Formalized Higher-Ranked Polymorphic Type Inference Algorithms

by

Jinxu Zhao
(赵锦煦)

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

"Formalized Higher-Ranked Polymorphic Type Inference Algorithms"

Submitted by

Jinxu Zhao

for the degree of Doctor of Philosophy

at The University of Hong Kong

in February 2021

Declaration

I declare that this thesis represents my own work, except where due acknowledgment is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

.....

Jinxu Zhao

February 2021

Acknowledgments

Contents

List of Figures

List of Tables

Part I

Prologue

1 Introduction

1.1 Type Systems and Type Inference Algorithms

Statically typed programming languages are widely used nowadays. Programs are categorized by various types before they are compiled and executed. Type errors caught before execution usually indicate potential bugs, letting the programmers realize and correct such errors in advance.

In the early stages, programming languages like Java (before 1.5) are built on a simple type system, where only features like primitive types, explicitly-typed functions, and non-generic classes are supported. People soon realize the need to generalize similar programs that have different types when used. For example, a simple way to define a function that swaps the first two items in an integer array is

```
void swap2(int[] arr) {
  int t = arr[0];
  arr[0] = arr[1];
  arr[1] = t;
}
```

Similarly, the swap function for a float array can be defined as

```
void swap2(float[] arr) {
  float t = arr[0];
  arr[0] = arr[1];
  arr[1] = t;
}
```

which mostly shares the same body with the above definition for integer array, except that the type of element in the array changes from `int` to `float`. If such functionality is a commonly used one, such as the sorting function, we definitely want to define it once and use it on many types, such as int, float, double, String, etc. Luckily, later versions of Java (1.5 and above) provides a way to define a *generic* function that accepts input of different types:

```
<T> void swap2_generic(T[] arr) {
  T t = arr[0];
  arr[0] = arr[1];
  arr[1] = t;
}
```

where `T` denotes a generic type that programmers may arbitrarily pick. The `swap2_generic` function utilizes the feature of Java's type system, generics, to improve modularity. The following program invokes the generic function (suppose we defined the `swap2_generic` function in a `Utils` class as a static method)

```
Double[] arr = new Double[2];
arr[0] = 1.0; arr[1] = 2.0;
Utils.<Double>swap2_generic(arr);
System.out.println(arr[0] + " " + arr[1]);
```

However, the type parameter `<Double>` seems a bit redundant: given the type of `arr` is `Double[]`, the generic variable `T` can only be `Double`. In fact, with the help of *type inference*, we can simply write

```
Utils.swap2_generic(arr);
```

to call the function. When compiling the above code, type inference algorithms help programmers to fill in the missing type automatically, thus saving them from writing redundant code.

From the example above, we learn that the generics of Java together with type inference algorithms used in the compiler help programmers to write generic functions, given that the functionality is generic in nature. In other words, the introduction of good features of type systems accepts more meaningful programs.

On the other hand, being able to accept all syntactically correct programs, as dynamically-typed programming languages do, is not desirable as well. Ill-typed programs are easy to write by mistake, like `"3" / 3`; or even well-typed programs with ambiguous/unexpected meaning like `"3" + 4 + 5` (for someone who is not familiar with the conventions, she might think this evaluates to ”39”). Or even more commonly seen in practice, an accidentally misspelled variable name might cause a runtime error until the line of code is actually executed. Type systems are designed to prevent such problems from happening, therefore statically-typed languages ensure type-safety, or “well-typed programs cannot go wrong”. Type inference algorithms that come along with modern type systems help eliminate trivial or redundant parts of programs to improve the conciseness.

1.1.1 Functional Programming and System F

Nowadays, more and more programming languages adopt the functional programming paradigm, where functions are first-class citizens, and programs are mostly constructed with function applications. Functional programming originates from the lambda calculus [Church 1932]. The simply-typed lambda calculus [Pierce 2002] extends the lambda calculus with a simple static type checking algorithm, preventing ill-typed programs before actual execution. However, the system does not allow polymorphic functions and thus is quite tedious to express higher-level logic.

In order to improve the expressiveness of functional programming languages, System F [Reynolds 1983] introduces polymorphism via the universal quantifier $\forall$ for types and the Λ binder (to introduce type-level functions) for expressions. For example, the identity function that can range over any type of the form $A \to A$ can be encoded in System F:

$$\mathrm{id} = \Lambda a.\, \lambda x : a.\, x : \forall a.\, a \to a$$

To enjoy the polymorphism of such a function, one needs to first supply a type argument like id @Int (we use the @ sign to denote a type-level application), so that the polymorphic function is *instantiated* with a concrete type.

Implicit Parametric Polymorphism Although being much more expressive than simply-typed systems, plain System F is still tedious to use. It feels a bit silly to write id @Int 3 compared with id 3, because the missing type is quite easy to figure out, in presence of the argument, which is of type Int and should represent a for the function application at the same time. With *implicit parametric polymorphism* [Reynolds 1983], type arguments like @Int are not written by the programmer explicitly, in contrast, it is the type inference algorithm's responsibility to guess them.

Practically speaking, in Java we can define the identity function as well,

```
<T> T id(T x) {
  return x;
}
```

And we typically use the function without specifying the type parameters, because the type system already supports implicit parametric polymorphism:

```
int n = id(3);
String s = id("3");
```

Theoretically speaking, it is unfortunate that there does not exist such a perfect algorithm that can automatically guess missing type applications for every possible System F program [Tiuryn and Urzyczyn 1996]. For example, the following expression is ambiguous when the implicit type argument is not given:

$$\mathrm{f} = (\mathrm{choose} : \forall a.\ a \to a \to a)\ (\mathrm{id} : \forall b.\ b \to b)$$

It is unclear how the type variable is instantiated during the polymorphic application: one possibility is that a is chosen to be the type of id, or $\forall b.\ b \to b$, resulting in the type $(\forall b.\ b \to b) \to (\forall b.\ b \to b)$ (Note that we cannot express this type in programming languages like Java, simply because the $\forall$ quantifiers do not appear at the top level, and its type system is already a restricted version of System F). However, another valid type is $\forall b.\ (b \to b) \to (b \to b)$, which is obtained by first instantiating id with a fresh type variable b, and generalizing the type after calculating the type of the application. Furthermore, between the two possible types, neither one is better: there exist programs that type check under either one of them and fail to type check under the other.

The fact that implicit parametric algorithm for full System F is impossible motivates people to discover restrictions on the type system under which type inference algorithms are capable of guessing the best types.

1.1.2 Hindley-Milner Type System

The Hindley-Milner (henceforth denoted as HM) type system [Damas and Milner 1982; Hindley 1969; Milner 1978] restricts System F types to *type schemes*, or *first-order polymorphism*, where polymorphic types can only have universal quantifiers in the top level. For example, $\forall a\ b.\ (a \to b) \to a \to b$ is allowed, but not $(\forall b.\ b \to b) \to (\forall b.\ b \to b)$. The type system of many programming languages like Java adopts the idea from HM, where generics is a syntax to express polymorphic types in HM: all the generic variables must be declared at the top level before the function return type. An important property of the HM type inference algorithm is *principality*, where any unannotated program can be inferred to a most general type within its type system. This supports full type-inference without any type annotations.

For example, the following function

$$\mathrm{g} = \lambda x.\ \lambda y.\ x$$

can be assigned to types Int $\to$ Bool $\to$ Int, Int $\to$ Int $\to$ Int or infinitely many others. The HM inference algorithm will infer a more general type $\forall a.\ \forall b.\ a \to b \to a$. In order to

use the function as the types mentioned above, a more-general-than relation $\leq$ is used to describe that the polymorphic type can be instantiated to more concrete types:

$$\forall a.\, \forall b.\, a \to b \to a \leq \text{Int} \to \text{Bool} \to \text{Int}$$
$$\forall a.\, \forall b.\, a \to b \to a \leq \text{Int} \to \text{Int} \to \text{Int}$$
$$\forall a.\, \forall b.\, a \to b \to a \leq \forall a.\, a \to a \to a$$

Predicativity In the HM system, $\forall$ quantifiers can appear only on the top level, type instantiations will always be *monotypes*, i.e. types without the $\forall$ quantifier. We refer to such a system as *predicative*. In contrast, System F does not restrict the types to instantiate, thus being an impredicative system. An important challenge is that full type inference for impredicative polymorphism is known to be undecidable [Wells 1999]. There are works that focus on practical inference of impredicative systems [Emrich et al. 2020; Le Botlan and Rémy 2003; Leijen 2008; Serrano et al. 2020, 2018; Vytiniotis et al. 2008]. However, throughout this work, we study predicative type systems only.

1.1.3 Higher-Ranked Polymorphism

As functional languages evolved, the need for more expressive power has motivated language designers to look beyond HM, where there is still one obvious weakness that prevents some useful programs to type check: HM only have types of rank-1, since all the $\forall$'s appear on the top level. Thus one expected feature is to allow *higher-ranked polymorphism* where polymorphic types can occur anywhere in a type signature. This enables more code reuse and more expressions to type check, and has numerous applications [Gill et al. 1993; Jones 1995; Lämmel and Jones 2003; Launchbury and Peyton Jones 1995].

One of the interesting examples is the ST monad [Launchbury and Peyton Jones 1994] of Haskell, where the `runST` function is only possible to express in a rank-2 type system:

$$\text{runST} :: \forall a.\, (\forall s.\, \text{ST}\ s\ a) \to a$$

The type is rank-2 because of the inner $\forall$ quantifier in the argument position of the type. Such a type encapsulates the state and makes sure that program states from different computation threads do not escape their scopes, otherwise the type checker should reject in advance.

In order to support higher-ranked types, we need to extend the type system of HM, but not taking a too big step since type inference for full System F would be impossible. A simple polymorphic subtyping relation proposed by Odersky and Läufer [1996] extends the HM system by allowing higher-ranked types, but instantiations are still limited to monotypes, thus the system remains predicative.

1.1.4 Bidirectional Typing

In order to improve the expressiveness for higher-ranked systems, some type annotations are necessary to guide type inference. In response to this challenge, several decidable type systems requiring some annotations have been proposed [Dunfield and Krishnaswami 2013; Le Botlan and Rémy 2003; Leijen 2008; Peyton Jones et al. 2007; Serrano et al. 2018; Vytiniotis et al. 2008].

As an example,

$$\text{hpoly} = \lambda(f : \forall a.\ a \to a).\ (f\ 1, f\ 'c')$$

the type of hpoly is $(\forall a.\ a \to a) \to (\text{Int}, \text{Char})$, which is a rank-2 type and is not typeable in HM. Notably (and unlike Hindley-Milner) the lambda argument f requires a *polymorphic* type annotation. This annotation is needed because the single universal quantifier does not appear at the top-level. Instead, it is used to quantify a type variable a used in the first argument of the function.

