac{\Gamma[\hat{\alpha}_1, \hat{\alpha}_2, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \vdash \hat{\alpha}_1 \approx \tau_1 \dashv \Delta_1 \quad \Delta_1 \vdash \hat{\alpha}_2 \approx [\Delta_1]\tau_1 \dashv \Delta_2}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} \approx \tau_1 \rightarrow \tau_2 \dashv \Delta_2}}$$

$$\frac{\Gamma[\hat{\alpha}_1, \hat{\alpha}_2, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \Vdash \hat{\alpha}_1 \approx \tau_1 \dashv \Delta_1 \quad \Delta_1 \Vdash \hat{\alpha}_2 \approx [\Delta_1]\tau_1 \dashv \Delta_2}{\Gamma[\hat{\alpha}] \Vdash \tau_1 \rightarrow \tau_2 \approx \hat{\alpha} \dashv \Delta_2}$$

2458 Worse, this kind of “duplication” would scale up with the number of type constructs in the
2459 system.

Second, while decomposition works for function types, it may not work easily for more complicated types, e.g., dependent types. For example, consider the following unification problem, in which $\Pi a : \tau_1.\tau_2$ is a dependent type where a may appear in τ_2 .

$$\hat{\alpha}, \hat{\beta} \Vdash \hat{\alpha} \approx \Pi a : \hat{\beta}.a$$

2460 Here because $\hat{\beta}$ appears after $\hat{\alpha}$, we cannot directly set $\hat{\alpha} = \Pi a : \hat{\beta}.a$, which is ill-typed.
2461 However, if we try to decompose the type $\Pi a : \hat{\beta}.a$ like before, in which case we have $\hat{\alpha} =$
2462 $\Pi a : \hat{\alpha}_1.\hat{\alpha}_2$, it is obvious that $\hat{\alpha}_2$ should be solved by a . Then, in order to make the solution
2463 well typed, we need to put a in the front of $\hat{\alpha}_2$ in the context. However, this means that a
2464 would remain in the context, and it would be available for any later unification variables that
2465 should not have access to a .

2466 6.1.2 OUR APPROACH: TYPE PROMOTION

2467 We propose the *promotion* process, which helps resolve the dependency between unification
2468 variables. Promotion combines the advantages of Gundry et al. [2010] and DK: it is a simple
2469 and predictable process, so that information increase can still be modeled as the syntactic
2470 context extension; moreover, it does not cause any duplication.

To understand how promotion works, let us consider again the unification problem

$$\hat{\alpha}, \hat{\beta} \Vdash \hat{\alpha} \approx \hat{\beta} \rightarrow \text{Int}$$

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 unification variables in the type, and solves those out-of-scope unification variables with fresh ones added to the front of $\hat{\alpha}$, such that unification 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

$$\hat{\alpha}_1, \hat{\alpha}, \hat{\beta} = \hat{\alpha}_1 \Vdash \hat{\alpha} \approx \hat{\alpha}_1 \rightarrow \hat{\beta}$$

Notice that $\hat{\alpha}_1$ is inserted right before $\hat{\alpha}$. Now after substitution $\hat{\beta}$ with $\hat{\alpha}_1$, the unification example becomes

$$\hat{\alpha}_1, \hat{\alpha}, \hat{\beta} = \hat{\alpha}_1 \vdash^u \hat{\alpha} \approx \hat{\alpha}_1 \rightarrow \text{Int}$$

and $\hat{\alpha}_1 \rightarrow \text{Int}$ is a valid solution for $\hat{\alpha}$. Therefore, we get a final solution context $\hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \text{Int}, \hat{\beta} = \hat{\alpha}_1$. Namely,

$$\hat{\alpha}, \hat{\beta} \vdash^u \hat{\alpha} \approx \hat{\beta} \rightarrow \text{Int} \dashv \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \text{Int}, \hat{\beta} = \hat{\alpha}_1$$

2471 Comparing the result with the solution context we get from DK ($\hat{\alpha}_1, \hat{\alpha}_2 = \text{Int}, \hat{\alpha} = \hat{\alpha}_1 \rightarrow$
2472 $\hat{\alpha}_2, \hat{\beta} = \hat{\alpha}_1$), it is obvious that these two solutions are equivalent up to substitution.

2473 **INTERPRETATION OF PROMOTION.** The key insight of the promotion process is that *the rela-*
2474 *tive order between unification variables does not matter for solving a constraint.* Thus its task is
2475 to “move” unification variables to suitable positions *indirectly*, by solving those out-of-scope
2476 unification variables with fresh in-scope ones.

2477 The observation seems to go against the design principle that the contexts are ordered.
2478 However, ordering is still important for variables whose order matters. For instance, for
2479 polymorphic types, the order between unification variables $\hat{\alpha}$ and type variables a is impor-
2480 tant, so we cannot set $\hat{\alpha}$ to a under the context $(\hat{\alpha}, a)$. Moreover, ordering still prevents
2481 invalid cyclic contexts, e.g., $\hat{\alpha} = \hat{\beta} \rightarrow \text{Int}, \hat{\beta} = \hat{\alpha} \rightarrow \text{Int}$.

2482 **UNIFICATION FOR THE SIMPLY TYPED LAMBDA CALCULUS.** As a first illustration of the pro-
2483 motion process, Section 6.2 recasts the unification process for the simply typed lambda cal-
2484 culus (STLC) using the promotion process. Note that using promotion for unification is
2485 overkill for STLC, but this system illustrates the key idea of promotion and sets us the stage
2486 for later advanced applications.

2487 6.1.3 POLYMORPHIC PROMOTION

The promotion process we described so far only works for unification. In this section, we dis-
cuss applying promotion to polymorphic subtyping in a higher-rank polymorphic calculus.
The challenge raised by higher-rank polymorphism is that, instead of unification for type
equality, higher-rank polymorphism takes care of *polymorphic subtyping*, which allows an

expression to have a subtype of the expected type. On the other hand, subtyping also needs to take unification into account at the same time. For example, for the subtyping constraint

$$\Gamma \vdash^{\text{sub}} \hat{\alpha} <: \text{Int}$$

2488 For this case, promotion works like before: it tries to promote unification variables in Int ,
 2489 where there is none so the promoted type remains Int . We can then solve $\hat{\alpha}$ by Int , which gives
 2490 us the output context $\hat{\alpha} = \text{Int}$. Indeed, the only valid solution in this case is $\hat{\alpha} = \text{Int}$. Recall
 2491 that unification variables can only be solved by *monotypes*, which is what makes unification
 2492 decidable and have principal solutions.

As the above example shows, for monotypes, promotion works just like before. However, with higher-rank polymorphism, we may have polymorphic types in the constraint. Consider

$$\hat{\alpha} \vdash^{\text{sub}} \forall a. a \rightarrow a <: \hat{\alpha}$$

2493 How can we promote the polymorphic type $\forall a. a \rightarrow a$ into a monotype which can serve as
 2494 a valid solution for $\hat{\alpha}$? One possible answer is to set $\hat{\alpha} = \text{Int} \rightarrow \text{Int}$, or $\hat{\alpha} = \text{Bool} \rightarrow \text{Bool}$.
 2495 In fact, the most general solution for this subtyping problem is $\hat{\alpha} = \hat{\beta} \rightarrow \hat{\beta}$ with fresh
 2496 $\hat{\beta}$. Namely, we remove the universal quantifier in $\forall a. a \rightarrow a$ and replace the variable a
 2497 with a fresh unification variable $\hat{\beta}$ added to the front of $\hat{\alpha}$, resulting in the solution context
 2498 $\hat{\beta}, \hat{\alpha} = \hat{\beta} \rightarrow \hat{\beta}$.

What if the subtyping relation is the other way around? Namely,

$$\hat{\alpha} \vdash^{\text{sub}} \hat{\alpha} <: \forall a. a \rightarrow a$$

2499 Now how can we promote the type $\forall a. a \rightarrow a$? It turns out that this subtyping is actually
 2500 unsolvable, as there is no monotype that can be a subtype of $\forall a. a \rightarrow a$. Therefore, in this
 2501 case, promoting $\forall a. a \rightarrow a$ will directly add the type variable a to the tail of the context to
 2502 promote $a \rightarrow a$. Since a is added to the tail, it means that a is out of the scope of $\hat{\alpha}$ and
 2503 promoting $a \rightarrow a$ would fail, which is exactly what we want. In fact, the promotion would
 2504 succeed only if the universally quantified variable is not used in the body of the polymorphic
 2505 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
 2506 $\hat{\alpha}$ in $\hat{\alpha} \vdash^{\text{sub}} \hat{\alpha} <: \forall a. \text{Int} \rightarrow \text{Int}$.

2507 From these observations, we extend promotion to *polymorphic promotion*, which is able to
 2508 resolve the polymorphic subtyping relation for unification variables. Depending on whether

the unification 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 $\hat{\alpha} \vdash^{\text{sub}} \forall a. a \rightarrow a <: \hat{\alpha}$, where the universal quantifier is removed and the type variable a is replaced by a fresh unification variable added to front of the unification variable being solved.

On the other hand, the covariant mode promotes types as $\forall a. a \rightarrow a$ in the case of $\hat{\alpha} \vdash^{\text{sub}} \hat{\alpha} <: \forall a. a \rightarrow a$. In this case, promoting $\forall a. a \rightarrow a$ will directly add the type variable a to the tail of the context. Since the type variable is out of the scope of the unification 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

$$\hat{\alpha} \vdash^{\text{sub}} \hat{\alpha} <: (\forall a. a \rightarrow a) \rightarrow (\forall a. \text{Int} \rightarrow \text{Int})$$

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 [Dunfield and Krishnaswami 2013] can be replaced by our polymorphic promotion process. Furthermore, we show that subtyping, which was built upon instantiation but now uses polymorphic promotion, remains sound and complete.

6.2 UNIFICATION FOR THE SIMPLY TYPED LAMBDA CALCULUS

This section first introduces the simply typed lambda calculus (STLC), and then presents a unification algorithm that uses the novel promotion mechanism. As we have mentioned before, using promotion for unification is overkill for STLC. So the purpose of this section is to illustrate the key idea of promotion in the context of STLC, which sets us the stage for later advanced applications.

Expressions	e	$::=$	$x \mid n \mid \lambda x : \tau. e \mid \lambda x. e \mid e_1 e_2$
Monotypes	τ	$::=$	$\text{Int} \mid \tau_1 \rightarrow \tau_2$
Contexts	Ψ	$::=$	$\bullet \mid \Psi, x : \tau$

$\Psi \vdash e : \tau$

(Typing)

$\frac{\text{T-VAR} \quad (x : \tau) \in \Psi}{\Psi \vdash x : \tau}$	$\frac{\text{T-INT}}{\Psi \vdash n : \text{Int}}$	$\frac{\text{T-LAMANN} \quad \Psi, x : \tau_1 \vdash e : \tau_2}{\Psi \vdash \lambda x : \tau. e : \tau_1 \rightarrow \tau_2}$	$\frac{\text{T-LAM} \quad \Psi, x : \tau_1 \vdash e : \tau_2}{\Psi \vdash \lambda x. e : \tau_1 \rightarrow \tau_2}$
$\frac{\text{T-APP} \quad \Psi \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Psi \vdash e_2 : \tau_1}{\Psi \vdash e_1 e_2 : \tau_2}$			

Figure 6.1: Syntax and declarative typing of the simply typed lambda calculus.

2533 6.2.1 DECLARATIVE SYSTEM

2534 The declarative specification of STLC is given in Figure 6.1.

2535 Expressions e include variables x , integers n , annotated lambda abstractions $\lambda x : \tau. e$,
 2536 un-annotated lambda abstractions $\lambda x. e$, and applications $e_1 e_2$. We have only monotypes τ ,
 2537 which includes the integer type Int and function types $\tau_1 \rightarrow \tau_2$. Contexts Ψ keep track of
 2538 the types of the variables.

2539 The typing rules are standard. For unannotated lambdas (rule [T-LAMANN](#)), the system
 2540 guesses a monotype τ .

2541 6.2.2 ALGORITHMIC SYSTEM

2542 The syntax of the algorithmic system is given in Figure 6.2. Following DK [Dunfield and
 2543 Krishnaswami 2013], algorithmic monotypes include unification type variables $\hat{\alpha}$. Algorithmic
 2544 contexts also contain declarations of unification type variables, either unsolved ($\hat{\alpha}$) or
 2545 solved ($\hat{\alpha} = \tau$). Complete contexts Ω contain only solved variables. We use the judgment
 2546 $\Gamma \vdash^{\text{wf}} \tau$ to indicate that all unification variables in τ are well-scoped, and $\Gamma \text{ ok}$ to indicate
 2547 well-formed contexts, where each binding is well-formed in its prefix context.

2548 Following Dunfield and Krishnaswami [2013], we use contexts as substitutions on types.
 2549 We write $[\Gamma]\tau$ to mean Γ applied as a substitution to type τ , whose definition is given below.

Expressions	$e ::= x \mid n \mid \lambda x : \tau. e \mid \lambda x. e \mid e_1 e_2$	
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	$\Omega ::= \bullet \mid \Omega, \hat{\alpha} = \tau$	

$\Gamma \vdash^{\text{wf}} \tau$

(Well-formedness)

$\frac{\text{WF-INT}}{\Gamma \vdash^{\text{wf}} \text{Int}}$	$\frac{\text{WF-ARROW} \quad \Gamma \vdash^{\text{wf}} \tau_1 \quad \Gamma \vdash^{\text{wf}} \tau_2}{\Gamma \vdash^{\text{wf}} \tau_1 \rightarrow \tau_2}$	$\frac{\text{WF-EVAR} \quad \hat{\alpha} \in \Gamma}{\Gamma \vdash^{\text{wf}} \hat{\alpha}}$
--	---	---

$\Gamma \text{ ok}$

(Well-formedness of algorithmic contexts)

$\frac{\text{WFC-EMPTY}}{\bullet \text{ ok}}$	$\frac{\text{WFC-EVAR} \quad \Gamma \text{ ok} \quad \hat{\alpha} \notin \text{FV}(\Gamma)}{\Gamma, \hat{\alpha} \text{ ok}}$	$\frac{\text{WFC-EVAR SOLVED} \quad \Gamma \text{ ok} \quad \Gamma \vdash^{\text{wf}} \tau \quad \hat{\alpha} \notin \text{FV}(\Gamma)}{\Gamma, \hat{\alpha} = \tau \text{ ok}}$
---	---	--

Figure 6.2: Syntax of algorithmic STLC

$[\Gamma] \text{Int}$	$=$	Int
$[\Gamma[\hat{\alpha}]] \hat{\alpha}$	$=$	$\hat{\alpha}$
$[\Gamma[\hat{\alpha} = \tau]] \hat{\alpha}$	$=$	$[\Gamma[\hat{\alpha} = \tau]] \tau$
$[\Gamma] \tau_1 \rightarrow \tau_2$	$=$	$[\Gamma] \tau_1 \rightarrow [\Gamma] \tau_2$

2551 **ALGORITHMIC TYPING.** Figure 6.3 defines algorithmic typing. The judgment $\Gamma \vdash e : \tau \dashv \Delta$
 2552 reads that under the input context Γ , the expression e has type τ , with the output context Δ .
 2553 Most rules are standard. In rule [AT-LAMANN](#), instead of guessing a type as in its declarative
 2554 counterpart, the rule generates a fresh unification variable $\hat{\alpha}$ as the type of x , and another
 2555 fresh unification variable $\hat{\beta}$ as the type of the body, and continues type-checking the body.
 2556 After getting the type τ of the body, we unify $\hat{\beta}$ with $[\Theta_1] \tau$, and return the result type $\hat{\alpha} \rightarrow \hat{\beta}$,
 2557 where the trailing context Θ is discarded.

2558 Here are several notable things about rule [AT-LAMANN](#). First, we do not directly return
 2559 $\hat{\alpha} \rightarrow \tau$ as the result type, as otherwise we cannot discard the type context Θ after x , since
 2560 the type τ may refer to unification variables in Θ . Discarding Θ is mainly an optimization
 2561 step. In order to safely remove Θ , we unify $\hat{\beta} \approx [\Theta_1] \tau$, and return $\hat{\alpha} \rightarrow \hat{\beta}$ as the result
 2562 type instead, which ensures that the result type is well-formed under the context Δ (where
 2563 both $\hat{\alpha}$ and $\hat{\beta}$ are bound). Second, we do not simply unify $\hat{\beta} \approx \tau$, but $\hat{\beta} \approx [\Theta_1] \tau$, where

$\Gamma \vdash e : \tau \dashv \Delta$

(Algorithmic typing)

$$\begin{array}{c}
 \text{AT-VAR} \quad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau \dashv \Gamma} \qquad \text{AT-INT} \quad \frac{}{\Gamma \vdash n : \text{Int} \dashv \Gamma} \qquad \text{AT-LAMANN} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \dashv \Delta}{\Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2 \dashv \Delta} \\
 \\
 \text{AT-LAM} \quad \frac{\Gamma, \hat{\alpha}, \hat{\beta}, x : \hat{\alpha} \vdash e : \tau \dashv \Theta_1 \quad \Theta_1 \Vdash \hat{\beta} \approx [\Theta_1]\tau \dashv \Delta, x : \hat{\alpha}, \Theta}{\Gamma \vdash \lambda x. e : \hat{\alpha} \rightarrow \hat{\beta} \dashv \Delta} \\
 \\
 \text{AT-APP} \quad \frac{\Gamma \vdash e_1 : \tau_1 \dashv \Theta_1 \quad \Theta_1 \vdash e_2 : \tau_2 \dashv \Theta_2 \quad \Theta_2, \hat{\alpha} \Vdash [\Theta_2]\tau_1 \approx [\Theta_2]\tau_2 \rightarrow \hat{\alpha} \dashv \Delta}{\Gamma \vdash e_1 e_2 : \hat{\alpha} \dashv \Delta}
 \end{array}$$

$\Gamma \Vdash \tau_1 \approx \tau_2 \dashv \Delta$

(Unification)

$$\begin{array}{c}
 \text{U-REFL} \quad \frac{}{\Gamma \Vdash \tau \approx \tau \dashv \Gamma} \qquad \text{U-ARROW} \quad \frac{\Gamma \Vdash \tau_1 \approx \tau_3 \dashv \Theta \quad \Theta \Vdash [\Theta]\tau_2 \approx [\Theta]\tau_4 \dashv \Delta}{\Gamma \Vdash \tau_1 \rightarrow \tau_2 \approx \tau_3 \rightarrow \tau_4 \dashv \Delta} \\
 \\
 \text{U-EVARL} \quad \frac{\Gamma \Vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta[\hat{\alpha}]}{\Gamma \Vdash \hat{\alpha} \approx \tau_1 \dashv \Delta[\hat{\alpha} = \tau_2]} \qquad \text{U-EVARR} \quad \frac{\Gamma \Vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta[\hat{\alpha}]}{\Gamma \Vdash \tau_1 \approx \hat{\alpha} \dashv \Delta[\hat{\alpha} = \tau_2]}
 \end{array}$$

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

(Promotion)

$$\begin{array}{c}
 \text{PR-INT} \quad \frac{}{\Gamma \Vdash_{\hat{\alpha}}^{\text{pr}} \text{Int} \rightsquigarrow \text{Int} \dashv \Gamma} \qquad \text{PR-ARROW} \quad \frac{\Gamma \Vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_3 \dashv \Theta \quad \Theta \Vdash_{\hat{\alpha}}^{\text{pr}} [\Theta]\tau_2 \rightsquigarrow \tau_4 \dashv \Delta}{\Gamma \Vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightarrow \tau_2 \rightsquigarrow \tau_3 \rightarrow \tau_4 \dashv \Delta} \\
 \\
 \text{PR-EVARL} \quad \frac{}{\Gamma[\hat{\beta}][\hat{\alpha}] \Vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \Gamma[\hat{\beta}][\hat{\alpha}]} \qquad \text{PR-EVARR} \quad \frac{}{\Gamma[\hat{\alpha}][\hat{\beta}] \Vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta}_1 \dashv \Gamma[\hat{\beta}_1, \hat{\alpha}][\hat{\beta} = \hat{\beta}_1]}
 \end{array}$$

Figure 6.3: Algorithmic typing, promotion and unification of algorithmic STLC

the context Θ_1 is applied to τ (we do not apply Θ_1 to $\hat{\beta}$ as given $\hat{\beta}$ is a fresh unification variable not referred anywhere yet, we know $[\Theta_1]\hat{\beta} = \hat{\beta}$). This maintains an invariant: in the unification derivation $\Gamma \Vdash \tau_1 \approx \tau_2 \dashv \Delta$, the types τ_1 and τ_2 are fully applied under Γ . This invariant simplifies the unification process: we know that every unification variable we encounter during unification is unsolved and we do not need to discuss the case when it is solved where we need to replace it with its solution.

In rule **AT-APP**, we first infer the type of the function τ_1 , and the type of the argument τ_2 . As we know that τ_1 must be an arrow type, we create a fresh unification variable $\hat{\alpha}$ as the result type of the application, and unify $[\Theta_2]\tau_1 \approx [\Theta_2]\tau_2 \rightarrow \hat{\alpha}$. Again notice we apply the context to the types before we unify them.

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

PROMOTION. The promotion relation $\Gamma \Vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta$ given at the bottom of Figure 6.3 reads that under the input context Δ , promoting type τ_1 yields type τ_2 , so that τ_2 is well-formed in the prefix context of $\hat{\alpha}$, while retaining $[\Delta]\tau_1 = [\Delta]\tau_2$. As τ_2 is well-formed in the prefix context of $\hat{\alpha}$, recall that in the unification rule (e.g. rule **U-EVARL**) it is well-scoped to solve $\hat{\alpha}$ by τ_2 .

At a high-level, \Vdash^{pr} looks for free unification variables in τ_1 . Integers are always well-formed (rule **PR-INT**). Promoting a function recursively promotes its components (rule **PR-ARROW**). Variables bound to the left of $\hat{\alpha}$ in Γ are unaffected (rule **PR-EVARL**), as they are already well-formed. In rule **PR-EVARR**, a unification variable $\hat{\beta}$ bound to the right of $\hat{\alpha}$ in Γ is replaced by a fresh variable introduced to $\hat{\alpha}$'s left. Promotion is a partial operation, as it requires $\hat{\beta}$ either to be to the right or to the left of $\hat{\alpha}$, i.e., $\hat{\beta} \neq \hat{\alpha}$. There is yet another possibility: if $\hat{\beta} = \hat{\alpha}$, then no rule applies. This is a desired property, as the $\hat{\beta} = \hat{\alpha}$ case exactly corresponds to the “occurs-check” in a more typical presentation of unification. By preventing promoting $\hat{\alpha}$ to the left of $\hat{\alpha}$, we prevent the possibility of an infinite substitution when applying an algorithmic context. Note that rule **U-REFL** solves the unification case $\hat{\alpha} \approx \hat{\alpha}$.

$$\boxed{\Gamma \longrightarrow \Delta} \quad (\text{Context Extension})$$

$$\begin{array}{c}
 \text{EXT-ID} \\
 \frac{}{\bullet \longrightarrow \bullet} \\
 \\
 \text{EXT-SOLVED} \\
 \frac{\Gamma \longrightarrow \Delta}{\Gamma, \hat{\alpha} = \tau \longrightarrow \Delta, \hat{\alpha} = \tau'} \\
 \\
 \text{EXT-VAR} \\
 \frac{\Gamma \longrightarrow \Delta \quad [\Delta]\tau = [\Delta]\tau'}{\Gamma, x : \tau \longrightarrow \Delta, x : \tau'} \\
 \\
 \text{EXT-EVAR} \\
 \frac{\Gamma \longrightarrow \Delta}{\Gamma, \hat{\alpha} \longrightarrow \Delta, \hat{\alpha}} \\
 \\
 \text{EXT-SOLVE} \\
 \frac{\Gamma \longrightarrow \Delta}{\Gamma, \hat{\alpha} \longrightarrow \Delta, \hat{\alpha} = \tau} \\
 \\
 \text{EXT-ADD} \\
 \frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \hat{\alpha}} \\
 \\
 \text{EXT-ADDSOLVE} \\
 \frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \hat{\alpha} = \tau}
 \end{array}$$

Figure 6.4: Context extension

EXAMPLES. Below we give the derivation of $\hat{\alpha}, \hat{\beta} \vdash^{\mu} \hat{\alpha} \approx \hat{\beta} \rightarrow \text{Int}$ discussed in Section 6.1.1. Notice how the type $\hat{\beta} \rightarrow \text{Int}$ is promoted to $\hat{\alpha}_1 \rightarrow \text{Int}$.

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

The algorithm (correctly) rejects to unify $\hat{\alpha} \approx \hat{\alpha} \rightarrow \text{Int}$, where promotions fails due to “occurs-check”.

$$\begin{array}{c}
 \frac{\hat{\alpha}, \hat{\beta} \parallel_{\hat{\alpha}}^{\text{pr}} \hat{\alpha} \rightsquigarrow ??? \quad \dots}{\hat{\alpha}, \hat{\beta} \parallel_{\hat{\alpha}}^{\text{pr}} \hat{\alpha} \rightarrow \text{Int} \rightsquigarrow} \text{PR-ARROW} \\
 \frac{\hat{\alpha}, \hat{\beta} \parallel_{\hat{\alpha}}^{\text{pr}} \hat{\alpha} \rightarrow \text{Int} \rightsquigarrow}{\Gamma \vdash^{\mu} \hat{\alpha} \approx \hat{\alpha} \rightarrow \text{Int}} \text{U-EVAL-R}
 \end{array}$$

The case for $\hat{\alpha} \approx \hat{\alpha}$ is fine:

$$\frac{}{\Gamma \vdash^{\mu} \hat{\alpha} \approx \hat{\alpha} \dashv \Gamma} \text{U-REFL}$$

2598 6.2.3 SOUNDNESS AND COMPLETENESS

2599 Following Dunfield and Krishnaswami [2013], we first define the *context extension* judgment
 2600 $\Gamma \longrightarrow \Delta$, given in Figure 6.4, which reads that Γ is extended by Δ . Context extension

captures the notion of *information increase*. Essentially, if $\Gamma \longrightarrow \Delta$, then Δ contains more information than Γ ; for example, solutions to existing unification variables (rule [EXT-SOLVE](#)), more unification variables, either unsolved (rule [EXT-ADD](#)) or solved (rule [EXT-ADD-SOLVE](#)).

We now 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 $\hat{\alpha}$.

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

With soundness of promotion, we can prove that the unification algorithm and the algorithmic typing are also sound:

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

Theorem 6.3 (Soundness of Typing). *Given $\Delta \longrightarrow \Omega$, if $\Gamma \vdash e : \tau \dashv \Delta$, then $[\Omega]\Delta \vdash e : [\Omega]\tau$.*

Now we move to completeness. Note that in the following completeness statement we require $\hat{\alpha} \notin \text{FV}(\tau_1)$, or otherwise promotion would fail.

Theorem 6.4 (Completeness of Promotion). *Given $\Gamma \longrightarrow \Omega$, and $\Gamma \vdash^{\text{wf}} \hat{\alpha}$, and $\Gamma \vdash^{\text{wf}} \tau_1$, and $[\Gamma]\hat{\alpha} = \hat{\alpha}$, and $[\Gamma]\tau_1 = \tau_1$, if $\hat{\alpha} \notin \text{FV}(\tau_1)$, there exist τ_2, Δ and Ω' such that $\Gamma \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$ and $\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \tau_1 \rightsquigarrow \tau_2 \dashv \Delta$.*

Theorem 6.5 (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 $\Gamma \vdash^{\text{u}} \tau_1 \approx \tau_2 \dashv \Delta$.*

Theorem 6.6 (Completeness of Typing). *Given $\Gamma \longrightarrow \Omega$, and $\Gamma \vdash^{\text{wf}} \tau$, if $[\Omega]\Gamma \vdash e : \tau$, then there exists Δ, Ω', τ' such that $\Delta \longrightarrow \Omega', \Omega \longrightarrow \Omega'$, and $\Gamma \vdash e : \tau' \dashv \Delta$, and $[\Omega]\tau' = \tau$,*

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.

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

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

(Subtyping)

DK-S-TVAR

$$\frac{a \in \Psi}{\Psi \vdash^{DK} a <: a}$$

DK-S-INT

$$\frac{}{\Psi \vdash^{DK} \text{Int} <: \text{Int}}$$

DK-S-ARROW

$$\frac{\Psi \vdash^{DK} \sigma_3 <: \sigma_1 \quad \Psi \vdash^{DK} \sigma_2 <: \sigma_4}{\Psi \vdash^{DK} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4}$$

DK-S-FORALLL

$$\frac{\Psi \vdash^{DK} \tau \quad \Psi \vdash^{DK} \sigma[a \mapsto \tau] <: \sigma_2}{\Psi \vdash^{DK} \forall a. \sigma_1 <: \sigma_2}$$

DK-S-FORALLR

$$\frac{\Psi, a \vdash^{DK} \sigma_1 <: \sigma_2}{\Psi \vdash^{DK} \sigma_1 <: \forall a. \sigma_2}$$

Figure 6.5: Syntax and subtyping for higher-rank polymorphism

2631 6.3.1 DECLARATIVE SYSTEM

2632 The definition of types in DK (Figure 2.6 in Section 2.3.2) is repeated in Figure 6.5. Comparing to STLC, we have polymorphic types $\forall a. \sigma$ and type variables a . The context Ψ keeps track of the types of term bindings $x : \sigma$, as well as type variables a .

2635 Figure 6.6 also gives the subtyping rules for higher-rank polymorphism, which is used in Odersky and Läufer [1996] (Section 2.2.2) and Dunfield and Krishnaswami [2013]. We use the judgment $\Psi \vdash^{DK} \sigma_1 <: \sigma_2$ to denote the subtyping relation in DK. In rule **DK-S-ARROW**, functions are contravariant on arguments, and covariant on return types. Rule **DK-S-FORALLR** simply puts the type variable a into the context, and rule **DK-S-FORALLL** guesses a monomorphic instantiation of the type variable.

2641 6.3.2 ALGORITHMIC SYSTEM

2642 The syntax of the algorithmic system is given in Figure 6.6. We distinguish between polymorphic types σ and monotypes τ . Following Dunfield and Krishnaswami [2013], types are extended with the unification variable $\hat{\alpha}$. Note that while the system includes polymorphic types, the solutions to unification variables $\hat{\alpha} = \tau$ are restricted to monotypes; that is the key to ensure algorithmic typing is decidable and have principal types. $\blacktriangleright_{\hat{\alpha}}$ is a scope marker, subscripted by the unification variable $\hat{\alpha}$, and is used only in and explained with subtyping.

2648 The promotion mode \otimes is either covariant (+) or contravariant (-). We can use $-\otimes$ to flip the promotion mode. Specifically,

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

$\Gamma \vdash^{\text{wf}} \tau$

(Well-formedness)

$\frac{\text{WF-INT}}{\Gamma \vdash^{\text{wf}} \text{Int}}$	$\frac{\text{WF-ARROW} \quad \Gamma \vdash^{\text{wf}} \tau_1 \quad \Gamma \vdash^{\text{wf}} \tau_2}{\Gamma \vdash^{\text{wf}} \tau_1 \rightarrow \tau_2}$	$\frac{\text{WF-EVAR} \quad \hat{\alpha} \in \Gamma}{\Gamma \vdash^{\text{wf}} \hat{\alpha}}$	$\frac{\text{WF-TVAR} \quad a \in \Gamma}{\Gamma \vdash^{\text{wf}} a}$
--	---	---	---

$\Gamma \text{ ok}$

(Well-formedness of algorithmic contexts)

$\frac{\text{WFC-EMPTY}}{\bullet \text{ ok}}$	$\frac{\text{WFC-EVAR} \quad \Gamma \text{ ok} \quad \hat{\alpha} \notin \text{FV}(\Gamma)}{\Gamma, \hat{\alpha} \text{ ok}}$	$\frac{\text{WFC-EVAR SOLVED} \quad \Gamma \text{ ok} \quad \Gamma \vdash^{\text{wf}} \tau \quad \hat{\alpha} \notin \text{FV}(\Gamma)}{\Gamma, \hat{\alpha} = \tau \text{ ok}}$
$\frac{\text{WFC-TVAR} \quad \Gamma \text{ ok} \quad a \notin \text{FV}(\Gamma)}{\Gamma, a \text{ ok}}$		

Figure 6.6: Types, contexts, and well-formedness of the algorithmic system

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

2651 SUBTYPING. Figure 6.6 also includes the subtyping judgment $\Gamma \vdash^{\text{sub}} \sigma_1 <: \sigma_2 \dashv \Delta$, which
 2652 reads that, under the input context Γ , type σ_1 is a subtype of σ_2 , with the output context Δ .