One of the key features that improve type inference algorithms with type annotations is bidirectional typing [Pierce and Turner 2000], a technique that combines two modes of typing: type checking, which checks an expression against a given type, and type synthesis (inference), which infers a type from an expression. Bidirectional typing is quite useful when the language supports type annotations, because those "hints" are handled with the checking mode. With bidirectional type-checking the typing judgment has two modes. In the checking mode $\Gamma \vdash e \Leftarrow A$ both e and A are inputs: i.e. we check whether expression e has some type A. In the synthesis mode $\Gamma \vdash e \Rightarrow A$ only e is input: we need to calculate the output type A from the input expression e. It is clear that, with less information, the synthesis mode is more difficult to implement than the checking mode. Therefore, bidirectional type checking with type annotations provides a way for the programmer to guide the type inference algorithm when the expression is tricky to analyse, especially in the case where higher-ranked types are involved. In addition, bidirectional typing algorithms improve the quality of error messages in practice, due to the fact that they report errors in a relatively local range, compared with global unification algorithms.

Two closely related type systems that support *predicative* higher-ranked type inference were proposed by Peyton Jones et al. [Peyton Jones et al. 2007] and Dunfield and Krishnaswami [Dunfield and Krishnaswami 2013] (henceforth denoted as DK). These type systems are popular among language designers and their ideas have been adopted by several modern functional languages, including Haskell, PureScript [Freeman 2017] and Unison [Chiusano and Bjarnason 2015] among others.

DK developed a higher-ranked global bidirectional type system based on the declarative system by Odersky and Läufer [Odersky and Läufer 1996]. Beyond the existing works, they

introduce a third form of judgment, the application inference $A \bullet e \Rrightarrow C$, where the function type A and argument expression e are input, and type C is the output representing the result type of the function application $(f :: A)\ e$. Note that f does not appear in this relation, and one can tell the procedure for type checking application expressions from the shape of the judgment — firstly, the type of the function is inferred to A; then the application inference judgment is reduced to a checking judgment to verify if the argument e checks against the argument part of A; finally, output the return part of A. Formally, the rules implement the procedure we described above:

$$\frac{\Psi \vdash e_1 \Rightarrow A \quad \Psi \vdash A \bullet e_2 \Rrightarrow C}{\Psi \vdash e_1\ e_2 \Rightarrow C}\ \texttt{Decl}{\rightarrow}\texttt{E} \qquad \frac{\Psi \vdash e \Leftarrow A}{\Psi \vdash A \rightarrow C \bullet e \Rrightarrow C}\ \texttt{Decl}{\rightarrow}\texttt{App}$$

The use of application inference judgment avoids implicit instantiations of types like HM, instead, when the function type A is a polymorphic type, it is explicitly instantiated by the application inference until it becomes a function type:

$$\frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \bullet e \Rrightarrow C}{\Psi \vdash \forall a.\ A \bullet e \Rrightarrow C}\ \texttt{Decl}\forall\texttt{App}$$

As a result, DK is in a more syntax-directed system compared with HM-like systems.

DK also provided an elegant formalization of their sound and complete algorithm, which has also inspired implementations of type inference in some polymorphic programming languages (such as PureScript [Freeman 2017] or DDC [Disciple Development Team 2017]).

The focus of this thesis is also on predicative implicit higher-ranked bidirectional type inference algorithms.

1.1.5 Subtyping

The term "subtyping" is used as two slightly different concepts in this thesis. One of them refers to the polymorphic subtyping relation used in polymorphic type systems, which compares the degree of polymorphism between types, i.e. the more-general-than relation. Chapters 3 and 4 only focus on this type of subtyping relation. The other one is the subtyping usually seen in object-oriented programming, where there are some built-in or user-defined type conversion rules.

Type system in presence of object-oriented-style subtyping allows programmers to abstract functionalities inside a class and thus benefit from another form of polymorphism. Introduced by Cardelli [1988]; Mitchell [1984]; Reynolds [1985], subtyping is studied for explicitly typed programs. Compared with functional programming languages, mainstream

object-oriented languages lacks competitive type inference algorithms. We will further introduce object-oriented subtyping as a feature in Chapter 5, where we also introduce the top and bottom types as the minimal support for subtyping.

1.2 Mechanical Formalizations and Theorem Provers

Although type inference is important in practice and receives a lot of attention in academic research, there is little work on mechanically formalizing such advanced forms of type inference in theorem provers. The remarkable exception is work done on the formalization of certain parts of Hindley-Milner type inference [Dubois 2000; Dubois and Menissier-Morain 1999; Garrigue 2015; Naraschewski and Nipkow 1999; Urban and Nipkow 2008]. However, there is still no formalization of the higher-ranked type systems that are employed by modern languages like Haskell. This is at odds with the current trend of mechanical formalizations in programming language research. In particular, both the POPLMark challenge [Aydemir et al. 2005] and CompCert [Leroy et al. 2012] have significantly promoted the use of theorem provers to model various aspects of programming languages. Today papers in various programming language venues routinely use theorem provers to mechanically formalize: *dynamic and static semantics* and their correctness properties [Aydemir et al. 2008], *compiler correctness* [Leroy et al. 2012], *correctness of optimizations* [Bertot et al. 2006], *program analysis* [Chang et al. 2006] or proofs involving *logical relations* [Abel et al. 2018].

Motivations for Mechanical Formalizations. The main argument for mechanical formalizations is a simple one. Proofs for programming languages tend to be *long*, *tedious*, and *error-prone*. In such proofs, it is very easy to make mistakes that may invalidate the whole development. Furthermore, readers and reviewers often do not have time to look at the proofs carefully to check their correctness. Therefore errors can go unnoticed for a long time. In fact, manual proofs are commonly observed to have flaws that invalidate the claimed properties. For instance, Klein et al. [2012] reproduced the proofs of nine ICFP 2009 papers in Redex, and found problems in each one of them. We also found false lemmas and incorrect proofs in DK's manual proof [Dunfield and Krishnaswami 2013]. Mechanical formalizations provide, in principle, a natural solution for these problems. Theorem provers can automatically check and validate the proofs, which removes the burden of checking from both the person doing the proofs as well as readers or reviewers.

Moreover, extending type-inference algorithms with new programming language features is often quite delicate. Studying the meta-theory for such extensions would be greatly aided by the existence of a mechanical formalization of the base language, which could then be reused and extended by the language designer. Compared with manual proofs which may

take a long time before one can fully understand every detail, theorem provers can quickly point out proofs that are invalidated after extensions.

Challenges in Variable Binding and Abella. Handling variable binding is particularly challenging in type inference, because the algorithms typically do not rely simply on local environments, but instead propagate information across judgments. Yet, there is little work on how to deal with these complex forms of binding in theorem provers. We believe that this is the primary reason why theorem provers have still not been widely adopted for formalizing type-inference algorithms.

The Abella theorem prover [Gacek 2008] is one that specifically eases formalization on-binders. Different from the two common treatments of binding which are to use the De Bruijn index [de Bruijn 1972] and the nominal logic frameworkof Pitts [Pitts 2003], Abella uses the abstraction operator in a typed λ-calculus to encode binding. Its λ-tree syntax, or HOAS, and features including the ∇ quantifier and higher-order unification, have better experiences than using Coq libraries utilizing other approaches. In practice, Abella uses the ∇ (nabla) quantifier and nominal constants to help quantify a "fresh" variable during formalization. For example, the common type checking rule

$$\frac{e \Leftarrow A \qquad a \text{ fresh}}{e \Leftarrow \forall a.\, A}$$

is encoded as

```
check E (all A) := nabla a, check E (A a)
```

in Abella, where the ∇ quantifier introduces a fresh type variable a and later use it to "open" the body of $\forall a.\, A$.

Throughout the thesis, all the type systems and declared properties are mechanically formalized in Abella.

1.3 Contributions and Outline

Contributions In this thesis, we propose variants of type inference algorithms for higher-ranked polymorphic type systems and formalize each of them in the Abella theorem prover. It is the first work on higher-ranked type inference that comes with mechanical formalizations. In summary, the main contributions of this thesis are:

- **Chapter 3** presents a predicative polymorphic subtyping algorithm.

  - We proved that our algorithm is *sound*, *complete*, and *decidable* with respect to OL's higher-ranked subtyping specification in the Abella theorem prover. And we are the first to formalize the meta-theoretical results of a polymorphic higher-ranked subtyping algorithm.
  - Similar to DK's algorithm, we employ an ordered context to collect type variables and existential variables (used as placeholders for guessing monotypes). However, our unification process is novel. DK's algorithm solves variables on-the-fly and communicates the partial solutions through an output context. In contrast, our algorithm collects a list of judgments and propagate partial solutions across them via eager substitutions. Such technique eliminates the use of output contexts, and thus simplifies the metatheory and makes mechanical formalizations easier. Besides, using only a single context keeps the definition of well-formedness simple, resulting in an easy and elegant algorithm.

- Chapter 4 presents a new bidirectional higher-ranked typing inference algorithm based on DK's declarative system.
  - We are the first to present a full mechanical formalization for a type inference algorithm of higher-ranked type system. The *soundness*, *completeness*, and *decidability* are shown in the Abella theorem prover, with respect to DK's declarative system.
  - We propose *worklist judgments*, a new technique that unifies ordered contexts and judgments. This enables precise scope tracking of variables in judgments and avoids the duplication of context information across judgments in worklists. Similar to the previous algorithm, partial solutions are propagated across judgments in a single list consist of both variable bindings and judgments. Nevertheless, the unification of worklists and contexts exploits the fact that judgments are usually sharing a large part of common information. And one can easily tell when a variable is no longer referred to.
  - Furthermore, we support inference judgments so that bidirectional typing can be encoded as worklist judgments. The idea is to use a continuation passing style to enable the transfer of inferred information across judgments.

- Chapter 5 further extends the higher-ranked system with object-oriented subtyping.
  - We propose a bidirectional declarative system extended with the top and bottom types and relevant subtyping and typing rules. Several desirable properties are satisfied and mechanically proven.

- A new backtracking-based worklist algorithm is presented and proven to be *sound* with respect to our declarative specification in the Abella theorem prover. Extended with subtyping relations of the top and bottom types, simple solutions such as $\top$ or $\bot$ satisfies subtyping constraints in parallel with other solutions witch does not involve object-oriented subtyping. Our backtracking technique is specifically well-suited for the non-deterministic trial without missing any of them.
- We also formalize the rank-1 restriction of subtyping relation, and proved that our algorithmic subtyping is *complete* under such restriction.

Prior Publications. This thesis is partially based on the publications by the author [Zhao et al. 2018, 2019], as indicated below.

Chapter 3: Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. "Formalization of a Polymorphic Subtyping Algorithm". In *International Conference on Interactive Theorem Proving (ITP)*.

Chapter 4: Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. "A Mechanical Formalization of Higher-Ranked Polymorphic Type Inference". In *International Conference on Functional Programming (ICFP)*.

Mechanized Proofs. All proofs in this thesis is mechanically formalized in the Abella theorem prover and are available online:

Chapter 3: `https://github.com/JimmyZJX/Abella-subtyping-algorithm`

Chapter 4: `https://github.com/JimmyZJX/TypingFormalization`

Chapter 5: `https://github.com/JimmyZJX/TODO` Jimmy: TODO URL

---

2 Background

In this chapter, we introduce some highly related type systems. They are basic concepts and should help the reader understand the rest of the thesis better. Section 2.1 introduces the Hindley-Milner type system [Damas and Milner 1982; Hindley 1969; Milner 1978], a widely used system that supports rank-1 (prenex) polymorphism. Section 2.2 presents the Odersky-Läufer type system, which is an extension of the Hindley-Milner by allowing higher-ranked types. Section 2.3 describes the Dunfield-Krishnaswami bidirectional type system, a bidirectional type system that further extends Odersky-Läufer's. Finally, in Section 2.4 we introduce one of the recent advancements in the ML family, namely the MLsub type system, which integrates object-oriented subtyping with the Hindley-Milner type inference.

2.1 Hindley-Milner Type System

The Hindley-Milner type system, hereafter called "HM" for short, is a classical lambda calculus with parametric polymorphism. Thanks to its simple formulation and powerful inference algorithm, many modern functional programming languages are still using HM as their base, including the ML family and Haskell. The system is also known as Damas–Milner or Damas–Hindley–Milner. Hindley [Hindley 1969] and Milner [Milner 1978] independently discovered equivalent algorithms for the polymorphic typing problem and also proved the soundness of their algorithms. Later on, Damas and Milner [1982] proved the completeness of their algorithm.

2.1.1 Declarative System

Syntax The declarative syntax is shown in Figure 2.1. The HM types are consist of polymorphic types (or type schemes) and monomorphic types. A polymorphic type contains zero or more universal quantifiers only at the top level. When no universal quantifier occurs, the type belongs to a mono-type. Mono-types are constructed by a unit type 1, a type variable a, or a function type $\tau_1 \to \tau_2$.

Expressions e includes variables x, literals $()$, lambda abstractions $\lambda x.\, e$, applications $e_1\, e_2$ and the let expression $\textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2$. A context Ψ is a collection of type bindings for variables.

$$
\begin{array}{rlll}
\text{Type variables} & a, b & & \\
\text{Types} & \sigma & ::= & \tau \mid \forall a.\, \sigma \\
\text{Monotypes} & \tau & ::= & 1 \mid a \mid \tau_1 \to \tau_2 \\
\text{Expressions} & e & ::= & x \mid () \mid \lambda x.\, e \mid e_1\, e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
\text{Contexts} & \Psi & ::= & \cdot \mid \Psi, x : \sigma
\end{array}
$$

Figure 2.1: HM Syntax

$\boxed{\sigma_1 \sqsubseteq \sigma_2}$ HM Type Instantiation

$$\frac{\tau' = [\overline{\tau}/\overline{a}]\tau \qquad \overline{b} \notin \mathrm{FV}(\forall \overline{a}.\, \tau)}{\forall \overline{a}.\, \tau \sqsubseteq \forall \overline{b}.\, \tau'}\ \texttt{HM-TInst}$$

$\boxed{\Psi \vdash_{HM} e : \sigma}$ HM Typing

$$\frac{(x : \sigma) \in \Psi}{\Psi \vdash_{HM} x : \sigma}\ \texttt{HM-Var} \qquad \frac{}{\Psi \vdash_{HM} () : 1}\ \texttt{HM-Unit} \qquad \frac{\Psi, x : \tau_1 \vdash_{HM} e : \tau_2}{\Psi \vdash_{HM} \lambda x.\, e : \tau_1 \to \tau_2}\ \texttt{HM-Abs}$$

$$\frac{\Psi \vdash_{HM} e_1 : \tau_1 \to \tau_2 \qquad \Psi \vdash_{HM} e_2 : \tau_1}{\Psi \vdash_{HM} e_1\, e_2 : \tau_2}\ \texttt{HM-App}$$

$$\frac{\Psi \vdash_{HM} e_1 : \sigma \qquad \Psi, x : \sigma \vdash_{HM} e_2 : \tau}{\Psi \vdash_{HM} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}\ \texttt{HM-Let}$$

$$\frac{\Psi \vdash_{HM} e : \sigma \qquad \overline{a} \notin \mathrm{FV}(\Psi)}{\Psi \vdash_{HM} e : \forall \overline{a}.\, \sigma}\ \texttt{HM-Gen} \qquad \frac{\Psi \vdash_{HM} e : \sigma_1 \qquad \sigma_1 \sqsubseteq \sigma_2}{\Psi \vdash_{HM} e : \sigma_2}\ \texttt{HM-Inst}$$

Figure 2.2: HM Type System

Type Instantiation The relations between types are described via type instantiations. The rule shown to the top of Figure 2.2 checks if $\forall \overline{a}.\, \tau$ is a *generic instance* of $\forall \overline{b}.\, \tau'$. This relation is valid when $\tau' = [\overline{\tau}/\overline{a}]\tau$ for a series of mono-types $\overline{\tau}$ and each variable in $\overline{b}$ is not free in $\forall \overline{a}.\, \tau$.

For example,

$$\forall a.\, a \to a \sqsubseteq 1 \to 1$$

is obtained by the substitution $[1/a]$, and

$$\forall a.\, a \to a \sqsubseteq \forall b.\, (b \to b) \to (b \to b)$$

substitutes a by $b \to b$, and generalizes b after the substitution.

TYPING The typing relation $\Psi \vdash_{HM} e : \sigma$ synthesizes a type σ for an expression e under the context Ψ. Rule `HM-Var` looks up the binding of a variable x in the context. Rule `HM-Unit` always gives the unit type 1 to the unit expression $()$. For a lambda abstraction $\lambda x.\ e$, rule `HM-Abs` guesses its input type (τ_1) and computes the type of its body (τ_2) as the return type. Rule `HM-App` eliminates a function type by an application $e_1\ e_2$, where the argument type must be the same as the input type of the function, and the type of the whole application is τ_2.

Rule `HM-Let` is also referred to as let-polymorphism. In (untyped) lambda calculus, **let** $x = e_1$ **in** e_2 behaves the same as $(\lambda x.\ e_2)\ e_1$. However, the HM let rule derives the type of e_1 first, and binds the polymorphic type into the context before e_2. This enables polymorphic expressions to be reused multiple times in different instantiated types.

Rules `HM-Gen` and `HM-Inst` change the type of an expression at any time during the derivation. Rule `HM-Gen` generalizes over fresh type variables $\overline{a}$. Rule `HM-Inst`, as opposed to generalization, specializes a type according to the type instantiation relation.

The type system of HM supports *implicit instantiation* through rule `HM-Inst`. This means that any expression (function) that has a polymorphic type can be automatically instantiated with a proper monotype for any reasonable application. The fact that only monotypes are guessed indicates that the system is *predicative*. In contrast, an *impredicative* system might guess polymorphic types. Unfortunately, type inference on impredicative systems is undecidable [Wells 1999]. In this thesis, we focus on predicative systems only.

2.1.2 Algorithmic System and Principality

SYNTAX-DIRECTED SYSTEM The declarative system is not syntax-directed due to rules `HM-Gen` and `HM-Inst`, which can be applied to any expression. A syntax-directed system can be obtained by replacing rules `HM-Var` and `HM-Let` by the following rules:

$$\frac{(x : \sigma) \in \Psi \qquad \sigma \sqsubseteq \tau}{\Psi \vdash^{S}_{HM} x : \tau}\ \texttt{HM-Var-Inst} \qquad \frac{\begin{array}{c}\Psi \vdash_{HM} e_1 : \sigma \\ \overline{a} = \mathrm{FV}(\sigma) - \mathrm{FV}(\Psi) \\ \Psi, x : \forall \overline{a}.\ \sigma \vdash_{HM} e_2 : \tau\end{array}}{\Psi \vdash^{S}_{HM} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}\ \texttt{HM-Let-Gen}$$

A generalization on σ, the synthesized type of e_1, is added to rule `HM-Let`, since it is the source place where a polymorphic type is generated. However, a too generalized type might reject applications due to its shape, therefore, an instantiation procedure is added to eliminate all the universal quantifiers on rule `HM-Var`. We omit rules `HM-Unit`, `HM-Abs`, and `HM-App`

for the syntax-directed system $\Psi \vdash^{S}_{HM}$. The following property shows that the new system is (almost) equivalent to the original declarative system.

Theorem 2.1 (Equivalence of Syntax-Directed System)**.**

1. *If* $\Psi \vdash^{S}_{HM} e : \sigma$ *then* $\Psi \vdash_{HM} e : \sigma$
2. *If* $\Psi \vdash_{HM} e : \sigma$ *then* $\Psi \vdash_{HM} e : \tau$*, and* $\forall \overline{a}.\ \tau \sqsubseteq \sigma$*, where* $\overline{a} = FV(\tau) - FV(\Psi)$.

Type Inference Algorithm Although being syntax-directed solves some problems, the rules still require some guessings, including rule `HM-Abs` and `HM-Var-Inst`. Algorithm W [Milner 1978], based on unification, is proven to be sound and complete w.r.t the declarative specifications.

Theorem 2.2 (Algorithmic Completeness (Principality))**.** *If* $\Psi \vdash_{HM} e : \sigma$*, then W computes a principal type scheme* σ_p*, i.e.*

1. $\Psi \vdash_{HM} e : \sigma_p$
2. $\sigma_p \sqsubseteq \sigma$.

2.2 Odersky-Läufer Type System

The HM type system is simple, powerful, and easy-to-use. However, it only accepts types of rank-1, i.e. the $\forall$ quantifier can only appear in the top-level. In practice, there are cases where *higher-ranked* types are needed. The rank-1 limitation prevents those programs from type check and thus loses expressiveness. The problem is that full type inference for System F is proven to be undecidable [Wells 1999]. Odersky and Läufer [1996] then proposed a system where programmers can make use of type annotations to guide the type system, especially on higher-ranked types. This extension of HM preserves nice properties of HM, while accepting higher-ranked types to be checked with the help of the programmers.

For example, consider the following function definition

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

This is not typeable in HM, because the argument of the lambda abstraction, f, is applied to both an integer and a character, which means that it should be of a polymorphic unknown type, thus the type of the lambda abstraction cannot be inferred by the HM type system. This seems reasonable, since there are several polymorphic types that fit the function, for example,

$$\lambda f.\ (f\ 1, f\ 'c') :: (\forall a.\ a \to a) \to (\text{Int}, \text{Char})$$

$$\lambda f.\ (f\ 1, f\ 'c') :: (\forall a.\ a \to \text{Int}) \to (\text{Int}, \text{Int})$$

The solution is also natural: if the programmer may pick the type of argument she wants, the type system can figure out the rest. By adding type annotation on f, OL now accepts the definition

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

and infers type $(\forall a.\ a \to a) \to (\text{Int}, \text{Char})$.