2653 The rules except the last two are the same as the algorithmic subtyping rules in DK. Let us
 2654 first briefly go through the subtyping rules before we introduce our two new rules. Rules **S-**
 2655 **TVAR**, **S-INT**, and **S-EVAR** are self-explanatory. In rule **S-ARROW**, we first solve the contravari-
 2656 ant constraint $\sigma_3 <: \sigma_1$, which returns the output context Θ . We then solve the covariant
 2657 constraint $[\Theta]\sigma_2 <: [\Theta]\sigma_4$. Note that we apply the context Θ before we check the subtyping
 2658 derivation. Just like before, this maintains a general invariant: in the subtyping derivation
 2659 $\Gamma \vdash^{\text{sub}} \sigma_1 <: \sigma_2 \dashv \Delta$, the types σ_1 and σ_2 are fully applied under Γ . Rule **S-FORALLR** simply
 2660 puts a into the context and continue to check $\sigma_1 <: \sigma_2$, which returns the output context
 2661 Θ, a, Δ . As a goes out of the scope in the conclusion, the rule then discards a as well as the
 2662 trailing context Θ after a . Rule **S-FORALLL** puts a new unification variable as the instantia-
 2663 tion of the quantifier, and continues to check $\sigma_1[a \mapsto \hat{\alpha}] <: \sigma_2$. Just like before, as $\hat{\alpha}$ goes
 2664 out of the scope in the conclusion, we want to discard $\hat{\alpha}$ as well as its trailing context in the
 2665 conclusion. However, during the derivation, we may actually add more unification variables
 2666 right in front of $\hat{\alpha}$ that should also be removed. Therefore, we insert a marker $\blacktriangleright_{\hat{\alpha}}$ that marks
 2667 the start of the scope that will get discarded in the conclusion. In the conclusion, we then
 2668 discard $\blacktriangleright_{\hat{\alpha}}$ together with its trailing context.

2669 Note that in the design of Dunfield and Krishnaswami [2013], rule **S-FORALLR** and rule **S-**
 2670 **FORALLL** are overlapped, when both types are quantifiers. In practice, since rule **S-FORALLR**
 2671 is invertible, one can apply it eagerly. A possible fix is to restrict rule **S-FORALLL** to only apply
 2672 when the right hand side is not a quantifier [Peyton Jones et al. 2007].

2673 Now we turn to rule **S-INSTL** and rule **INSTR**, which are specific to this system. Recall that
 2674 in DK, subtyping between $\hat{\alpha}$ and σ relies on the instantiation rules, which are duplicated
 2675 for the case when $\hat{\alpha}$ is on the left and the case when $\hat{\alpha}$ is on the right. Here, instead of
 2676 instantiation, we directly use polymorphic promotion to promote the possibly polymorphic
 2677 type σ into a monotype τ . Specifically, rule **S-INSTL** uses polymorphic promotion under the
 2678 covariant mode (+) and rule **S-INSTR** uses polymorphic promotion under the contravariant
 2679 mode (-). If promotion succeeds, we can directly set $\hat{\alpha}$ to τ .

2680 POLYMORPHIC PROMOTION. The judgment $\Gamma \vdash_{\hat{\alpha}}^{\otimes} \sigma \rightsquigarrow \tau \dashv \Delta$ reads that under the input
 2681 context Γ , promoting σ under promotion mode \otimes yields type τ , so that τ is well-formed in
 2682 the prefix context of $\hat{\alpha}$.

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

(Subtyping)

S-TVAR $\frac{}{\Gamma[a] \vdash^{\text{sub}} a <: a \dashv \Gamma[a]}$	S-INT $\frac{}{\Gamma \vdash^{\text{sub}} \text{Int} <: \text{Int} \dashv \Gamma}$	S-EVAR $\frac{}{\Gamma[\hat{\alpha}] \vdash^{\text{sub}} \hat{\alpha} <: \hat{\alpha} \dashv \Gamma[\hat{\alpha}]}$
S-ARROW $\frac{\Gamma \vdash^{\text{sub}} \sigma_3 <: \sigma_1 \dashv \Theta \quad \Theta \vdash^{\text{sub}} [\Theta]\sigma_2 <: [\Theta]\sigma_4 \dashv \Delta}{\Gamma \vdash^{\text{sub}} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4 \dashv \Delta}$	S-FORALLR $\frac{\Gamma, a \vdash^{\text{sub}} \sigma_1 <: \sigma_2 \dashv \Delta, a, \Theta}{\Gamma \vdash^{\text{sub}} \sigma_1 <: \forall a. \sigma_2 \dashv \Delta}$	
S-FORALLL $\frac{\Gamma, \blacktriangleright_{\hat{\alpha}}, \hat{\alpha} \vdash^{\text{sub}} \sigma_1[a \mapsto \hat{\alpha}] <: \sigma_2 \dashv \Delta, \blacktriangleright_{\hat{\alpha}}, \Theta}{\Gamma \vdash^{\text{sub}} \forall a. \sigma_1 <: \sigma_2 \dashv \Delta}$		

S-INSTL $\frac{\Gamma[\hat{\alpha}] \vdash_{\hat{\alpha}}^+ \sigma \rightsquigarrow \tau \dashv \Delta[\hat{\alpha}]}{\Gamma[\hat{\alpha}] \vdash^{\text{sub}} \hat{\alpha} <: \sigma \dashv \Delta[\hat{\alpha} = \tau]}$	S-INSTR $\frac{\Gamma[\hat{\alpha}] \vdash_{\hat{\alpha}}^- \sigma \rightsquigarrow \tau \dashv \Delta[\hat{\alpha}]}{\Gamma[\hat{\alpha}] \vdash^{\text{sub}} \sigma <: \hat{\alpha} \dashv \Delta[\hat{\alpha} = \tau]}$
--	--

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

(Polymorphic Promotion)

P-PR-FORALLL $\frac{\Gamma[\hat{\beta}, \hat{\alpha}] \vdash_{\hat{\alpha}}^- \sigma[a \mapsto \hat{\beta}] \rightsquigarrow \tau \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash_{\hat{\alpha}}^- \forall a. \sigma \rightsquigarrow \tau \dashv \Delta}$	P-PR-FORALLR $\frac{\Gamma, a \vdash_{\hat{\alpha}}^+ \sigma \rightsquigarrow \tau \dashv \Delta, a}{\Gamma \vdash_{\hat{\alpha}}^+ \forall a. \sigma \rightsquigarrow \tau \dashv \Delta}$
P-PR-ARROW $\frac{\Gamma \vdash_{\hat{\alpha}}^{\otimes} \sigma_1 \rightsquigarrow \tau_1 \dashv \Theta \quad \Theta \vdash_{\hat{\alpha}}^{\otimes} [\Theta]\sigma_2 \rightsquigarrow \tau_2 \dashv \Delta}{\Gamma \vdash_{\hat{\alpha}}^{\otimes} \sigma_1 \rightarrow \sigma_2 \rightsquigarrow \tau_1 \rightarrow \tau_2 \dashv \Delta}$	P-PR-INT $\frac{}{\Gamma \vdash_{\hat{\alpha}}^{\otimes} \text{Int} \rightsquigarrow \text{Int} \dashv \Gamma}$
P-PR-EVARL $\frac{}{\Gamma[\hat{\beta}][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\otimes} \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \Gamma[\hat{\beta}][\hat{\alpha}]}$	P-PR-EVARR $\frac{}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash_{\hat{\alpha}}^{\otimes} \hat{\beta} \rightsquigarrow \hat{\beta}_1 \dashv \Gamma[\hat{\beta}_1, \hat{\alpha}][\hat{\beta} = \hat{\beta}_1]}$
P-PR-TVAR $\frac{}{\Gamma[a][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\otimes} a \rightsquigarrow a \dashv \Gamma[a][\hat{\alpha}]}$	

Figure 6.7: Unification, subtyping and polymorphic promotion of the algorithmic system)

The only difference between these two promotion modes is how to promote polymorphic types. Under the contravariant mode (rule **P-PR-FORALL**), a monotype would make the final type more polymorphic. Therefore, we replace the universal binder a with a fresh unification 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.

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

2683 Rule **P-PR-ARROW** flips the mode for codomains, and uses the same mode for domains.
 2684 When the type to be promoted is a monotype, rule **P-PR-MONO** uses the promotion judgment
 2685 (\Vdash^{pr}) directly. Note that for a monotype the mode does not matter, so the rest of the rules
 2686 apply in both modes. Rules **P-PR-INT**, **P-PR-EVARL**, and **P-PR-EVARR** are the same as before.

2687 Importantly, we now have a new rule **PR-TVAR** for type variables a . Note again that pro-
 2688 motion is a partial operation, as it requires a to be the left of $\hat{\alpha}$, since the order of variable
 2689 matters. This effectively solves the scoping issue between type variables and unification vari-
 2690 ables. For example, if we want to derive $\hat{\alpha}, a \vdash^{\text{sub}} \hat{\alpha} <: a$, we should fail – a is not in the
 2691 scope of $\hat{\alpha}$ and we cannot set $\hat{\alpha} = a$! This is exactly what rule **PR-TVAR** does – promoting a
 2692 fails as this is no way we can promote it to be well-typed in the prefix context of $\hat{\alpha}$.

2693 **EXAMPLES.** Below we give some examples.

The first example is the subtyping derivation of $\hat{\alpha} \vdash^{\text{sub}} \forall a. a \rightarrow a <: \hat{\alpha}$. According to rule **S-INST**, we start by promoting $\forall a. a \rightarrow a$ under the contravariant mode $(-)$. We then call rule **P-PR-FORALL**, which adds a fresh unification variable $\hat{\beta}$ in front of $\hat{\alpha}$ and returns the promoted type $\hat{\beta} \rightarrow \hat{\beta}$. Note that $\hat{\beta}$ promotes to itself under both modes as it appears to

the left of $\hat{\alpha}$ (rule **P-PR-EVARL**). We now return back to the subtyping derivation, and set $\hat{\alpha} = \hat{\beta} \rightarrow \hat{\beta}$.

$$\begin{array}{c}
\text{P-PR-EVARL} \qquad \qquad \text{P-PR-EVARL} \\
\frac{\hat{\beta}, \hat{\alpha} \vdash_{\hat{\alpha}}^+ \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \hat{\beta}, \hat{\alpha}}{\hat{\beta}, \hat{\alpha} \vdash_{\hat{\alpha}} \hat{\beta} \rightarrow \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \hat{\beta}, \hat{\alpha}} \text{P-PR-ARROW} \qquad \frac{\hat{\beta}, \hat{\alpha} \vdash_{\hat{\alpha}}^- \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \hat{\beta}, \hat{\alpha}}{\hat{\beta}, \hat{\alpha} \vdash_{\hat{\alpha}} \hat{\beta} \rightarrow \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \hat{\beta}, \hat{\alpha}} \text{P-PR-ARROW} \\
\frac{\hat{\beta}, \hat{\alpha} \vdash_{\hat{\alpha}} \hat{\beta} \rightarrow \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \hat{\beta}, \hat{\alpha}}{\hat{\alpha} \vdash_{\hat{\alpha}} \forall a. a \rightarrow a \rightsquigarrow \hat{\beta} \rightarrow \hat{\beta} \dashv \hat{\beta}, \hat{\alpha}} \text{P-PR-FORALLL} \\
\frac{\hat{\alpha} \vdash_{\hat{\alpha}} \forall a. a \rightarrow a \rightsquigarrow \hat{\beta} \rightarrow \hat{\beta} \dashv \hat{\beta}, \hat{\alpha}}{\hat{\alpha} \vdash^{\text{sub}} \forall a. a \rightarrow a <: \hat{\alpha} \dashv \hat{\beta}, \hat{\alpha} = \hat{\beta} \rightarrow \hat{\beta}} \text{S-INSTL}
\end{array}$$

The second example is the subtyping derivation of $\hat{\alpha} \vdash^{\text{sub}} \hat{\alpha} <: \forall a. a \rightarrow a$. According to rule **S-INSTL**, we start by promoting $\forall a. a \rightarrow a$ under the covariant mode (+). We then call rule **P-PR-FORALLR**. In this case we directly add the type variable a into the type context, and continue to promote $a \rightarrow a$. After one step of rule **P-PR-ARROW**, we got stuck – note how rule **P-PR-TVAR** fails to apply in this case as a appears to the right of $\hat{\alpha}$. So promotion (correctly) rejects this example.

$$\begin{array}{c}
\frac{\hat{\alpha}, a \vdash_{\hat{\alpha}}^+ a \rightsquigarrow ???}{\hat{\alpha}, a \vdash_{\hat{\alpha}}^+ a \rightarrow a \rightsquigarrow ???} \text{P-PR-ARROW} \\
\frac{\hat{\alpha}, a \vdash_{\hat{\alpha}}^+ a \rightarrow a \rightsquigarrow ???}{\hat{\alpha} \vdash_{\hat{\alpha}}^+ \forall a. a \rightarrow a \rightsquigarrow ???} \text{P-PR-FORALLR} \\
\frac{\hat{\alpha} \vdash_{\hat{\alpha}}^+ \forall a. a \rightarrow a \rightsquigarrow ???}{\hat{\alpha} \vdash^{\text{sub}} \hat{\alpha} <: \forall a. a \rightarrow a} \text{S-INSTL}
\end{array}$$

2694 6.3.3 SOUNDNESS AND COMPLETENESS

2695 The statement of soundness of promotion remains the same as before.

2696 **Theorem 6.7** (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*
 2697 *$\Delta_1 \vdash^{\text{wf}} \tau_2$, and $[\Delta]\tau_1 = [\Delta]\tau_2$.*

2698 Based on soundness of promotion, we prove that after polymorphic promotion, the pro-
 2699 moted type is also well-formed under the prefix context of $\hat{\alpha}$. Moreover, polymorphic pro-
 2700 motion builds a subtyping relation according to the promotion mode: under the contravari-
 2701 ant mode (-), the original type is a subtype of the promoted type; under the covariant mode
 2702 (+), the promoted type is a subtype of the original type.

2703 **Theorem 6.8** (Soundness of Polymorphic Promotion). *If $\Gamma \vdash_{\hat{\alpha}}^{\otimes} \sigma \rightsquigarrow \tau \dashv \Delta$, and $\Delta =$*
 2704 *$\Delta_1, \hat{\alpha}, \Delta_2$, then $\Delta_1 \vdash^{\text{wf}} \tau_2$. Moreover, given $\Delta \rightarrow \Omega$,*

2705 *• if $\otimes = \neg$, then $[\Omega]\Gamma \vdash^{\text{DK}} [\Omega]\sigma <: [\Omega]\tau$; and*

2706 • if $\otimes = \vdash$, then $[\Omega]\Gamma \vdash^{DK} [\Omega]\tau <: [\Omega]\sigma$.

2707 With soundness of polymorphic promotion, next we show that the new subtyping judg-
2708 ment using polymorphic promotion instead of instantiation remains sound.

2709 **Theorem 6.9** (Soundness of Subtyping). *If $\Gamma \vdash^{\text{sub}} \sigma_1 <: \sigma_2 \dashv \Delta$, and $\Delta \longrightarrow \Omega$, then*
2710 *$[\Omega]\Gamma \vdash^{DK} [\Omega]\sigma_1 <: [\Omega]\sigma_2$.*

2711 Now we turn to completeness. The completeness of promotion is the same as before.

2712 **Theorem 6.10** (Completeness of Promotion). *Given $\Gamma \longrightarrow \Omega$, and $\Gamma \vdash^{\text{wf}} \hat{\alpha}$, and $\Gamma \vdash^{\text{wf}} \tau$,*
2713 *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*
2714 *$\Omega \longrightarrow \Omega'$ and $\Gamma \vdash_{\hat{\alpha}}^{\text{pr}} \tau \rightsquigarrow \tau_2 \dashv \Delta$.*

2715 Completeness of polymorphic promotion has two parts. If the unification variable ap-
2716 pears on the left, then we promote the type under the covariant mode (+), or otherwise the
2717 contravariant mode (-). Moreover, it also requires $\hat{\alpha} \notin \text{FV}(\sigma)$.

2718 **Theorem 6.11** (Completeness of Polymorphic Promotion). *Given $\Gamma \longrightarrow \Omega$, and $\Gamma \vdash^{\text{wf}} \hat{\alpha}$,*
2719 *and $\Gamma \vdash^{\text{wf}} \sigma$, and $[\Gamma]\hat{\alpha} = \hat{\alpha}$, and $[\Gamma]\tau = \sigma$, and $\hat{\alpha} \notin \text{FV}(\sigma)$,*

2720 • if $[\Omega]\Gamma \vdash^{DK} [\Omega]\hat{\alpha} <: [\Omega]\sigma$, then there exist τ, Δ and Ω' such that $\Gamma \vdash_{\hat{\alpha}}^+ \sigma \rightsquigarrow \tau \dashv \Delta$;
2721 and

2722 • if $[\Omega]\Gamma \vdash^{DK} [\Omega]\sigma <: [\Omega]\hat{\alpha}$, then there exist τ, Δ and Ω' such that $\Gamma \vdash_{\hat{\alpha}}^- \sigma \rightsquigarrow \tau \dashv \Delta$.

2723 Finally, we prove that our subtyping is complete. With this, we have proved our claim
2724 that the original instantiation relation in DK can be replaced by the polymorphic promo-
2725 tion process, as the subtyping algorithm using polymorphic promotion remains sound and
2726 complete.

2727 **Theorem 6.12** (Completeness of Subtyping). *Given $\Gamma \longrightarrow \Omega$, and $\Gamma \vdash^{\text{wf}} \sigma_1$, and $\Gamma \vdash^{\text{wf}} \sigma_2$,*
2728 *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*
2729 *$\Gamma \vdash^{\text{sub}} [\Gamma]\sigma_1 <: [\Gamma]\sigma_2 \dashv \Delta$.*

2730 6.4 DISCUSSION

2731 This section discusses two extensions of promotion, and we can see that promotion scales
2732 naturally to other type features. The first extension explores dependent types, while the sec-
2733 ond extension considers gradual types.

2734 6.4.1 PROMOTING DEPENDENT TYPES

2735 In Section 6.1.1 we mentioned the drawback of decomposing type constructs that it cannot
 2736 be easily applied to more advanced types like dependent types. In this section, we discuss
 2737 how we can apply promotion to dependent types.

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

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

2738 Here we first promote τ_1 , returning τ_3 . Then we add a into the context to promote τ_2 . Finally,
 2739 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 $\Pi a : \hat{\beta}. a$.

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

2740 We expect that the promotion would return $\Pi a : \hat{\beta}. a$. However, after we add a into the
 2741 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 unification 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.,

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

$$\frac{\text{s-INSTRR} \quad \Gamma[\hat{\alpha}] \vdash_{\hat{\alpha}}^- \sigma \rightsquigarrow \tau \dashv \Delta_1, \hat{\alpha}, \Delta_2 \quad \Delta_1 \vdash^{\text{wf}} \tau}{\Gamma[\hat{\alpha}] \vdash^{\text{sub}} \sigma <: \hat{\alpha} \dashv \Delta_1, \hat{\alpha} = \tau, \Delta_2}$$

2742 Xie and Oliveira [2017] include a more detailed discussion and formalization of applying
2743 promotion to a dependently typed lambda calculus.

2744 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 unification variable:

P-PR-UNKNOWN

$$\frac{}{\Gamma[\hat{\alpha}] \vdash_{\alpha}^{\otimes} ? \rightsquigarrow \hat{\beta} \dashv \Gamma[\hat{\beta}, \hat{\alpha}]}$$

2745 For example, we have $\hat{\alpha} \vdash_{\alpha}^{\text{pr}} \text{Int} \rightarrow ? \rightsquigarrow \text{Int} \rightarrow \hat{\beta} \dashv \hat{\beta}, \hat{\alpha}$.

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

P-PR-UNKNOWN_G

$$\frac{}{\Gamma[\hat{\alpha}] \vdash_{\alpha}^{\otimes} ? \rightsquigarrow \hat{\beta}_G \dashv \Gamma[\hat{\beta}_G, \hat{\alpha}]}$$

2746 With these rules it would be possible to apply polymorphic promotion to GPC. Note this
2747 discussion is a sketch and we have not fully worked out the full algorithm yet.

2748 6.5 PROMOTION IN ACTION

2749 The promotion operation is actually present in the *Glasgow Haskell Compiler* (GHC), the
2750 state-of-the-art compiler for the Haskell programming language, though its form is different
2751 than what we have presented.

2752 In particular, instead of keeping all the unification variables ordered in the context, which
2753 can be quite inefficient, GHC only keeps them in order with respect to type variables in the
2754 context, by assigning a *level number* to each unification variable and each type variable, and
2755 incrementing the level number whenever we enter a new skolemisation scope, i.e., when a
2756 new type variable is brought into scope. When unifying $\hat{\alpha} \approx \tau$ where $\hat{\alpha}$ is level N , to ensure
2757 well-scopedness, GHC checks that all free variables in τ are from level N or lower, and update
2758 levels when needed: if there is a unification variable $\hat{\beta}$ in τ_2 , where $\hat{\beta}$ has level $M > N$, then
2759 it allocates $\hat{\beta}_1$ at level N and set $\hat{\beta} = \hat{\beta}_1$, which essentially implements the idea of promotion.

2760 In the following discussion, we refer to the promotion process described in this chapter as
 2761 *order promotion*, and the process in GHC as *level promotion*.

As an example, consider during type inference, we need to prove the following subtyping constraint:

$$\bullet \vdash^{\text{sub}} \forall a. \text{Int} \rightarrow a \rightarrow \forall b. b <: \text{Int} \rightarrow \forall c. (\forall d. d \rightarrow d) \rightarrow c$$

2762 Below, we work through the example using order promotion on the left, and then work
 2763 through the example again using level promotion on the right¹. The reader is encouraged to
 2764 read the left part first before moving to the right.

2765 With promotion, we start with

$$\bullet \vdash^{\text{sub}} \forall a. \text{Int} \rightarrow a \rightarrow \forall b. b <: \text{Int} \rightarrow \forall c. (\forall d. d \rightarrow d) \rightarrow c$$

Now let us go through the example again with levels. We keep track of the current global level, starting from 0:

$$(0) \bullet \vdash^{\text{sub}} \forall a. \text{Int} \rightarrow a \rightarrow \forall b. b <: \text{Int} \rightarrow \forall c. (\forall d. d \rightarrow d) \rightarrow c$$

2766 First, we apply rule **S-FORALLL** and instantiate a with $\hat{\alpha}$:

$$\hat{\alpha} \vdash^{\text{sub}} \text{Int} \rightarrow \hat{\alpha} \rightarrow \forall b. b <: \text{Int} \rightarrow \forall c. (\forall d. d \rightarrow d) \rightarrow c$$

Now as before, we first instantiating a with fresh unification variables $\hat{\alpha}$, but unlike before, we record that $\hat{\alpha}$ is created at level 0. In this case, we only created new unification variables, so the global level number stays the same:

$$(0) \hat{\alpha}(0) \vdash^{\text{sub}} \text{Int} \rightarrow \hat{\alpha} \rightarrow \forall b. b <: \text{Int} \rightarrow \forall c. (\forall d. d \rightarrow d) \rightarrow c$$

2767 We then apply rule **S-ARROW** and split the constraint into two constraints. As the contravariant constraint on arguments is simply $\text{Int} <: \text{Int}$, we proceed by solving the re-simply $\text{Int} <: \text{Int}$, we continue with the turn type: covariant constraint on the return type:

$$\hat{\alpha} \vdash^{\text{sub}} \hat{\alpha} \rightarrow \forall b. b <: \forall c. (\forall d. d \rightarrow d) \rightarrow c$$

$$(0) \hat{\alpha}(0) \vdash^{\text{sub}} \hat{\alpha} \rightarrow \forall b. b <: \forall c. (\forall d. d \rightarrow d) \rightarrow c$$

¹The process we are going to describe is not exactly how GHC works for this example, but it is sufficient to illustrate the key idea of levels.

2768 We apply rule **S-FORALLR** and skolemise c : We now skolemise c on the right, and also remember that c is skolemised at level 0. Moreover, since we created a new type variable, we increase the global level number by 1:

$$(1) \quad \hat{\alpha}(0), c(0) \vdash^{\text{sub}} \hat{\alpha} \rightarrow \forall b. b <: (\forall d. d \rightarrow d) \rightarrow c$$

2769 We apply rule **S-ARROW** again, and we first We split the function types, and the first solve the constraint on the argument type: constraint we get is:

$$\hat{\alpha}, c \vdash^{\text{sub}} \forall d. d \rightarrow d <: \hat{\alpha} \quad (1) \quad \hat{\alpha}(0), c(0) \vdash^{\text{sub}} \forall d. d \rightarrow d <: \hat{\alpha}$$

2770 where d is instantiated with $\hat{\beta}$: We again instantiate d with fresh unification variables $\hat{\beta}$. Since the global level number is 1, $\hat{\beta}$ is at level 1:

$$\hat{\alpha}, c, \hat{\beta} \vdash^{\text{sub}} \hat{\beta} \rightarrow \hat{\beta} <: \hat{\alpha} \quad (1) \quad \hat{\alpha}(0), c(0), \hat{\beta}(1) \vdash^{\text{sub}} \hat{\beta} \rightarrow \hat{\beta} <: \hat{\alpha}$$

2771 At this point, we want to solve $\hat{\alpha}$ with $\hat{\beta} \rightarrow \hat{\beta}$. Now we want to solve $\hat{\alpha}$ with $\hat{\beta} \rightarrow \hat{\beta}$. We $\hat{\beta}$. However, $\hat{\beta}$ is not in the scope of $\hat{\alpha}$. So check their level numbers. Because $\hat{\beta}$ has we promote $\hat{\beta}$ by creating a fresh $\hat{\alpha}_2$ to the a level greater than 0, we promote $\hat{\beta}$ by left of $\hat{\alpha}$ and setting $\hat{\beta} = \hat{\alpha}_2$. We can then creating a fresh $\hat{\alpha}_2$ at level 0, and setting solve $\hat{\alpha} = \hat{\alpha}_2 \rightarrow \hat{\alpha}_2$. $\hat{\beta} = \hat{\alpha}_2$. But unlike before, since the context is not ordered, we can freely put $\hat{\alpha}_2$ at the end of the context:

$$\hat{\alpha}, c, \hat{\beta} \vdash^{\text{sub}} \hat{\beta} \rightarrow \hat{\beta} <: \hat{\alpha} \dashv \hat{\alpha}_2, \hat{\alpha} = \hat{\alpha}_2 \rightarrow \quad (1) \quad \hat{\alpha}(0), c(0), \hat{\beta}(1) \vdash^{\text{sub}} \hat{\beta} \rightarrow \hat{\beta} <: \hat{\alpha} \dashv \hat{\alpha}_2$$

$$\hat{\alpha}_2, c, \hat{\beta} = \hat{\alpha}_2 \quad \hat{\alpha}(0) = \hat{\alpha}_2 \rightarrow \hat{\alpha}_2, c(0), \hat{\beta}(1) = \hat{\alpha}_2, \hat{\alpha}_2(0)$$

2772 Going back to the covariant constraint $\forall d. d \rightarrow d <: c$ on the left, which can be simply solved by instantiating d with $\hat{\beta}_2$ and solving $\hat{\beta}_2 = c$: Going back to the covariant constraint $\forall d. d \rightarrow d <: c$ on the left, which can be simply solved by instantiating d with $\hat{\beta}_2(1)$ and solving $\hat{\beta}_2 = c$, as c has a smaller level:

$$\hat{\alpha}_2, \hat{\alpha} = \hat{\alpha}_2 \rightarrow \hat{\alpha}_2, c, \hat{\beta} = \hat{\alpha}_2 \vdash^{\text{sub}} \forall b. b <: \quad (1) \quad \hat{\alpha}(0) = \hat{\alpha}_2 \rightarrow \hat{\alpha}_2, c(0), \hat{\beta}(1) = c \dashv \hat{\alpha}_2, \hat{\alpha}_2(0) \vdash^{\text{sub}} \forall b. b <: c \dashv \hat{\alpha}(0) = \hat{\alpha}_2 \rightarrow$$

$$\hat{\alpha}_2, \hat{\alpha} = \hat{\alpha}_2 \rightarrow \hat{\alpha}_2, c, \hat{\beta} = \hat{\alpha}_2, \hat{\beta}_2 = c \quad \hat{\alpha}_2, c(0), \hat{\beta}(1) = \hat{\alpha}_2, \hat{\alpha}_2(0), \hat{\beta}_2(1) = c$$

2773 A key observation is that just like order promotion, level promotion captures precisely
 2774 that *the relative order between unification variables does not matter*, and well-scopedness are
 2775 ensured using levels. The advantage with level promotion is that the whole type context does
 2776 not need to be ordered, which can be implemented more efficiently in practice. While not
 2777 shown in this example, just like order promotion, level promotion can fail for type variables.
 2778 For example, order promotion fails to promote c in $\widehat{\alpha}, c \vdash^{\text{sub}} \widehat{\alpha} <: c$; similarly, level promo-
 2779 tion fails to promote c in $c(1), \widehat{\alpha}(0) \vdash^{\text{sub}} \widehat{\alpha} <: c$, where c has a level greater than $\widehat{\alpha}$.

2780 6.6 LEVELS AND GENERALIZATION

2781 Levels are first introduced in Rémy [1992] to implement generalization more efficiently. Be-
 2782 sides GHC, they have also been implemented in OCaml; Kiselyov [2013] provides a detailed
 2783 description of levels in the OCaml type checker. In this section, we first describe how lev-
 2784 els are used to implement efficient generalization, and then discuss how we can implement
 2785 generalization in our setting of ordered type contexts. Efficiency is not a major focus in this
 2786 dissertation, but we hope this discussion with levels and generalization can help provide a
 2787 deeper understanding of ordered type contexts and promotion.

Let us first recall the declarative generalization rule used in the let expression in Hindley-
 Milner:

$$\frac{\text{HM-LET} \quad \Psi \vdash^{HM} e_1 : \sigma \quad \Psi, x : \sigma \vdash^{HM} e_2 : \tau}{\Psi \vdash^{HM} \text{let } x = e_1 \text{ in } e_2 : \tau}$$

How can we implement such a rule in an algorithmic system? In type inference literature
 where efficiency is not the main concern, it is usually implemented as:

$$\frac{\Gamma \vdash^{HM} e_1 : \tau_1 \quad \overline{\alpha_i}^i = \text{FV}(\tau_1) - \text{FV}(\Gamma) \quad \Gamma, x : \forall \overline{a_i}^i. \tau_1[\overline{\alpha_i} \mapsto \overline{a_i}^i] \vdash^{HM} e_2 : \tau_2}{\Gamma \vdash^{HM} \text{let } x = e_1 \text{ in } e_2 : \tau_2}$$

2788 where Γ does not track unification variable, and we calculate the free variables in τ_1 that are
 2789 not referred in Γ , and generalize type τ_1 over $\overline{a_i}^i$.

2790 However, this implementation requires a linear scan of the type context, which can be very
 2791 large for big programs. For nested let expressions, type checking will be quadratic. In fact, it
 2792 was one of the main reasons for the slow speed of Caml compilation [Rémy 1992].

Rémy resolves the problem by using levels to speed up the process of generalization. The key insight is to use levels to keep track of the scope of let expressions (in the previous section, we showed how GHC uses levels to keep track of the scope of skolemisation). For example,

```

2793  -- starting from level 0
2794  let x1 =      -- level 1
2795      let x2 = e2 -- level 2
2796      in e3      -- level 1
2797  in e4          -- level 0
    
```

When we enter a let expression, we increase the level, and when we get out of the scope of a let expression, we decrease the level. Every unification variable is associated with a level at which it is created. Now generalization can be implemented in an efficient way: instead of looking through the context, we directly look at the type to be generalized, and compare the level of each unification variable with the current level. If a unification variable has a level greater than the current level, it is created in a deeper level, but since that level is gone, it is now a free variable and thus can be generalized over. Such an implementation can be further optimized by updating levels lazily; we refer the interested reader to Kiselyov [2013]; Rémy [1992] for more detailed explanations.

In our setting with ordered type contexts that tracks unification variables, we can also implement generalization in a way more efficient than linearly scanning the type context: as we ensure that all unification variables are put in order, we know exactly what unification variables can be generalized over. With ordered type context, the let generalization rule looks like:

$$\frac{\Gamma, \blacktriangleright_{\hat{\alpha}} \vdash^{HM} e_1 : \tau_1 \dashv \Delta, \blacktriangleright_{\hat{\alpha}}, \Theta \quad \widehat{\alpha}_i^i = \mathbf{unsolved}(\Theta) \quad \Gamma, x : \forall \bar{a}_i^i. ([\Theta]\tau_1)[\widehat{\alpha}_i \mapsto a_i^i] \vdash^{HM} e_2 : \tau_2 \dashv \Delta_2, x : \forall \bar{a}_i^i. \tau_1[\bar{X}_i \mapsto a_i^i], \Theta_2}{\Gamma \vdash^{HM} \mathbf{let} x = e_1 \mathbf{in} e_2 : \tau_2 \dashv \Delta_2}$$

Note that to generalize τ_1 , we do not need to traverse the whole context $\Delta, \blacktriangleright_{\hat{\alpha}}, \Theta$. That is because promotion has already guaranteed that any unification variables that Γ depends on will already have been promoted to the left of the $\blacktriangleright_{\hat{\alpha}}$, and so all unsolved unification variables in Θ are exactly those unification variables that can be generalized over. Note that we apply Θ to τ_1 so that all solved unification variables are substituted in τ_1 before we throw them away. We can also be more precise by generalizing only $\mathbf{unsolved}(\Theta) \cap \text{FV}([\Theta]\tau_1)$.