In what follows, we will first formally define the rank of a type, and then introduce the declarative system of OL, finally discuss the relationship between OL and HM.

2.2.1 Higher-Ranked Types

The *rank* of a type represents how deep a universal quantifier appears at the contravariant position [Kfoury and Tiuryn 1992]. Formally speaking,

$$\begin{array}{rrcl} \text{Rank 0 / Monotypes} & \tau, \sigma^0 & ::= & 1 \mid a \mid \tau_1 \to \tau_2 \\ \text{Rank } k (k \geq 1), \text{ Polytypes} & \sigma^k & ::= & \sigma^{k-1} \mid \sigma^{k-1} \to \sigma^k \mid \forall a.\ \sigma^k \end{array}$$

The following example illustrates what rank a type belongs to:

$$\begin{array}{rl} 1 \to 1 & \text{Rank 0} \\ \forall a.\ a \to a & \text{Rank 1} \\ 1 \to \forall a.\ a \to a & \text{Rank 1} \\ (\forall a.\ a \to a) \to (\forall a.\ a \to a) & \text{Rank 2} \end{array}$$

According to the definition, monotypes are types that does not contain any universal quantifier. In the HM type system, all polymorphic types have rank 1.

2.2.2 Declarative System

The syntax of Odersky-Läufer system is shown in Figure 2.3. There are several differences compared to the HM system.

First, polymorphic types can be of arbitrary rank, i.e. the forall quantifier may occur at any part of a type. Yet, mono-type remains the same definition as HM's.

Second, expressions now allows annotations $e : \sigma$ and (argument) annotated lambda functions $\lambda x : \sigma.\ e$. Annotations on expressions help guide the type system properly, acting as a machine-checked document by the programmers. By annotating the argument of a lambda

$$
\begin{array}{rlll}
\text{Type variables} & a, b & & \\
\text{Types} & \sigma & ::= & 1 \mid a \mid \sigma_1 \to \sigma_2 \mid \forall a.\, \sigma \\
\text{Monotypes} & \tau & ::= & 1 \mid a \mid \tau_1 \to \tau_2 \\
\text{Expressions} & e & ::= & x \mid () \mid \lambda x : \sigma.\, e \mid e : \sigma \mid \lambda x.\, e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
\text{Contexts} & \Psi & ::= & \cdot \mid \Psi, x : \sigma \mid \Psi, a
\end{array}
$$

Figure 2.3: Syntax of Odersky-Läufer System

$$
\frac{}{\Psi \vdash_{OL} 1}\ \texttt{OL-WF-Unit} \qquad \frac{a \in \Psi}{\Psi \vdash_{OL} a}\ \texttt{OL-WF-TVar}
$$

$$
\frac{\Psi \vdash_{OL} \sigma_1 \qquad \Psi \vdash_{OL} \sigma_2}{\Psi \vdash_{OL} \sigma_1 \to \sigma_2}\ \texttt{OL-WF-Arr} \qquad \frac{\Psi, a \vdash_{OL} \sigma}{\Psi \vdash_{OL} \forall a.\, \sigma}\ \texttt{OL-WF-Forall}
$$

Figure 2.4: Well-formedness of types in the Odersky-Läufer System

function with a polymorphic type σ, one may encode a function of higher rank in this system compared to HM's.

Finally, contexts consist of not only variable bindings, but also type variable declarations. Here we adopt a slightly different approach than the original work [Odersky and Läufer 1996], which does not track type variables explicitly in a context. Such explicit declarations reduce formalization difficulties, especially when dealing with freshness conditions or variable name encodings. This also enables us to formally define the well-formedness of types, shown in Figure 2.4.

Subtyping The subtyping relation, defined in Figure 2.5, is more powerful than that (type instantiation) of HM. In contrast to HM's subtyping, higher-ranked types can be compared thanks to rule `OL-SUB-Arr`. Functions are contravariant on argument types and covariant on return types. Rule `OL-SUB-∀L` instantiates a polymorphic type by a monotype τ. Rule `OL-SUB-∀R` picks a fresh type variable for the right-hand-side polymorphic type, which is made possible by renaming according to alpha-equivalence. Given that such implicit freshness condition is widely adopted, we omit that throughout the thesis.

Typing The type system of Odersky-Läufer, shown in Figure 2.6, extends HM's type system in the following aspects.

Rule `OL-Lam` now accepts polymorphic return type, because such type is well-formed. The guess on parameter types is still limited to monotypes like HM's. However, if a parameter

$$\frac{}{\Psi \vdash_{OL} 1 \leq 1}\ \texttt{OL-SUB-Unit} \qquad \frac{a \in \Psi}{\Psi \vdash_{OL} a \leq a}\ \texttt{OL-SUB-Var}$$

$$\frac{\Psi \vdash_{OL} \sigma_1' \leq \sigma_1 \qquad \Psi \vdash_{OL} \sigma_2 \leq \sigma_2'}{\Psi \vdash_{OL} \sigma_1 \to \sigma_2 \leq \sigma_1' \to \sigma_2'}\ \texttt{OL-SUB-Arr}$$

$$\frac{\Psi \vdash_{OL} \tau \qquad \Psi \vdash_{OL} [\tau/a]\sigma \leq \sigma'}{\Psi \vdash_{OL} \forall a.\ \sigma \leq \sigma'}\ \texttt{OL-SUB-}\forall\texttt{L} \qquad \frac{\Psi, a \vdash_{OL} \sigma \leq \sigma'}{\Psi \vdash_{OL} \sigma \leq \forall a.\ \sigma'}\ \texttt{OL-SUB-}\forall\texttt{R}$$

Figure 2.5: Subtyping of the Odersky-Läufer System

$$\frac{(x : \sigma) \in \Psi}{\Psi \vdash_{OL} x : \sigma}\ \texttt{OL-Var} \qquad \frac{}{\Psi \vdash_{OL} () : 1}\ \texttt{OL-Unit} \qquad \frac{\Psi \vdash_{OL} e : \sigma}{\Psi \vdash_{OL} (e : \sigma) : \sigma}\ \texttt{OL-Anno}$$

$$\frac{\Psi \vdash_{OL} \tau \qquad \Psi, x : \tau \vdash_{OL} e : \sigma}{\Psi \vdash_{OL} \lambda x.\ e : \tau \to \sigma}\ \texttt{OL-Lam} \qquad \frac{\Psi, x : \sigma_1 \vdash_{OL} e : \sigma_2}{\Psi \vdash_{OL} \lambda x : \sigma_1.\ e : \sigma_1 \to \sigma_2}\ \texttt{OL-LamAnno}$$

$$\frac{\Psi \vdash_{OL} e_1 : \sigma_1 \to \sigma_2 \qquad \Psi \vdash_{OL} e_2 : \sigma_1}{\Psi \vdash_{OL} e_1\ e_2 : \sigma_2}\ \texttt{OL-App} \qquad \frac{\Psi, a \vdash_{OL} e : \sigma}{\Psi \vdash_{OL} e : \forall a.\ \sigma}\ \texttt{OL-Gen}$$

$$\frac{\Psi \vdash_{OL} e_1 : \sigma_1 \qquad \Psi, x : \sigma_1 \vdash_{OL} e_2 : \sigma_2}{\Psi \vdash_{OL} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \sigma_2}\ \texttt{OL-Let} \qquad \frac{\Psi \vdash_{OL} e : \sigma_1 \qquad \Psi \vdash_{OL} \sigma_1 \leq \sigma_2}{\Psi \vdash_{OL} e : \sigma_2}\ \texttt{OL-Sub}$$

Figure 2.6: Typing of the Odersky-Läufer System

type is specified in advance, the type system accepts polymorphic argument type with rule `OL-LamAnno`. Functions of arbitrary rank can be encoded through proper annotations. The application and let-generalization rules also accept polymorphic return types.

Rule `OL-Gen` encodes the generalization rule of HM in a different way under explicit type variable declarations. A fresh type variable is introduced into the context before the type of expression e is calculated. Then we conclude that e has a polymorphic type by generalizing the type variable. For example, the type of the identity function is derived as follows

$$\dfrac{\dfrac{\dfrac{}{\cdot, a \vdash_{OL} a} \qquad \dfrac{(x : a) \in (\cdot, a, x : a)}{\cdot, a, x : a \vdash_{OL} x : a}\ \texttt{OL-Var}}{\cdot, a \vdash_{OL} \lambda x.\ x : a \to a}\ \texttt{OL-Lam}}{\cdot \vdash_{OL} \lambda x.\ x : \forall a.\ a \to a}\ \texttt{OL-Gen}$$

The subsumption rule `OL-Sub` converts the type of an expression with the help of the subtyping relation.

Type variables	a, b		
Types	A, B, C	$::=$	$1 \mid a \mid \forall a.\ A \mid A \to B$
Monotypes	τ, σ	$::=$	$1 \mid a \mid \tau \to \sigma$
Expressions	e	$::=$	$x \mid () \mid \lambda x.\ e \mid e_1\ e_2 \mid (e : A)$
Contexts	Ψ	$::=$	$\cdot \mid \Psi, a \mid \Psi, x : A$

Figure 2.7: Syntax of Declarative System

2.2.3 Relating to HM

The OL type system accepts higher-ranked types, but it only tries to instantiate monotypes like HM. Therefore, conservatively extends HM, such that every typed expression in HM is also typed in OL. In the meantime, all the "guessing" jobs OL needs to do remains in instantiating monotypes, thus the algorithm can be extended directly from any one for HM. In other words, it is the typing rules that help the type system to reason about higher-ranked types, without actually complicating the type inference algorithm.

2.3 Dunfield-Krishnaswami Bidirectional Type System

Bidirectional typing is popular among new type systems. Compared with the ML-style systems, bidirectional typing additionally makes use of checking mode, which checks an expression against a known type. This is especially helpful in dealing with unannotated lambda functions, and when the type of the function can be inferred from the neighbor nodes in the syntax tree. For example,

$$\lambda f.\ (f\ 1, f\ 'c') : (\forall a.\ a \to a) \to (\text{Int} \to \text{Char})$$

is typeable in higher-ranked bidirectional systems, as the outer type annotation may act as if both the argument type and return type of the lambda is given. The Dunfield-Krishnaswami type system [Dunfield and Krishnaswami 2013], hereafter referred to as DK, extends the OL system by exploiting bidirectional typing. In this section, we only introduce the declarative system and leave the discussion of their algorithmic system to Chapters 3 and 4.

2.3.1 Declarative System

Syntax. The syntax of DK's declarative system [Dunfield and Krishnaswami 2013] is shown in Figure 3.1. A declarative type A is either the unit type 1, a type variable a, a universal quantification $\forall a.\ A$ or a function type $A \to B$. Nested universal quantifiers are allowed for types, but monotypes τ do not have any universal quantifier. Terms include a unit term $()$,

$\boxed{\Psi \vdash A}$ Well-formed declarative type

$$\frac{}{\Psi \vdash 1}\ \mathtt{wf_d unit} \qquad \frac{a \in \Psi}{\Psi \vdash a}\ \mathtt{wf_d var} \qquad \frac{\Psi \vdash A \quad \Psi \vdash B}{\Psi \vdash A \to B}\ \mathtt{wf_d \to} \qquad \frac{\Psi, a \vdash A}{\Psi \vdash \forall a.A}\ \mathtt{wf_d \forall}$$

$\boxed{\Psi \vdash e}$ Well-formed declarative expression

$$\frac{x : A \in \Psi}{\Psi \vdash x}\ \mathtt{wf_d tmvar} \qquad \frac{}{\Psi \vdash ()}\ \mathtt{wf_d tmunit} \qquad \frac{\Psi, x : A \vdash e}{\Psi \vdash \lambda x.\, e}\ \mathtt{wf_d abs}$$

$$\frac{\Psi \vdash e_1 \quad \Psi \vdash e_2}{\Psi \vdash e_1\ e_2}\ \mathtt{wf_d app} \qquad \frac{\Psi \vdash A \quad \Psi \vdash e}{\Psi \vdash (e : A)}\ \mathtt{wf_d anno}$$

$\boxed{\Psi \vdash A \leq B}$ Declarative subtyping

$$\frac{a \in \Psi}{\Psi \vdash a \leq a} \leq\mathtt{Var} \qquad \frac{}{\Psi \vdash 1 \leq 1} \leq\mathtt{Unit} \qquad \frac{\Psi \vdash B_1 \leq A_1 \quad \Psi \vdash A_2 \leq B_2}{\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2} \leq\!\to$$

$$\frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \leq B}{\Psi \vdash \forall a.\, A \leq B} \leq\forall\mathtt{L} \qquad \frac{\Psi, b \vdash A \leq B}{\Psi \vdash A \leq \forall b.\, B} \leq\forall\mathtt{R}$$

Figure 2.8: Declarative Well-formedness and Subtyping

variables x, lambda-functions $\lambda x.\, e$, applications $e_1\ e_2$ and annotations $(e : A)$. Contexts Ψ are sequences of type variable declarations and term variables with their types declared $x : A$.

Well-formedness Well-formedness of types and terms is shown at the top of Figure 2.8. The rules are standard and simply ensure that variables in types and terms are well-scoped.

Declarative Subtyping The bottom of Figure 2.8 shows DK's declarative subtyping judgment $\Psi \vdash A \leq B$, which was adopted from Odersky and Läufer [1996]. It compares the degree of polymorphism between A and B in DK's implicit polymorphic type system. Essentially, if A can always be instantiated to match any instantiation of B, then A is "at least as polymorphic as" B. We also say that A is "more polymorphic than" B and write $A \leq B$.

Subtyping rules $\leq\mathtt{Var}$, $\leq\mathtt{Unit}$ and $\leq\!\to$ handle simple cases that do not involve universal quantifiers. The subtyping rule for function types $\leq\!\to$ is standard, being covariant on the return type and contravariant on the argument type. Rule $\leq\forall\mathtt{R}$ states that if A is a subtype of B in the context Ψ, a, where a is fresh in A, then $A \leq \forall a.\, B$. Intuitively, if A is more general than $\forall a.\, B$ (where the universal quantifier already indicates that $\forall a.\, B$ is a general type), then A must instantiate to $[\tau/a]B$ for every τ.

$\boxed{\Psi \vdash e \Leftarrow A}$ e checks against input type A.

$\boxed{\Psi \vdash e \Rightarrow A}$ e synthesizes output type A.

$\boxed{\Psi \vdash A \bullet e \Rightarrow\!\!\!\Rightarrow C}$ Applying a function of type A to e synthesizes type C.

$$\frac{(x : A) \in \Psi}{\Psi \vdash x \Rightarrow A}\ \texttt{DeclVar} \qquad \frac{\Psi \vdash e \Rightarrow A \quad \Psi \vdash A \leq B}{\Psi \vdash e \Leftarrow B}\ \texttt{DeclSub}$$

$$\frac{\Psi \vdash A \quad \Psi \vdash e \Leftarrow A}{\Psi \vdash (e : A) \Rightarrow A}\ \texttt{DeclAnno} \qquad \frac{}{\Psi \vdash () \Leftarrow 1}\ \texttt{Decl1I} \qquad \frac{}{\Psi \vdash () \Rightarrow 1}\ \texttt{Decl1I}\Rightarrow$$

$$\frac{\Psi, a \vdash e \Leftarrow A}{\Psi \vdash e \Leftarrow \forall a.\ A}\ \texttt{Decl}\forall\texttt{I} \qquad \frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \bullet e \Rightarrow\!\!\!\Rightarrow C}{\Psi \vdash \forall a.\ A \bullet e \Rightarrow\!\!\!\Rightarrow C}\ \texttt{Decl}\forall\texttt{App}$$

$$\frac{\Psi, x : A \vdash e \Leftarrow B}{\Psi \vdash \lambda x.\ e \Leftarrow A \to B}\ \texttt{Decl}\to\texttt{I} \qquad \frac{\Psi \vdash \sigma \to \tau \quad \Psi, x : \sigma \vdash e \Leftarrow \tau}{\Psi \vdash \lambda x.\ e \Rightarrow \sigma \to \tau}\ \texttt{Decl}\to\texttt{I}\Rightarrow$$

$$\frac{\Psi \vdash e_1 \Rightarrow A \quad \Psi \vdash A \bullet e_2 \Rightarrow\!\!\!\Rightarrow C}{\Psi \vdash e_1\ e_2 \Rightarrow C}\ \texttt{Decl}\to\texttt{E} \qquad \frac{\Psi \vdash e \Leftarrow A}{\Psi \vdash A \to C \bullet e \Rightarrow\!\!\!\Rightarrow C}\ \texttt{Decl}\to\texttt{App}$$

Figure 2.9: Declarative Typing

The most interesting rule is $\leq\forall\texttt{L}$. If some instantiation of $\forall a.\ A$, $[\tau/a]A$, is a subtype of B, then $\forall a.\ A \leq B$. The monotype τ we used to instantiate a is *guessed* in this declarative rule, but the algorithmic system does not guess and defers the instantiation until it can determine the monotype deterministically. The fact that τ is a monotype rules out the possibility of polymorphic (or impredicative) instantiation. However this restriction ensures that the subtyping relation remains decidable. Allowing an arbitrary type (rather than a monotype) in rule $\leq\forall\texttt{L}$ is known to give rise to an undecidable subtyping relation [Tiuryn and Urzyczyn 1996]. Peyton Jones et al. [2007] also impose the restriction of predicative instantiation in their type system. Both systems are adopted by several practical programming languages.

Note that when we introduce a new binder in the premise, we implicitly pick a fresh one. This applies to rules such as $\texttt{wf}_\texttt{d}\forall$, $\texttt{wf}_\texttt{d}\texttt{abs}$, $\leq\forall\texttt{R}$, throughout the whole text.

Declarative Typing The bidirectional type system, shown in Figure 2.9, has three judgments. The checking judgment $\Psi \vdash e \Leftarrow A$ checks expression e against the type A in the context Ψ. The synthesis judgment $\Psi \vdash e \Rightarrow A$ synthesizes the type A of expression e in the context Ψ. The application judgment $\Psi \vdash A \bullet e \Rightarrow\!\!\!\Rightarrow C$ synthesizes the type C of the application of a function of type A (which could be polymorphic) to the argument e.

Many rules are standard. Rule `DeclVar` looks up term variables in the context. Rules `Decl1I` and `Decl1I`$\Rightarrow$ respectively check and synthesize the unit type. Rule `DeclAnno` synthesizes the annotated type A of the annotated expression $(e : A)$ and checks that e has type

$$
\begin{array}{lllll}
\text{Types} & \tau & ::= & 1 \mid a \mid \top \mid \bot \mid \tau_1 \to \tau_2 \mid \tau_1 \sqcup \tau_2 \mid \tau_1 \sqcap \tau_2 \\
\text{Positive Types} & \tau^+ & ::= & 1 \mid a \mid \bot \mid \tau_1^- \to \tau_2^+ \mid \tau_1^+ \sqcup \tau_2^+ \\
\text{Negative Types} & \tau^- & ::= & 1 \mid a \mid \top \mid \tau_1^+ \to \tau_2^- \mid \tau_1^- \sqcap \tau_2^-
\end{array}
$$

Figure 2.10: Types of MLsub

A. Checking an expression e against a polymorphic type $\forall a.\ A$ in the context Ψ succeeds if e checks against A in the extended context (Ψ, a). The subsumption rule `DeclSub` depends on the subtyping relation, and changes mode from checking to synthesis: if e synthesizes type A and $A \leq B$ (A is more polymorphic than B), then e checks against B. If a checking problem does not match any other rules, this rule can be applied to synthesize a type instead and then check whether the synthesized type entails the checked type. Lambda abstractions are the hardest construct of the bidirectional type system to deal with. Checking $\lambda x.\ e$ against function type $A \to B$ is easy: we check the body e against B in the context extended with $x : A$. However, synthesizing a lambda-function is a lot harder, and this type system only synthesizes monotypes $\sigma \to \tau$.

Application $e_1\ e_2$ is handled by Rule `Decl→E`, which first synthesizes the type A of the function e_1. If A is a function type $B \to C$, Rule `Decl→App` is applied; it checks the argument e_2 against B and returns type C. The synthesized type of function e_1 can also be polymorphic, of the form $\forall a.\ A$. In that case, we instantiate A to $[\tau/a]A$ with a monotype τ using according to Rule `Decl→I⇒`. If $[\tau/a]A$ is a function type, Rule `Decl→App` proceeds; if $[\tau/a]A$ is another universal quantified type, Rule `Decl→I⇒` is recursively applied.

To conclude, DK employs a bidirectional declarative type system. The type system is mostly syntax-directed, but there are still some guesses of monotypes that need to be resolved by an algorithm. We will continue to discuss DK's algorithm in Chapters 3 and 4.

2.4 MLsub

MLsub [Dolan and Mycroft 2017] extends the HM type system with object-oriented subtyping. In presence of subtyping, type inference does not simply handle equality during unification. Therefore, types are extended with lattice operations to express bounds properly. Furthermore, polar types are introduced to help separate input and output types, which simplifies the type inference algorithm. Like HM's type inference, MLsub always infers a principal type.