2817 In recent years, languages like Haskell have seen a dramatic surge of new features that signif-
 2818 ically extends the expressive power of their type systems. With these features, the challenge
 2819 of *kind inference* for datatype declarations has presented itself and become a worthy research
 2820 problem on its own.

2821 In this chapter, we apply promotion to kind inference for datatypes. Inspired by previous
 2822 research on type-inference, we offer declarative specifications for what datatype declarations
 2823 should be accepted, both for Haskell98 and for a more advanced system we call PolyKinds,
 2824 based on the extensions in modern Haskell, including a limited form of dependent types.
 2825 We believe these formulations to be novel and without precedent, even for Haskell98. These
 2826 specifications are complemented with implementable algorithmic versions. We study *sound-*
 2827 *ness*, *completeness* and the existence of *principal kinds* in these systems, proving the proper-
 2828 ties where they hold. This work can serve as a guide both to language designers who wish
 2829 to formalize their datatype declarations and also to implementors keen to have principled
 2830 inference of principal types.

2831 7.1 INTRODUCTION AND MOTIVATION

2832 The global type-inference algorithms employed in modern functional languages such as Haskell,
 2833 ML, and OCaml are derived from the Hindley-Milner type system (HM) [Damas and Milner
 2834 1982; Hindley 1969], with multiple extensions. Common extensions of HM include *higher-*
 2835 *ranked polymorphism* [Odersky and Läufer 1996; Peyton Jones et al. 2007] and *type-inference*
 2836 *for GADTs* [Peyton Jones et al. 2006], which have both been formally studied thoroughly.

2837 Most research work for extensions of HM so far (including OL, DK, AP and GPC) has
 2838 focused on forms of polymorphism, where type variables all have the same kind. In these
 2839 systems, the type variables introduced by universal quantifiers and/or type declarations all
 2840 stand for proper types (i.e., they have kind \star). In such a simplified setting, datatype declara-
 2841 tions such as

2842 `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 $K :: (\star \rightarrow \star) \rightarrow \star$.

2901 **DEFAULTING.** Definitions like k (in STLC) or K (in Haskell98) do not have a principal
 2902 type/kind, which raises the immediate question of what type/kind to infer. Haskell98 solves
 2903 this problem by using a *defaulting* strategy: *if the kind of a type variable cannot be inferred,*
 2904 *then it is defaulted to \star .* Therefore the kind of K in Haskell98 is $\star \rightarrow \star$. From the perspec-
 2905 tive of type inference, such defaulting strategy may seem somewhat ad-hoc, but due to the
 2906 role that \star plays at the type level it seems a defensible design for kind inference. Defaulting
 2907 brings complications in writing a declarative specification. We discuss this point further in
 2908 Section 7.4.3.

2909 7.2.2 KIND INFERENCE IN MODERN GHC HASKELL

2910 The type and kind languages for modern GHC are *unified* (i.e., types and kinds are indistin-
 2911 guishable), *dependently typed*, and the kind system includes the $\star :: \star$ axiom Cardelli [1986];
 2912 Weirich et al. [2013]. We informally use the word *type* or *kind* where we find it appropriate.
 2913 Unlike Haskell98’s datatypes, whose inference problem is quite closely related to the well-
 2914 studied inference problem for STLC, type inference for various features in modern Haskell
 2915 is not well-studied. While we are motivated concretely by Haskell, many of the challenges we
 2916 face would be present in any dependently typed language seeking principled type inference.
 2917 We use the term PolyKinds to refer to the fragment of modern Haskell that we model.² We
 2918 enumerate the key features of this fragment below.

2919 **KIND POLYMORPHISM AND DEPENDENT TYPES** Global type inference, in the style of Damas
 2920 and Milner [1982], allows polymorphic kinds to be assigned to datatype definitions. For
 2921 instance, reconsider

2922 `data K a = K Int`

2923 In PolyKinds, K can be given the kind $K :: \forall\{k\}. k \rightarrow \star$. This example shows one of the
 2924 interesting new features of PolyKinds over Haskell98: *kind polymorphism* [Yorgey et al. 2012].
 2925 The polymorphic kind is obtained via *generalization*, which is a standard feature in Damas-
 2926 Milner algorithms. Polymorphic types are helpful for recovering principal types, since they
 2927 generalize many otherwise incomparable monomorphic types.

2928 System-F-based languages do not have dependent types. In contrast, PolyKinds supports
 2929 dependent kinds such as

2930 `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. Appendix C.2 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 $\forall(k :: \star). k \rightarrow \star$.

```

2965   data P a = MkP (Q a)
2966   data Q a = MkQ (P a)

```

2967 The recursion is simple here: all recursive occurrences are at the same type. In existing
 2968 type-inference algorithms, such recursive definitions are well understood and do not bring
 2969 considerable complexity to type inference. However, we must also consider *polymorphic re-*
 2970 *cursion* as in *Poly*:

```

2971   data Poly :: ∀k. k → ★
2972   data Poly k = C1 (Poly Int) | C2 (Poly Maybe)

```

2973 This example includes a *kind signature*³, meaning that we must *check* the kind of the datatype,
 2974 not *infer* it. In the definition of *Poly*, the type *Poly Int* requires an instantiation $k \mapsto \star$,
 2975 while the type *Poly Maybe* requires an instantiation of $k \mapsto (\star \rightarrow \star)$. These differing
 2976 instantiations mean that the declaration employs polymorphic recursion.

2977 PolyKinds deals with such cases of polymorphic recursion, which also appear at the term
 2978 level—for example, when writing recursive functions over GADTs or nested datatypes [Bird
 2979 and Meertens 1998]. Polymorphic recursion is known to render type-inference undecid-
 2980 able [Henglein 1993]. Furthermore, most existing formalizations of type inference avoid
 2981 the question entirely, either by not modeling recursion at all or not allowing polymorphic
 2982 recursion. Our PolyKinds system follows Jones [1999] to support polymorphic recursion
 2983 where there are kind signatures that cut the loops. Polymorphic recursion is allowed only on
 2984 datatypes with a kind signature; other datatypes are treated as monomorphic during infer-
 2985 ence.

2986 **VISIBLE KIND APPLICATION** PolyKinds lifts *visible type application* (VTA) [Eisenberg et al.
 2987 2016], whereby we can explicitly instantiate a function call, as in *id @Bool True*, to kinds,
 2988 giving us *visible kind application* (VKA). Following the design of VTA, we distinguish *speci-*
 2989 *fied variables* (*a*) from *inferred variables* ($\{a\}$). As described by Eisenberg et al. [2016, Sec-
 2990 tion 3.1], only specified variables can be instantiated via VKA. Instantiation of variables is
 2991 inferred when no explicit kind application is given. To illustrate, consider

```

2992   data T :: ∀a b. a b → ★

```

2993 Here, *a* and *b* are specified variables. Because their order is given, explicit instantiation of
 2994 *a* must happen before *b*. For example, *T @Maybe* instantiates *a* to *Maybe*. On the other

³Recently GHC has implemented *standalone kind signatures* [Eisenberg 2018], where instead of *data* it uses the keyword *type*.

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

OPEN KIND SIGNATURES AND GENERALIZATION ORDER Echoing the design of Haskell, PolyKinds supports *open kind signatures*. We say a signature is *closed* if it contains no free variables, e.g.,

`data T :: $\forall a. a \rightarrow \star$`

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

`data Q :: $\forall(a :: (f \ b)) (c :: k). f \ c \rightarrow \star$`

Free variables (in this case, f , b , k) will be generalized over; that is, Haskell implicitly adds universal quantifiers for each variable mentioned in the open kind signature that is not otherwise bound⁴ [Marlow et al. 2010]. We have a decision to make: in which order do we generalize the free variables? This question is non-trivial, as there can be dependency between the variables. We infer $k :: \star$, $f :: k \rightarrow \star$, $b :: k$. Even though f and b appear before k , their kinds end up depending on k and we must quantify k before f and b . Inferring this order is a challenge: we cannot know the correct order before completing inference. We thus introduce *local scopes*, which are sets of variables that may be reordered. Since the ordering is not fixed by the programmer, these variables are considered *inferred*, not *specified*, with respect to VKA.

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

`data T1 = $\forall a. MkT1 \ a$`

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

`data P1 :: $\forall(a :: \star). \star$`

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

⁴GHC actually implements the *forall-or-nothing* rule, which states that either *all* variables are quantified by a user-written \forall or none are. Our treatment here is more general than *forall-or-nothing*. Appendix C.2.10 includes more discussion.

3024 7.2.3 DESIRABLE PROPERTIES FOR KIND INFERENCE

3025 The goal of this work is to provide concrete, principled guidance to implementors of depen-
 3026 dently typed languages, such as GHC/Haskell. It is thus important to be able to describe our
 3027 inference algorithm as sound and complete against a *declarative specification*. This declar-
 3028 ative specification is what we might imagine a programmer to have in her head as she pro-
 3029 grams. This system should be designed with a minimum of low-level detail and a minimum
 3030 of surprises. It is then up to an algorithm to live up to the expectations set by the specification.
 3031 The algorithm is sound when all programs it accepts are also accepted by the specification; it
 3032 is complete when all programs accepted by the specification are accepted by the algorithm.

3033 Why choose the particular set of features described here? Because they lead to interesting
 3034 kind inference challenges. We have found that the features above are sufficient in exploring
 3035 kind inference in modern Haskell. We consider unformalized extensions in Section 7.8.

3036 7.3 DATATYPES IN HASKELL98

3037 We begin our formal presentation with Haskell98. The fragment of the syntax of Haskell98
 3038 that concerns us appears at the top of Figure 7.1, including datatype declarations, types,
 3039 kinds, and contexts. The metavariable e refers to expressions, but we do not elaborate the
 3040 details of expressions' syntax or typing rules here. A program pgm is a sequence of groups
 3041 (defined below) of datatype declarations \mathcal{T} , followed by an expression e . We write $\tau_1 \rightarrow \tau_2$
 3042 as an abbreviation for $(\rightarrow)\tau_1 \tau_2$.

3043 7.3.1 GROUPS AND DEPENDENCY ANALYSIS

3044 Users are free to write declarations in any order: earlier declarations can depend on later
 3045 ones in the same compilation unit. However, any kind-checking algorithm must process
 3046 the declarations in dependency order. Complicating this is that type declarations may be
 3047 mutually recursive. A formal analysis of this dependency analysis is not enlightening, so
 3048 we consider it to be a preprocessing step that produces the grammar in Figure 7.1. This
 3049 dependency analysis breaks up the (unordered) raw input into mutually recursive groups
 3050 (potentially containing just one declaration), and puts these in dependency order. We use
 3051 the term *group* to describe a set of mutually recursive declarations.

3052 7.3.2 DECLARATIVE TYPING RULES

3053 The declarative typing rules are in Figure 7.1. There are no surprises here; we review these
 3054 rules briefly. The top judgment is $\Sigma; \Psi \vdash^{pgm} pgm : \sigma$. Its rule **PGM-DT** extends the input

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	$::=$	$\bullet \mid \Sigma, a : \kappa \mid \Sigma, T : \kappa$

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

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

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

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

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

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

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

$\text{K-VAR} \quad \frac{(a : \kappa) \in \Sigma}{\Sigma \vdash^k a : \kappa}$
 $\text{K-TCON} \quad \frac{(T : \kappa) \in \Sigma}{\Sigma \vdash^k T : \kappa}$
 $\text{K-NAT} \quad \frac{}{\Sigma \vdash^k \mathbf{Int} : \star}$
 $\text{K-ARROW} \quad \frac{}{\Sigma \vdash^k \rightarrow : \star \rightarrow \star \rightarrow \star}$

$\text{K-APP} \quad \frac{\Sigma \vdash^k \tau_2 : \kappa_1 \quad \Sigma \vdash^k \tau_1 : \kappa_1 \rightarrow \kappa_2}{\Sigma \vdash^k \tau_1 \tau_2 : \kappa_2}$

Figure 7.1: Declarative specification of Haskell98 datatype declarations

type context Σ with kinds for the datatype declarations to form Σ' , which is used to check both the datatype declarations and the rest of the program. In rule **PGM-DT**, we implicitly extract the names \overline{T}^i from the declarations $\overline{\mathcal{T}}^i$ (and use this abuse of notation throughout our work, relating T to \mathcal{T} and D to \mathcal{D}). The kinds are *guessed* for an entire group all at once: they are added to the context *before* looking at the declarations. This is needed because the declarations in the group refer to one another. Guessing the right answer is typical of declarative type systems. The algorithmic system presented in Section 7.4 provides a mechanism for an implementation. Although there is no special judgment for typing a group of mutually recursive datatypes, we use $\Sigma \vdash^{\text{grp}} \text{rec } \overline{\mathcal{T}}^i \rightsquigarrow \overline{\kappa}_i^i ; \overline{\Psi}_i^i$ 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 type 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 (i.e., Ψ in Figure 7.1), note that the grammar for κ has been extended; accordingly, algorithmic term contexts Γ might include kinds with unification variables, while declarative contexts Ψ do not. This approach of recording unification variables and their solutions in the contexts is inspired by Gundry et al. [2010] and Dunfield and Krishnaswami [2013]. Importantly, an algorithmic context is an *ordered* list, which en-

forces that given $\Delta_1, \hat{\alpha} = \kappa, \Delta_2$, the kind κ must be well-formed under Δ_1 . This rules out solutions like $\hat{\alpha} = \hat{\alpha} \rightarrow \star$ or $\hat{\alpha} = \hat{\beta}, \hat{\beta} = \hat{\alpha}$. Complete contexts Ω are contexts with all unification variables solved.

We use a hole notation for inserting or replacing declarations in the middle of a context. $\Delta[\Theta]$ means that Δ is of the form $\Delta_1, \Theta, \Delta_2$. To reduce clutter, when we have $\Delta[\hat{\alpha}]$, we also use only Δ to refer to the same context. If we have $\Delta[\hat{\alpha}] = \Delta_1, \hat{\alpha}, \Delta_2$, then $\Delta[\hat{\alpha} = \kappa] = \Delta_1, \hat{\alpha} = \kappa, \Delta_2$. This notation allows multiple holes: $\Delta[\Theta_1][\Theta_2]$ means that Δ is of the form $\Delta_1, \Theta_1, \Delta_2, \Theta_2, \Delta_3$. For example, $\Delta[\hat{\alpha}][\hat{\beta}]$ is $\Delta_1, \hat{\alpha}, \Delta_2, \hat{\beta}, \Delta_3$. Critically, $\hat{\alpha}$ appears before $\hat{\beta}$.

Since type contexts carry solutions for unification variables, we use contexts as substitutions: $[\Delta]\kappa$ applies Δ to kind κ . Applying Δ substitutes all solved unification variables in its argument idempotently. If under a complete context Ω , a kind κ is well-formed, then $[\Omega]\kappa$ contains no unification variables and is thus a well-formed declarative kind.

$$\begin{array}{lcl} & \hline [\Delta]\star & = & \star \\ & [\Delta]\kappa_1 \rightarrow \kappa_2 & = & [\Delta]\kappa_1 \rightarrow [\Delta]\kappa_2 \\ & [\Delta[\hat{\alpha}]]\hat{\alpha} & = & \hat{\alpha} \\ & [\Delta[\hat{\alpha} = \kappa]]\hat{\alpha} & = & [\Delta[\hat{\alpha} = \kappa]]\kappa \\ & \hline \end{array}$$

For term contexts, $[\Delta]\Gamma$ applies Δ to each kind in Γ . Similarly, if under Ω , a term context Γ is well-formed, then $[\Omega]\Gamma$ gives back a declarative term context.

$$\begin{array}{lcl} & \hline [\Delta]\bullet & = & \bullet \\ & [\Delta](\Gamma, D : \sigma) & = & [\Delta]\Gamma, D : [\Delta]\sigma \\ & \hline \end{array}$$

The notation $[\Omega]\Delta$ applies a complete context Ω to Δ . We apply Ω to the kind of type variables and type constructors in Δ and remove the binding of solved unification variables from Δ . As above, $[\Omega]\Delta$ is a declarative type context.

$$\begin{array}{lcl} & \hline [\bullet]\bullet & = & \bullet \\ & [\Omega, a : \kappa](\Delta, a : \kappa) & = & [\Omega]\Delta, a : [\Omega]\kappa \\ & [\Omega, T : \kappa](\Delta, T : \kappa) & = & [\Omega]\Delta, T : [\Omega]\kappa \\ & [\Omega, \hat{\alpha} = \kappa](\Delta, \hat{\alpha}) & = & [\Omega]\Delta \\ & [\Omega, \hat{\alpha} = \kappa](\Delta, \hat{\alpha} = \kappa') & = & [\Omega]\Delta \quad \text{if } [\Omega]\kappa = [\Omega]\kappa' \\ & [\Omega, \hat{\alpha} = \kappa]\Delta & = & [\Omega]\Delta \quad \text{if } \hat{\alpha} \notin \Delta \\ & \hline \end{array}$$

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

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	$::=$	$\bullet \mid \Omega, a : \kappa \mid \Omega, T : \kappa \mid \Omega, \widehat{\alpha} = \kappa$

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

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

A-PGM-DT

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

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

A-DT-DECL

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

$$\boxed{\Delta \Vdash_{\tau}^{\text{dc}} \mathcal{D} \rightsquigarrow \tau' \dashv \Theta} \quad (\text{Typing Data Constructor Decl.})$$

A-DC-DECL

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

Figure 7.2: Algorithmic program typing in Haskell98

$$\boxed{\Delta \longrightarrow \Omega} \quad (\text{Defaulting})$$

$ \begin{array}{c} \text{A-CTXDE-EMPTY} \\ \hline \bullet \longrightarrow \bullet \end{array} $	$ \begin{array}{c} \text{A-CTXDE-TVAR} \\ \hline \frac{\Delta \longrightarrow \Omega}{\Delta, a : \kappa \longrightarrow \Omega, a : \kappa} \end{array} $	$ \begin{array}{c} \text{A-CTXDE-TCON} \\ \hline \frac{\Delta \longrightarrow \Omega}{\Delta, T : \kappa \longrightarrow \Omega, T : \kappa} \end{array} $
$ \begin{array}{c} \text{A-CTXDE-KUVAR} \\ \hline \frac{\Delta \longrightarrow \Omega}{\Delta, \hat{\alpha} = \kappa \longrightarrow \Omega, \hat{\alpha} = \kappa} \end{array} $	$ \begin{array}{c} \text{A-CTXDE-SOLVE} \\ \hline \frac{\Delta \longrightarrow \Omega}{\Delta, \hat{\alpha} \longrightarrow \Omega, \hat{\alpha} = \star} \end{array} $	

Figure 7.3: Defaulting in Haskell98.

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 $\hat{\alpha}$, and then type-check ($\|\text{dt}\rangle$) each datatype definition (Section 7.4.4), producing the context Θ_{n+1} . Then we default (Section 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 unsolved unification variables of one group do not propagate to the rest of the program; accordingly, the input context of $\|\text{pgm}\rangle$ is always a complete context. Echoing the notation for the declarative system, we write $\Omega \|\text{grp}\rangle \text{rec } \overline{\mathcal{T}}_i^i \rightsquigarrow \overline{\kappa}_i^i; \overline{\Gamma}_i^i \dashv \Theta$ 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, $\Delta \longrightarrow \Omega$, given in Figure 7.3. 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 .

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.

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

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 $\widehat{\alpha}_i^i \rightarrow \star$. The rule then unifies κ with $\widehat{\alpha}_i^i \rightarrow \star$. Before unification, we apply the context Δ to κ , thus $[\Delta]\kappa$; 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

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

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

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

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

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

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

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

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

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

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

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

Figure 7.4: Algorithmic kinding, unification and promotion in Haskell98.

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 $\Delta \Vdash_{\tau}^{\text{dc}} \mathcal{D} \rightsquigarrow \tau' \dashv \Theta$ 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 $\Delta \Vdash^k \tau : \kappa \dashv \Theta$ is given in Figure 7.4. 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 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 $\Delta \Vdash^u \kappa_1 \approx \kappa_2 \dashv \Theta$ is given in Figure 7.4. 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](#).

$$\boxed{\Delta \longrightarrow \Theta} \quad (\text{Context Extension})$$

$$\begin{array}{c}
\text{A-CTXE-EMPTY} \\
\hline
\bullet \longrightarrow \bullet
\end{array}
\quad
\begin{array}{c}
\text{A-CTXE-TVAR} \\
\hline
\frac{\Delta \longrightarrow \Theta}{\Delta, a : \kappa \longrightarrow \Theta, a : \kappa}
\end{array}
\quad
\begin{array}{c}
\text{A-CTXE-TCON} \\
\hline
\frac{\Delta \longrightarrow \Theta}{\Delta, T : \kappa \longrightarrow \Theta, T : \kappa}
\end{array}$$

$$\begin{array}{c}
\text{A-CTXE-KUVAR} \\
\hline
\frac{\Delta \longrightarrow \Theta}{\Delta, \hat{\alpha} \longrightarrow \Theta, \hat{\alpha}}
\end{array}
\quad
\begin{array}{c}
\text{A-CTXE-KUVAR SOLVED} \\
\hline
\frac{\Delta \longrightarrow \Theta \quad [\Theta]\kappa_1 = [\Theta]\kappa_2}{\Delta, \hat{\alpha} = \kappa_1 \longrightarrow \Theta, \hat{\alpha} = \kappa_2}
\end{array}
\quad
\begin{array}{c}
\text{A-CTXE-SOLVE} \\
\hline
\frac{\Delta \longrightarrow \Theta \quad \Theta \Vdash^{\text{kv}} \kappa}{\Delta, \hat{\alpha} \longrightarrow \Theta, \hat{\alpha} = \kappa}
\end{array}$$

$$\begin{array}{c}
\text{A-CTXE-ADD} \\
\hline
\frac{\Delta \longrightarrow \Theta}{\Delta \longrightarrow \Theta, \hat{\alpha}}
\end{array}
\quad
\begin{array}{c}
\text{A-CTXE-ADD SOLVED} \\
\hline
\frac{\Delta \longrightarrow \Theta \quad \Theta \Vdash^{\text{kv}} \kappa}{\Delta \longrightarrow \Theta, \hat{\alpha} = \kappa}
\end{array}$$

Figure 7.5: Context extension in Haskell98

PROMOTION. As described in Section 6.1.2, the crucial insight of \Vdash^{pr} is that *the relative order between unification variables does not matter for solving a constraint*. The promotion judgment $\Delta \Vdash_{\hat{\alpha}}^{\text{pr}} \kappa_1 \rightsquigarrow \kappa_2 \dashv \Theta$ 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 6.3. Importantly, in rule **A-PR-KUVAR**, 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 $\Omega \Vdash^{\text{ctx}} \Gamma$, 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 $\Omega \Vdash^{\text{ctx}} \Gamma$; 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*. The formal definition of context extension is given in Figure 7.5, and its treatment is as Dunfield

and Krishnaswami [2013]. Intuitively, context extension preserves all information in Δ , and may increase the information by adding or solving more unification variables. 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 $[\Omega]\Omega \vdash^{\text{grp}} \text{rec } \overline{T}_i^i \rightsquigarrow \overline{\kappa}_i^i; \overline{\Psi}_i^i$, then there exists $\overline{\kappa}_i^i, \overline{\Gamma}_i^i, \Theta$, and Ω' , such that $\Omega \Vdash^{\text{grp}} \text{rec } \overline{T}_i^i \rightsquigarrow \overline{\kappa}_i^i; \overline{\Gamma}_i^i \dashv \Theta$, where $\Theta \longrightarrow \Omega'$, and $[\Omega']\overline{\kappa}_i^i = \overline{\kappa}_i^i$, 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.) 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.

7.5.1 TYPE PARAMETERS

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

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

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

Definition 18 (Principal Kind in Haskell98 with Kind Parameters). Given a context Σ , a 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 $\text{rec } \overline{\mathcal{T}}_i^i$, denoted as $\Sigma \vdash \text{rec } \overline{\mathcal{T}}_i^i \rightsquigarrow^p \overline{\kappa}_i^i$, if $\Sigma \vdash^{\text{grp}} \text{rec } \overline{\mathcal{T}}_i^i \rightsquigarrow \overline{\kappa}_i^i; \overline{\Psi}_i^i$, and whenever $\Sigma \vdash^{\text{grp}} \text{rec } \overline{\mathcal{T}}_i^i \rightsquigarrow \overline{\kappa}'_i^i; \overline{\Psi}'_i^i$ holds, there exists some substitution S , such that $\overline{S(\kappa_i)} = \overline{\kappa}'_i^i$ and $\overline{S(\Psi_i)} = \overline{\Psi}'_i^i$.

Theorem 7.3 (Principality of Haskell98 with Kind Parameters). If $\Sigma \vdash^{\text{grp}} \text{rec } \overline{\mathcal{T}}_i^i \rightsquigarrow \overline{\kappa}_i^i; \overline{\Psi}_i^i$, then there exists some $\overline{\kappa}'_i^i$ such that $\Sigma \vdash \text{rec } \overline{\mathcal{T}}_i^i \rightsquigarrow^p \overline{\kappa}'_i^i$.

7.5.2 PRINCIPAL KINDS AND DEFAULTING

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

Definition 19 (Defaulting Kind Parameter Substitution). Let $S^* \in \text{KPARAM} \rightarrow \kappa$ denote 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 $S^*(\kappa_i)$.

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

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 7.11. Compared to Haskell98, programs pgm now include datatype signatures S^5 . 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 (monomorphic) 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 20 (Dependency Analysis in PolyKinds). All definitions form a graph, where

- notes are either (a) a signature, or (b) a definition; and
- edges indicate the dependency relation:
 - (i) If the signature/definition of T_1 mentions T_2 , then:
 - a) if T_2 has a signature, the signature/definition of T_1 depends on the signature of T_2 ;
 - b) otherwise, the signature/definition of T_1 depends on the definition of T_2 .

⁵Our use of **sig**, just like **rec**, is for the purpose of formalism only, not as a required keyword in the source program.

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 7.6: Syntax of PolyKinds

(ii) A definition depends on its signature.

We then perform *strongly connected component* analysis on the group, where mutually recursive definitions form a group, producing the grammar in Figure 7.11.

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

```

data T1 :: T2 a → ★
data T2 :: T1 → ★

```

is rejected, lest $T1$ be assigned type $\forall(a :: T1). T2 a \rightarrow \star$. In other words, signatures do not form groups: they are always processed individually. Moreover, the definition of a datatype which has a signature does not join others in a group, as according to Definition 20, there will be no dependency from datatypes on it. This simplifies the kinding procedure, as we will see in the coming section.

7.6.2 CHECKING PROGRAMS

The declarative typing rules appear in Figure 7.7, with auxiliary functions defined in Figure 7.8. The judgment $\Sigma; \Psi \vdash^{pgm} pgm : \sigma$ checks the program. From now on we omit the typing rule for expressions in programs, which is essentially the same as in Haskell98. Rule **PGM-SIG** processes kind signatures by elaborating and generalizing the kind, then adding it to the context Σ . The helper judgment $\Sigma \vdash^{sig} \mathcal{S} \rightsquigarrow T : \eta$ checks a kind signature $\mathbf{data} T : \sigma$.

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

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

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

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

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

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

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

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

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

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

$$\boxed{\Sigma \vdash_{\phi^c}^{\text{gen}} \Psi_1 \rightsquigarrow \Psi_2} \quad (\text{Generalization})$$

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

Figure 7.7: Declarative specification of PolyKinds

$\boxed{\lceil \sigma \rceil}$
(Kind results in \star)

SR-STAR	SR-ARROW	SR-FORALL
$\frac{}{\lceil \star \rceil}$	$\frac{\lceil \kappa_2 \rceil}{\lceil \kappa_1 \rightarrow \kappa_2 \rceil}$	$\frac{\lceil \sigma \rceil}{\lceil \forall \phi. \sigma \rceil}$

$\mathbf{fkv}(\forall a. \sigma)$	$=$	$\mathbf{fkv}(\sigma) \setminus_a$
$\mathbf{fkv}(\forall a : \kappa. \sigma)$	$=$	$\mathbf{fkv}(\sigma) \setminus_a \cup \mathbf{fkv}(\kappa)$
$\mathbf{fkv}(\tau_1 \tau_2)$	$=$	$\mathbf{fkv}(\tau_1) \cup \mathbf{fkv}(\tau_2)$
$\mathbf{fkv}(\tau_1 @ \tau_2)$	$=$	$\mathbf{fkv}(\tau_1) \cup \mathbf{fkv}(\tau_2)$
$\mathbf{fkv}(a)$	$=$	$\{a\}$
$\mathbf{fkv}(\star)$	$=$	\emptyset
$\mathbf{fkv}(\text{Int})$	$=$	\emptyset
$\mathbf{fkv}(T)$	$=$	\emptyset
$\mathbf{fkv}(\rightarrow)$	$=$	\emptyset

$\mathcal{Q}(\Sigma, \sigma)$ is the set of all well-formed orderings of $\mathcal{Q}_\kappa(\Sigma, \sigma)$

$\mathcal{Q}_\kappa(\Sigma, \forall a. \sigma)$	$=$	$\mathcal{Q}_\kappa(\Sigma, \sigma) \setminus_a$
$\mathcal{Q}_\kappa(\Sigma, \forall a : \kappa. \sigma)$	$=$	$\mathcal{Q}_\kappa(\Sigma, \sigma) \setminus_a \cup \mathcal{Q}_\kappa(\Sigma, \kappa)$
$\mathcal{Q}_\kappa(\Sigma, \tau_1 \tau_2)$	$=$	$\mathcal{Q}_\kappa(\Sigma, \tau_1) \cup \mathcal{Q}_\kappa(\Sigma, \tau_2)$
$\mathcal{Q}_\kappa(\Sigma, \tau_1 @ \tau_2)$	$=$	$\mathcal{Q}_\kappa(\Sigma, \tau_1) \cup \mathcal{Q}_\kappa(\Sigma, \tau_2)$
$\mathcal{Q}_\kappa(\Sigma, a)$	$=$	$\mathcal{Q}_\kappa(\Sigma, \rho) \cup \{a : \rho\}$ where $a : \rho \in \Sigma$
$\mathcal{Q}_\kappa(\Sigma, \star)$	$=$	\emptyset
$\mathcal{Q}_\kappa(\Sigma, \text{Int})$	$=$	\emptyset
$\mathcal{Q}_\kappa(\Sigma, T)$	$=$	\emptyset
$\mathcal{Q}_\kappa(\Sigma, \rightarrow)$	$=$	\emptyset

Figure 7.8: Auxiliary functions of declarative specification in PolyKinds

First, it uses $\lceil \sigma \rceil$ to ensure σ returns \star : $\lceil \sigma \rceil$ simply traverses over arrows and forall, checking that the final kind of σ is \star . Then, as σ may be an open kind signature, it extracts the free kind variables $\phi = \mathbf{fkv}(\sigma)$, so that ϕ is a well-formed ordering of the free variables. 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 ϕ_1^c 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. Thus we get rid of the issues of defaulting; that is, the example of *Q1* and *Q2* in Section 7.4.3 will be accepted in PolyKinds as instead of defaulting *Q2* will get a polymorphic type. 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 $\mathcal{Q}(\Sigma, \sigma)$ is the set of all well-formed orderings of the free variables (transitively looking into variables' kinds) of σ . 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 .

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

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

$$\begin{array}{c} \text{INST-FORALL-INFER} \\ \hline \Sigma \vdash^{\text{ela}} \rho : \omega_1 \quad \Sigma \vdash^{\text{inst}} \mu_1 @ \rho : \eta[a \mapsto \rho] <: \omega_2 \rightsquigarrow \mu_2 \\ \hline \Sigma \vdash^{\text{inst}} \mu_1 : \forall \{a : \omega_1\}. \eta <: \omega_2 \rightsquigarrow \mu_2 \end{array}$$

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

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

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

$$\begin{array}{c} \text{KTT-STAR} \quad \text{KTT-NAT} \quad \text{KTT-VAR} \quad \text{KTT-ARROW} \\ \hline \Sigma \vdash^{\text{k}} \star : \star \rightsquigarrow \star \quad \Sigma \vdash^{\text{k}} \text{Int} : \star \rightsquigarrow \text{Int} \quad \frac{(a : \omega) \in \Sigma}{\Sigma \vdash^{\text{k}} a : \omega \rightsquigarrow a} \quad \Sigma \vdash^{\text{k}} \rightarrow : \star \rightarrow \star \rightarrow \star \rightsquigarrow \rightarrow \end{array}$$

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

$$\begin{array}{c} \text{KTT-TCON} \quad \text{KTT-KAPP} \\ \hline \frac{(T : \eta) \in \Sigma}{\Sigma \vdash^{\text{k}} T : \eta \rightsquigarrow T} \quad \frac{\Sigma \vdash^{\text{k}} \kappa_1 : \forall a : \omega. \eta \rightsquigarrow \rho_1 \quad \Sigma \vdash^{\text{kc}} \kappa_2 \Leftarrow \omega \rightsquigarrow \rho_2}{\Sigma \vdash^{\text{k}} \kappa_1 @ \kappa_2 : \eta[a \mapsto \rho_2] \rightsquigarrow \rho_1 @ \rho_2} \end{array}$$

$$\begin{array}{c} \text{KTT-KAPP-INFER} \\ \hline \Sigma \vdash^{\text{k}} \kappa_1 : \forall \{\overline{a_i} : \overline{\omega_i}^i\}. \forall a : \omega. \eta \rightsquigarrow \rho'_1 \\ \frac{\Sigma \vdash^{\text{ela}} \rho_i : \omega_i[\overline{a_i} \mapsto \overline{\rho_i}^i] \quad \Sigma \vdash^{\text{kc}} \kappa_2 \Leftarrow \omega[\overline{a_i} \mapsto \overline{\rho_i}^i] \rightsquigarrow \rho'_2}{\Sigma \vdash^{\text{k}} \kappa_1 @ \kappa_2 : \eta[\overline{a_i} \mapsto \overline{\rho_i}^i][a \mapsto \rho_2] \rightsquigarrow \rho'_1 @ \overline{\rho_i}^i @ \rho'_2} \end{array}$$

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

$$\begin{array}{c} \text{KTT-FORALL} \\ \hline \Sigma \vdash^{\text{kc}} \kappa \Leftarrow \star \rightsquigarrow \omega \quad \Sigma, a : \omega \vdash^{\text{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu \\ \hline \Sigma \vdash^{\text{k}} \forall a : \kappa. \sigma : \star \rightsquigarrow \forall a : \omega. \mu \end{array}$$

Figure 7.9: Declarative kind-checking in PolyKinds

$\boxed{\Sigma \vdash^{\text{ela}} \mu : \eta}$		<i>(Elaborated Kinding)</i>	
$\frac{\text{ELA-STAR}}{\Sigma \vdash^{\text{ela}} \star : \star}$	$\frac{\text{ELA-NAT}}{\Sigma \vdash^{\text{ela}} \text{Int} : \star}$	$\frac{\text{ELA-VAR} \quad (a : \omega) \in \Sigma}{\Sigma \vdash^{\text{ela}} a : \omega}$	$\frac{\text{ELA-TCON} \quad (T : \eta) \in \Sigma}{\Sigma \vdash^{\text{ela}} T : \eta}$
$\frac{\text{ELA-ARROW}}{\Sigma \vdash^{\text{ela}} \rightarrow : \star \rightarrow \star \rightarrow \star}$	$\frac{\text{ELA-APP} \quad \Sigma \vdash^{\text{ela}} \rho_1 : \omega_1 \rightarrow \omega_2 \quad \Sigma \vdash^{\text{ela}} \rho_2 : \omega_1}{\Sigma \vdash^{\text{ela}} \rho_1 \rho_2 : \omega_2}$		
$\frac{\text{ELA-KAPP} \quad \Sigma \vdash^{\text{ela}} \rho_1 : \forall a : \omega. \eta \quad \Sigma \vdash^{\text{ela}} \rho_2 : \omega}{\Sigma \vdash^{\text{ela}} \rho_1 @ \rho_2 : \eta[a \mapsto \rho_2]}$		$\frac{\text{ELA-KAPP-INFER} \quad \Sigma \vdash^{\text{ela}} \rho_1 : \forall \{a : \omega\}. \eta \quad \Sigma \vdash^{\text{ela}} \rho_2 : \omega}{\Sigma \vdash^{\text{ela}} \rho_1 @ \rho_2 : \eta[a \mapsto \rho_2]}$	
$\frac{\text{ELA-FORALL} \quad \Sigma \vdash^{\text{ela}} \omega : \star \quad \Sigma, a : \omega \vdash^{\text{ela}} \mu : \star}{\Sigma \vdash^{\text{ela}} \forall a : \omega. \mu : \star}$		$\frac{\text{ELA-FORALL-INFER} \quad \Sigma \vdash^{\text{ela}} \omega : \star \quad \Sigma, a : \omega \vdash^{\text{ela}} \mu : \star}{\Sigma \vdash^{\text{ela}} \forall \{a : \omega\}. \mu : \star}$	

Figure 7.10: Declarative elaborated kinding in PolyKinds

3351 7.6.3 CHECKING KINDS

3352 The kinding judgment \vdash^k appears in Figure 7.9. Kinding $\Sigma \vdash^k \sigma : \eta \rightsquigarrow \mu$ infers the type σ
3353 to have kind η , and it elaborates σ to μ . The kinding rules are built upon the axiom $\Sigma \vdash^k \star :$
3354 $\star \rightsquigarrow \star$ (rule **KTT-STAR**). While this axiom is known to violate logical consistency, as Haskell
3355 is already logically inconsistent because of its general recursion, we do not consider it as an
3356 issue here⁶. Rule **KTT-APP** concerns applications $\tau_1 \tau_2$. It first infers the kind of τ_1 to be η_1 .
3357 The kind η_1 can be a polymorphic kind headed by a \forall , though it is expected to be a function
3358 kind. Thus the rule uses \vdash^{inst} to instantiate η_1 to $\omega_1 \rightarrow \omega_2$. The instantiation judgment
3359 $\Sigma \vdash^{\text{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2$ instantiates a kind η to a monokind ω , where if μ_1 has kind η
3360 then μ_2 has kind ω . After instantiation, rule **KTT-APP** checks (\vdash^{kc}) the argument τ_2 against
3361 the expected argument kind ω_1 . The kind checking judgment \vdash^{kc} simply delegates the work
3362 to kinding and instantiation. Rule **KTT-KAPP** checks visible kind applications. Note in the
3363 return kind η , the variable a is substituted by the elaborated argument ρ_2 . Rule **KTT-KAPP-**
3364 **INFER** is similar to rule **KTT-KAPP**, but before we instantiated specified variables, we must first
3365 instantiate all inferred variables. Rule **KTT-FORALLI** elaborates an unannotated type $\forall a. \sigma$ to

⁶This design is different from terminating dependently-typed languages, which require polymorphism to be stratified into a hierarchy of levels, or otherwise they permit an encoding of Girard's paradox [Girard 1972]. But as Haskell is already non-terminating, there is no motivation for stratification. See [Weirich et al. 2017] for more discussion.

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 7.11: Algorithmic syntax in PolyKinds

3366 $\forall a : \omega. \mu$, where ω is an *elaborated* kind (\vdash^{ela}) guessed for a . Rule **KTT-FORALL** is similar,
 3367 elaborating an annotated type $\forall a : \kappa. \sigma$ to $\forall a : \omega. \mu$, where ω is the elaborated result of κ .

3368 The stand-alone elaborated kinding judgment \vdash^{ela} type-checks elaborated types, given in
 3369 Figure 7.10. As all necessary instantiation has been done, type-checking for elaborated types
 3370 is easy. For example, rule **ELA-APP** concerns applications $\rho_1 \rho_2$. Compared to rule **KTT-APP**,
 3371 here ρ_1 has an arrow kind, and takes exactly the kind of ρ_2 . All judgments output well-formed
 3372 elaborated types, as the following lemma states:

3373 **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*
 3374 *$\Sigma \vdash^{kc} \sigma \Leftarrow \eta \rightsquigarrow \mu$, 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*
 3375 *$\Sigma \vdash^{\text{ela}} \mu_2 : \omega$.*

3376 7.7 KIND INFERENCE FOR POLYKINDS

3377 We now describe the *algorithmic* counterpart of the PolyKinds system. Figure 7.11 presents
 3378 the syntax of kinds and contexts in the algorithmic system for PolyKinds. Elaborated mono-
 3379 types are extended with unification variables $\hat{\alpha}$. Echoing the algorithm for Haskell98, type
 3380 contexts are extended with unification variables, which now have kinds ($\hat{\alpha} : \omega$ and $\hat{\alpha} : \omega =$
 3381 ρ). Also added to contexts are local scopes $\{\Delta\}$. These are special type contexts, where
 3382 *variables can be reordered*. Recall the kind $\forall(a :: (f\ b))\ (c :: k). f\ c \rightarrow \star$ in Section 7.2.2,
 3383 where f and b appear before k , but end up depending on k . In which order should we put
 3384 f , b and k in the algorithmic context to kind-check the signature? We cannot have a correct
 3385 order before completing inference. Therefore, we put them into a local scope, knowing we
 3386 can reorder the variables during kind-checking according to the dependency information.
 3387 The well-formedness judgment for local scopes requires them to be well-scoped, leading to
 3388 the fact that $\Delta, \{\Delta'\}$ is well-formed iff Δ, Δ' is. The marker \blacktriangleright_D , subscripted by the name of
 3389 a data constructor, is used only in and explained with rule **A-DC-TT**.

3390 7.7.1 ALGORITHMIC PROGRAM TYPING

3391 The algorithmic typing rules appear in Figure 7.12, with some of the auxiliary definitions
 3392 given in Figure 7.13. The judgment $\Omega; \Gamma \Vdash^{\text{pgm}} \text{pgm} : \mu$ checks the program. The rule **A-**
 3393 **PGM-SIG** and rule **A-PGM-DT-TTS** correspond directly to the declarative rules. Note that as
 3394 the datatype declaration in rule **A-PGM-DT-TTS** already has a signature, the output type con-
 3395 text remains unchanged. Rule **A-PGM-DT-TT** concerns a group (without kind signatures).
 3396 Like in Haskell98, it first assigns a fresh unification variable $\hat{\alpha}_i : \star$ as the kind of each type
 3397 constructor, and then type-checks each datatype declaration, yielding the output context
 3398 Θ_{n+1} . Unlike Haskell98 which then uses defaulting, here from each $\hat{\alpha}_i$ we get their unsolved
 3399 unification variables $\hat{\phi}_i^c$ and generalize the kind of each type constructor as well as the type
 3400 of each data constructor. The **unsolved** (Δ) metafunction (Figure 7.13) simply extracts a set
 3401 of free unification variables in Δ , with their kinds substituted by Δ . Before generalization,
 3402 we apply Θ_{n+1} to the results so all solved unification variables get substituted away. We use
 3403 the notation $\hat{\phi}_i^c \mapsto \phi_i^c$ to mean that all unification variables in $\hat{\phi}_i^c$ are replaced by fresh type
 3404 variables in ϕ_i^c . The algorithmic generalization judgment \Vdash^{gen} corresponds straightforwardly
 3405 to the declarative rule, and thus is omitted. Though they appear daunting, the extended con-
 3406 texts used in the last premise to this rule are unsurprising: they just apply the relevant sub-
 3407 stitutions (the solved unification variables in Θ_{n+1} , the replacement of unification variables
 3408 with fresh proper type variables $\hat{\phi}_i^c \mapsto \phi_i^c$, and the generalization of the kinds of the group
 3409 of datatypes $T_i \mapsto T_i @ \phi_i^c$).

3410 The judgment $\Omega \Vdash^{\text{sig}} \mathcal{S} \rightsquigarrow T : \eta$ type-checks a signature definition (rule **A-SIG-TT**).
 3411 We get all free variables in σ using $\text{fkv}(\sigma)$ and assign each variable a_i a kind $\hat{\alpha}_i : \star$. Those
 3412 variables are put into a local scope to kind-check σ . Then, we use `scoped_sort`—a standard
 3413 topological sort—to return an ordering of the variables that respects dependencies. Finally,
 3414 we substitute away solved unification variables in the result kind μ and generalize over the
 3415 unsolved variables $\hat{\phi}_2^c$ in Δ . As $\hat{\phi}_2^c$ is generalized outside ϕ_1^c , we use the *quantification check*
 3416 $\Delta \hookrightarrow \bar{a}_i^i$ (Section 7.7.2) to ensure the result kind is well-ordered.

3417 Rule **A-DT-TT** is a straightforward generalization of rule **A-DT-DECL** to polymorphic kinds.
 3418 Here T can have a polymorphic kind from kind signatures.

3419 Rule **A-DC-TT** checks a data constructor declaration. It first puts a marker into the context
 3420 before kinding. After kinding, it substitutes away all the solved unification variables to the
 3421 right of the marker, and generalizes over all unsolved unification variables to the right of the
 3422 marker. The fact that the context is ordered gives us precise control over variables that need
 3423 generalization.

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

(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

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

A-PGM-DT-TT

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

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

(Typing Signature)

A-SIG-TT

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

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

(Typing Datatype Decl.)

A-DT-TT

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

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

(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 7.12: Algorithmic program typing in PolyKinds

unsolved (\bullet)	=	\bullet
unsolved ($\Delta, a : \omega$)	=	unsolved (Δ)
unsolved ($\Delta, T : \eta$)	=	unsolved (Δ)
unsolved ($\Delta, \{\Delta^{\text{lo}}\}$)	=	unsolved ($\Delta, \Delta^{\text{lo}}$)
unsolved ($\Delta, \hat{\alpha} : \omega_1 = \omega_2$)	=	unsolved (Δ)
unsolved ($\Delta, \hat{\alpha} : \omega$)	=	unsolved (Δ), $\hat{\alpha} : [\Delta]\omega$

$\Delta \hookrightarrow \phi$

(Quantification Checking)

A-QC-TVAR $\frac{\Delta \hookrightarrow \phi}{\Delta, a : \omega \hookrightarrow \phi}$	A-QC-TCON $\frac{\Delta \hookrightarrow \phi}{\Delta, T : \eta \hookrightarrow \phi}$	A-QC-LO $\frac{\Delta, \Delta^{\text{lo}} \hookrightarrow \phi}{\Delta, \{\Delta^{\text{lo}}\} \hookrightarrow \phi}$	A-QC-KUVAR SOLVED $\frac{\Delta \hookrightarrow \phi}{\Delta, \hat{\alpha} : \omega_1 = \omega_2 \hookrightarrow \phi}$
A-QC-KUVAR $\frac{\Delta \hookrightarrow \phi \quad \mathbf{fkv}([\Delta]\omega) \# \phi}{\Delta, \hat{\alpha} : \omega \hookrightarrow \phi}$			

Figure 7.13: Unsolved set of unification variables and quantification check in PolyKinds

7.7.2 THE QUANTIFICATION CHECK

In rule **A-SIG-TT**, we reject ill-ordered kinds. Consider the following example:

```

3424 data Proxy :: ∀k. k → ★
3425 data Relate :: ∀a (b :: a). a → Proxy b → ★
3426 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 $\hat{\alpha}$. We discover that $\hat{\alpha} = \text{Proxy } \hat{\beta}$, where $\hat{\beta} :: a$. There are no further constraints on $\hat{\beta}$. Naïvely, we would generalize over $\hat{\beta}$, 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 $\hat{\beta}$ would become if generalized) before other ones. Recall in Section 7.6.2, we always put quantified variables in this order: variables arising from generalization, from implicit quantification, and from explicit quantification. This order is intuitive: variables arising from generalization is invisible to the users and thus are always at the front; variables from implicit quantification refer to user-written variables and follow next; and variables from explicit quantification correspond to a user-specified telescope and thus should not be modified.

3442 The quantification-checking metafunction $\Delta \hookrightarrow \phi$, given in Figure 7.13, essentially ex-
 3443 presses $\text{fkv}(\text{unsolved}(\Delta)) \# \phi$, which ensures that free variables in $\text{unsolved}(\Delta)$ are disjoint
 3444 ($\#$) with ϕ , so that we can safely generalize $\text{unsolved}(\Delta)$ outside ϕ .

3445 Unfortunately, quantification check also makes us lose principality, as we will discuss in
 3446 Section 7.7.6. We conjecture, though, that lifting quantification check and allowing for in-
 3447 terleaving specified/inferred variables may restore principality. That means, for the above
 3448 example, we can get

3449 $\text{data } T :: \forall(a :: \star) (b :: a) (c :: a) \{f :: a\} (d :: \text{Proxy } f). \text{Relate } @a @f b d \rightarrow \star$

3450 We also consider alternative design in Appendix C.2.8.

3451 7.7.3 KINDING

3452 Figure 7.14 presents the selected rules for kinding judgment \Vdash^k , along with the auxiliary
 3453 judgments. Full rules can be found in Appendix C.3. Most rules correspond directly to their
 3454 declarative counterparts. For applications $\tau_1 \tau_2$, rule **A-KTT-APP** first synthesizes the kind of
 3455 τ_1 to be η_1 , then uses \Vdash^{kapp} to type-check τ_2 . The judgment $\Delta \Vdash^{\text{kapp}} (\rho_1 : \eta) \bullet \tau : \omega \rightsquigarrow \rho_2 \dashv \Theta$
 3456 is interpreted as, under context Δ , applying the type ρ_1 of kind η to the type τ returns kind
 3457 ω , the elaboration result ρ_2 , and an output context Θ . When η_1 is polymorphic (rule **A-KAPP-**
 3458 **TT-FORALL**), we instantiate it with a fresh unification variable. Rule **A-KTT-FORALLI** checks
 3459 a polymorphic type. We assign a unification variable as the kind of a , bring $\hat{\alpha} : \star, a : \hat{\alpha}$ into
 3460 scope to check the body against \star , yielding the output context $\Delta_2, a : \hat{\alpha}, \Delta_3$. As a goes out of
 3461 the scope in the conclusion, we need to drop a in the concluding context. To make sure that
 3462 dropping a outputs a well-formed context, we substitute away all solved unification variables
 3463 in Δ_3 for the return kind, and keep only $\text{unsolved}(\Delta_3)$, which are ensured ($\Delta_3 \hookrightarrow a$) to
 3464 have no dependency on a .

3465 In the algorithmic elaborated kinding judgment $\Delta \Vdash^{\text{ela}} \mu : \eta$, we keep the invariant:
 3466 $[\Delta]\eta = \eta$. That is why in rule **A-ELA-APP** we substitute a with $[\Delta]\rho_2$.

3467 Instantiation (\Vdash^{inst}) contains the only entry to unification (rule **A-INST-REFL**).

3468 7.7.4 UNIFICATION

3469 The judgments of unification and promotion are excerpted in Figure 7.15. Most rules are
 3470 natural extensions of those in Haskell98.

3471 **PROMOTION** The promotion judgment $\Delta \Vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta$ is extended with kind an-
 3472 notations for unification variables. As our unification variables have kinds now, rule **A-PR-**
 3473 **KUVAR-**TT**** must also promote the kind of $\hat{\beta}$, so that $\hat{\beta}_1 : \rho_1$ in the context is well-formed.

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

$$\begin{array}{c} \text{A-INST-REFL} \\ \frac{\Delta \Vdash^{\text{u}} \omega_1 \approx \omega_2 \dashv \Theta}{\Delta \Vdash^{\text{inst}} \mu : \omega_1 <: \omega_2 \rightsquigarrow \mu \dashv \Theta} \end{array} \quad \begin{array}{c} \text{A-INST-FORALL} \\ \frac{\Delta, \hat{\alpha} : \omega_1 \Vdash^{\text{inst}} \mu_1 @ \hat{\alpha} : \eta[a \mapsto \hat{\alpha}] <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta}{\Delta \Vdash^{\text{inst}} \mu_1 : \forall a : \omega_1. \eta <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta} \end{array}$$

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

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

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

$$\text{A-KTT-STAR} \quad \frac{}{\Delta \Vdash^{\text{k}} \star : \star \rightsquigarrow \star \dashv \Delta}$$

$$\text{A-KTT-APP} \quad \frac{\Delta \Vdash^{\text{k}} \tau_1 : \eta_1 \rightsquigarrow \rho_1 \dashv \Delta_1 \quad \Delta_1 \Vdash^{\text{kapp}} (\rho_1 : [\Delta_1] \eta_1) \bullet \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\text{k}} \tau_1 \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}$$

$$\text{A-KTT-FORALLI} \quad \frac{\Delta, \hat{\alpha} : \star, a : \hat{\alpha} \Vdash^{\text{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu \dashv \Delta_2, a : \hat{\alpha}, \Delta_3 \quad \Delta_3 \hookrightarrow a}{\Delta \Vdash^{\text{k}} \forall a. \sigma : \star \rightsquigarrow \forall a : \hat{\alpha}. [\Delta_3] \mu \dashv \Delta_2, \text{unsolved}(\Delta_3)}$$

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

$$\text{A-KAPP-TT-ARROW} \quad \frac{\Delta \Vdash^{\text{kc}} \tau \Leftarrow \omega_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta \Vdash^{\text{kapp}} (\rho_1 : \omega_1 \rightarrow \omega_2) \bullet \tau : \omega_2 \rightsquigarrow \rho_1 \rho_2 \dashv \Theta}$$

$$\text{A-KAPP-TT-FORALL} \quad \frac{\Delta, \hat{\alpha} : \omega_1 \Vdash^{\text{kapp}} (\rho_1 @ \hat{\alpha} : \eta[a \mapsto \hat{\alpha}]) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\text{kapp}} (\rho_1 : \forall a : \omega_1. \eta) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}$$

$$\text{A-KAPP-TT-KUVAR} \quad \frac{\Delta_1, \hat{\alpha}_1 : \star, \hat{\alpha}_2 : \star, \hat{\alpha} : \omega = (\hat{\alpha}_1 \rightarrow \hat{\alpha}_2), \Delta_2 \Vdash^{\text{kc}} \tau \Leftarrow \hat{\alpha}_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta_1, \hat{\alpha} : \omega, \Delta_2 \Vdash^{\text{kapp}} (\rho_1 : \hat{\alpha}) \bullet \tau : \hat{\alpha}_2 \rightsquigarrow \rho_1 \rho_2 \dashv \Theta}$$

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

$$\begin{array}{c} \text{A-ELA-APP} \\ \frac{\Delta \Vdash^{\text{ela}} \rho_1 : \omega_1 \rightarrow \omega_2 \quad \Delta \Vdash^{\text{ela}} \rho_2 : \omega_1}{\Delta \Vdash^{\text{ela}} \rho_1 \rho_2 : \omega_2} \end{array} \quad \begin{array}{c} \text{A-ELA-KAPP} \\ \frac{\Delta \Vdash^{\text{ela}} \rho_1 : \forall a : \omega. \eta \quad \Delta \Vdash^{\text{ela}} \rho_2 : \omega}{\Delta \Vdash^{\text{ela}} \rho_1 @ \rho_2 : \eta[a \mapsto [\Delta] \rho_2]} \end{array}$$

Figure 7.14: Selected rules for algorithmic kinding in PolyKinds

$\Delta \Vdash \omega_1 \approx \omega_2 \dashv \Theta$ (Unification)

$$\frac{\text{A-U-REFL-TT}}{\Delta \Vdash \omega \approx \omega \dashv \Delta} \quad \frac{\text{A-U-APP} \quad \Delta \Vdash \rho_1 \approx \rho_3 \dashv \Delta_1 \quad \Delta_1 \Vdash [\Delta_1] \rho_2 \approx [\Delta_1] \rho_4 \dashv \Theta}{\Delta \Vdash \rho_1 \rho_2 \approx \rho_3 \rho_4 \dashv \Theta}$$

$$\frac{\text{A-U-KVARL-TT} \quad \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \hat{\alpha} : \omega_1, \Theta_2 \quad \Theta_1 \Vdash^{\text{ela}} \rho_2 : \omega_2 \quad \Theta_1 \Vdash [\Theta_1] \omega_1 \approx \omega_2 \dashv \Theta_3}{\Delta \Vdash \hat{\alpha} \approx \rho_1 \dashv \Theta_3, \hat{\alpha} : \omega_1 = \rho_2, \Theta_2}$$

$$\frac{\text{A-U-KVARL-LO-TT} \quad \Delta_1, \Delta_2 \Vdash^{\text{mv}} \hat{\alpha} : \omega_1 \rightsquigarrow \Theta \quad \Delta[\{\Theta\}] \vdash_{\hat{\alpha}}^{\text{pr}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \{\Theta_2, \hat{\alpha} : \omega_1, \Theta_3\}, \Theta_4 \quad \Theta_1, \{\Theta_2\} \Vdash^{\text{ela}} \rho_2 : \omega_2 \quad \Theta_1, \{\Theta_2\} \Vdash [\Theta_1, \Theta_2] \omega_1 \approx \omega_2 \dashv \Theta_5, \{\Theta_6\}}{\Delta[\{\Delta_1, \hat{\alpha} : \omega_1, \Delta_2\}] \Vdash \hat{\alpha} \approx \rho_1 \dashv \Theta_5, \{\Theta_6, \hat{\alpha} : \omega_1 = \rho_2, \Theta_3\}, \Theta_4}$$

$\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta$ (Promotion)

$$\frac{\text{A-PR-TVAR}}{\Delta[a][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\text{pr}} a \rightsquigarrow a \dashv \Delta[a][\hat{\alpha}]} \quad \frac{\text{A-PR-KUVARR-TT} \quad \Delta \vdash_{\hat{\alpha}}^{\text{pr}} [\Delta] \rho \rightsquigarrow \rho_1 \dashv \Theta[\hat{\alpha}][\hat{\beta} : \rho]}{\Delta[\hat{\alpha}][\hat{\beta} : \rho] \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta}_1 \dashv \Theta[\hat{\beta}_1 : \rho_1, \hat{\alpha}][\hat{\beta} : \rho = \hat{\beta}_1]}$$

$\Delta_1 \Vdash^{\text{mv}} \Delta_2 \rightsquigarrow \Theta$ (Moving)

$$\frac{\text{A-MV-EMPTY}}{\bullet \Vdash^{\text{mv}} \Delta \rightsquigarrow \Delta} \quad \frac{\text{A-MV-KUVAR} \quad \text{var}(\omega) \# \text{dom}(\Delta_2) \quad \Delta_1 \Vdash^{\text{mv}} \Delta_2 \rightsquigarrow \Theta}{\hat{\alpha} : \omega, \Delta_1 \Vdash^{\text{mv}} \Delta_2 \rightsquigarrow \hat{\alpha} : \omega, \Theta}$$

$$\frac{\text{A-MV-KUVARM} \quad \neg(\text{var}(\omega) \# \text{dom}(\Delta_2)) \quad \Delta_1 \Vdash^{\text{mv}} \Delta_2, \hat{\alpha} : \omega \rightsquigarrow \Theta}{\hat{\alpha} : \omega, \Delta_1 \Vdash^{\text{mv}} \Delta \rightsquigarrow \Theta}$$

Figure 7.15: Selected rules for unification, promotion, and moving in PolyKinds

Promotion now has a new failure mode: it cannot move proper quantified type variables. In rule [A-PR-TVAR](#), the variable a must be to the left of $\hat{\alpha}$.

Unfortunately, now we cannot easily tell whether promoting is terminating. In particular, the convergence of promotion in Haskell98 is built upon the obvious fact that the size of the kind being promoted always gets smaller from the conclusion to the hypothesis. However, rule [A-PR-KUVAR-TT](#) breaks this invariant, as the judgment recurs into the kinds of unification variables, and the size of the kinds may be larger than the unification variables. As shown in Section 7.7.5, we prove that promotion is terminating.

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 $\Delta_1 \dashv^{\text{mv}} \Delta_2 \leadsto \Theta$ 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.

3508 7.7.5 TERMINATION

3509 Now the challenge is to prove that our unification algorithm terminates, which relies on the
 3510 convergence of the promotion algorithm. Next, we first discuss the termination of unifi-
 3511 cation, and then move to the more complicated proof for promotion. Let $\langle \Delta \rangle$ denote the
 3512 number of unsolved unification variables in Δ .

3513 **Lemma 7.6** (Promotion Preserves $\langle \Delta \rangle$). *If $\Delta \vdash_{\alpha}^{\text{pr}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta$, then $\langle \Delta \rangle = \langle \Theta \rangle$.*

3514 **Lemma 7.7** (Unification Makes Progress). *If $\Delta \Vdash \omega_1 \approx \omega_2 \dashv \Theta$, then either $\Theta = \Delta$, or*
 3515 *$\langle \Theta \rangle < \langle \Delta \rangle$.*

3516 Now we measure unification $\Delta \Vdash \omega_1 \approx \omega_2 \dashv \Theta$ using the lexicographic order of the
 3517 pair $(\langle \Delta \rangle, |\omega_1|)$, where $|\omega_1|$ computes the standard size of ω_1 . We prove the pair always gets
 3518 smaller from the conclusion to the hypothesis. Formally, assuming promotion terminates,
 3519 we have

3520 **Theorem 7.8** (Unification Terminates). *Given a context Δ ok, and kinds ρ_1 and ρ_2 , where*
 3521 *$[\Delta]\rho_1 = \rho_1$, and $[\Delta]\rho_2 = \rho_2$, it is decidable whether there exists Θ such that $\Delta \Vdash \rho_1 \approx \rho_2 \dashv$*
 3522 *Θ .*

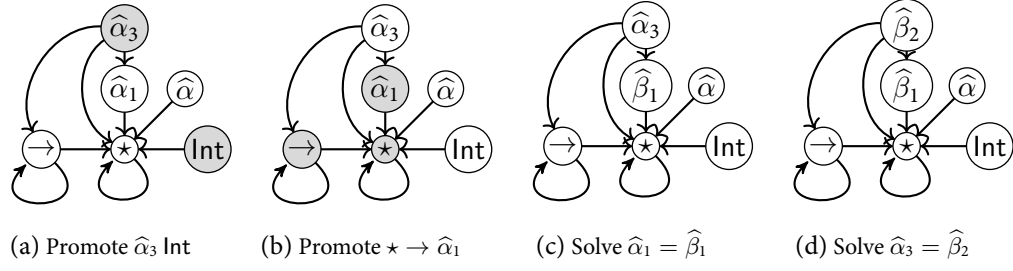
3523 We are not yet done, since Theorem 7.8 depends on the convergence of promotion. As
 3524 observed in rule [A-PR-KUVARR](#), the size of the type being promoted increases from the con-
 3525 clusion to the hypothesis. Worse, the context never decreases. How do we prove promotion
 3526 terminates? The crucial observation for rule [A-PR-KUVARR](#) is that, when we move from the
 3527 conclusion to the hypothesis, we also move from a unification variable to its kind. Since
 3528 the kind is well-formed under the prefix context of the variable, we are somehow moving
 3529 leftward in the context.

3530 To formalize the observation, we define the *dependency graph* of a context.

3531 **Definition 21** (Dependency Graph). The dependency graph of a context Δ is a *directed* graph
 3532 where:

- 3533 1. Nodes are all type variables and unsolved unification variables of Δ , and the terminal
 3534 symbols \star , \rightarrow and Int .
- 3535 2. Edges indicate the dependency from a type to its substituted kind. For example, if
 3536 $\hat{\alpha} : \omega$, then there is a directed edge from $\hat{\alpha}$ to all the nodes appearing in $[\Delta]\omega$.

3537 As an illustration, consider the context $\Delta = \hat{\alpha} : \star, \hat{\alpha}_1 : \star, \hat{\alpha}_2 : \star = \hat{\alpha}_1, \hat{\alpha}_3 : \star \rightarrow \hat{\alpha}_2$,
 3538 whose dependency graph is given in Figure 7.16a (the reader is advised to ignore the color



$$\begin{aligned}
 \Theta_1 &= \hat{\beta}_1 : \star, \hat{\alpha} : \star, \hat{\alpha}_1 : \star = \hat{\beta}_1, \hat{\alpha}_2 : \star = \hat{\alpha}_1, \hat{\alpha}_3 : \star \rightarrow \hat{\alpha}_2 \\
 \Theta_2 &= \hat{\beta}_1 : \star, \hat{\beta}_2 : \star \rightarrow \hat{\beta}_1, \hat{\alpha} : \star, \hat{\alpha}_1 : \star = \hat{\beta}_1, \hat{\alpha}_2 : \star = \hat{\alpha}_1, \hat{\alpha}_3 : \star \rightarrow \hat{\alpha}_2 = \hat{\beta}_2 \\
 &\dots \frac{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \star \rightsquigarrow \star \dashv \Delta}{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\alpha}_1 \rightsquigarrow \hat{\beta}_1 \dashv \Theta_1} \text{A-PR-KUVarR} \quad \boxed{7.16c} \\
 &\frac{\boxed{7.16b} \quad \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \star \rightarrow \hat{\alpha}_1 \rightsquigarrow \star \rightarrow \hat{\beta}_1 \dashv \Theta_1}{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\alpha}_3 \rightsquigarrow \hat{\beta}_2 \dashv \Theta_2} \text{A-PR-APP} \quad \boxed{7.16d} \\
 &\frac{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\alpha}_3 \rightsquigarrow \hat{\beta}_2 \dashv \Theta_2}{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\alpha}_3 \text{ Int} \rightsquigarrow \hat{\beta}_2 \text{ Int} \dashv \Theta_2} \text{A-PR-APP} \quad \dots
 \end{aligned}$$

Figure 7.16: Example of dependency graph

for now). There are several notable properties. First, as long as the context is well-formed, the graph is *acyclic* except for the self-loop of \star and \rightarrow . Second, solved unification variables never appear in the graph. The kind of $\hat{\alpha}_3$ depends on $\hat{\alpha}_2$, which is already solved by $\hat{\alpha}_1$, so the dependency goes from $\hat{\alpha}_3$ to $\hat{\alpha}_1$.

Now let us consider how promotion works in terms of the dependency graph, by trying 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 given at the bottom of Figure 7.16. We omit some details via (\dots) as promoting constants (\star, \rightarrow and Int) is trivial. At the top of Figure 7.16 we give the dependency graph at certain points in the derivation, where the part being promoted is highlighted in gray. At the beginning we are at Figure 7.16a. 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 7.16b). 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 7.16c). 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 7.16d). Going back to unification, we solve $\hat{\alpha} = \hat{\beta}_2 \text{ Int}$.

We have several key observations. First, when we move from Figure 7.16a to Figure 7.16b 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 7.16c and Figure 7.16d), 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-KUVR 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, as the declarative system lacks principal kinds. 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.