2.4.1 Types and Polar Types

In comparison to the type system of HM, types (Figure 2.10) now include $\top$ and $\bot$, as minimal components to support subtyping. Besides, the least-upper-bound ($\sqcup$) and greatest-lower-bound ($\sqcap$) lattice operations are used to represent a bound expressed by two types. For finite types, a distributive lattice can be defined via a set of equivalence classes of $\equiv$ [Dolan and Mycroft 2017]. The most interesting equations are the distributivity rule and rules for function types:

$$\begin{aligned}
\tau_1 \sqcup (\tau_2 \sqcap \tau_3) &\equiv (\tau_1 \sqcup \tau_2) \sqcap (\tau_1 \sqcup \tau_3) \\
\tau_1 \sqcap (\tau_2 \sqcup \tau_3) &\equiv (\tau_1 \sqcap \tau_2) \sqcup (\tau_1 \sqcap \tau_3) \\
(\tau_1 \to \tau_2) \sqcup (\tau_1' \to \tau_2') &\equiv (\tau_1 \sqcap \tau_1') \to (\tau_2 \sqcup \tau_2') \\
(\tau_1 \to \tau_2) \sqcap (\tau_1' \to \tau_2') &\equiv (\tau_1 \sqcup \tau_1') \to (\tau_2 \sqcap \tau_2')
\end{aligned}$$

The partial order $\tau_1 \leq \tau_2$ is defined as $\tau_1 \sqcup \tau_2 \equiv \tau_2$ or $\tau_1 \sqcap \tau_2 \equiv \tau_1$. $\top$ and $\bot$ are the least and greatest types. The above rules on function types imply the usual subtyping rule for function types, considering the definition of partial order:

$$\frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

Type schemes are not defined as the usual σ. Instead, a monotype τ already represents a type scheme by omitting the $\forall$ quantifiers—all the free type variables are implicitly generalized.

Recursive types play an important role regarding the principality of type inference, but we omit them for simplicity.

Polar Types Polar types are restrictions on the lattice operations; they should not occur arbitrarily in any position. Specifically, function outputs consist of types (τ_1, τ_2) from different branches, resulting in $\tau_1 \sqcup \tau_2$; a function input might be used in various ways (under different constraints), thus $\tau_1 \sqcap \tau_2$ is more suitable. In summary, $\sqcup$ only arises in return types, while $\sqcap$ only arises in argument types. Figure 2.10 formally defines the restriction, where positive types τ^+ describe return types, and negative types τ^- describe argument types.

An important consequence is that all the constraints are of the form $\tau^+ \leq \tau^-$, which represents the subtyping relation when using an output expression in a function application

as an argument. The following subtyping rules involving the lattice operations reflects their basic properties:

$$\frac{\tau_1^+ \le \tau^- \qquad \tau_2^+ \le \tau^-}{\tau_1^+ \sqcup \tau_2^+ \le \tau^-} \qquad \frac{\tau^+ \le \tau_1^- \qquad \tau^+ \le \tau_2^-}{\tau^- \le \tau_1^- \sqcap \tau_2^-} \quad .$$

Interestingly, the polar subtyping judgments avoids difficult judgments like $\tau_1 \sqcap \tau_2 \le \tau$ or $\tau \le \tau_1 \sqcup \tau_2$ through its syntactic restriction.

2.4.2 Biunification

Type inference for MLsub, similar to that for ML, is mainly a unification algorithm. However, in presence of subtyping, equality-based unification loses information about subtyping constraints.

For the atomic constraint $\widehat{\alpha} = \tau$ where $\widehat{\alpha} \notin \mathrm{FV}(\tau)$, the ML unification algorithm produces the substitution $[\tau/\widehat{\alpha}]$. In contrast, an MLsub atomic constraint might be $\widehat{\alpha} \le \tau$, and the substitution $[\tau/\widehat{\alpha}]$ treat the subtyping constraint as an equality constraint, which eliminates a whole set of possibilities.

Luckily, lattices in MLsub helps express subtyping constraints on types directly. The constraint $\widehat{\alpha} \le \tau$ ($\widehat{\alpha} \notin \mathrm{FV}(\tau)$) may produce the substitution $[\widehat{\alpha} \sqcap \tau/\widehat{\alpha}]$, since $\widehat{\alpha} \sqcap \tau \le \tau$. In the meantime, $\widehat{\alpha} \sqcap \tau$ does not lose any expressiveness: for any τ_0 s.t. $\tau_0 \le \tau$, picking $\widehat{\alpha} = \tau_0$ gives $\widehat{\alpha} \sqcap \tau = \tau_0$, and the substitution $[\widehat{\alpha} \sqcap \tau/\widehat{\alpha}]$ is equivalent to $[\tau_0/\widehat{\alpha}]$.

In presence of polar types, the biunification algorithm of MLsub produces a *bisubstitution* $[\widehat{\alpha} \sqcap \tau^-/\widehat{\alpha}^-]$ against the constraint $\widehat{\alpha} \le \tau^-$, where only negative occurrences are substituted, keeping polar types properly "polarized". For example, a positive type $\widehat{\alpha} \to \widehat{\alpha}$ becomes $(\widehat{\alpha} \sqcap \tau^-) \to \widehat{\alpha}$ under such substitution and remains a positive type. A more important fact is that this type is equivalent to the original type with the constraint $\widehat{\alpha} \le \tau^-$. Similarly, a constraint like $\tau^+ \le \widehat{\alpha}$ is reduced to a substitution $[\widehat{\alpha} \sqcup \tau^+/\widehat{\alpha}^+]$.

For example, the *choose* function is typed $\forall a.\ a \to a \to a$ in ML. However, MLsub might also infer an equivalent type $\forall a\ b.\ a \to b \to a \sqcup b$. One can easily read the MLsub type in a form where constraints are explicitly stated

$$\forall a\ b\ c.\ a \to b \to c \text{ where } a \le c, b \le c.$$

Therefore, MLsub encodes the constraints directly onto types with the help of the lattice operations. Furthermore, a simplification step is taken after the type inference algorithm, reducing the size and improving readablity of the type inferred.

As a result, biunification for MLsub extends unification for ML, accepting subtyping in addition to type schemes, while maintaining principality.

Part II

Higher-Ranked Type Inference Algorithms

3 Higher-Rank Polymorphism Subtyping Algorithm

In this chapter, we present a new algorithm for polymorphic subtyping with mechanical formalizations in the Abella theorem prover. There is little work on formalizing type inference algorithms before, especially for higher-ranked systems, due to the fact that environments and variable bindings are tricky to mechanize in theorem provers. In order to overcome the difficulty in formalization, we propose the novel algorithm by means of *worklist judgments*. Worklist judgments turn complicated global propagation of unification constraints into simple local substitutions. Moreover, we exploit several ideas in the recent inductive formulation of a type-inference algorithm by Dunfield and Krishnaswami [2013], which turn out to be useful for mechanization in a theorem prover.

Building on these ideas we develop a complete formalization of polymorphic subtyping in the Abella theorem prover. Moreover, we show that the algorithm is *sound*, *complete*, and *decidable* with respect to the well-known declarative formulation of polymorphic subtyping by Odersky and Läufer [1996]. While these meta-theoretical results are not new, as far as we know our work is the first to mechanically formalize them.

3.1 Overview: Polymorphic Subtyping

This section discusses Odersky and Läufer declarative subtyping rules further in depth, and identifies the challenges in formalizing a corresponding algorithmic version. Then the key ideas of our approach that address those challenges are introduced.

3.1.1 Declarative Polymorphic Subtyping

In implicitly polymorphic type systems, the subtyping relation compares the degree of polymorphism of types. In short, if a polymorphic type A can always be instantiated to any instantiation of B, then A is "at least as polymorphic as" B, or we just say that A is "more polymorphic than" B, or $A \leq B$.

There is a very simple declarative formulation of polymorphic subtyping due to Odersky and Läufer [1996]. The syntax of this declarative system is shown in Figure 3.1. Types, rep-

$$
\begin{array}{lllll}
\text{Type variables} & a, b & & & \\
\text{Types} & A, B, C & ::= & 1 \mid a \mid \forall a.A \mid A \to B \\
\text{Monotypes} & \tau & ::= & 1 \mid a \mid \tau_1 \to \tau_2 \\
\text{Contexts} & \Psi & ::= & \cdot \mid \Psi, a
\end{array}
$$

Figure 3.1: Syntax of Declarative System

$$\boxed{\Psi \vdash A}$$

$$\frac{}{\Psi \vdash 1}\ \mathtt{wf_d unit} \qquad \frac{a \in \Psi}{\Psi \vdash a}\ \mathtt{wf_d var} \qquad \frac{\Psi \vdash A \quad \Psi \vdash B}{\Psi \vdash A \to B}\ \mathtt{wf_d}\to \qquad \frac{\Psi, a \vdash A}{\Psi \vdash \forall a.A}\ \mathtt{wf_d}\forall$$

$$\boxed{\Psi \vdash A \leq B}$$

$$\frac{a \in \Psi}{\Psi \vdash a \leq a}\ \leq\mathtt{Var} \qquad \frac{}{\Psi \vdash 1 \leq 1}\ \leq\mathtt{Unit} \qquad \frac{\Psi \vdash B_1 \leq A_1 \quad \Psi \vdash A_2 \leq B_2}{\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2}\ \leq\to$$

$$\frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \leq B}{\Psi \vdash \forall a.A \leq B}\ \leq\forall\mathtt{L} \qquad \frac{\Psi, a \vdash A \leq B}{\Psi \vdash A \leq \forall a.B}\ \leq\forall\mathtt{R}$$

Figure 3.2: Well-formedness of Declarative Types and Declarative Subtyping

resented by A, B, C, are the unit type 1, type variables a, b, universal quantification $\forall a.A$ and function type $A \to B$. We allow nested universal quantifiers to appear in types, but not in monotypes. Contexts Ψ collect a list of type variables.

In Figure 3.2, we give the well-formedness and subtyping relation for the declarative system, which is identical to the subtyping relation introduced in Subsection 2.3.1.

3.1.2 Finding Solutions for Variable Instantiation

The declarative system specifies the behavior of subtyping relations, but is not directly implementable: the rule $\leq\forall\mathtt{L}$ requires guessing a monotype τ. The core problem that an algorithm for polymorphic subtyping needs to solve is to find an algorithmic way to compute the monotypes, instead of guessing them. An additional challenge is that the declarative rule $\leq\to$ splits one judgment into two, and the (partial) solutions found for existential variables when processing the first judgment should be transfered to the second judgment.

Dunfield and Krishnaswami's Approach An elegant algorithmic solution to computing the monotypes is presented by Dunfield and Krishnaswami [2013]. Their algorithmic subtyping judgment has the form:

$$\Psi \vdash A \leq B \dashv \Phi$$

A notable difference to the declarative judgment is the presence of a so-called *output context* Φ, which refines the *input context* Ψ with solutions for existential variables found while processing the two types being compared for subtyping. Both Ψ and Φ are *ordered contexts* with the same structure. Ordered contexts are particularly useful to keep track of the correct scoping for variables, and are a notable difference to older type-inference algorithms [Damas and Milner 1982] that use global unification variables or constraints collected in a set.