```

3578   data Eq ::  $\forall k. k \rightarrow k \rightarrow \star$ 
3579   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*. To properly state the guarantee, we first define the notion of *kind instantiation* and *partial order of kinds*:

Definition 22 (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 $\omega_1[\phi_1 \mapsto \overline{\rho}_1][\phi_2 \mapsto \overline{\rho}_2] = \omega$ for some $\overline{\rho}_1$ and $\overline{\rho}_2$.

The relation is embedded in $\Sigma \vdash^{\text{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2$ (Figure 7.9), where we ignore μ_1 and μ_2 .

Definition 23 (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 $\Sigma \vdash \eta_1 <: \omega$.

To understand the definition, consider that if the program type-checks under $T : \eta_2$, then it must type-check under $T : \eta_1$, as $T : \eta_1$ can be instantiated to all monokinds that $T : \eta_2$ is used at.

Now we lift the definition of \Vdash^{grp} to be the generalized result of kinds and contexts. And we are ready now to formally prove our guarantee of principality:

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 $[\Omega]\Omega \vdash^{\text{grp}} \text{rec } \overline{\tau}_i^i \rightsquigarrow \overline{\eta}_i^i ; \overline{\Psi}_i^i$ holds, we have $[\Omega]\Omega \vdash [\Omega]\eta_i \preceq \eta_i'$.*

This result echoes the result in the term-level type inference algorithm for Haskell ([Vytiniotis et al. 2011, Section 6.5]): our algorithm does not infer every kind acceptable by the declarative system, but the kinds it does infer are always the best (principal) ones.

7.8 LANGUAGE EXTENSIONS

We have seen that the PolyKinds system incorporates many features and enjoys desirable properties. In this section, we discuss how the PolyKinds system can be extended with more related language features. Appendix C.1 contains a few more, less impactful extensions.

7.8.1 HIGHER-RANK POLYMORPHISM

The system can be extended naturally to support higher-rank polymorphism [Dunfield and Krishnaswami 2013; Peyton Jones et al. 2007]. With higher-rank polymorphism, every type can have a polymorphic kind. For example, data constructor declarations become $\forall\phi. D \overline{\sigma}_i^i$ instead of $\forall\phi. D \overline{\tau}_i^i$.

Unfortunately, higher-rank polymorphism breaks principality. Consider:

```
data Q1 ::  $\forall k_1 k_2. k_1 \rightarrow \star$ 
data Q2 ::  $(\forall (k_1 : \star) (k_2 : k_1). k_1 \rightarrow \star) \rightarrow \star$ 
```

First, we modify the definition of partial order of kinds (Definition 23) to state that one kind is more general than another if it can be instantiated to all *polykinds* that the other kind can be instantiated to. Now consider the kind of *Q1*, which under the algorithm is generalized to $\forall\{k3 :: \star\} (k1 :: \star) (k2 :: k3). k1 \rightarrow \star$. In Theorem 7.11, we guarantee that this kind is a principal kind as it can be instantiated to all monokinds that other possible kinds for *Q1*, e.g., $\forall(k1 :: \star) (k2 :: k1). k1 \rightarrow \star$, can be instantiated to. However, under the new definition, $\forall\{k3 :: \star\} (k1 :: \star) (k2 :: k3). k1 \rightarrow \star$ is no longer more general than $\forall(k1 :: \star) (k2 :: k1). k1 \rightarrow \star$, as there is no way to instantiate the former to the latter. To see why we need to modify the definition at all, consider the rank-2 kind of *Q2*, which expects exactly an argument of kind $\forall(k1 :: \star) (k2 :: k1). k1 \rightarrow \star$.

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]. The lack of lambda type-level abstractions largely simplifies the kind inference algorithm (e.g., it allows us to decompose type applications as in rule A-U-APP). However, it also 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

```

3650     data G a where
3651         MkG :: G Bool

```

3652 to be

```

3653     data G a = (a ~ Bool) ⇒ MkG

```

3654 where \sim describes an equality constraint. For our purposes of doing kind inference, these
3655 equality constraints are uninteresting: the \sim operator simply relates two types of the same
3656 kind and can be processed as any polykinded type constructor would be. Modeling con-
3657 straints to the left of a \Rightarrow similarly would add a little clutter to our rules, but would offer no
3658 real challenges.

3659 The unexpected simplicity of adding GADTs to our system arises from a key fact: we do
3660 not ever allow *pattern-matching*. A GADT pattern-match brings a local equality assumption
3661 into scope, which would influence the unification algorithm. However, as pattern matching
3662 does not happen in the context of datatype declarations, we avoid this wrinkle here.

3663 SYNTAX The implementation of GADTs in GHC has an unusual syntax:

```

3664     data G a where
3665         MkG :: a → G Int

```

3666 The surprising aspect of this syntax is that the two *as* above are *different*: the *a* in the header
3667 is unrelated to the *a* in the data constructor. This seemingly inconsequential design choice
3668 makes kind inference for GADTs very challenging, as constructors have no way to refer back
3669 to the datatype parameters. Given that this aspect of GADTs is a quirk of GHC’s design—and
3670 is not repeated in other languages that support GADTs—we remark here that it is odd and
3671 perhaps should be remedied. We will return back to this discussion in Section 9.4.

3672 7.8.3 TYPE FAMILIES

3673 Type families [Chakravarty et al. 2005] are, effectively, type-level functions. Kind inference
3674 of type families thus can be designed much like type inference for ordinary functions. How-
3675 ever, as they can have dependency, the complications we describe in this paper would arise
3676 here, too. In particular, unification would have to be kind-directed, as we have described.
3677 The current syntax for closed type families [Eisenberg et al. 2014] shares the same scoping
3678 problem as the syntax for GADTs, so our arguments above apply to closed type families
3679 equally.

3680 The challenge with type families is that they indeed do pattern-matching, and thus (in
3681 concert with GADTs) can bring local equalities into scope. A full analysis of the ramifications
3682 here is beyond the scope of this paper, but we believe the literature on type inference in the
3683 presence of local equalities would be helpful. Principal among these is the work of Vytiniotis
3684 et al. [2011], but Gundry [2013] and Eisenberg [2016] also approach this problem in the
3685 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 skolemisation, examples like $\forall a. \text{Int} \rightarrow \text{Int} <: \text{Int} \rightarrow \forall a. a$ are allowed. We believe deep skolemisation is compatible with our subtyping definition (i.e., our subtyping can naturally integrate deep skolemisation). Note, though, recently GHC has resiled from deep skolemisation [Peyton Jones 2020], as it is believed that the cost of deep skolemisation is large compared to its benefit.

Dunfield and Krishnaswami [2013] build a simple and concise algorithm for higher-rank polymorphism based on traditional bidirectional type checking. They deal with the same language of Peyton Jones et al. [2007], except they do not have **let** expressions nor generalization (though it is discussed in design variations). Built upon some of these techniques, Dunfield and Krishnaswami [2019] extend the system to a much richer type language that includes existentials, indexed types, and equations over type variables.

IMPREDICATIVE HIGHER-RANK TYPE INFERENCE. While our work focuses on predicative higher-rank types, there are also a lot of work on type inference for *impredicative* higher-rank types. Many of these work replies on new forms of types. ML^F [Le Botlan and Rémy 2003, 2009; Rémy and Yakobowski 2008] generalizes ML with first-class polymorphism. ML^F introduces a new type of bounded quantification (either rigid or flexible) for polymorphic types so that instantiation of polymorphic bindings is delayed until a principal type is found. higher-rank types. The HML system [Leijen 2009] is proposed as a simplification and restric-

tion of ML^F . HML only uses flexible types, which simplifies the type inference algorithm, but retains many interesting properties and features.

The FPH system [Vytiniotis et al. 2008] introduces boxy monotypes into System F types. One critique of boxy type inference is that the impredicativity is deeply hidden in the algorithmic type inference rules, which makes it hard to understand the interaction between its predicative constraints and impredicative instantiations [Rémy 2005].

Recently, Serrano et al. [2020, 2018] exploit impredicative instantiations of type variables that appears under a type constructor (i.e., type variables are *guarded*) Serrano et al. [2018] is strictly subsumed by Serrano et al. [2020]. The former distinguishes variables using three *sorts*, so that certain sorts of variables can be instantiated with higher-rank polymorphic types, while the latter inspects the function arguments and assigns impredicative instantiations before monomorphic ones.

8.2 BIDIRECTIONAL TYPE CHECKING

Bidirectional type checking was popularized by the work of Pierce and Turner [2000]. It has since been applied to many type systems with advanced features. The alternative application mode introduced in Chapter 3 enables a variant of bidirectional type checking. There are many other efforts to refine bidirectional type checking.

Colored local type inference [Odersky et al. 2001] allows partial type information to be propagated, by distinguishing inherited types (known from the context) and synthesized types (inferred from terms). A similar distinction is achieved in Dunfield and Krishnaswami [2013] by manipulating type variables.

Tridirectional type checking [Dunfield and Pfenning 2004] is based on bidirectional type checking and has a rich set of property types including intersections, unions and quantified dependent types, but without parametric polymorphism. Tridirectional type checking has a new direction for supporting type checking unions and existential quantification.

Greedy bidirectional polymorphism [Dunfield 2009] adopts a greedy idea from Cardelli [1993] on bidirectional type checking with higher-rank types, where type variables in instantiations are determined by their first constraint. In this way, they support some uses of impredicative polymorphism. However, the greediness also makes many obvious programs rejected.

A detailed survey of the development of bidirectional type checking is given by Dunfield and Krishnaswami [2020], which collect and explain the design principles of bidirectional type checking, and summarize past research related to bidirectional type checking.

8.3 GRADUAL TYPING

The seminal paper by Siek and Taha [2006] is the first to propose gradual typing, which enables programmers to mix static and dynamic typing in a program by providing a mechanism to control which parts of a program are statically checked. The original proposal extends the simply typed lambda calculus by introducing the unknown type ? and replacing type equality with type consistency. Casts are introduced to mediate between statically and dynamically typed code. Later Siek and Taha [2007] incorporated gradual typing into a simple object oriented language, and showed that subtyping and consistency are orthogonal – an insight that partly inspired our work on GPC. We show that subtyping and consistency are orthogonal in a much richer type system with higher-rank polymorphism. Siek et al. [2009] explores the design space of different dynamic semantics for simply typed lambda calculus with casts and unknown types. In the light of the ever-growing popularity of gradual typing, and its somewhat murky theoretical foundations, Siek et al. [2015] felt the urge to have a complete formal characterization of what it means to be gradually typed. They proposed a set of criteria that provides important guidelines for designers of gradually typed languages. Cimini and Siek [2016] introduced the *Gradualizer*, a general methodology for generating gradual type systems from static type systems. Later they also develop an algorithm to generate dynamic semantics [Cimini and Siek 2017]. Garcia et al. [2016] introduced the AGT approach based on abstract interpretation. As we discussed, none of these approaches instructed us how to define consistent subtyping for polymorphic types.

There is some work on integrating gradual typing with rich type disciplines. Bañados Schwerter et al. [2014] establish a framework to combine gradual typing and effects, with which a static effect system can be transformed to a dynamic effect system or any intermediate blend. Jafery and Dunfield [2017] present a type system with *gradual sums*, which combines refinement and imprecision. We have discussed the interesting definition of *directed consistency* in Section 4.2. Castagna and Lanvin [2017] develop a gradual type system with intersection and union types, with consistent subtyping defined by following the idea of Garcia et al. [2016]. Eremondi et al. [2019] develop a gradual dependently-typed language, where compile-time normalization and run-time execution are distinguished to account for nontermination and failure. TypeScript [Bierman et al. 2014] has a distinguished dynamic type, written `any`, whose fundamental feature is that any type can be implicitly converted to and from `any`. Our treatment of the unknown type in Figure 4.6 is similar to their treatment of `any`. However, their type system does not have polymorphic types. Also, unlike our consistent subtyping which inserts runtime casts, in TypeScript, type information is erased after compilation so there are no runtime casts, which makes runtime type errors possible.

3781 8.4 GRADUAL TYPE SYSTEMS WITH EXPLICIT POLYMORPHISM

3782 Morris [1973] dynamically enforces parametric polymorphism and uses *sealing* functions as
 3783 the dynamic type mechanism. More recent works on integrating gradual typing with para-
 3784 metric polymorphism include the dynamic type of Abadi et al. [1995] and the *Sage* language
 3785 of Gronski et al. [2006]. None of these has carefully studied the interaction between statically
 3786 and dynamically typed code.

3787 Ahmed et al. [2009] proposed λB that extends the blame calculus [Wadler and Findler
 3788 2009] to incorporate polymorphism. The key novelty of their work is to use dynamic seal-
 3789 ing to enforce parametricity. As such, they end up with a sophisticated dynamic seman-
 3790 tics. Later, Ahmed et al. [2017] prove that with more restrictions, λB satisfies parametricity.
 3791 Compared to their work, our GPC type system can catch more errors earlier since, as we
 3792 argued, their notion of *compatibility* is too permissive. For example, the following is rejected
 3793 (more precisely, the corresponding source program never gets elaborated) by our type sys-
 3794 tem:

$$(\lambda x : ?. x + 1) : \forall a. a \rightarrow a \rightsquigarrow \langle ? \rightarrow \text{Int} \hookrightarrow \forall a. a \rightarrow a \rangle (\lambda x : ?. x + 1)$$

3795 while the type system of λB would accept the translation, though at runtime, the program
 3796 would result in a cast error as it violates parametricity. We emphasize that it is the combina-
 3797 tion of our powerful type system together with the powerful dynamic semantics of λB that
 3798 makes it possible to have implicit higher-rank polymorphism in a gradually typed setting.
 3799 Devriese et al. [2017] proved that embedding of System F terms into λB is not fully abstract.
 3800 Igarashi et al. [2017] also studied integrating gradual typing with parametric polymorphism.
 3801 They proposed System F_G , a gradually typed extension of System F, and System F_C , a new
 3802 polymorphic blame calculus. As has been discussed extensively, their definition of type con-
 3803 sistency does not apply to our setting (implicit polymorphism). All of these approaches mix
 3804 consistency with subtyping to some extent, which we argue should be orthogonal. On a side
 3805 note, it seems that our calculus can also be safely translated to System F_C . However we do
 3806 not understand all the tradeoffs involved in the choice between λB and System F_C as a target.

3807 Recently, Toro et al. [2019] applied AGT to designing a gradual language with explicit
 3808 parametric polymorphism, claiming that graduality and parametricity are inherently incom-
 3809 patible. However, later New et al. [2019] show that by modifying System F's syntax to make
 3810 the sealing visible, both graduality and parametricity can be achieved.

3811 8.5 GRADUAL TYPE INFERENCE

3812 Siek and Vachharajani [2008] studied unification-based type inference for gradual typing,
 3813 where they show why three straightforward approaches fail to meet their design goals. One
 3814 of their main observations is that simply ignoring dynamic types during unification does
 3815 not work. Therefore, their type system assigns unknown types to type variables and infers
 3816 gradual types, which results in a complicated type system and inference algorithm. In our
 3817 algorithm presented in Chapter 5, comparisons between existential variables and unknown
 3818 types are emphasized by the distinction between static existential variables and gradual ex-
 3819 istential variables. By syntactically refining unsolved gradual existential variables with un-
 3820 known types, we gain a similar effect as assigning unknown types, while keeping the algo-
 3821 rithm relatively simple. Garcia and Cimini [2015] presented a new approach where gradual
 3822 type inference only produces static types, which is adopted in our type system. They also
 3823 deal with let-polymorphism (rank 1 types). They proposed the distinction between static
 3824 and gradual type parameters, which inspired our extension to restore the dynamic gradual
 3825 guarantee. Although those existing works all involve gradual types and inference, none of
 3826 these works deal with higher-rank implicit polymorphism.

3827 8.6 HASKELL AND GHC

3828 THE GLASGOW HASKELL COMPILER. The systems we present in Chapter 7 are inspired by
 3829 the algorithms implemented in GHC. However, our goal in the design of these systems is to
 3830 produce a sound and (nearly) complete pair of specification and implementation, not simply
 3831 to faithfully record what is implemented. We have identified ways that the GHC implemen-
 3832 tation can improve in the future. For example, GHC quantifies over local scopes as *specified*
 3833 where we believe they should be *inferred*; and the tight connection in our system between
 3834 unification and promotion may improve upon GHC’s approach, which separates the two.
 3835 The details of the relationship between our work and GHC (including a myriad of ways our
 3836 design choices differ in small ways from GHC’s) appear in Appendix C.2.

3837 TYPE INFERENCE IN HASKELL. Type inference in Haskell is inspired by Damas and Milner
 3838 [1982] and Pottier and Rémy [2005], extended with various type features, including higher
 3839 rank polymorphism [Peyton Jones et al. 2007] and local assumptions [Schrijvers et al. 2009;
 3840 Simonet and Pottier 2007; Vytiniotis et al. 2011], among others. However, none of these
 3841 works describe an inference algorithm for datatypes, nor do they formalize type variables of
 3842 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, following Jones [1999].

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.

3875 Ziliani and Sozeau [2015] present the higher-order unification algorithm for CIC, the base
 3876 logic of Coq. They favor syntactic equality by trying first-order unification, as they argue
 3877 the first-order solution gives the most *natural* solution. However, they omit a correctness
 3878 proof for their algorithm. Coen [2004] also considers first-order unification, but only the
 3879 soundness lemma is proved. Different from their systems, our system is based on the novel
 3880 promotion judgment, and correctness including soundness and termination is proved.

3881 The technique of *suspended substitutions* [Eisenberg 2016; Gundry and McBride 2013]
 3882 is widely adopted in unification algorithms. Our system provides a design alternative, our
 3883 *quantification check*. Choosing between suspended substitutions and the quantification check
 3884 is a user-facing language design decision, as suspended substitutions can accept some more
 3885 programs. The quantification check means that the kind of a locally quantified variable a
 3886 must be fully determined in a 's scope; it may *not* be influenced by usage sites of the con-
 3887 struct that depends on a . Suspended substitutions relax this restriction. We conjecture that
 3888 suspended substitutions can yield a complete algorithm. However, that mechanism is com-
 3889 plex. Moreover, unification based on suspended substitutions is only decidable for the pat-
 3890 tern fragment. Our system, in contrast, avoids all the complication introduced by suspended
 3891 substitutions through its quantification check. Our unification terminates for all inputs, pre-
 3892 serving backward compatibility to Hindley-Milner-style inference. Although we reject the
 3893 definition of T (Section 7.7.2), we can solve more constraints outside the pattern fragment.
 3894 We conjecture that those constraints are much more common than definitions like T . Sus-
 3895 pended substitutions often come with a *pruning* process [Abel and Pientka 2011], which
 3896 produces a valid solution before solving a unification variable. Our promotion process has a
 3897 similar effect.

3898 HOMOGENEOUS KIND-PRESERVING UNIFICATION. Jones [1995] proposed a homogeneous
 3899 kind-preserving unification between two types. Kinds κ are defined only as \star or $\kappa_1 \rightarrow \kappa_2$.
 3900 As the kind system is much simpler, kind-preserving unification \sim_κ is simply subscripted
 3901 by the kind, and working out the kinds is straightforward. Our unification subsumes Jones's
 3902 algorithm.

3903 CONTEXT EXTENSION. Our approach of recording unification variables and their solutions
 3904 in the contexts is inspired by Gundry et al. [2010] and Dunfield and Krishnaswami [2013].
 3905 Gundry and McBride [2013] applied the approach to unification in dependent types, where
 3906 the context also records constraints; constraints also appear in context in Eisenberg [2016].
 3907 Further, in PolyKinds, we extend the context extension approach with local scopes, support-
 3908 ing groups of order-insensitive variables.

9 SUMMARY AND FUTURE DIRECTIONS

3910 In summary, this dissertation has pushed the research on predicative implicit higher-rank
 3911 polymorphism further, and we believe that contributions in this dissertation can be used to
 3912 guide the continued evolution of (functional) programming language design and implemen-
 3913 tations. Specifically, with the new bidirectional type checking algorithm using the applica-
 3914 tion mode, we were able to type-check programs that traditional type inference algorithms
 3915 cannot, and thus provide new insights for inference algorithm design with bidirectional type
 3916 checking. With the integration of higher-rank polymorphism and gradual typing, we pro-
 3917 vided a step forward in gradualizing modern functional programming languages like Haskell.
 3918 Moreover, the work on *type promotion* simplified type inference algorithms with tricky de-
 3919 pendency and scoping issues, and the kind inference for datatypes presented a first known,
 3920 detailed account of datatypes, which can serve as a guide for future development of datatypes.

3921 In this section we discuss some future directions we would like to pursue.

3922 9.1 DEPENDENT TYPE SYSTEMS WITH APPLICATION MODE

3923 The application mode is possibly applicable to systems with advanced features, where type
 3924 inference is sophisticated or even undecidable. One promising application is, for instance,
 3925 dependent type systems [Xi and Pfenning 1999]. Type systems with dependent types usually
 3926 unify the syntax for terms and types, with a single lambda abstraction generalizing both type
 3927 and lambda abstractions. Unfortunately, this means that the **let** desugar is not valid in those
 3928 systems. As a concrete example, consider desugaring the expression **let** $a = \text{Int}$ **in** $\lambda x :$
 3929 $a. x + 1$ into $(\lambda a. \lambda x : a. x + 1) \text{Int}$, which is ill-typed because the type of x in the abstraction
 3930 body is a and not Int .

3931 Because **let** cannot be encoded, declarations cannot be encoded either. Modeling decla-
 3932 rations in dependently typed languages is a subtle matter, and normally requires some addi-
 3933 tional complexity [Severi and Poll 1994].

We believe that the same technique presented in Section 3.5.3 can be adapted into a de-
 pendently typed language to enable a **let** encoding. In a dependent type system with uni-
 fied 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](#).

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

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

3934 With such rules it would be possible to handle declarations easily in dependent type sys-
3935 tems.

3936 9.2 TYPE INFERENCE FOR INTERSECTION TYPE SYSTEMS

3937 Another type system that could possibly benefit from the application mode is intersection
3938 type systems [Coppo et al. 1979; Pottinger 1980; Salle 1978]. In particular, we consider in-
3939 tersection type systems with an explicit *merge operator* [Dunfield 2014]. In such a system,
3940 we can construct terms of an intersection type, like $1, , \text{true}$ of type $\text{Int} \& \text{Bool}$. Thanks to
3941 *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
3942 Bool . Calculi with *disjoint intersection types* [Alpuim et al. 2017; Bi et al. 2019; Oliveira et al.
3943 2016] incorporate a *coherent* merge operator. In such calculi the merge operator can merge
3944 two terms with *arbitrary* types as long as their types are disjoint; disjointness conflicts are
3945 reported as type-errors. As illustrated by Xie et al. [2020], the expressive power of disjoint
3946 intersection types can encode diverse programming language features, promising an econ-
3947 omy of theory and implementation.

3948 Disjoint intersection types also pose challenges to type inference. Supposing that we have
3949 $\text{succ} : \text{Int} \rightarrow \text{Int}$ and $\text{not} : \text{Bool} \rightarrow \text{Bool}$, consider the following term:

3950 $(\text{succ} , , \text{not}) 3$

3951 We expect the expression to type-check, as according to subtyping, the term $(\text{succ} , , \text{not})$
3952 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
3953 typing to automatically pick succ and apply it to 3. To this end, we need to push the type
3954 information of the argument (3) into the function $(\text{succ} , , \text{not})$.

3955 Future work is required to explore how well the application mode can be used for type
3956 inference in intersection type systems, and whether it can be integrated with the distributivity
3957 subtyping rules of intersection types [Bi et al. 2019].

9.3 GRADUALIZING TYPE CLASSES

In Section 4.1.2, we discussed about gradualizing modern functional programming languages like Haskell. One of its core abstraction features in Haskell is *type classes*. Type classes [Wadler and Blott 1989] were initially introduced in Haskell to make ad-hoc overloading less ad-hoc, and since then have been adopted in many languages including Mercury [Henderson et al. 1996], Coq [Sozeau and Oury 2008], PureScript [Freeman 2017], and Lean [de Moura et al. 2015]. An interesting future direction then is to gradualizing type classes.

Consider again the example used in Section 4.1.2:

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

While $f : \forall a. a \rightarrow a$ is of course a valid type annotation, it unfortunately rules out many valid arguments that may have type class constraints in their types, e.g.,

```
show      :: Show a => a -> String
-- rejected
(\f :: ∀a. a -> a. (f 1, f 'a')) show
```

With gradual typing, if we annotation f with the the unknown type $?$, we expect that the following expression can type-check.

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

However, a nontrivial challenge in gradualizing type classes is that the dynamic semantics of type classes is not expressed directly but rather by type-directed elaboration into a simpler language without type classes. Thus the dynamic semantics of type classes is given indirectly as the dynamic semantics of their elaborated forms. Consider `show` as an example. The *dictionary-passing* elaboration of type-classes translates the type of `show` into the following one, supposing `ShowD` is the dictionary type of the type class `show`.

```
show :: ShowD a -> a -> String
```

Now with the unknown type, we cannot predict how to elaborate the original expression. In particular, if f is applied to `show`, it means that f needs to be elaborated into a function that actually takes two arguments, first the dictionary and then the argument.

```
(\f. (f showInt 1, f showChar 'a')) show
```

This kind of uncertainty in elaboration brings extra complexity and may interact with explicit casts in the target blame calculi.

3988 9.4 GENERALIZED ALGEBRAIC DATATYPES (GADTs)

3989 A natural extension of PolyKinds is to include GADTs. We have briefly discussed GADTs in
 3990 Section 7.8.2. In particular, we are interested in finding the right formalization of GADTs.

3991 Haskell’s *syntax* for GADT declarations is quite troublesome. Consider these examples:

3992 `data R a where`
 3993 `MkR :: b → R b`

3994 `data S a where`
 3995 `MkS :: S b`

3996 `data T a where`
 3997 `MkT :: ∀(k :: ★) (b :: k). T b`

3998 In GHC’s implementation of GADTs, any variables declared in the header (between `data` and
 3999 `where`) *do not scope*. In all the examples above, the type variable `a` does not scope over the
 4000 constructor declarations. This is why we have written the variable `b` in those types, to make
 4001 it clear that `b` is distinct from `a`. We could have written `a`—it would still be a distinct `a` from
 4002 that in the header—but it would be more confusing.

4003 The question is: how do we determine the kind of the parameter to the datatype? One
 4004 possibility is to look only in the header. In all cases above, we would infer no constraints and
 4005 would give each type a kind of $\forall(k :: \star). k \rightarrow \star$. This is unfortunate, as it would make `R`
 4006 a kind-indexed GADT: the `MkR` constructor would carry a proof that the kind of its type
 4007 parameter is \star . This, in turn, wreaks havoc with type inference, as it is hard to infer the result
 4008 type of a pattern-match against a GADT Vytiniotis et al. [2011].

4009 Furthermore, this approach might accept *more* programs than the user wants. Consider
 4010 this definition:

4011 `data P a where`
 4012 `MkP1 :: b → P b`
 4013 `MkP2 :: f a → P f`

4014 Does the user want a kind-indexed GADT, noting that `b` and `f` have different kinds? Or
 4015 would the user want this rejected? If we make the fully general kind $\forall k. k \rightarrow \star$ for `P`, this
 4016 would be accepted, perhaps surprising users.

4017 It thus seems we wish to look at the data constructors when inferring the kind of the
 4018 datatype. The challenge in looking at data constructors is that their variables are *locally*

4019 bound. In MkR and MkS , we implicitly quantify over b . In MkR , we discover that $b :: \star$, and
 4020 thus that R must have kind $\star \rightarrow \star$. In MkS , we find no constraints on b 's kind, and thus no
 4021 constraints on S 's argument's kind, and so we can generalize to get $S :: \forall (k :: \star). k \rightarrow \star$. Let
 4022 us now examine MkT : it explicitly brings k and b into scope. Thus, the argument to T has
 4023 *local* kind k . It would be impossible to unify the kind of T 's argument—call it $\hat{\alpha}$ —with k ,
 4024 because k would be bound to the *right* of $\hat{\alpha}$ in an inference context. Thus it seems we would
 4025 reject T .