Output contexts are useful to transfer information across judgments in Dunfield and Krishnaswami's approach. For example, the algorithmic rule corresponding to $\leq\!\!\rightarrow$ in their approach is:

$$\frac{\Psi \vdash B_1 <: A_1 \dashv \Phi \quad \Phi \vdash [\Phi]A_2 <: [\Phi]B_2 \dashv \Phi'}{\Psi \vdash A_1 \to A_2 <: B_1 \to B_2 \dashv \Phi'} <:\!\!\rightarrow$$

The information gathered by the output context when comparing the input types of the functions for subtyping is transferred to the second judgment by becoming the new input context, while any solution derived from the first judgment is applied to the types of the second judgment.

Example If we want to show that $\forall a.a \to a$ is a subtype of $1 \to 1$, the declarative system will guess the proper $\tau = 1$ for Rule $\leq\forall\mathtt{L}$:

$$\frac{\cdot \vdash 1 \quad \cdot \vdash 1 \to 1 \leq 1 \to 1}{\cdot \vdash \forall a.a \to a \leq 1 \to 1} \leq\forall\mathtt{L}$$

Dunfield and Krishnaswami introduce an *existential variable*—denoted with $\widehat{\alpha}, \widehat{\beta}$—whenever a monotype τ needs to be guessed. Below is a sample derivation of their algorithm:

$$\dfrac{\dfrac{\dfrac{}{\widehat{\alpha} \vdash 1 \leq \widehat{\alpha} \dashv \widehat{\alpha} = 1}\,\mathtt{InstRSolve} \quad \dfrac{}{\widehat{\alpha} = 1 \vdash 1 \leq 1 \dashv \widehat{\alpha} = 1}\,<:\mathtt{Unit}}{\widehat{\alpha} \vdash \widehat{\alpha} \to \widehat{\alpha} \leq 1 \to 1 \dashv \widehat{\alpha} = 1}\,<:\!\!\rightarrow}{\cdot \vdash \forall a.a \to a \leq 1 \to 1 \dashv \cdot}\,<:\forall\mathtt{L}$$

The first step applies Rule $<:\forall\mathtt{L}$, which introduces a fresh existential variable, $\widehat{\alpha}$, and opens the left-hand-side $\forall$-quantifier with it. Next, Rule $<:\!\!\rightarrow$ splits the judgment in two. For the first branch, Rule `InstRSolve` satisfies $1 \leq \widehat{\alpha}$ by solving $\widehat{\alpha}$ to 1, and stores the solution in its output context. The output context of the first branch is used as the input context of

the second branch, and the judgment is updated according to current solutions. Finally, the second branch becomes a base case, and Rule <:`Unit` finishes the derivation, makes no change to the input context and propagates the output context back.

Dunfield and Krishnaswami's algorithmic specification is elegant and contains several useful ideas for a mechanical formalization of polymorphic subtyping. For example, *ordered contexts* and *existential variables* enable a purely inductive formulation of polymorphic subtyping. However, the binding/scoping structure of their algorithmic judgment is still fairly complicated and poses challenges when porting their approach to a theorem prover.

3.1.3 The Worklist Approach

We inherit Dunfield and Krishnaswami's ideas of ordered contexts, existential variables and the idea of solving those variables, but drop output contexts. Instead, our algorithmic rule has the form:

$$\Gamma \vdash \Omega$$

where Ω is a list of judgments $A \leq B$ instead of a single one. This judgment form, which we call *worklist judgment*, simplifies two aspects of Dunfield and Krishnaswami's approach.

Firstly, as already stated, there are no output contexts. Secondly, the form of the ordered contexts becomes simpler. The transfer of information across judgments is simplified because all judgments share the input context. Moreover, the order of the judgments in the list allows information discovered when processing the earlier judgments to be easily transferred to the later judgments. In the worklist approach the rule for function types is:

$$\frac{\Gamma \vdash B_1 \leq A_1; A_2 \leq B_2; \Omega}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2; \Omega} \leq_{\mathtt{a}}\!\to$$

The derivation of the previous example with the worklist approach is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash \cdot}\,\mathtt{a\_nil}}{\cdot \vdash 1 \leq 1; \cdot} \leq_{\mathtt{a}}\mathtt{unit}}{\widehat{\alpha} \vdash 1 \leq \widehat{\alpha}; \widehat{\alpha} \leq 1; \cdot} \leq_{\mathtt{a}}\mathtt{solve\_ex}}{\widehat{\alpha} \vdash \widehat{\alpha} \to \widehat{\alpha} \leq 1 \to 1; \cdot} \leq_{\mathtt{a}}\!\to}{\cdot \vdash \forall a.a \to a \leq 1 \to 1; \cdot} \leq_{\mathtt{a}}\forall\mathtt{L}$$

To derive $\cdot \vdash \forall a.a \to a \leq 1 \to 1$ with the worklist approach, we first introduce an existential variable and change the judgment to $\widehat{\alpha} \vdash \widehat{\alpha} \to \widehat{\alpha} \leq 1 \to 1; \cdot$. Then, we split the judgment in two for the function types and the derivation comes to $\widehat{\alpha} \vdash 1 \leq \widehat{\alpha}; \widehat{\alpha} \leq 1; \cdot$. When the first judgment is solved with $\widehat{\alpha} = 1$, we immediately remove $\widehat{\alpha}$ from the context, while propagating the solution as a substitution to the rest of the judgment list, resulting in $\cdot \vdash 1 \leq 1; \cdot$, which finishes the derivation in two trivial steps.

With this form of eager propagation, solutions no longer need to be recorded in contexts, simplifying the encoding and reasoning in a proof assistant.

Key Results Both the declarative and algorithmic systems are formalized in Abella. We have proven 3 important properties for this algorithm: *decidability*, ensuring that the algorithm always terminates; and *soundness* and *completeness*, showing the equivalence of the declarative and algorithmic systems.

3.2 A Worklist Algorithm for Polymorphic Subtyping

This section presents our algorithm for polymorphic subtyping. A novel aspect of our algorithm is the use of worklist judgments: a form of judgment that facilitates the propagation of information.

3.2.1 Syntax and Well-Formedness of the Algorithmic System

Figure 4.4 shows the syntax and the well-formedness judgment.

Existential Variables In order to solve the unknown types τ, the algorithmic system extends the declarative syntax of types with *existential variables* $\widehat{\alpha}$. They behave like unification variables, but are not globally defined. Instead, the ordered *algorithmic context*, inspired by Dunfield and Krishnaswami [2013], defines their scope. Thus the type τ represented by the corresponding existential variable is always bound in the corresponding declarative context Ψ.

Worklist judgments The form of our algorithmic judgments is non-standard. Our algorithm keeps track of an explicit list of outstanding work: the list Ω of (reified) *algorithmic judgments* of the form $A \leq B$, to which a substitution can be applied once and for all to propagate the solution of an existential variable.

Type variables	a, b		
Existential variables	$\widehat{\alpha}, \widehat{\beta}$		
Algorithmic types	A, B, C	$::=$	$1 \mid a \mid \widehat{\alpha} \mid \forall a.A \mid A \to B$
Algorithmic context	Γ	$::=$	$\cdot \mid \Gamma, a \mid \Gamma, \widehat{\alpha}$
Algorithmic judgments	Ω	$::=$	$\cdot \mid A \leq B; \Omega$

$$\boxed{\Gamma \vdash A}$$

$$\frac{}{\Gamma \vdash 1}\ \mathtt{wf_a unit} \qquad \frac{a \in \Gamma}{\Gamma \vdash a}\ \mathtt{wf_a var} \qquad \frac{\widehat{\alpha} \in \Gamma}{\Gamma \vdash \widehat{\alpha}}\ \mathtt{wf_a exvar}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}\ \mathtt{wf_a}{\to} \qquad \frac{\Gamma, a \vdash A}{\Gamma \vdash \forall a.A}\ \mathtt{wf_a}\forall$$

Figure 3.3: Syntax and Well-Formedness judgment for the Algorithmic System.

Hole Notation To facilitate context manipulation, we use the syntax $\Gamma[\Gamma_M]$ to denote a context of the form $\Gamma_L, \Gamma_M, \Gamma_R$ where Γ is the context $\Gamma_L, \bullet, \Gamma_R$ with a hole ($\bullet$). Hole notations with the same name implicitly share the same Γ_L and Γ_R. A multi-hole notation like $\Gamma[\widehat{\alpha}][\widehat{\beta}]$ means $\Gamma_1, \widehat{\alpha}, \Gamma_2, \widehat{\beta}, \Gamma_3$.

3.2.2 Algorithmic Subtyping

The algorithmic subtyping judgment, defined in Figure 4.5, has the form $\Gamma \vdash \Omega$, where Ω collects multiple subtyping judgments $A \leq B$. The algorithm treats Ω as a worklist. In every step it takes one task from the worklist for processing, possibly pushes some new tasks on the worklist, and repeats this process until the list is empty. This last and single base case is handled by Rule `a_nil`. The remaining rules all deal with the first task in the worklist. Logically we can discern 3 groups of rules.

Firstly, we have five rules that are similar to those in the declarative system, mostly just adapted to the worklist style. For instance, Rule $\leq_{\mathtt{a}}{\to}$ consumes one judgment and pushes two to the worklist. A notable difference with the declarative Rule $\leq\forall\mathtt{L}$ is that Rule $\leq_{\mathtt{a}}\forall\mathtt{L}$ requires no guessing of a type τ to instantiate the polymorphic type $\forall a.A$, but instead introduces an existential variable $\widehat{\alpha}$ to the context and to A. In accordance with the declarative system, where the monotype τ should be bound in the context Ψ, here $\widehat{\alpha}$ should only be solved to a monotype bound in Γ. More generally, for any algorithmic context $\Gamma[\widehat{\alpha}]$, the algorithmic variable $\widehat{\alpha}$ can only be solved to a monotype that is well-formed with respect to Γ_L.