4026 Our conclusion here is that the design of GADTs in GHC/Haskell is flawed: the type vari-
 4027 ables mentioned in the header should indeed scope over the constructors. This would mean
 4028 we could reject T : if the user wanted to explicitly make T polymorphically kinded, they
 4029 could do so right in the header. So one possible application of our work is to apply our
 4030 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 192]

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 194 and 195]

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 192]

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 192]

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint polymorphism. In *European Symposium on Programming (ESOP)*. [cited on page 198]

P. B. Andrews. 1971. Resolution in type Theory. *Journal of Symbolic Logic* 36 (1971), 414–432. [cited on page 149]

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 191]

- 4057 Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive dis-
4058 joint polymorphism for compositional programming. In *European Symposium on Pro-*
4059 *gramming (ESOP)*. [cited on page 198]
- 4060 Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In
4061 *Proceedings of the 28th European Conference on Object-Oriented Programming*. [cited on
4062 pages 8, 61, and 191]
- 4063 Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C#.
4064 In *Proceedings of the European Conference on Object-Oriented Programming*. [cited on
4065 page 61]
- 4066 Richard S. Bird and Lambert Meertens. 1998. Nested datatypes. In *LNCS 1422: Pro-*
4067 *ceedings of Mathematics of Program Construction*, Johan Jeuring (Ed.). Springer-Verlag,
4068 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)
4069 [online/BirdMeertens98Nested.pdf](http://www.cs.ox.ac.uk/people/richard.bird/online/BirdMeertens98Nested.pdf) [cited on page 150]
- 4070 Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Op-
4071 tional Types for Clojure. In *Programming Languages and Systems*. [cited on pages 8 and 61]
- 4072 Joachim Breitner, Richard A Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016.
4073 Safe zero-cost coercions for Haskell. *Journal of Functional Programming* 26 (2016). [cited
4074 on page 194]
- 4075 L. Cardelli. 1986. *A polymorphic lambda-calculus with Type:Type*. Technical Report 10. SRC.
4076 [cited on page 148]
- 4077 Luca Cardelli. 1993. *An implementation of FSub*. Technical Report. Research Report 97,
4078 Digital Equipment Corporation Systems Research Center. [cited on pages 89 and 190]
- 4079 Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection
4080 Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 41 (Aug. 2017), 28 pages. [cited on
4081 pages 61, 75, 99, and 191]
- 4082 Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated type
4083 synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Func-*
4084 *tional Programming* (Tallinn, Estonia) (ICFP '05). ACM, New York, NY, USA, 241–253.
4085 <https://doi.org/10.1145/1086365.1086397> [cited on page 184]
- 4086 Gang Chen. 2003. Coercive Subtyping for the Calculus of Constructions (POPL '03).
4087 10 pages. [cited on page 53]

- 4088 Alonzo Church. 1941. *The calculi of lambda-conversion*. Number 6. Princeton University
4089 Press. [cited on page 65]
- 4090 Matteo Cimini and Jeremy G. Siek. 2016. The Gradualizer: A Methodology and Algorithm
4091 for Generating Gradual Type Systems. In *Proceedings of the 43rd Symposium on Principles*
4092 *of Programming Languages*. [cited on pages 61, 75, 83, and 191]
- 4093 Matteo Cimini and Jeremy G. Siek. 2017. Automatically Generating the Dynamic Seman-
4094 tics of Gradually Typed Languages. In *Proceedings of the 44th Symposium on Principles of*
4095 *Programming Languages*. [cited on pages 61 and 191]
- 4096 Jesper Cockx, Dominique Devriese, and Frank Piessens. 2016. Unifiers as equivalences:
4097 proof-relevant unification of dependently typed data. In *Proceedings of the 21st ACM*
4098 *SIGPLAN International Conference on Functional Programming (Nara, Japan) (ICFP*
4099 *2016)*. ACM, New York, NY, USA, 270–283. [https://doi.org/10.1145/2951913.](https://doi.org/10.1145/2951913.2951917)
4100 2951917 [cited on page 194]
- 4101 Claudio Sacerdoti Coen. 2004. *Mathematical knowledge management and interactive theorem*
4102 *proving*. Ph.D. Dissertation. University of Bologna, 2004. Technical Report UBLCS 2004-
4103 5. [cited on page 195]
- 4104 Marco Comini, Ferruccio Damiani, and Samuel Vrech. 2008. On polymorphic recursion,
4105 type systems, and abstract interpretation. In *International Static Analysis Symposium*.
4106 Springer, 144–158. [cited on page 194]
- 4107 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Patrick Sallé. 1979. Functional charac-
4108 terization of some semantic equalities inside λ -calculus. In *International Colloquium on*
4109 *Automata, Languages, and Programming*. Springer, 133–146. [cited on page 198]
- 4110 Thierry Coquand. 1996. An algorithm for type-checking dependent types. *Science of Com-*
4111 *puter Programming* 26, 1-3 (1996), 167–177. [cited on page 24]
- 4112 Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin.
4113 1958. *Combinatory logic*. Vol. 1. North-Holland Amsterdam. [cited on page 65]
- 4114 Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs.
4115 In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Program-*
4116 *ming Languages (Albuquerque, New Mexico) (POPL '82)*. Association for Computing Ma-
4117 chinery, New York, NY, USA, 207–212. <https://doi.org/10.1145/582153.582176>
4118 [cited on pages 3, 4, 15, 18, 145, 148, and 193]

- 4119 Ferruccio Damiani. 2003. Rank 2 intersection types for local definitions and conditional
4120 expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 4
4121 (2003), 401–451. [cited on page 194]
- 4122 Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In
4123 *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Program-*
4124 *ming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 198–208.
4125 <https://doi.org/10.1145/351240.351259> [cited on pages 24 and 99]
- 4126 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von
4127 Raumer. 2015. The Lean theorem prover. (2015). [cited on page 199]
- 4128 Dominique Devriese, Marco Patrignani, and Frank Piessens. 2017. Parametricity versus the
4129 universal type. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 38.
4130 [cited on page 192]
- 4131 Joshua Dunfield. 2009. Greedy Bidirectional Polymorphism. In *Workshop on ML*. [cited on
4132 page 190]
- 4133 Joshua Dunfield. 2014. Elaborating intersection and union types. *Journal of Functional Pro-*
4134 *gramming (JFP)* 24, 2-3 (2014), 133–165. [cited on page 198]
- 4135 Jana Dunfield and Neel Krishnaswami. 2020. Bidirectional Typing. arXiv:1908.05839 [cs.PL]
4136 [cited on page 190]
- 4137 Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional
4138 Typechecking for Higher-Rank Polymorphism. In *Proceedings of the 18th ACM SIGPLAN*
4139 *International Conference on Functional Programming (Boston, Massachusetts, USA) (ICFP*
4140 *'13)*. Association for Computing Machinery, New York, NY, USA, 429–442. [https://](https://doi.org/10.1145/2500365.2500582)
4141 doi.org/10.1145/2500365.2500582 [cited on pages 3, 6, 7, 12, 23, 25, 28, 30, 61, 81,
4142 85, 95, 109, 114, 121, 125, 126, 130, 131, 132, 134, 154, 161, 182, 189, 190, and 195]
- 4143 Joshua Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and Complete Bidi-
4144 rectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed
4145 Types. *Proc. ACM Program. Lang.* 3, POPL, Article 9 (Jan. 2019), 28 pages. [https:](https://doi.org/10.1145/3290322)
4146 [//doi.org/10.1145/3290322](https://doi.org/10.1145/3290322) [cited on page 189]
- 4147 Joshua Dunfield and Frank Pfenning. 2004. Tridirectional Typechecking. *SIGPLAN Not.*
4148 39, 1 (Jan. 2004), 281–292. <https://doi.org/10.1145/982962.964025> [cited on
4149 pages 24, 34, 35, 36, and 190]

- Richard A Eisenberg. 2016. *Dependent types in haskell: Theory and practice*. Ph.D. Dissertation. University of Pennsylvania. [cited on pages 185, 194, 195, and 231]
- Richard A. Eisenberg. 2018. Standalone kind signatures. <https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0054-kind-signatures.rst> [cited on page 150]
- 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 184]
- Richard A Eisenberg, Stephanie Weirich, and Hamidhasan G Ahmed. 2016. Visible type application. In *European Symposium on Programming*. Springer, 229–254. [cited on page 150]
- 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 191]
- Phil Freeman. 2017. *PureScript by Example*. Leanpub. <https://leanpub.com/purescript>. [cited on page 199]
- 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, 162, and 193]
- 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 191]
- 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. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. Dissertation. Éditeur inconnu. [cited on page 170]

- 4181 Jean-Yves Girard. 1986. The System F of Variable Types, Fifteen Years Later. *Theoretical*
4182 *computer science* 45 (1986), 159–192. [cited on pages 5, 23, and 51]
- 4183 Warren D Goldfarb. 1981. The undecidability of the second-order unification problem. *The-*
4184 *oretical Computer Science* 13, 2 (1981), 225–230. [cited on pages 149 and 194]
- 4185 Roberta Gori and Giorgio Levi. 2002. An experiment in type inference and verification by
4186 abstract interpretation. In *International Workshop on Verification, Model Checking, and*
4187 *Abstract Interpretation*. Springer, 225–239. [cited on page 194]
- 4188 Roberta Gori and Giorgio Levi. 2003. Properties of a type abstract interpreter. In *Interna-*
4189 *tional Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer,
4190 132–145. [cited on page 194]
- 4191 Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N Freund, and Cormac Flanagan.
4192 2006. Sage: Hybrid Checking for Flexible Specifications. In *Scheme and Functional Pro-*
4193 *gramming Workshop*. [cited on page 192]
- 4194 Adam Gundry and Conor McBride. 2013. A tutorial implementation of dynamic pattern
4195 unification. *Unpublished draft* (2013). [cited on pages 194 and 195]
- 4196 Adam Gundry, Conor McBride, and James McKinna. 2010. Type inference in context. In
4197 *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional*
4198 *programming*. ACM, 43–54. [cited on pages 12, 119, 120, 121, 122, 154, and 195]
- 4199 Adam Michael Gundry. 2013. *Type inference, Haskell and dependent types*. Ph.D. Disserta-
4200 tion. University of Strathclyde. [cited on pages 149, 185, and 194]
- 4201 Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon
4202 Taylor, and Chris Speirs. 1996. *The Mercury Language Reference Manual*. Technical
4203 Report. [cited on page 199]
- 4204 Fritz Henglein. 1993. Type inference with polymorphic recursion. *ACM Trans. Program.*
4205 *Lang. Syst.* 15, 2 (April 1993), 253–289. <https://doi.org/10.1145/169701.169692>
4206 [cited on pages 150 and 236]
- 4207 J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic.
4208 *Trans. Amer. Math. Soc.* 146 (1969), 29–60. [cited on pages 3, 4, 15, and 145]
- 4209 G. Huet. 1973. A unification algorithm for typed lambda calculus. *Theoretical Computer*
4210 *Science* 1, 1 (1973), 27–57. [cited on page 149]

- 4211 Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing.
 4212 In *Proceedings of the 22nd International Conference on Functional Programming*. [cited on
 4213 pages 68, 70, 71, 75, 103, 107, and 192]
- 4214 Khurram A. Jafery and Joshua Dunfield. 2017. Sums of Uncertainty: Refinements Go
 4215 Gradual. In *Proceedings of the 44th Symposium on Principles of Programming Languages*.
 4216 14 pages. [cited on pages 61, 75, and 191]
- 4217 Trevor Jim. 1996. What are principal typings and what are they good for?. In *Proceedings*
 4218 *of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.
 4219 ACM, 42–53. [cited on page 194]
- 4220 Mark P Jones. 1995. A system of constructor classes: overloading and implicit higher-order
 4221 polymorphism. *Journal of functional programming* 5, 1 (1995), 1–35. [cited on pages 146,
 4222 149, 183, and 195]
- 4223 Mark P. Jones. 1996. Using Parameterized Signatures to Express Modular Structure. In
 4224 *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Program-*
 4225 *ming Languages* (St. Petersburg Beach, Florida, USA) (POPL '96). 68–78. [https:](https://doi.org/10.1145/237721.237731)
 4226 [//doi.org/10.1145/237721.237731](https://doi.org/10.1145/237721.237731) [cited on page 5]
- 4227 Mark P. Jones. 1999. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop*
 4228 *(Haskell '99)*, Erik Meijer (Ed.). Paris, France, pp. 9–22. University of Utrecht Technical
 4229 Report UU-CS-1999-28. [cited on pages 150, 164, and 194]
- 4230 Mark P Jones. 2000. Type classes with functional dependencies. In *European Symposium on*
 4231 *Programming*. Springer, 230–244. [cited on page 67]
- 4232 Assaf J Kfoury and Jerzy Tiuryn. 1992. Type reconstruction in finite rank fragments of the
 4233 second-order λ -calculus. *Information and computation* 98, 2 (1992), 228–257. [cited on
 4234 page 19]
- 4235 Oleg Kiselyov. 2013. How OCaml type checker works. [http://okmij.org/ftp/ML/](http://okmij.org/ftp/ML/generalization.html#levels)
 4236 [generalization.html#levels](http://okmij.org/ftp/ML/generalization.html#levels) [cited on pages 143 and 144]
- 4237 Oleg Kiselyov, Ralf Lämmel, and Kean Schupke. 2004. Strongly typed heterogeneous col-
 4238 lections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. ACM, 96–107.
 4239 [cited on page 67]
- 4240 Csongor Kiss, Susan Eisenbach, Tony Field, and Simon Peyton Jones. 2019. Higher-order
 4241 type-level programming in Haskell. In *Proceedings of the 24th ACM SIGPLAN Interna-*
 4242 *tional Conference on Functional Programming (ICFP 2019)*. ACM. [cited on page 183]

- 4243 Pieter Koopman, Rinus Plasmeijer, and Jan Martin Jansen. 2014. Church Encoding of Data
4244 Types Considered Harmful for Implementations: Functional Pearl. In *Proceedings of the*
4245 *26nd 2014 International Symposium on Implementation and Application of Functional Lan-*
4246 *guages* (Boston, MA, USA) (*IFL '14*). Association for Computing Machinery, New York,
4247 NY, USA, Article 4, 12 pages. <https://doi.org/10.1145/2746325.2746330> [cited
4248 on page 65]
- 4249 Didier Le Botlan and Didier Rémy. 2003. MLF: Raising ML to the Power of System F (*ICFP*
4250 '03). 12 pages. [cited on pages 6 and 189]
- 4251 Didier Le Botlan and Didier Rémy. 2009. Recasting MLF. *Information and Computation* 207,
4252 6 (2009), 726–785. [cited on page 189]
- 4253 Jukka Lehtosalo et al. 2006. Mypy. <http://www.mypy-lang.org/> [cited on pages 8
4254 and 61]
- 4255 Daan Leijen. 2009. Flexible Types: Robust Type Inference for First-class Polymorphism
4256 (*POPL '09*). 12 pages. [cited on pages 6 and 189]
- 4257 Andres Löb, Conor McBride, and Wouter Swierstra. 2010. A tutorial implementation of a
4258 dependently typed lambda calculus. *Fundamenta informaticae* 102, 2 (2010), 177–207.
4259 [cited on page 24]
- 4260 Simon Marlow et al. 2010. Haskell 2010 language report. Available on: [https://www.haskell.](https://www.haskell.org/onlinereport/haskell2010)
4261 [org/onlinereport/haskell2010](https://www.haskell.org/onlinereport/haskell2010) (2010). [cited on page 151]
- 4262 Jacob Matthews and Amal Ahmed. 2008. Parametric polymorphism through run-time seal-
4263 ing or, theorems for low, low prices!. In *European Symposium on Programming*. Springer,
4264 16–31. [cited on page 80]
- 4265 Conor McBride. 2002. Faking it Simulating dependent types in Haskell. *Journal of functional*
4266 *programming* 12, 4-5 (2002), 375–392. [cited on page 67]
- 4267 McCracken. 1984. The typechecking of programs with implicit type structure. In *Lecture*
4268 *Notes in Computer Science (Semantics of Data Types, Vol. 173)*. [cited on page 5]
- 4269 Dale Miller. 1991. Unification of simply typed lambda-terms as logic programming. (1991).
4270 [cited on page 194]
- 4271 Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer*
4272 *and system sciences* 17, 3 (1978), 348–375. [cited on page 15]

- James H. Morris, Jr. 1973. Types Are Not Sets. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) (POPL '73). ACM, New York, NY, USA, 120–124. <https://doi.org/10.1145/512927.512938> [cited on page 192]
- James Hiram Morris Jr. 1969. *Lambda-calculus models of programming languages*. Ph.D. Dissertation. Massachusetts Institute of Technology. [cited on page 82]
- Alan Mycroft. 1984. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*. Springer, 217–228. [cited on page 194]
- Georg Neis, Derek Dreyer, and Andreas Rossberg. 2009. Non-parametric Parametricity. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) (ICFP '09). ACM, New York, NY, USA, 135–148. <https://doi.org/10.1145/1596550.1596572> [cited on page 80]
- Max S. New, Dustin Jamner, and Amal Ahmed. 2019. Graduality and Parametricity: Together Again for the First Time. *Proc. ACM Program. Lang.* 4, POPL, Article 46 (Dec. 2019), 32 pages. <https://doi.org/10.1145/3371114> [cited on page 192]
- Martin Odersky and Konstantin Läuffer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) (POPL '96). Association for Computing Machinery, New York, NY, USA, 54–67. <https://doi.org/10.1145/237721.237729> [cited on pages 3, 7, 15, 18, 19, 20, 68, 132, 145, and 189]
- Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored Local Type Inference (POPL '01). 13 pages. [cited on pages 37 and 190]
- Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*. [cited on page 198]
- Michel Parigot. 1992. Recursive programming with proofs. *Theoretical Computer Science* 94, 2 (1992), 335–356. [cited on page 65]
- Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press. [cited on page 147]
- Simon Peyton Jones. 2020. Simplify subsumption. <https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0287-simplify-subsumption.rst> [cited on pages 23 and 189]

- 4304 Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: exploring the design
4305 space. In *Haskell workshop*, Vol. 1997. [cited on page 67]
- 4306 Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Prac-
4307 tical type inference for arbitrary-rank types. *Journal of functional programming* 17, 1
4308 (2007), 1–82. [cited on pages 3, 6, 7, 15, 18, 24, 28, 29, 35, 38, 39, 45, 46, 47, 53, 81,
4309 134, 145, 149, 182, 183, 189, and 193]
- 4310 Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006.
4311 Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM*
4312 *SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA)
4313 (ICFP ’06). ACM, New York, NY, USA, 50–61. [https://doi.org/10.1145/1159803.](https://doi.org/10.1145/1159803.1159811)
4314 1159811 [cited on pages 145 and 183]
- 4315 Benjamin C Pierce. 2002. *Types and programming languages*. [cited on page 81]
- 4316 Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Pro-*
4317 *gram. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. <https://doi.org/10.1145/345099.345100>
4318 [cited on pages 7, 11, 23, 56, and 190]
- 4319 François Pottier and Didier Rémy. 2005. The essence of ML type inference. *Advanced Topics*
4320 *in Types and Programming Languages* (2005). [cited on pages 193, 194, and 231]
- 4321 Garrel Pottinger. 1980. A type assignment for the strongly normalizable λ -terms. *To HB*
4322 *Curry: essays on combinatory logic, lambda calculus and formalism* (1980), 561–577. [cited
4323 on page 198]
- 4324 Jason Reed. 2009. Higher-order constraint simplification in dependent type theory. In *Pro-*
4325 *ceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages:*
4326 *Theory and Practice*. ACM, 49–56. [cited on page 194]
- 4327 Didier Rémy. 1992. Extension of ml type system with a sorted equational theory on types.
4328 (1992). [cited on pages 143 and 144]
- 4329 Didier Rémy. 2005. Simple, Partial Type-inference for System F Based on Type-containment
4330 (ICFP ’05). 14 pages. [cited on page 190]
- 4331 Didier Rémy and Boris Yakobowski. 2008. From ML to MLF: Graphic Type Constraints with
4332 Efficient Type Inference (ICFP ’08). 12 pages. [cited on page 189]
- 4333 John C Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium*.
4334 Springer, 408–425. [cited on page 5]

- John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Proceedings of the IFIP 9th World Computer Congress*. [cited on page 80]
- John C. Reynolds. 1991. The Coherence of Languages with Intersection Types. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*. [cited on page 82]
- Patrick Salle. 1978. Une Extension De La Theorie Des Types En lambda-Calcul. In *Proceedings of the Fifth Colloquium on Automata, Languages and Programming*. Springer-Verlag, London, UK, UK, 398–410. [cited on page 198]
- Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (Edinburgh, Scotland) (ICFP '09)*. ACM, New York, NY, USA, 341–352. <https://doi.org/10.1145/1596550.1596599> [cited on page 193]
- Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A Quick Look at Impredicativity. *Proc. ACM Program. Lang.* 4, ICFP, Article 89 (Aug. 2020), 29 pages. <https://doi.org/10.1145/3408971> [cited on pages 6, 23, and 190]
- Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (Philadelphia, PA, USA) (PLDI 2018)*. ACM, New York, NY, USA, 783–796. <https://doi.org/10.1145/3192366.3192389> [cited on pages 6, 23, and 190]
- Paula Severi and Erik Poll. 1994. Pure Type Systems with Definitions. *Logical Foundations of Computer Science* (1994), 316–328. [cited on page 197]
- Jeremy Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the design space of higher-order casts. In *European Symposium on Programming*. 17–31. [cited on page 191]
- Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings of the 2006 Scheme and Functional Programming Workshop*. [cited on pages 8, 11, 61, 101, and 191]
- Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *European Conference on Object-Oriented Programming*. [cited on pages 6, 9, 11, 61, 62, 63, 64, 68, 71, 73, 98, 99, 101, and 191]

- 4366 Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-based In-
4367 ference. In *Proceedings of the 2008 Symposium on Dynamic Languages*. [cited on page 193]
- 4368 Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined
4369 criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL*
4370 *2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. [cited on pages 9, 11, 12, 75,
4371 78, 83, 85, and 191]
- 4372 Vincent Simonet and François Pottier. 2007. A constraint-based approach to guarded alge-
4373 braic data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*
4374 29, 1 (2007), 1. [cited on page 193]
- 4375 Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *TPHOLs '08* (Montreal,
4376 P.Q., Canada). Springer-Verlag, 278–293. [cited on page 199]
- 4377 Sam Tobin-Hochstadt. 2019. Gradual Typing from Theory to Practice. [https://blog.](https://blog.sigplan.org/2019/07/12/gradual-typing-theory-practice/)
4378 [sigplan.org/2019/07/12/gradual-typing-theory-practice/](https://blog.sigplan.org/2019/07/12/gradual-typing-theory-practice/) [cited on page 8]
- 4379 Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual Parametricity, Revisited.
4380 *Proc. ACM Program. Lang.* 3, POPL, Article 17 (Jan. 2019), 30 pages. [https://doi.](https://doi.org/10.1145/3290330)
4381 [org/10.1145/3290330](https://doi.org/10.1145/3290330) [cited on page 192]
- 4382 Julien Verlauguet. 2013. Facebook: Analyzing PHP statically. In *Proceedings of Commercial*
4383 *Users of Functional Programming*. [cited on page 61]
- 4384 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and
4385 Evaluation of Gradual Typing for Python. In *Proceedings of the 10th Symposium on Dy-*
4386 *namic languages*. [cited on pages 8 and 61]
- 4387 Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. Out-
4388 sideIn (X) Modular type inference with local assumptions. *Journal of functional program-*
4389 *ming* 21, 4-5 (2011), 333–412. [cited on pages 149, 182, 183, 185, 193, 200, 231, and 232]
- 4390 Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2006. Boxy types: in-
4391 ference for higher-rank types and impredicativity. In *Proceedings of the Eleventh ACM*
4392 *SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA)
4393 *(ICFP '06)*. ACM, New York, NY, USA. [https://doi.org/10.1145/1159803.](https://doi.org/10.1145/1159803.1159811)
4394 [1159811](https://doi.org/10.1145/1159803.1159811) [cited on page 37]
- 4395 Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: First-class
4396 Polymorphism for Haskell (*ICFP '08*). 12 pages. [cited on page 190]

- 4397 P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *POPL*
4398 '89. ACM. [cited on page 199]
- 4399 Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed.
4400 In *Proceedings of the 18th European Symposium on Programming Languages and Systems*.
4401 [cited on page 192]
- 4402 Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. 2019. A
4403 Role for dependent types in Haskell. *Proc. ACM Program. Lang.* 3, ICFP, Article 101 (July
4404 2019), 29 pages. <https://doi.org/10.1145/3341705> [cited on page 194]
- 4405 Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind
4406 Equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Func-*
4407 *tional Programming* (Boston, Massachusetts, USA) (ICFP '13). ACM, New York, NY, USA,
4408 275–286. <https://doi.org/10.1145/2500365.2500599> [cited on pages 146, 148,
4409 and 194]
- 4410 Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A
4411 Eisenberg. 2017. A specification for dependent types in Haskell. In *Proceedings of the 22th*
4412 *ACM SIGPLAN International Conference on Functional Programming* (ICFP '17). ACM.
4413 [cited on pages 170 and 194]
- 4414 Joe B Wells. 1999. Typability and Type Checking in System F are Equivalent and Undecidable.
4415 *Annals of Pure and Applied Logic* 98, 1-3 (1999), 111–156. [cited on pages 6, 7, 19, and 91]
- 4416 Thomas Winant, Dominique Devriese, Frank Piessens, and Tom Schrijvers. 2014. Partial
4417 type signatures for haskell. In *International Symposium on Practical Aspects of Declarative*
4418 *Languages*. Springer, 17–32. [cited on page 230]
- 4419 A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.*
4420 115, 1 (Nov. 1994), 38–94. <https://doi.org/10.1006/inco.1994.1093> [cited on
4421 page 51]
- 4422 Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors.
4423 In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Program-*
4424 *ming Languages* (New Orleans, Louisiana, USA) (POPL '03). ACM, New York, NY, USA,
4425 224–235. <https://doi.org/10.1145/604131.604150> [cited on page 183]
- 4426 Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In
4427 *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

- 4428 *Languages* (San Antonio, Texas, USA) (*POPL '99*). Association for Computing Machinery,
4429 New York, NY, USA, 214–227. <https://doi.org/10.1145/292540.292560> [cited
4430 on pages 24 and 197]
- 4431 Ningning Xie, Xuan Bi, and Bruno C d S Oliveira. 2018. Consistent Subtyping for All. In
4432 *European Symposium on Programming*. Springer, 3–30. [cited on pages 13 and 27]
- 4433 Ningning Xie, Xuan Bi, Bruno C. D. S. Oliveira, and Tom Schrijvers. 2019a. Consistent
4434 Subtyping for All. *ACM Transactions on Programming Languages and Systems* 42, 1, Article
4435 2 (Nov. 2019), 79 pages. <https://doi.org/10.1145/3310339> [cited on pages 13
4436 and 27]
- 4437 Ningning Xie, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers. 2020. Row and
4438 Bounded Polymorphism via Disjoint Polymorphism. In *34th European Conference on
4439 Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Infor-
4440 matics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–
4441 Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 27:1–27:30. [https://doi.org/
4442 10.4230/LIPIcs.ECOOP.2020.27](https://doi.org/10.4230/LIPIcs.ECOOP.2020.27) [cited on page 198]
- 4443 Ningning Xie and Richard A Eisenberg. 2018. Coercion Quantification. In *Haskell Imple-
4444 mentors' Workshop*. [cited on page 194]
- 4445 Ningning Xie, Richard A Eisenberg, and Bruno CDS Oliveira. 2019b. Kind Inference for
4446 Datatypes: Technical Supplement. *arXiv preprint arXiv:1911.06153* (2019). [https://
4447 arxiv.org/abs/1911.06153](https://arxiv.org/abs/1911.06153) [cited on pages 13 and 194]
- 4448 Ningning Xie, Richard A. Eisenberg, and Bruno C. d. S. Oliveira. 2019c. Kind Inference for
4449 Datatypes. *Proc. ACM Program. Lang.* 4, POPL, Article 53 (Dec. 2019), 28 pages. [https :
4450 //doi.org/10.1145/3371121](https://doi.org/10.1145/3371121) [cited on page 13]
- 4451 Ningning Xie and Bruno C d S Oliveira. 2017. Towards Unification for Dependent Types. In
4452 *Draft Proceedings of the 18th Symposium on Trends in Functional Programming (TFP '18)*.
4453 Extended abstract. [cited on pages 13 and 140]
- 4454 Ningning Xie and Bruno C d S Oliveira. 2018. Let Arguments Go First. In *European Sympo-
4455 sium on Programming*. Springer, 272–299. [cited on pages 13, 38, and 57]
- 4456 Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis,
4457 and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th
4458 ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Philadelphia,

- 4459 Pennsylvania, USA) (*TLDI '12*). ACM, New York, NY, USA, 53–66. [https://doi.org/](https://doi.org/10.1145/2103786.2103795)
4460 [10.1145/2103786.2103795](https://doi.org/10.1145/2103786.2103795) [cited on pages [146](#), [148](#), and [230](#)]
- 4461 Beta Ziliani and Matthieu Sozeau. 2015. A Unification Algorithm for Coq Featuring Universe
4462 Polymorphism and Overloading. In *Proceedings of the 20th ACM SIGPLAN International*
4463 *Conference on Functional Programming* (Vancouver, BC, Canada) (*ICFP 2015*). ACM, New
4464 York, NY, USA, 179–191. <https://doi.org/10.1145/2784731.2784751> [cited on
4465 page [194](#)]

PART VI

TECHNICAL APPENDIX

4466

A

FULL RULES FOR ALGORITHMIC AP

4467

$$\boxed{(S_1, N_1) \vdash^{AP} \sigma <: \sigma_2 \hookrightarrow (S_2, N_2)}$$

(Algorithmic Subtyping)

4468

$$\frac{\text{AP-A-S-MONO} \quad S_0 \vdash^{AP} \tau_1 \approx \tau_2 \hookrightarrow S_1}{(S_0, N_0) \vdash^{AP} \tau_1 <: \tau_2 \hookrightarrow (S_1, N_0)}$$

4469

$$\frac{\text{AP-A-S-ARROWL} \quad \begin{array}{l} (S_0, N_0) \vdash^{AP} \sigma \triangleright \sigma_3 \rightarrow \sigma_4 \hookrightarrow (S_1, N_1) \\ (S_1, N_1) \vdash^{AP} \sigma_3 <: \sigma_1 \hookrightarrow (S_2, N_2) \quad (S_2, N_2) \vdash^{AP} \sigma_2 <: \sigma_4 \hookrightarrow (S_3, N_3) \end{array}}{(S_0, N_0) \vdash^{AP} \sigma_1 \rightarrow \sigma_2 <: \sigma \hookrightarrow (S_3, N_3)}$$

4470

$$\frac{\text{AP-A-S-ARROWR} \quad \begin{array}{l} (S_0, N_0) \vdash^{AP} \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \hookrightarrow (S_1, N_1) \\ (S_1, N_1) \vdash^{AP} \sigma_3 <: \sigma_1 \hookrightarrow (S_2, N_2) \quad (S_2, N_2) \vdash^{AP} \sigma_2 <: \sigma_4 \hookrightarrow (S_3, N_3) \end{array}}{(S_0, N_0) \vdash^{AP} \sigma <: \sigma_3 \rightarrow \sigma_4 \hookrightarrow (S_3, N_3)}$$

4471

$$\frac{\text{AP-A-S-FORALLL} \quad (S_0, N_0) \vdash^{AP} \sigma_1[a \mapsto \widehat{\beta}] <: \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0 \widehat{\beta}) \vdash^{AP} \forall a. \sigma_1 <: \sigma_2 \hookrightarrow (S_1, N_1)}$$

4472

$$\frac{\text{AP-A-S-FORALLR} \quad (S_0, N_0) \vdash^{AP} \sigma_1 <: \sigma_2[a \mapsto b] \hookrightarrow (S_1, N_1) \quad b \notin \text{FV}(S(\sigma_1)) \quad b \notin \text{FV}(S(\forall a. \sigma_2))}{(S_0, N_0 b) \vdash^{AP} \sigma_1 <: \forall a. \sigma_2 \hookrightarrow (S_1, N_1)}$$

4473

$$\boxed{(S_1, N_1); \Sigma \vdash^{AP} \sigma <: \sigma_2 \hookrightarrow (S_2, N_2)}$$

(Algorithmic Application Subtyping)

4474

$$\frac{\text{AP-A-AS-EMPTY}}{(S_0, N_0); \bullet \vdash^{AP} \sigma <: \sigma \hookrightarrow (S_0, N_0)}$$

4475

$$\frac{\text{AP-A-AS-FORALL} \quad (S_0, N_0); \Sigma, \sigma_3 \vdash^{AP} \sigma_1[a \mapsto \widehat{\beta}] <: \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0 \widehat{\beta}); \Sigma, \sigma_3 \vdash^{AP} \forall a. \sigma_1 <: \sigma_2 \hookrightarrow (S_1, N_1)}$$

A Full Rules for Algorithmic AP

4476	$\frac{\text{AP-A-AS-ARROW} \quad (S_0, N_0) \vdash^{AP} \sigma_3 <: \sigma_1 \hookrightarrow (S_1, N_1) \quad (S_1, N_1); \Sigma \vdash^{AP} \sigma_2 <: \sigma_4 \hookrightarrow (S_2, N_2)}{(S_0, N_0); \Sigma, \sigma_3 \vdash^{AP} \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4 \hookrightarrow (S_2, N_2)}$	
4477	$\frac{\text{AP-A-AS-MONO} \quad (S_0, N_0) \vdash^{AP} \tau \triangleright \tau_1 \rightarrow \tau_2 \hookrightarrow (S_1, N_1) \quad (S_1, N_1); \Sigma, \sigma_3 \vdash^{AP} \tau_1 \rightarrow \tau_2 <: \sigma \hookrightarrow (S_2, N_2)}{(S_0, N_0 \hat{\beta}); \Sigma, \sigma_3 \vdash^{AP} \tau <: \sigma \hookrightarrow (S_2, N_2)}$	
4478	$\boxed{(S_1, N_1) \vdash^{AP} \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \hookrightarrow (S_2, N_2)}$	(Matching)
4479	$\frac{\text{AP-A-M-TVAR} \quad S_0 \vdash^{AP} \hat{\alpha} \approx \hat{\alpha}_1 \rightarrow \hat{\alpha}_2 \hookrightarrow S_1}{(S_0, N_0 \hat{\alpha}_1 \hat{\alpha}_2) \vdash^{AP} \hat{\alpha} \triangleright \hat{\alpha}_1 \rightarrow \hat{\alpha}_2 \hookrightarrow (S_1, N_0)}$	
4480	$\frac{\text{AP-A-M-ARROW}}{(S_0, N_0) \vdash^{AP} \sigma_1 \rightarrow \sigma_2 \triangleright \sigma_1 \rightarrow \sigma_2 \hookrightarrow (S_0, N_0)}$	
4481	$\boxed{S_1 \vdash^{AP} \tau_1 \approx \tau_2 \hookrightarrow S_2}$	(Unification)
4482	$\frac{\text{AP-A-U-REFL}}{S_0 \vdash^{AP} \tau \approx \tau \hookrightarrow S_0} \quad \frac{\text{AP-A-U-SOLVED\texttt{EVARL}} \quad \hat{\alpha} \in S_0 \quad S_0 \vdash^{AP} S_0(\hat{\alpha}) \approx \tau \hookrightarrow S_1}{S_0 \vdash^{AP} \hat{\alpha} \approx \tau \hookrightarrow S_1}$	
4483	$\frac{\text{AP-A-U-EVARL} \quad \hat{\alpha} \notin S_0 \quad \hat{\alpha} \notin \text{FV}(S_0(\tau))}{S_0 \vdash^{AP} \hat{\alpha} \approx \tau \hookrightarrow [\hat{\alpha} \mapsto S_0(\tau)] \cdot S_1} \quad \frac{\text{AP-A-U-SOLVED\texttt{EVARR}} \quad \hat{\alpha} \in S_0 \quad S_0 \vdash^{AP} \tau \approx S_0(\hat{\alpha}) \hookrightarrow S_1}{S_0 \vdash^{AP} \tau \approx \hat{\alpha} \hookrightarrow S_1}$	
4484	$\frac{\text{AP-A-U-EVARR} \quad \hat{\alpha} \notin S_0 \quad \hat{\alpha} \notin \text{FV}(S_0(\tau))}{S_0 \vdash^{AP} \tau \approx \hat{\alpha} \hookrightarrow [\hat{\alpha} \mapsto S_0(\tau)] \cdot S_1} \quad \frac{\text{AP-A-U-ARROW} \quad S_0 \vdash^{AP} \tau_1 \approx \tau_3 \hookrightarrow S_1 \quad S_1 \vdash^{AP} \tau_2 \approx \tau_4 \hookrightarrow S_2}{S_0 \vdash^{AP} \tau_1 \rightarrow \tau_2 \approx \tau_3 \rightarrow \tau_4 \hookrightarrow S_2}$	
4485	$\boxed{(S_1, N_1); \Psi \vdash^{AP} e \Rightarrow \sigma \hookrightarrow (S_2, N_2)}$	(Algorithmic Typing Inference)

AP-A-INF-INT

$$\frac{}{(S_0, N_0); \Psi \vdash^{AP} n \Rightarrow \text{Int} \hookrightarrow (S_0, N_0)}$$

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

$$\frac{\text{AP-A-INF-LAMANN} \quad (S_0, N_0); \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0); \Psi \vdash^{AP} \lambda x : \sigma_1. e \Rightarrow \sigma_1 \rightarrow \sigma_2 \hookrightarrow (S_1, N_1)}$$

$$\boxed{(S_1, N_1); \Psi; \Sigma \vdash^{AP} e \Rightarrow \sigma \hookrightarrow (S_2, N_2)} \quad (\text{Algorithmic Typing Application Mode})$$

$$\frac{\text{AP-A-APP-VAR} \quad (x : \sigma_1) \in \Psi \quad (S_0, N_0); \Sigma \vdash^{AP} \sigma_1 <: \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0); \Psi; \Sigma \vdash^{AP} x \Rightarrow \sigma_2 \hookrightarrow (S_1, N_1)}$$

$$\frac{\text{AP-A-APP-LAM} \quad (S_0, N_0); \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_2 \hookrightarrow (S_1, N_1)}{(S_0, N_0); \Psi; \Sigma, \sigma_1 \vdash^{AP} \lambda x. e \Rightarrow \sigma_1 \rightarrow \sigma_2 \hookrightarrow (S_1, N_1)}$$

$$\frac{\text{AP-A-APP-LAMANN} \quad (S_0, N_0) \vdash^{AP} \sigma_2 <: \sigma_1 \hookrightarrow (S_1, N_1) \quad (S_1, N_1); \Psi, x : \sigma_1 \vdash^{AP} e \Rightarrow \sigma_3 \hookrightarrow (S_2, N_2)}{(S_0, N_0); \Psi; \Sigma, \sigma_2 \vdash^{AP} \lambda x : \sigma_1. e \Rightarrow \sigma_2 \rightarrow \sigma_3 \hookrightarrow (S_2, N_2)}$$

$$\frac{\text{AP-A-APP-APP} \quad (S_0, N_0); \Psi \vdash^{AP} e_2 \Rightarrow \sigma_1 \hookrightarrow (S_1, N_1 \overline{a_i^i}) \quad \overline{\alpha_i^i} = \text{FV}(S_1(\sigma_1)) - \text{FV}(S_1(\Psi)) \quad \sigma_2 = \forall \overline{a_i^i}. \sigma_1[\overline{\alpha_i^i} \mapsto \overline{a_i^i}] \quad (S_1, N_1); \Psi; \Sigma, \sigma_2 \vdash^{AP} e_1 \Rightarrow \sigma_2 \rightarrow \sigma_3 \hookrightarrow (S_2, N_2)}{(S_0, N_0); \Psi; \Sigma \vdash^{AP} e_1 e_2 \Rightarrow \sigma_3 \hookrightarrow (S_2, N_2)}$$

4494

B THE EXTENDED ALGORITHMIC GPC

4495

B.1 SYNTAX

4496

4497

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	$\Omega ::= \bullet \mid \Omega, x : \sigma \mid \Omega, a \mid \Omega, \hat{\alpha}_S = \tau \mid \Omega, \hat{\alpha}_G = t \mid \Omega, \blacktriangleright_{\hat{\alpha}}$

4498

B.2 TYPE SYSTEM

4499

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

(Algorithmic Consistent Subtyping)

4500

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

$$\frac{}{\Gamma \vdash^G \mathbf{Int} \lesssim \mathbf{Int} \dashv \Gamma}$$

4501

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

$$\frac{\Gamma, a \vdash^G \sigma_1 \lesssim \sigma_2 \dashv \Delta, a, \Theta}{\Gamma \vdash^G \sigma_1 \lesssim \forall a. \sigma_2 \dashv \Delta}$$

4502

GPC-AS-FORALLL

$$\frac{\Gamma, \blacktriangleright_{\hat{\alpha}_S}, \hat{\alpha}_S \vdash^G \sigma_1[a \mapsto \hat{\alpha}_S] \lesssim \sigma_2 \dashv \Delta, \blacktriangleright_{\hat{\alpha}_S}, \Theta}{\Gamma \vdash^G \forall a. \sigma_1 \lesssim \sigma_2 \dashv \Delta}$$

GPC-AS-SPAR

$$\frac{}{\Gamma \vdash^G \mathcal{S} \lesssim \mathcal{S} \dashv \Gamma}$$

4503

GPC-AS-GPAR

$$\frac{}{\Gamma \vdash^G \mathcal{G} \lesssim \mathcal{G} \dashv \Gamma}$$

GPC-AS-UNKNOWNLL

$$\frac{}{\Gamma \vdash^G ? \lesssim \mathbb{C} \dashv \mathbf{contaminate}(\Gamma, \mathbb{C})}$$

4504	$\frac{\text{GPC-AS-UNKNOWNRR}}{\Gamma \vdash^G \mathbb{C} \lesssim ? \dashv \text{contaminate}(\Gamma, \mathbb{C})}$	$\frac{\text{GPC-AS-INSTL} \quad \hat{\alpha} \notin \text{FV}(\sigma) \quad \Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta}$
4505	$\frac{\text{GPC-AS-INSTR} \quad \hat{\alpha} \notin \text{FV}(\sigma) \quad \Gamma[\hat{\alpha}] \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta}$	
4506	$\boxed{\Gamma \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta}$	(Instantiation I)
4507	$\frac{\text{GPC-INSTL-SOLVE S} \quad \Gamma \vdash^G \tau}{\Gamma, \hat{\alpha}_S, \Gamma' \vdash^G \hat{\alpha}_S \lesssim \tau \dashv \Gamma, \hat{\alpha}_S = \tau, \Gamma'}$	$\frac{\text{GPC-INSTL-SOLVE G} \quad \Gamma \vdash^G t}{\Gamma, \hat{\alpha}_G, \Gamma' \vdash^G \hat{\alpha}_G \lesssim t \dashv \Gamma, \hat{\alpha}_G = t, \Gamma'}$
4508	$\frac{\text{GPC-INSTL-SOLVE US}}{\Gamma[\hat{\alpha}_S] \vdash^G \hat{\alpha}_S \lesssim ? \dashv \Gamma[\hat{\alpha}_G, \hat{\alpha}_S = \hat{\alpha}_G]}$	$\frac{\text{GPC-INSTL-SOLVE UG}}{\Gamma[\hat{\alpha}_G] \vdash^G \hat{\alpha}_G \lesssim ? \dashv \Gamma[\hat{\alpha}_G]}$
4509	$\frac{\text{GPC-INSTL-REACH SG1}}{\Gamma[\hat{\alpha}_S][\hat{\beta}_G] \vdash^G \hat{\alpha}_S \lesssim \hat{\beta}_G \dashv \Gamma[\hat{\alpha}_G, \hat{\alpha}_S = \hat{\alpha}_G][\hat{\beta}_G = \hat{\alpha}_G]}$	
4510	$\frac{\text{GPC-INSTL-REACH SG2}}{\Gamma[\hat{\beta}_S][\hat{\alpha}_G] \vdash^G \hat{\alpha}_G \lesssim \hat{\beta}_S \dashv \Gamma[\hat{\beta}_G, \hat{\beta}_S = \hat{\beta}_G][\hat{\alpha}_G = \hat{\beta}_G]}$	
4511	$\frac{\text{GPC-INSTL-REACH OTHER}}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash^G \hat{\alpha} \lesssim \hat{\beta} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]}$	
4512	$\frac{\text{GPC-INSTL-ARR} \quad \Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \vdash^G \sigma_1 \lesssim \hat{\alpha}_1 \dashv \Theta \quad \Theta \vdash^G \hat{\alpha}_2 \lesssim [\Theta]\sigma_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \sigma_1 \rightarrow \sigma_2 \dashv \Delta}$	
4513	$\frac{\text{GPC-INSTL-FORALL R} \quad \Gamma[\hat{\alpha}], b \vdash^G \hat{\alpha} \lesssim \sigma \dashv \Delta, b, \Theta}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \lesssim \forall b. \sigma \dashv \Delta}$	
4514	$\boxed{\Gamma \vdash^G \sigma \lesssim \hat{\alpha} \dashv \Delta}$	(Instantiation II)

4515	$\frac{\text{GPC-INSTR-SOLVES} \quad \Gamma \vdash^G \tau}{\Gamma, \hat{\alpha}_S, \Gamma' \vdash^G \tau \lesssim \hat{\alpha}_S \dashv \Gamma, \hat{\alpha}_S = \tau, \Gamma'}$	$\frac{\text{GPC-INSTR-SOLVEG} \quad \Gamma \vdash^G t}{\Gamma, \hat{\alpha}_G, \Gamma' \vdash^G t \lesssim \hat{\alpha}_G \dashv \Gamma, \hat{\alpha}_G = t, \Gamma'}$
4516	$\frac{\text{GPC-INSTR-SOLVEUS}}{\Gamma[\hat{\alpha}_S] \vdash^G ? \lesssim \hat{\alpha}_S \dashv \Gamma[\hat{\alpha}_G, \hat{\alpha}_S = \hat{\alpha}_G]}$	$\frac{\text{GPC-INSTR-SOLVEUG}}{\Gamma[\hat{\alpha}_G] \vdash^G ? \lesssim \hat{\alpha}_G \dashv \Gamma[\hat{\alpha}_G]}$
4517	$\frac{\text{GPC-INSTR-REACHSG1}}{\Gamma[\hat{\alpha}_S][\hat{\beta}_G] \vdash^G \hat{\beta}_G \lesssim \hat{\alpha}_S \dashv \Gamma[\hat{\alpha}_G, \hat{\alpha}_S = \hat{\alpha}_G][\hat{\beta}_G = \hat{\alpha}_G]}$	
4518	$\frac{\text{GPC-INSTR-REACHSG2}}{\Gamma[\hat{\beta}_S][\hat{\alpha}_G] \vdash^G \hat{\beta}_S \lesssim \hat{\alpha}_G \dashv \Gamma[\hat{\beta}_G, \hat{\beta}_S = \hat{\beta}_G][\hat{\alpha}_G = \hat{\beta}_G]}$	
4519	$\frac{\text{GPC-INSTR-REACHOTHER}}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash^G \hat{\beta} \lesssim \hat{\alpha} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]}$	
4520	$\frac{\text{GPC-INSTR-ARR} \quad \Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \vdash^G \hat{\alpha}_1 \lesssim \sigma_1 \dashv \Theta \quad \Theta \vdash^G [\Theta]\sigma_2 \lesssim \hat{\alpha}_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash^G \sigma_1 \rightarrow \sigma_2 \lesssim \hat{\alpha} \dashv \Delta}$	
4521	$\frac{\text{GPC-INSTR-FORALLLL} \quad \Gamma[\hat{\alpha}], \blacktriangleright_{\hat{b}_S}, \hat{\beta}_S \vdash^G \sigma[b \mapsto \hat{\beta}_S] \lesssim \hat{\alpha} \dashv \Delta, \blacktriangleright_{\hat{b}_S}, \Theta}{\Gamma[\hat{\alpha}] \vdash^G \forall b. \sigma \lesssim \hat{\alpha} \dashv \Delta}$	
4522	$\boxed{\Gamma \vdash^G e \Rightarrow \sigma \dashv \Delta} \quad (\text{Inference})$	
4523	$\frac{\text{GPC-INF-VAR} \quad (x : \sigma) \in \Gamma}{\Gamma \vdash^G x \Rightarrow \sigma \dashv \Gamma}$	$\frac{\text{GPC-INF-INT}}{\Gamma \vdash^G n \Rightarrow \text{Int} \dashv \Gamma}$
4524	$\frac{\text{GPC-INF-LAMANN2} \quad \Gamma \vdash^G \sigma \quad \Gamma, \hat{\beta}_S, x : \sigma \vdash^G e \Leftarrow \hat{\beta}_S \dashv \Delta, x : \sigma, \Theta}{\Gamma \vdash^G \lambda x : \sigma. e \Rightarrow \sigma \rightarrow \hat{\beta}_S \dashv \Delta}$	
4525	$\frac{\text{GPC-INF-LAM2} \quad \Gamma, \hat{\alpha}_S, \hat{\beta}_S, x : \hat{\alpha}_S \vdash^G e \Leftarrow \hat{\beta}_S \dashv \Delta, x : \hat{\alpha}_S, \Theta}{\Gamma \vdash^G \lambda x. e \Rightarrow \hat{\alpha}_S \rightarrow \hat{\beta}_S \dashv \Delta}$	$\frac{\text{GPC-INF-ANNO} \quad \Gamma \vdash^G \sigma \quad \Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta}{\Gamma \vdash^G e : \sigma \Rightarrow \sigma \dashv \Delta}$

B The Extended Algorithmic GPC

4526	$\frac{\text{GPC-INF-APP} \quad \Gamma \vdash^G e_1 \Rightarrow \sigma \dashv \Theta_1 \quad \Theta_1 \vdash^G [\Theta_1] \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Theta_2 \quad \Theta_2 \vdash^G e_2 \Leftarrow [\Theta_2] \sigma_1 \dashv \Delta}{\Gamma \vdash^G e_1 e_2 \Rightarrow \sigma_2 \dashv \Delta}$	
4527	$\frac{\text{GPC-INF-LET2} \quad \Gamma \vdash^G e_1 \Rightarrow \sigma \dashv \Theta_1 \quad \Theta_1, \hat{\alpha}_S, x : \sigma \vdash^G e_2 \Leftarrow \hat{\alpha}_S \dashv \Delta, x : \sigma, \Theta_2}{\Gamma \vdash^G \text{let } x = e_1 \text{ in } e_2 \Rightarrow \hat{\alpha}_S \dashv \Delta}$	
4528	$\boxed{\Gamma \vdash^G e \Leftarrow \sigma \dashv \Delta}$	(Checking)
4529	$\frac{\text{GPC-CHK-LAM} \quad \Gamma, x : \sigma_1 \vdash^G e \Leftarrow \sigma_2 \dashv \Delta, x : \sigma_1, \Theta}{\Gamma \vdash^G \lambda x. e \Leftarrow \sigma_1 \rightarrow \sigma_2 \dashv \Delta} \quad \frac{\text{GPC-CHK-GEN} \quad \Gamma, a \vdash^G e \Leftarrow \sigma \dashv \Delta, a, \Theta}{\Gamma \vdash^G e \Leftarrow \forall a. \sigma \dashv \Delta}$	
4530	$\frac{\text{GPC-CHK-SUB} \quad \Gamma \vdash^G e \Rightarrow \sigma_1 \dashv \Theta \quad \Theta \vdash^G [\Theta] \sigma_1 \lesssim [\Theta] \sigma_2 \dashv \Delta}{\Gamma \vdash^G e \Leftarrow \sigma_2 \dashv \Delta}$	
4531	$\boxed{\Gamma \vdash^G \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta}$	(Algorithmic Matching)
4532	$\frac{\text{GPC-AM-FORALL} \quad \Gamma, \hat{\alpha}_S \vdash^G \sigma[a \mapsto \hat{\alpha}_S] \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta}{\Gamma \vdash^G \forall a. \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Delta} \quad \frac{\text{GPC-AM-ARR}}{\Gamma \vdash^G \sigma_1 \rightarrow \sigma_2 \triangleright \sigma_1 \rightarrow \sigma_2 \dashv \Gamma}$	
4533	$\frac{\text{GPC-AM-UNKNOWN}}{\Gamma \vdash^G ? \triangleright ? \rightarrow ? \dashv \Gamma} \quad \frac{\text{GPC-AM-VAR}}{\Gamma[\hat{\alpha}] \vdash^G \hat{\alpha} \triangleright \hat{\alpha}_1 \rightarrow \hat{\alpha}_2 \dashv \Gamma[\hat{\alpha}_1, \hat{\alpha}_2, \hat{\alpha} = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2]}$	

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 :: $\forall(k :: \star) \rightarrow k \rightarrow \star$` , with which users are forced to provide type arguments to `T`. That is, one would use `T` with, e.g., `T \star Int` and `T ($\star \rightarrow \star$) Maybe`, never just `T Int`. Visible dependent quantification is Haskell's equivalent to routine dependent quantification in dependently typed languages.

To support VDQ, rule `DT-TT` needs to be extended, as VDQ brings variables into scope for later reference. For example, given

```
data T ::  $\forall(k :: \star) \rightarrow k \rightarrow \star$ 
data T k a = MkT
```

We should get a context `k :: \star , a :: k` when checking `MkT`.

VDQ opens an interesting design choice: should unannotated type variables be able to introduce VDQ? For example, in the definition of `P` below, we use `f` and `a` as the arguments to `T`. To make it type-check, we need to infer `P :: $\forall(f :: \star) \rightarrow f \rightarrow \star$` .

```
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., $\hat{\alpha} \rightarrow \hat{\beta} \rightarrow \star$ or $\forall(f : \hat{\alpha}) \rightarrow \hat{\beta} \rightarrow \star$, when type-checking the `rec` group of `P`, until we type-check `P`'s body. In order to avoid this challenge, we support GHC's current ruling on the matter: *dependent variables must be manifestly so*. That is, the

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 24 (Dependency Analysis with Type-Level Data). We extend Definition 20 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  $App$  ( $f :: \star \rightarrow \star$ )  $a = A$  ( $f$   $a$ )
```

To deal with such a construct we first need to amend the syntax of a datatype declaration to support kind annotations for variables.

$$\text{datatype decl. } \mathcal{T} ::= \mathbf{data} T \phi = \overline{\mathcal{D}}_j^j$$

Kind annotations can also contain free variables, which need to be generalized in a similar way as signatures. For example, $T2$ has kind $\forall\{k :: \star\}. \forall(f :: k). \star$.

```
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 7.12), 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 7.15 would become part of the constraint solver, so substituting here is safe, too. A potential problem arises in rule `A-KTT-APP` (Figure 7.14), 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

4618 is that the compiler must maintain an extra invariant that all occurrences of a variable store
 4619 the same kind; this is straightforward to maintain in practice.

4620 Beyond just storing variables' kinds, the typing context in this work also critically stores
 4621 variables' ordering. Lacking contexts, GHC uses a different mechanism: *level numbers*, orig-
 4622 inally invented to implement untouchability [Vytiniotis et al. 2011, Section 5.1]. Every type
 4623 variable in GHC is assigned a level number during inference. Type variables contain a struc-
 4624 ture that includes level numbers. Roughly, the level number of a type variable *a* corresponds
 4625 to the number of type variables in scope before *a*. Accordingly, we can tell the relative order
 4626 (in a hypothetical context, according to the systems in this work) of two variables simply by
 4627 comparing their level numbers. One of GHC's invariants is that a unification variable at level
 4628 *n* is never unified with a type that mentions a variable with a level number $m > n$; this is
 4629 much like the extra checks in the unification judgments in our work.

4630 The *local scopes* of this work are also tracked by GHC. All the variables in the same local
 4631 scope are assigned the same level number, and they are flagged as reorderable. After inference
 4632 is complete, GHC does a topological sort to get the final order.

4633 A final role that contexts play in our formalism is that they store solutions for unification
 4634 variables; we apply contexts as a substitution. In GHC, unification variables store mutable
 4635 cells that get filled in. It has a process called *zonking*², which is exactly analogous to our
 4636 use of contexts as substitutions. Zonking a unification variable replaces the variable with its
 4637 solution, if any.

4638 C.2.3 UNIFICATION

4639 The solver in GHC still has to carry out unification, much along the lines of the unification
 4640 judgment we present here. This algorithm has to deal with the heterogeneous unification
 4641 problems we consider, as well. Indeed, GHC's unification algorithm recurs into the kinds of a
 4642 unification variable and the type it is unifying with, just as ours does. As implied by our focus
 4643 on decidability of unification, there have been a number of bugs in GHC's implementation
 4644 that led to loops in the type checker; the most recent is #16902.

4645 GHC actually uses several unification algorithms internally. It has an eager unifier, much
 4646 like the one we describe. When that unifier fails, it generates the constraint that is sent to the
 4647 solver. (The eager unifier is meant solely to be an optimization.) There is also a unifier meant
 4648 to work after type inference is complete; it checks for instance overlap, for example. All the
 4649 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.

- 4650 • The eager unifier recurs into kinds.
- 4651 • The unifier in the solver recurs into kinds.
- 4652 • The pure unifier uses an invariant that the kinds are related before looking at the types.
- 4653 It must recur when decomposing applications.

4654 In addition, GHC also has an overlap problem within unification, as exhibited in our work
 4655 by the overlap between rules [A-U-KVARL](#) and [A-U-KVARR](#) in Figure 7.4. Both the eager unifier
 4656 and the constraint-solver unifier deal with this ambiguity by using heuristics to choose which
 4657 variable might be more suitable for unification. This particular issue—which variable to unify
 4658 when there is a choice—has been the subject of some amount of churn over the years.

4659 C.2.4 COMPLETE USER-SUPPLIED KINDS

4660 Along with stand-alone kind signatures, as described in this work, GHC supports *complete*
 4661 *user-supplied kinds*, or CUSKs. A datatype has a CUSK when certain syntactic conditions are
 4662 satisfied; GHC detects these conditions *before* doing any kind inference. These CUSKs are a
 4663 poor substitute for proper kind signatures, as the syntactic cues are fragile and unexpected:
 4664 users sometimes write a CUSK without meaning to, and also sometimes leave out a necessary
 4665 part of a CUSK when they intend to specify the kind. Stand-alone kind signatures are a new
 4666 feature; they begin with the keyword `type` instead of `data`, as we have used in our work.

4667 Interestingly, it would be wrong to support CUSKs in a system without polymorphic kinds.
 4668 Consider this example:

```
4669     data S1 a = MkT1 S2
4670     data S2 = MkS2 (S1 Maybe)
```

4671 The types `S1` and `S2` form a group. We put `S2` (which has a CUSK) into the context with
 4672 kind \star . When we check `S1`, we find no constraints on `a` (in the constraint-generation pass;
 4673 see the general approach below). The kind of `S1` is then defaulted to $\star \rightarrow \star$. Checking
 4674 `S2` fails. Instead, we wish to pretend that `S2` does not have a CUSK. This would mean that
 4675 constraint-generation happens for all the constructors in both `S1` and `S2`, and `S1` would get
 4676 its correct kind $(\star \rightarrow \star) \rightarrow \star$.

4677 With kind-polymorphism, we have no problem because the kind of `T1` will be generalized
 4678 to $\forall(k :: \star). k \rightarrow \star$.

4679 This was reported as a bug #16609.

4680 C.2.5 DEPENDENCY ANALYSIS

4681 The algorithm implemented in GHC for processing datatype declarations starts with depen-
 4682 dency analysis, as ours does. The dependency analysis is less fine-grained than what we have
 4683 proposed in this work: signatures are ignored in the dependency analysis, and so datatypes
 4684 with signatures are processed alongside all the others. This means that the kinds in the ex-
 4685 ample below have more restrictive kinds in GHC than they do in our system:

```
4686   data S1 :: ∀k. k → ★
4687   data S1 a = MkS1 (S2 Int)
4688   data S2 a = MkS2 (S3 Int)
4689   data S3 a = MkS3 (S1 Int)
```

4690 A naïve dependency analysis would put all three definitions in the same group. The kind for
 4691 *S1* is given; it would indeed have that kind. The parameters of *S2* and *S3* would initially have
 4692 an unknown kind, but when occurrences of *S2* and *S3* are processed (in the definitions of
 4693 *S1* and *S2*, respectively), this unknown kind would become ★. Neither *S2* nor *S3* would be
 4694 generalized.

4695 There is a ticket to improve the dependency analysis: #9427.

4696 C.2.6 APPROACH TO KIND-CHECKING DATATYPES

4697 In GHC’s approach, after dependency analysis, so-called *initial kinds* are produced for all the
 4698 datatypes in the group. These either come from a datatype’s CUSK or from a simple analysis
 4699 of the header of the datatype (without looking at constructors). This step corresponds to our
 4700 algorithm’s placing a binding for the datatype in the context, either with the kind signature
 4701 or with a unification variable (rules [A-PGM-DT-TTS](#) and [A-PGM-DT-TT](#)).

4702 If there is no CUSK, GHC then passes over all the datatype’s constructors, collecting
 4703 constraints on unification variables. After solving these constraints, GHC generalizes the
 4704 datatype kind.

4705 For all datatypes, now with generalized kinds, all data constructors are checked (again,
 4706 for non-CUSK types). Because the kinds of the types are now generalized, a pass infers any
 4707 invisible parameters to polykinded types. For non-CUSK types, this second pass using gener-
 4708 alized kinds replaces the $T_i \mapsto T_i @ \phi_i^c$ substitution in the context in the last premise to rule [A-](#)
 4709 [PGM-DT-TT](#). Performing a substitution—instead of re-generating and solving constraints—
 4710 may be an opportunity for improvement in GHC.

4711 C.2.7 POLYMORPHIC RECURSION

4712 One challenge in kind inference is in the handling of polymorphic recursion. Although non-
 4713 CUSK types are indeed monomorphic during the constraint-generation pass, some limited
 4714 form of polymorphic recursion can get through. This is because all type variables are repre-
 4715 sented by a special form of unification variable called a TyVarTv. TyVarTvs can unify only
 4716 with other type variables. This design is motivated by the following examples:

```
4717     data T1 (a :: k) b = MkT1 (T2 a b)
4718     data T2 (c :: j) d = MkT2 (T1 c d)
```

```
4719     data T3 a where
4720       MkT3 ::  $\forall (k :: \star) (b :: k). T3 b$ 
```

4721 We want to accept all of these definitions. The first two, *T1* and *T2*, form a mutually
 4722 recursive group. Neither has a CUSK. However, the recursive occurrences are not polymor-
 4723 phically recursive: both recursive occurrences are at the *same* kind as the definition. Yet the
 4724 first parameter to *T1* is declared to have kind *k* while the first parameter to *T2* is declared
 4725 to have kind *j*. The solution: allow *k* to unify with *j* during the constraint-generation pass.
 4726 We would *not* want to allow either *k* or *j* to unify with a non-variable, as that would seem to
 4727 go against the user's wishes. But they must be allowed to unify with each other to accept this
 4728 example.

4729 With *T3* (identical to *T* from Section 9.4), we have a different motivation. During infer-
 4730 ence, we will guess the kind of *a*; call it $\hat{\alpha}$. When checking the *MkT3* constructor, we will
 4731 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*,
 4732 bound to $\hat{\alpha}$'s *right* in the context. Instead, we must set $k := \hat{\alpha}$. This is possible only if *k* is
 4733 represented by a unification variable.

4734 There are two known problems with this approach:

- 4735 1. It sometimes accepts polymorphic recursion, even without a CUSK. Here is an exam-
 4736 ple:

```
4737     data T4 a =  $\forall (k :: \star) (b :: k). MkT4 (T4 b)$ 
```

4738 The definition of *T4* is polymorphically recursive: the occurrence *T4 b* is specialized
 4739 to a kind other than the kind of *a*. Yet this definition is accepted. The two kinds unify
 4740 (as *k* becomes a unification variable, set to the guessed kind of *a*) during the constraint-
 4741 generation pass. Then, *T4* is generalized to get the kind $\forall k. k \rightarrow \star$, at which point the
 4742 last pass goes through without a hitch.

4743 The reason this acceptance is troublesome is not that *T4* is somehow dangerous or
 4744 unsafe. It is that we know that polymorphic recursion cannot be inferred Henglein
 4745 [1993], and yet GHC does it. Invariably, this must mean that GHC’s algorithm will be
 4746 hard to specify beyond its implementation.

4747 2. In rare cases, the constraint-generation pass will succeed, while the final pass—meant
 4748 to be redundant—will fail. Here is an example:

```
4749     data SameKind :: k → k → Type
4750     data Bad a where
4751       MkBad :: ∀k1 k2 (a :: k1) (b :: k2). Bad (SameKind a b)
```

4752 During the constraint-generation pass, the kinds *k*₁ and *k*₂ are allowed to unify, ac-
 4753 cepting the definition of *Bad*. During the final pass, however, *k*₁ and *k*₂ are proper
 4754 quantified type variables, always distinct. Thus the *SameKind a b* type is ill-kinded
 4755 and rejected.

4756 The fact that this final pass can fail means that we cannot implement it via a simple
 4757 substitution, as we do in rule *A-PGM-DT-TT*. One possible solution is our suggestion
 4758 to change the scoping of type parameters to GADT-syntax datatype declarations. With
 4759 that change, our second motivation above for TyVarTvs would disappear. GHC could
 4760 then use TyVarTvs only for kind variables in the head of a datatype declaration, using
 4761 proper quantified type variables in constructors. Of course, this change would break
 4762 much code in the wild, and we do not truly expect it to ever be adopted.

4763 C.2.8 THE QUANTIFICATION CHECK

4764 Our quantification check (Section 7.7.2) also has a parallel in GHC, but GHC’s solution to the
 4765 problem differed from ours. Instead of rejecting programs that fail the quantification check,
 4766 GHC accepted them, replacing the variables that would be (but cannot be) quantified with
 4767 its constant *Any* :: ∀*k*. *k*. The *Any* type is uninhabited, but exists at all kinds. As such, it is
 4768 an appropriate replacement for unquantifiable, unconstrained unification variables. Yet this
 4769 decision in GHC had unfortunate consequences: the *Any* type can appear in error messages,
 4770 and its introduction induces hard-to-understand type errors.

4771 We have later implemented our quantification check in GHC; see #16775.

4772 Another design alternative is to generalize the variable to the leftmost position where it is
 4773 still well-formed. Recall the example in Section 7.7.2:

```

4774 data Proxy :: ∀k. k → ★
4775 data Relate :: ∀a (b :: a). a → Proxy b → ★
4776 data T :: ∀(a :: ★) (b :: a) (c :: a) d. Relate b d → ★

```

4777 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
 4778 definition of T is rejected by the quantification check.

4779 Instead of rejecting the program, or solving $\hat{\beta}$ using *Any*, we can generalize over $\hat{\beta}$ as a
 4780 fresh variable f , which is put after a to make it well-kinded. Namely, we get

```

4781 data T :: ∀(a :: ★) {f :: a} (b :: a) (c :: a) (d :: Proxy f). Relate @a @f b d → ★

```

4782 However, this ordering of the variables violates our declarative specification. Moreover,
 4783 this type requires an inferred variable to be between specified variables. With higher-rank
 4784 polymorphism, due to the fact that GHC does not support first-class type-level abstraction
 4785 (i.e., Λ in types), this type cannot be instantiated to

```

4786 ∀(a :: ★) (b :: a) (c :: a) (d :: Proxy b). Relate @a @b b d → ★

```

4787 or

```

4788 ∀(a :: ★) (b :: a) (c :: a) (d :: Proxy c). Relate @a @c b d → ★

```

4789 which makes the generalization less useful.

4790 C.2.9 SCOPEDSORT

4791 When GHC deals with a local scope—a set of variables that may be reordered—it does a
 4792 topological sort on the variables at the end. However, not any topological sort will do: it must
 4793 use one that preserves the left-to-right ordering of the variables as much as possible. This is
 4794 because GHC considers these implicitly bound variables to be *specified*: they are available
 4795 for visible type application. For example, recall the example from Section 7.2.2, modified
 4796 slightly:

```

4797 data Q (a :: (f b)) (c :: k) (x :: f c)

```

4798 Inference will tell us that k must come before f and b , but the order of f and b is immaterial.
 4799 Our approach here is to make f , b , and k *inferred* variables: users of Q will not be able to
 4800 instantiate these parameters with visible type application. However, GHC takes a different
 4801 view: because the user has written the names of f , b , and k , they will be *specified*. This choice
 4802 means that the precise sorting algorithm GHC uses to fix the order of local scopes becomes

part of the *specification* of the language. Indeed, GHC documents the precise algorithm in its manual. If we followed suit, the algorithm would have to appear in our declarative specification, which goes against the philosophy of a declarative system.

Some recent debate led to a conclusion (see #16726) that we would change the interpretation of the \mathbb{Q} example from the main work, meaning that its kind variables would indeed become *inferred*. However, the problem with ScopedSort still exists in type signatures, where type variables may be implicitly bound.

C.2.10 THE “FORALL-OR-NOTHING” RULE

GHC implements the so-called *forall-or-nothing* rule, which states that either *all* variables are quantified by a user-written \forall or none are. These examples illustrate the effect:

```

4813   ex1 :: a → b → a
4814   ex2 :: ∀a b. a → b → a
4815   ex3 :: ∀a. a → b → a
4816   ex4 :: (∀a. a → b → a)

```

The signatures for both *ex1* and *ex2* are accepted: *ex1* quantifies none, while *ex2* quantifies all. The signature for *ex3* is rejected, as GHC rejects a mixed economy. However, and perhaps surprisingly, *ex4* is accepted. The only difference between *ex3* and *ex4* is the seemingly-redundant parentheses. However, because the forall-or-nothing rule applies only at the top level of a signature, the rule is not in effect for the \forall in *ex4*.

This rule interacts with the main work only in that our formalism (and some of our examples) does not respect it. This may be the cause of differing behavior between GHC and the examples we present.

C.3 COMPLETE SET OF RULES

In this section we include missing rules for Chapter 7. Some of the rules are repeated from those in the chapter.

C.3.1 DECLARATIVE HASKELL98

4829 $\boxed{\Sigma \vdash^k \sigma : \kappa}$

(Kinding for Polymorphic Types)

$$\frac{\text{K-FORALL} \quad \Sigma, a : \kappa \vdash^k \sigma : \star}{\Sigma \vdash^k \forall a : \kappa. \sigma : \star}$$

4831 $\boxed{\Sigma \vdash \Psi}$ (Well-formed Term Contexts)

$$\begin{array}{c}
 \text{ECTX-EMPTY} \\
 \hline
 \Sigma \vdash \bullet
 \end{array}
 \qquad
 \begin{array}{c}
 \text{ECTX-DCON} \\
 \frac{\Sigma \vdash \Psi \quad \Sigma \vdash^k \sigma : \star}{\Sigma \vdash \Psi, D : \sigma}
 \end{array}$$

4832

4833 C.3.2 ALGORITHMIC HASKELL98

4834 $\boxed{\Delta \Vdash^k \sigma : \kappa \dashv \Theta}$ (Kinding for Polymorphic Types)

$$\begin{array}{c}
 \text{A-K-FORALL} \\
 \frac{\Delta \Vdash^{kv} \kappa \quad \Delta, a : \kappa \Vdash^k \sigma : \kappa_2 \dashv \Theta, a : \kappa \quad [\Theta]\kappa_2 = \star}{\Delta \Vdash^k \forall a : \kappa. \sigma : \star \dashv \Theta}
 \end{array}$$

4835

4836 $\boxed{\Delta \Vdash^{kc} \sigma \Leftarrow \kappa}$ (Checking)

$$\begin{array}{c}
 \text{A-KC-EQ} \\
 \frac{\Delta \Vdash^k \sigma : \kappa_1 \dashv \Delta \quad [\Delta]\kappa_1 = [\Delta]\kappa_2}{\Delta \Vdash^{kc} \sigma \Leftarrow \kappa_2}
 \end{array}$$

4837

4838 $\boxed{\Delta \Vdash^{kv} \kappa}$ (Well-formed Kinds)

$$\begin{array}{c}
 \text{A-KV-STAR} \\
 \hline
 \Delta \Vdash^{kv} \star
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-KV-ARROW} \\
 \frac{\Delta \Vdash^{kv} \kappa_1 \quad \Delta \Vdash^{kv} \kappa_2}{\Delta \Vdash^{kv} \kappa_1 \rightarrow \kappa_2}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-KV-KUVAR} \\
 \frac{\hat{\alpha} \in \Delta}{\Delta \Vdash^{kv} \hat{\alpha}}
 \end{array}$$

4839

4840 $\boxed{\Delta \text{ ok}}$ (Well-formed Type Contexts)

$$\begin{array}{c}
 \text{A-TCTX-EMPTY} \\
 \hline
 \bullet \text{ ok}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-TCTX-TVAR} \\
 \frac{\Delta \text{ ok} \quad \Delta \Vdash^{kv} \kappa}{\Delta, a : \kappa \text{ ok}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-TCTX-TCON} \\
 \frac{\Delta \text{ ok} \quad \Delta \Vdash^{kv} \kappa}{\Delta, T : \kappa \text{ ok}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-TCTX-KUVAR} \\
 \frac{\Delta \text{ ok}}{\Delta, \hat{\alpha} \text{ ok}}
 \end{array}$$

4841

$$\begin{array}{c}
 \text{A-TCTX-KUVAR SOLVED} \\
 \frac{\Delta \text{ ok} \quad \Delta \Vdash^{kv} \kappa}{\Delta, \hat{\alpha} = \kappa \text{ ok}}
 \end{array}$$

4842

4843 $\boxed{\Delta \Vdash^{\text{ectx}} \Gamma}$ (Well-formed Term Contexts)

$$\begin{array}{c}
 \text{A-ECTX-EMPTY} \\
 \hline
 \Delta \Vdash^{\text{ectx}} \bullet
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-ECTX-DCON} \\
 \frac{\Delta \Vdash^{\text{ectx}} \Gamma \quad \Delta \Vdash^{kc} \sigma \Leftarrow \star}{\Delta \Vdash^{\text{ectx}} \Gamma, D : \sigma}
 \end{array}$$

4844

4845 C.3.3 DECLARATIVE POLYKINDS

4846 $\boxed{\Sigma \text{ ok}}$ (Well-formed Type Contexts)

$$\begin{array}{c}
 \text{TCTX-EMPTY} \\
 \hline
 \bullet \text{ ok}
 \end{array}
 \quad
 \begin{array}{c}
 \text{TCTX-TVAR-TT} \\
 \Sigma \text{ ok} \quad \Sigma \vdash^{\text{ela}} \rho : \star \\
 \hline
 \Sigma, a : \rho \text{ ok}
 \end{array}
 \quad
 \begin{array}{c}
 \text{TCTX-TCON-TT} \\
 \Sigma \text{ ok} \quad \Sigma \vdash^{\text{ela}} \eta : \star \\
 \hline
 \Sigma, T : \eta \text{ ok}
 \end{array}$$

4847

4848 $\boxed{\Sigma \vdash \Psi}$ (Well-formed Term Contexts)

$$\begin{array}{c}
 \text{ECTX-EMPTY} \\
 \hline
 \Sigma \vdash \bullet
 \end{array}
 \quad
 \begin{array}{c}
 \text{ECTX-DCON-TT} \\
 \Sigma \vdash \Psi \quad \Sigma \vdash^{\text{ela}} \mu : \star \\
 \hline
 \Sigma \vdash \Psi, D : \mu
 \end{array}$$

4849

4850 C.3.4 ALGORITHMIC POLYKINDS

4851 $\boxed{\Delta \Vdash^{\text{inst}} \mu_1 : \eta <: \omega \rightsquigarrow \mu_2 \dashv \Theta}$ (Instantiation)

$$\begin{array}{c}
 \text{A-INST-REFL} \\
 \Delta \Vdash^{\text{u}} \omega_1 \approx \omega_2 \dashv \Theta \\
 \hline
 \Delta \Vdash^{\text{inst}} \mu : \omega_1 <: \omega_2 \rightsquigarrow \mu \dashv \Theta
 \end{array}
 \quad
 \begin{array}{c}
 \text{A-INST-FORALL} \\
 \Delta, \hat{\alpha} : \omega_1 \Vdash^{\text{inst}} \mu_1 @ \hat{\alpha} : \eta[a \mapsto \hat{\alpha}] <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta \\
 \hline
 \Delta \Vdash^{\text{inst}} \mu_1 : \forall a : \omega_1. \eta <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta
 \end{array}$$

4852

$$\begin{array}{c}
 \text{A-INST-FORALL-INFER} \\
 \Delta, \hat{\alpha} : \omega_1 \Vdash^{\text{inst}} \mu_1 @ \hat{\alpha} : \eta[a \mapsto \hat{\alpha}] <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta \\
 \hline
 \Delta \Vdash^{\text{inst}} \mu_1 : \forall \{a : \omega_1\}. \eta <: \omega_2 \rightsquigarrow \mu_2 \dashv \Theta
 \end{array}$$

4853

4854 $\boxed{\Delta \Vdash^{\text{kc}} \sigma \Leftarrow \eta \rightsquigarrow \mu \dashv \Theta}$ (Kind Checking)

$$\begin{array}{c}
 \text{A-KC-SUB} \\
 \Delta \Vdash^{\text{k}} \sigma : \eta \rightsquigarrow \mu_1 \dashv \Delta_1 \quad \Delta_1 \Vdash^{\text{inst}} \mu_1 : [\Delta_1] \eta <: [\Delta_1] \omega \rightsquigarrow \mu_2 \dashv \Delta_2 \\
 \hline
 \Delta \Vdash^{\text{kc}} \sigma \Leftarrow \omega \rightsquigarrow \mu_2 \dashv \Delta_2
 \end{array}$$

4855

4856 $\boxed{\Delta \Vdash^{\text{k}} \sigma : \eta \rightsquigarrow \mu \dashv \Theta}$ (Kinding)

$$\begin{array}{c}
 \text{A-KTT-STAR} \\
 \hline
 \Delta \Vdash^{\text{k}} \star : \star \rightsquigarrow \star \dashv \Delta
 \end{array}
 \quad
 \begin{array}{c}
 \text{A-KTT-NAT} \\
 \hline
 \Delta \Vdash^{\text{k}} \text{Int} : \star \rightsquigarrow \text{Int} \dashv \Delta
 \end{array}
 \quad
 \begin{array}{c}
 \text{A-KTT-VAR} \\
 (a : \omega) \in \Delta \\
 \hline
 \Delta \Vdash^{\text{k}} a : \omega \rightsquigarrow a \dashv \Delta
 \end{array}$$

4857

$$\begin{array}{c}
 \text{A-KTT-TCON} \\
 (T : \eta) \in \Delta \\
 \hline
 \Delta \Vdash^{\text{k}} T : \eta \rightsquigarrow T \dashv \Delta
 \end{array}
 \quad
 \begin{array}{c}
 \text{A-KTT-ARROW} \\
 \hline
 \Delta \Vdash^{\text{k}} \rightarrow : \star \rightarrow \star \rightarrow \star \rightsquigarrow \rightarrow \dashv \Delta
 \end{array}$$

4858

A-KTT-FORALL

$$\frac{\Delta \Vdash^{\text{kc}} \kappa \Leftarrow \star \rightsquigarrow \omega \dashv \Delta_1 \quad \Delta_1, a : \omega \Vdash^{\text{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu \dashv \Delta_2, a : \omega, \Delta_3 \quad \Delta_3 \hookrightarrow a}{\Delta \Vdash^{\text{k}} \forall a : \kappa. \sigma : \star \rightsquigarrow \forall a : \omega. [\Delta_3] \mu \dashv \Delta_2, \text{unsolved}(\Delta_3)}$$

4859

A-KTT-APP

$$\frac{\Delta \Vdash^{\text{k}} \tau_1 : \eta_1 \rightsquigarrow \rho_1 \dashv \Delta_1 \quad \Delta_1 \Vdash^{\text{kapp}} (\rho_1 : [\Delta_1] \eta_1) \bullet \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\text{k}} \tau_1 \tau_2 : \omega \rightsquigarrow \rho \dashv \Theta}$$

4860

A-KTT-FORALLI

$$\frac{\Delta, \hat{\alpha} : \star, a : \hat{\alpha} \Vdash^{\text{kc}} \sigma \Leftarrow \star \rightsquigarrow \mu \dashv \Delta_2, a : \hat{\alpha}, \Delta_3 \quad \Delta_3 \hookrightarrow a}{\Delta \Vdash^{\text{k}} \forall a. \sigma : \star \rightsquigarrow \forall a : \hat{\alpha}. [\Delta_3] \mu \dashv \Delta_2, \text{unsolved}(\Delta_3)}$$

4861

A-KTT-KAPP

$$\frac{\Delta \Vdash^{\text{k}} \tau_1 : \eta \rightsquigarrow \rho_1 \dashv \Delta_1 \quad [\Delta_1] \eta = \forall a : \omega. \eta_2 \quad \Delta_1 \Vdash^{\text{kc}} \tau_2 \Leftarrow \omega \rightsquigarrow \rho_2 \dashv \Delta_2}{\Delta \Vdash^{\text{k}} \tau_1 @ \tau_2 : \eta_2[a \mapsto \rho_2] \rightsquigarrow \rho_1 @ \rho_2 \dashv \Delta_2}$$

4862

A-KTT-KAPP-INFER

$$\frac{\Delta \Vdash^{\text{k}} \tau_1 : \eta \rightsquigarrow \rho_1 \dashv \Delta_1 \quad [\Delta_1] \eta = \forall \{\overline{a_i : \omega_i^i}\}. \forall a : \omega. \eta_2}{\Delta_1, \hat{\alpha}_i : \omega_i[\overline{a_i \mapsto \hat{\alpha}_i^i}] \Vdash^{\text{kc}} \tau_2 \Leftarrow \omega[\overline{a_i \mapsto \hat{\alpha}_i^i}] \rightsquigarrow \rho_2 \dashv \Delta_2} \Delta \Vdash^{\text{k}} \tau_1 @ \tau_2 : \eta_2[\overline{a_i \mapsto \hat{\alpha}_i^i}][a \mapsto \rho_2] \rightsquigarrow \rho_1 @ \hat{\alpha}_i^i @ \rho_2 \dashv \Delta_2$$

4863

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

4864

(Application Kinding)

A-KAPP-TT-ARROW

$$\frac{\Delta \Vdash^{\text{kc}} \tau \Leftarrow \omega_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta \Vdash^{\text{kapp}} (\rho_1 : \omega_1 \rightarrow \omega_2) \bullet \tau : \omega_2 \rightsquigarrow \rho_1 \rho_2 \dashv \Theta}$$

4865

A-KAPP-TT-FORALL

$$\frac{\Delta, \hat{\alpha} : \omega_1 \Vdash^{\text{kapp}} (\rho_1 @ \hat{\alpha} : \eta[a \mapsto \hat{\alpha}]) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\text{kapp}} (\rho_1 : \forall a : \omega_1. \eta) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}$$

4866

A-KAPP-TT-FORALL-INFER

$$\frac{\Delta, \hat{\alpha} : \omega_1 \Vdash^{\text{kapp}} (\rho_1 @ \hat{\alpha} : \eta[a \mapsto \hat{\alpha}]) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}{\Delta \Vdash^{\text{kapp}} (\rho_1 : \forall \{a : \omega_1\}. \eta) \bullet \tau : \omega \rightsquigarrow \rho \dashv \Theta}$$

4867

A-KAPP-TT-KUVAR

$$\frac{\Delta_1, \hat{\alpha}_1 : \star, \hat{\alpha}_2 : \star, \hat{\alpha} : \omega = (\hat{\alpha}_1 \rightarrow \hat{\alpha}_2), \Delta_2 \Vdash^{\text{kc}} \tau \Leftarrow \hat{\alpha}_1 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta_1, \hat{\alpha} : \omega, \Delta_2 \Vdash^{\text{kapp}} (\rho_1 : \hat{\alpha}) \bullet \tau : \hat{\alpha}_2 \rightsquigarrow \rho_1 \rho_2 \dashv \Theta}$$

4868

$$\boxed{\Delta \Vdash^{\text{ela}} \mu : \eta}$$

4869

(Elaborated Kinding)

4870	$\frac{\text{A-ELA-STAR}}{\Delta \Vdash^{\text{ela}} \star : \star}$	$\frac{\text{A-ELA-KUVAR} \quad (\widehat{\alpha} : \omega) \in \Delta}{\Delta \Vdash^{\text{ela}} \widehat{\alpha} : [\Delta]\omega}$	$\frac{\text{A-ELA-NAT}}{\Delta \Vdash^{\text{ela}} \text{Int} : \star}$	$\frac{\text{A-ELA-VAR} \quad (a : \omega) \in \Delta}{\Delta \Vdash^{\text{ela}} a : [\Delta]\omega}$
4871	$\frac{\text{A-ELA-TCON} \quad (T : \eta) \in \Delta}{\Delta \Vdash^{\text{ela}} T : [\Delta]\eta}$	$\frac{\text{ELA-ARROW}}{\Delta \Vdash^{\text{ela}} \rightarrow : \star \rightarrow \star \rightarrow \star}$	$\frac{\text{A-ELA-FORALL} \quad \Delta \Vdash^{\text{ela}} \omega : \star \quad \Delta, a : \omega \Vdash^{\text{ela}} \mu : \star}{\Delta \Vdash^{\text{ela}} \forall a : \omega. \mu : \star}$	
4872	$\frac{\text{A-ELA-FORALL-INFER} \quad \Delta \Vdash^{\text{ela}} \omega : \star \quad \Delta, a : \omega \Vdash^{\text{ela}} \mu : \star}{\Delta \Vdash^{\text{ela}} \forall \{a : \omega\}. \mu : \star}$		$\frac{\text{A-ELA-APP} \quad \Delta \Vdash^{\text{ela}} \rho_1 : \omega_1 \rightarrow \omega_2 \quad \Delta \Vdash^{\text{ela}} \rho_2 : \omega_1}{\Delta \Vdash^{\text{ela}} \rho_1 \rho_2 : \omega_2}$	
4873	$\frac{\text{A-ELA-KAPP} \quad \Delta \Vdash^{\text{ela}} \rho_1 : \forall a : \omega. \eta \quad \Delta \Vdash^{\text{ela}} \rho_2 : \omega}{\Delta \Vdash^{\text{ela}} \rho_1 @ \rho_2 : \eta[a \mapsto [\Delta]\rho_2]}$		$\frac{\text{A-ELA-KAPP-INFER} \quad \Delta \Vdash^{\text{ela}} \rho_1 : \forall \{a : \omega\}. \eta \quad \Delta \Vdash^{\text{ela}} \rho_2 : \omega}{\Delta \Vdash^{\text{ela}} \rho_1 @ \rho_2 : \eta[a \mapsto [\Delta]\rho_2]}$	
4874	$\boxed{\Delta \Vdash_{\phi^c}^{\text{gen}} \Gamma_1 \rightsquigarrow \Gamma_2}$		(Generalization)	
4875	$\frac{\text{A-GEN} \quad \overline{\widehat{\phi}_i^c = \text{unsolved}(\mu_i)}^i}{\Delta \Vdash_{\phi^c}^{\text{gen}} \overline{D_i : \mu_i}^i \rightsquigarrow D : \forall \{\phi^c\}. \forall \{\phi_i^c\}. (\mu[\widehat{\phi}_i^c \mapsto \phi_i^c])^i}$			
4876	$\boxed{\Delta \text{ ok}}$		(Well-formed Type Contexts)	
4877	$\frac{\text{A-TCTX-EMPTY}}{\bullet \text{ ok}}$	$\frac{\text{A-TCTX-TVAR-TT} \quad \Delta \text{ ok} \quad \Delta \Vdash^{\text{ela}} \omega : \star}{\Delta, a : \omega \text{ ok}}$	$\frac{\text{A-TCTX-TCON-TT} \quad \Delta \text{ ok} \quad \Delta \Vdash^{\text{ela}} \eta : \star}{\Delta, T : \eta \text{ ok}}$	
4878	$\frac{\text{A-TCTX-KUVAR-TT} \quad \Delta \text{ ok} \quad \Delta \Vdash^{\text{ela}} \omega : \star}{\Delta, \widehat{\alpha} : \omega \text{ ok}}$	$\frac{\text{A-TCTX-KUVARSOLVED-TT} \quad \Delta \text{ ok} \quad \Delta \Vdash^{\text{ela}} \omega_2 : [\Delta]\omega_1}{\Delta, \widehat{\alpha} : \omega_1 = \omega_2 \text{ ok}}$	$\frac{\text{A-TCTX-LO} \quad \Delta, \Delta^{\text{lo}} \text{ ok}}{\Delta, \{\Delta^{\text{lo}}\} \text{ ok}}$	
4879	$\frac{\text{A-TCTX-MARKER} \quad \Delta \text{ ok}}{\Delta, \blacktriangleright_D \text{ ok}}$			
4880	$\boxed{\Delta \Vdash^{\text{ectx}} \Gamma}$		(Well-formed Term Contexts)	
4881	$\frac{\text{A-ECTX-EMPTY}}{\Delta \Vdash^{\text{ectx}} \bullet}$		$\frac{\text{A-ECTX-DCON-TT} \quad \Delta \Vdash^{\text{ectx}} \Gamma \quad \Delta \Vdash^{\text{ela}} \mu : \star}{\Delta \Vdash^{\text{ectx}} \Gamma, D : \mu}$	

4882 $\boxed{\Delta \Vdash^\mu \omega_1 \approx \omega_2 \dashv \Theta}$ (Unification)

$$\begin{array}{c}
 \text{A-U-REFL-TT} \\
 \hline
 \Delta \Vdash^\mu \omega \approx \omega \dashv \Delta
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-U-APP} \\
 \hline
 \Delta \Vdash^\mu \rho_1 \approx \rho_3 \dashv \Delta_1 \quad \Delta_1 \Vdash^\mu [\Delta_1]\rho_2 \approx [\Delta_1]\rho_4 \dashv \Theta \\
 \hline
 \Delta \Vdash^\mu \rho_1 \rho_2 \approx \rho_3 \rho_4 \dashv \Theta
 \end{array}$$

4883

$$\begin{array}{c}
 \text{A-U-KAPP} \\
 \hline
 \Delta \Vdash^\mu \rho_1 \approx \rho_3 \dashv \Delta_1 \quad \Delta_1 \Vdash^\mu [\Delta_1]\rho_2 \approx [\Delta_1]\rho_4 \dashv \Theta \\
 \hline
 \Delta \Vdash^\mu \rho_1 @ \rho_2 \approx \rho_3 @ \rho_4 \dashv \Theta
 \end{array}$$

4884

$$\begin{array}{c}
 \text{A-U-KVARL-TT} \\
 \hline
 \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \hat{\alpha} : \omega_1, \Theta_2 \quad \Theta_1 \Vdash^{\text{ela}} \rho_2 : \omega_2 \quad \Theta_1 \Vdash^\mu [\Theta_1]\omega_1 \approx \omega_2 \dashv \Theta_3 \\
 \hline
 \Delta \Vdash^\mu \hat{\alpha} \approx \rho_1 \dashv \Theta_3, \hat{\alpha} : \omega_1 = \rho_2, \Theta_2
 \end{array}$$

4885

$$\begin{array}{c}
 \text{A-U-KVARR-TT} \\
 \hline
 \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \hat{\alpha} : \omega_1, \Theta_2 \quad \Theta_1 \Vdash^{\text{ela}} \rho_2 : \omega_2 \quad \Theta_1 \Vdash^\mu [\Theta_1]\omega_1 \approx \omega_2 \dashv \Theta_3 \\
 \hline
 \Delta \Vdash^\mu \rho_1 \approx \hat{\alpha} \dashv \Theta_3, \hat{\alpha} : \omega_1 = \rho_2, \Theta_2
 \end{array}$$

4886

$$\begin{array}{c}
 \text{A-U-KVARL-LO-TT} \\
 \hline
 \Delta_1, \Delta_2 \vdash^{\text{mv}} \hat{\alpha} : \omega_1 \rightsquigarrow \Theta \quad \Delta[\{\Theta\}] \vdash_{\hat{\alpha}}^{\text{pr}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \{\Theta_2, \hat{\alpha} : \omega_1, \Theta_3\}, \Theta_4 \\
 \Theta_1, \{\Theta_2\} \Vdash^{\text{ela}} \rho_2 : \omega_2 \quad \Theta_1, \{\Theta_2\} \Vdash^\mu [\Theta_1, \Theta_2]\omega_1 \approx \omega_2 \dashv \Theta_5, \{\Theta_6\} \\
 \hline
 \Delta[\{\Delta_1, \hat{\alpha} : \omega_1, \Delta_2\}] \Vdash^\mu \hat{\alpha} \approx \rho_1 \dashv \Theta_5, \{\Theta_6, \hat{\alpha} : \omega_1 = \rho_2, \Theta_3\}, \Theta_4
 \end{array}$$

4887

$$\begin{array}{c}
 \text{A-U-KVARR-LO-TT} \\
 \hline
 \Delta_1, \Delta_2 \vdash^{\text{mv}} \hat{\alpha} : \omega_1 \rightsquigarrow \Theta \quad \Delta[\{\Theta\}] \vdash_{\hat{\alpha}}^{\text{pr}} \rho_1 \rightsquigarrow \rho_2 \dashv \Theta_1, \{\Theta_2, \hat{\alpha} : \omega_1, \Theta_3\}, \Theta_4 \\
 \Theta_1, \{\Theta_2\} \Vdash^{\text{ela}} \rho_2 : \omega_2 \quad \Theta_1, \{\Theta_2\} \Vdash^\mu [\Theta_1, \Theta_2]\omega_1 \approx \omega_2 \dashv \Theta_5, \{\Theta_6\} \\
 \hline
 \Delta[\{\Delta_1, \hat{\alpha} : \omega_1, \Delta_2\}] \Vdash^\mu \rho_1 \approx \hat{\alpha} \dashv \Theta_5, \{\Theta_6, \hat{\alpha} : \omega_1 = \rho_2, \Theta_3\}, \Theta_4
 \end{array}$$

4888

4889 $\boxed{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 \rightsquigarrow \omega_2 \dashv \Theta}$ (Promotion)

$$\begin{array}{c}
 \text{A-PR-STAR} \\
 \hline
 \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \star \rightsquigarrow \star \dashv \Delta
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-PR-ARROW} \\
 \hline
 \Delta \Vdash_{\hat{\alpha}}^{\text{pr}} \rightarrow \rightsquigarrow \rightarrow \dashv \Delta
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-PR-TCON} \\
 \hline
 \Delta[T][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\text{pr}} T \rightsquigarrow T \dashv \Delta[T][\hat{\alpha}]
 \end{array}$$

4890

$$\begin{array}{c}
 \text{A-PR-NAT} \\
 \hline
 \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \text{Int} \rightsquigarrow \text{Int} \dashv \Delta
 \end{array}
 \qquad
 \begin{array}{c}
 \text{A-PR-APP} \\
 \hline
 \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 \rightsquigarrow \rho_1 \dashv \Delta_1 \quad \Delta_1 \vdash_{\hat{\alpha}}^{\text{pr}} [\Delta_1]\omega_2 \rightsquigarrow \rho_2 \dashv \Theta \\
 \hline
 \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 \omega_2 \rightsquigarrow \rho_1 \rho_2 \dashv \Theta
 \end{array}$$

4891

4892	$\frac{\text{A-PR-KAPP} \quad \Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 \rightsquigarrow \rho_1 \dashv \Delta_1 \quad \Delta_1 \vdash_{\hat{\alpha}}^{\text{pr}} [\Delta_1] \omega_2 \rightsquigarrow \rho_2 \dashv \Theta}{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} \omega_1 @ \omega_2 \rightsquigarrow \rho_1 @ \rho_2 \dashv \Theta} \quad \text{A-PR-TVAR} \quad \frac{}{\Delta[a][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\text{pr}} a \rightsquigarrow a \dashv \Delta[a][\hat{\alpha}]}$
4893	$\frac{\text{A-PR-KUVARL} \quad \Delta[\hat{\beta}][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \Delta[\hat{\beta}][\hat{\alpha}]}{\Delta[\hat{\beta}][\hat{\alpha}] \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta} \dashv \Delta[\hat{\beta}][\hat{\alpha}]} \quad \text{A-PR-KUVARR-TT} \quad \frac{\Delta \vdash_{\hat{\alpha}}^{\text{pr}} [\Delta] \rho \rightsquigarrow \rho_1 \dashv \Theta[\hat{\alpha}][\hat{\beta} : \rho]}{\Delta[\hat{\alpha}][\hat{\beta} : \rho] \vdash_{\hat{\alpha}}^{\text{pr}} \hat{\beta} \rightsquigarrow \hat{\beta}_1 \dashv \Theta[\hat{\beta}_1 : \rho_1, \hat{\alpha}][\hat{\beta} : \rho = \hat{\beta}_1]}$
4894	$\boxed{\Delta_1 ++^{\text{mv}} \Delta_2 \rightsquigarrow \Theta} \quad (\text{Moving})$
4895	$\frac{\text{A-MV-EMPTY} \quad \bullet ++^{\text{mv}} \Delta \rightsquigarrow \Delta}{\bullet ++^{\text{mv}} \Delta \rightsquigarrow \Delta} \quad \text{A-MV-KUVAR} \quad \frac{\text{var}(\omega) \# \text{dom}(\Delta_2) \quad \Delta_1 ++^{\text{mv}} \Delta_2 \rightsquigarrow \Theta}{\hat{\alpha} : \omega, \Delta_1 ++^{\text{mv}} \Delta_2 \rightsquigarrow \hat{\alpha} : \omega, \Theta}$
4896	$\frac{\text{A-MV-KUVARM} \quad \neg(\text{var}(\omega) \# \text{dom}(\Delta_2)) \quad \Delta_1 ++^{\text{mv}} \Delta_2, \hat{\alpha} : \omega \rightsquigarrow \Theta}{\hat{\alpha} : \omega, \Delta_1 ++^{\text{mv}} \Delta_2 \rightsquigarrow \Theta}$
4897	$\frac{\text{A-MV-TVAR} \quad \text{var}(\omega) \# \text{dom}(\Delta_2) \quad \Delta_1 ++^{\text{mv}} \Delta_2 \rightsquigarrow \Theta}{a : \omega, \Delta_1 ++^{\text{mv}} \Delta_2 \rightsquigarrow a : \omega, \Theta}$
4898	$\frac{\text{A-MV-TVARM} \quad \neg(\text{var}(\omega) \# \text{dom}(\Delta_2)) \quad \Delta_1 ++^{\text{mv}} \Delta_2, a : \omega \rightsquigarrow \Theta}{a : \omega, \Delta_1 ++^{\text{mv}} \Delta_2 \rightsquigarrow \Theta}$

4899 C.3.5 CONTEXT APPLICATION IN POLYKINDS

4900	<hr/> <p>$[\Delta]\eta$ applies Δ as a substitution to η.</p> $\begin{aligned} [\Delta]\star &= \star \\ [\Delta]\text{Int} &= \text{Int} \\ [\Delta]a &= a \\ [\Delta]T &= T \\ [\Delta] \rightarrow &= \rightarrow \\ [\Delta]\forall a : \omega. \eta &= \forall a : [\Delta]\omega. [\Delta]\eta \\ [\Delta]\forall \{a : \omega\}. \eta &= \forall \{a : [\Delta]\omega\}. [\Delta]\eta \\ [\Delta](\rho_1 \rho_2) &= ([\Delta]\rho_1) ([\Delta]\rho_2) \\ [\Delta](\rho_1 @ \rho_2) &= ([\Delta]\rho_1) @ ([\Delta]\rho_2) \\ [\Delta][\hat{\alpha}]\hat{\alpha} &= \hat{\alpha} \\ [\Delta][\hat{\alpha} : \omega = \rho]\hat{\alpha} &= [\Delta][\hat{\alpha} : \omega = \rho]\rho \end{aligned}$ <hr/>
------	--

4901

 $[\Delta]\Gamma$ applies Δ as a substitution to Γ .

$$[\Omega]\bullet = \bullet$$

$$[\Omega](\Gamma, D : \mu) = [\Omega]\Gamma, D : [\Omega]\mu$$

4902

 $[\Omega]\Delta$ applies Ω as a substitution to Δ .

$$[\Omega]\bullet = \bullet$$

$$[\Omega, a : \omega](\Delta, a : \omega) = [\Omega]\Delta, a : [\Omega]\omega$$

$$[\Omega, T : \omega](\Delta, T : \omega) = [\Omega]\Delta, T : [\Omega]\omega$$

$$[\Omega, \hat{\alpha} : \omega = \rho](\Delta, \hat{\alpha} : \omega) = [\Omega]\Delta$$

$$[\Omega, \hat{\alpha} : \omega = \rho_1](\Delta, \hat{\alpha} : \omega = \rho_2) = [\Omega]\Delta \quad \text{if } [\Omega]\rho_1 = [\Omega]\rho_2$$

$$[\Omega, \hat{\alpha} : \omega = \rho]\Delta = [\Omega]\Delta \quad \text{if } \hat{\alpha} \notin \Delta$$

$$[\Omega, \blacktriangleright_D](\Delta, \blacktriangleright_D) = [\Omega]\Delta$$

$$[\Omega, \{\Omega_1\}](\Delta, \{\Delta_1\}) = [\Omega, \Omega_1](\Delta, \Delta')$$

where $\Delta' = \text{topo}(\Delta_1)$

4903 C.3.6 CONTEXT EXTENSION IN POLYKINDS

4904

$$\boxed{\Delta \longrightarrow \Theta}$$

(Context Extension)

4905

A-CTXE-EMPTY

$$\bullet \longrightarrow \bullet$$

A-CTXE-TVAR-TT

$$\Delta \longrightarrow \Theta$$

$$\Delta, a : \omega \longrightarrow \Theta, a : \omega$$

A-CTXE-TCON-TT

$$\Delta \longrightarrow \Theta$$

$$\Delta, T : \eta \longrightarrow \Theta, T : \eta$$

4906

A-CTXE-KUVAR-TT

$$\Delta \longrightarrow \Theta$$

$$\Delta, \hat{\alpha} : \omega \longrightarrow \Theta, \hat{\alpha} : \omega$$

A-CTXE-KUVAR SOLVED-TT

$$\Delta \longrightarrow \Theta \quad [\Theta]\rho_1 = [\Theta]\rho_2$$

$$\Delta, \hat{\alpha} : \omega = \rho_1 \longrightarrow \Theta, \hat{\alpha} : \omega = \rho_2$$

4907

A-CTXE-SOLVE-TT

$$\Delta \longrightarrow \Theta \quad \Theta \Vdash^{\text{ela}} \rho : [\Theta]\omega$$

$$\Delta, \hat{\alpha} : \omega \longrightarrow \Theta, \hat{\alpha} : \omega = \rho$$

A-CTXE-ADD-TT

$$\Delta \longrightarrow \Theta \quad \Theta \Vdash^{\text{ela}} \omega : \star$$

$$\Delta \longrightarrow \Theta, \hat{\alpha} : \omega$$

4908

A-CTXE-ADD SOLVED-TT

$$\Delta \longrightarrow \Theta \quad \Theta \Vdash^{\text{ela}} \rho : [\Theta]\omega$$

$$\Delta \longrightarrow \Theta, \hat{\alpha} : \omega = \rho$$

A-CTXE-MARKER

$$\Delta \longrightarrow \Theta$$

$$\Delta, \blacktriangleright_D \longrightarrow \Theta, \blacktriangleright_D$$

4909

A-CTXE-LO

$$\Delta \longrightarrow \Theta \quad \Delta, \text{topo}(\Delta_1) \longrightarrow \Theta, \Theta_1$$

$$\Delta, \{\Delta_1\} \longrightarrow \Theta, \{\Theta_1\}$$