$$\boxed{\Gamma \vdash \Omega}$$

$$\frac{}{\Gamma \vdash \cdot} \leq_{\mathtt{a}}\mathtt{nil}$$

$$\frac{\Gamma \vdash \Omega}{\Gamma \vdash 1 \leq 1; \Omega} \leq_{\mathtt{a}}\mathtt{unit} \qquad \frac{a \in \Gamma \qquad \Gamma \vdash \Omega}{\Gamma \vdash a \leq a; \Omega} \leq_{\mathtt{a}}\mathtt{var} \qquad \frac{\widehat{\alpha} \in \Gamma \qquad \Gamma \vdash \Omega}{\Gamma \vdash \widehat{\alpha} \leq \widehat{\alpha}; \Omega} \leq_{\mathtt{a}}\mathtt{exvar}$$

$$\frac{\Gamma \vdash B_1 \leq A_1; A_2 \leq B_2; \Omega}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2; \Omega} \leq_{\mathtt{a}}\to$$

$$\frac{\widehat{\alpha} \text{ fresh} \qquad \Gamma, \widehat{\alpha} \vdash [\widehat{\alpha}/a]A \leq B; \Omega}{\Gamma \vdash \forall a.A \leq B; \Omega} \leq_{\mathtt{a}}\forall\mathtt{L} \qquad \frac{b \text{ fresh} \qquad \Gamma, b \vdash A \leq B; \Omega}{\Gamma \vdash A \leq \forall b.B; \Omega} \leq_{\mathtt{a}}\forall\mathtt{R}$$

$$\frac{\widehat{\alpha} \notin FV(A) \cup FV(B) \quad \Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2] \vdash \widehat{\alpha}_1 \to \widehat{\alpha}_2 \leq A \to B; [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}]\Omega}{\Gamma[\widehat{\alpha}] \vdash \widehat{\alpha} \leq A \to B; \Omega} \leq_{\mathtt{a}}\mathtt{instL}$$

$$\frac{\widehat{\alpha} \notin FV(A) \cup FV(B) \quad \Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2] \vdash A \to B \leq \widehat{\alpha}_1 \to \widehat{\alpha}_2; [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}]\Omega}{\Gamma[\widehat{\alpha}] \vdash A \to B \leq \widehat{\alpha}; \Omega} \leq_{\mathtt{a}}\mathtt{instR}$$

$$\frac{\Gamma[\widehat{\alpha}][] \vdash [\widehat{\alpha}/\widehat{\beta}]\Omega}{\Gamma[\widehat{\alpha}][\widehat{\beta}] \vdash \widehat{\alpha} \leq \widehat{\beta}; \Omega} \leq_{\mathtt{a}}\mathtt{solve\_ex} \qquad \frac{\Gamma[\widehat{\alpha}][] \vdash [\widehat{\alpha}/\widehat{\beta}]\Omega}{\Gamma[\widehat{\alpha}][\widehat{\beta}] \vdash \widehat{\beta} \leq \widehat{\alpha}; \Omega} \leq_{\mathtt{a}}\mathtt{solve\_ex}'$$

$$\frac{\Gamma[a][] \vdash [a/\widehat{\beta}]\Omega}{\Gamma[a][\widehat{\beta}] \vdash a \leq \widehat{\beta}; \Omega} \leq_{\mathtt{a}}\mathtt{solve\_var} \qquad \frac{\Gamma[a][] \vdash [a/\widehat{\beta}]\Omega}{\Gamma[a][\widehat{\beta}] \vdash \widehat{\beta} \leq a; \Omega} \leq_{\mathtt{a}}\mathtt{solve\_var}'$$

$$\frac{\Gamma[] \vdash [1/\widehat{\alpha}]\Omega}{\Gamma[\widehat{\alpha}] \vdash \widehat{\alpha} \leq 1; \Omega} \leq_{\mathtt{a}}\mathtt{solve\_unit} \qquad \frac{\Gamma[] \vdash [1/\widehat{\alpha}]\Omega}{\Gamma[\widehat{\alpha}] \vdash 1 \leq \widehat{\alpha}; \Omega} \leq_{\mathtt{a}}\mathtt{solve\_unit}'$$

Figure 3.4: Algorithmic Subtyping

Secondly, Rules $\leq_{\mathtt{a}}\mathtt{instL}$ and $\leq_{\mathtt{a}}\mathtt{instR}$ partially instantiate existential types $\widehat{\alpha}$, to function types. The domain and range of the new function type are undetermined: they are set to two fresh existential variables $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$. To make sure that $\widehat{\alpha}_1 \to \widehat{\alpha}_2$ has the same scope as $\widehat{\alpha}$, the new variables $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ are inserted in the same position in the context where the old variable $\widehat{\alpha}$ was. To propagate the instantiation to the remainder of the worklist, $\widehat{\alpha}$ is substituted for $\widehat{\alpha}_1 \to \widehat{\alpha}_2$ in Ω. The *occurs-check* side-condition is necessary to prevent a diverging infinite instantiation. For example $1 \to \widehat{\alpha} \leq \widehat{\alpha}$ would diverge with no such check.

Thirdly, in the remaining six rules an existential variable can be immediately solved. Each of the six similar rules removes an existential variable from the context, performs a substitution on the remainder of the worklist and continues.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\widehat{\alpha}_1 \vdash \cdot}\,\texttt{a\_nil}}{\widehat{\alpha}_1 \vdash 1 \leq 1; \cdot}\,\leq_{\texttt{a}}\texttt{unit}}{\widehat{\alpha}_1, \widehat{\alpha}_2 \vdash \widehat{\alpha}_1 \leq \widehat{\alpha}_2; 1 \leq 1; \cdot}\,\leq_{\texttt{a}}\texttt{solve\_ex}}{\widehat{\alpha}_1, \widehat{\alpha}_2, \widehat{\beta} \vdash \widehat{\alpha}_1 \leq \widehat{\beta}; \widehat{\beta} \leq \widehat{\alpha}_2; 1 \leq 1; \cdot}\,\leq_{\texttt{a}}\texttt{solve\_ex}}{\widehat{\alpha}_1, \widehat{\alpha}_2, \widehat{\beta} \vdash \widehat{\beta} \to \widehat{\beta} \leq \widehat{\alpha}_1 \to \widehat{\alpha}_2; 1 \leq 1; \cdot}\,\leq_{\texttt{a}}\to}{\widehat{\alpha}, \widehat{\beta} \vdash \widehat{\beta} \to \widehat{\beta} \leq \widehat{\alpha}; 1 \leq 1; \cdot}\,\leq_{\texttt{a}}\texttt{instR}}{\widehat{\alpha} \vdash \forall a.\, a \to a \leq \widehat{\alpha}; 1 \leq 1; \cdot}\,\leq_{\texttt{a}}\forall\texttt{L}}{\widehat{\alpha} \vdash \widehat{\alpha} \to 1 \leq (\forall a.\, a \to a) \to 1; \cdot}\,\leq_{\texttt{a}}\to}{\cdot \vdash \forall a.\, a \to 1 \leq (\forall a.\, a \to a) \to 1; \cdot}\,\leq_{\texttt{a}}\forall\texttt{L}$$

Figure 3.5: A Success Derivation for the Algorithmic Subtyping Relation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\textit{stuck}}\,?}{\widehat{\alpha}, b \vdash \widehat{\alpha} \leq b; \cdot}\,\leq_{\texttt{a}}\forall\texttt{R}}{\widehat{\alpha} \vdash \widehat{\alpha} \leq \forall b.\, b; \cdot}\,\leq_{\texttt{a}}\texttt{unit}}{\widehat{\alpha} \vdash 1 \leq 1; \widehat{\alpha} \leq \forall b.\, b; \cdot}\,\leq_{\texttt{a}}\to}{\widehat{\alpha} \vdash 1 \to \widehat{\alpha} \leq 1 \to \forall b.\, b; \cdot}\,\leq_{\texttt{a}}\forall\texttt{L}}{\cdot \vdash \forall a.\, 1 \to a \leq 1 \to \forall b.\, b; \cdot}$$

Figure 3.6: A Failing Derivation for the Algorithmic Subtyping Relation

The algorithm on judgment list is designed to share the context across all judgments. However, the declarative system does not share a single context in its derivation. This gap is filled by strengthening and weakening lemmas of both systems, where most of them are straightforward to prove, except for the strengthening lemma of the declarative system, which is a little trickier.

Example We illustrate the subtyping rules through a sample derivation in Figure 3.5, which shows that that $\forall a.\, a \to 1 \leq (\forall a.\, a \to a) \to 1$. Thus the derivation starts with an empty context and a judgment list with only one element.

In step 1, we have only one judgment, and that one has a top-level $\forall$ on the left hand side. So the only choice is rule $\leq_{\texttt{a}}\forall\texttt{L}$, which opens the universally quantified type with an unknown existential variable $\widehat{\alpha}$. Variable $\widehat{\alpha}$ will be solved later to some monotype that is well-formed within the context before $\widehat{\alpha}$. That is, the empty context $\cdot$ in this case. In step 2, rule $\leq_{\texttt{a}}\to$ is applied to the worklist, splitting the first judgment into two. Step 3 is similar to step 1, where the left-hand-side $\forall$ of the first judgment is opened according to rule $\leq_{\texttt{a}}\forall\texttt{L}$

with a fresh existential variable. In step 4, the first judgment has an arrow on the left hand side, but the right-hand-side type is an existential variable. It is obvious that $\widehat{\alpha}$ should be solved to a monotype of the form $\sigma \to \tau$. Rule `instR` implements this, but avoids guessing σ and τ by "splitting" $\widehat{\alpha}$ into two existential variables, $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$, which will be solved to some σ and τ later. Step 5 applies Rule $\leq_{\mathtt{a}}\!\to$ again. Notice that after the split, $\widehat{\beta}$ appears in two judgments. When the first $\widehat{\beta}$ is solved during any step of the derivation, the next $\widehat{\beta}$ will be substituted by that solution. This propagation mechanism ensures the consistent solution of the variables, while keeping the context as simple as possible. Steps 6 and 7 solve existential variables. The existential variable that is right-most in the context is always solved in terms of the other. Therefore in step 6, $\widehat{\beta}$ is solved in terms of $\widehat{\alpha}_1$, and in step 7, $\widehat{\alpha}_2$ is solved in terms of $\widehat{\alpha}_1$. Additionally, in step 6, when $\widehat{\beta}$ is solved, the substitution $[\widehat{\alpha}_1/\widehat{\beta}]$ is propagated to the rest of the judgment list, and thus the second judgment becomes $\widehat{\alpha}_1 \leq \widehat{\alpha}_2$. Steps 8 and 9 trivially finish the derivation. Notice that $\widehat{\alpha}_1$ is not instantiated at the end. This means that any well-scoped instantiation is fine.

A Failing Derivation We illustrate the role of ordered contexts through another example: $\forall a.\ 1 \to a \leq 1 \to \forall b.\ b$. From the declarative perspective, a should be instantiated to some τ first, then b is introduced to the context, so that $b \notin FV(\tau)$. As a result, we cannot find τ such that $\tau \leq b$. Figure 3.6 shows the algorithmic derivation, which also fails due to the scoping—$\widehat{\alpha}$ is introduced earlier than b, thus it cannot be solved to b.

3.3 Metatheory

This section presents the three main meta-theoretical results that we have proved in Abella. The first two are soundness and completeness of our algorithm with respect to Odersky and Läufer's declarative subtyping. The third result is our algorithm's decidability.

3.3.1 Transfer to the Declarative System

To state the correctness of the algorithmic subtyping rules, Figure 3.7 introduces two *transfer* judgments to relate the declarative and the algorithmic system. The first judgment, transfer of contexts $\Gamma \to \Psi$, removes existential variables from the algorithmic context Γ to obtain a declarative context Ψ. The second judgment, transfer of the judgment list $\Gamma \mid \Omega \rightsquigarrow \Omega'$, replaces all occurrences of existential variables in Ω by well-scoped mono-types. Notice that this judgment is not decidable, i.e. a pair of Γ and Ω may be related with multiple Ω'. However, if there exists some substitution that transforms Ω to Ω', and each subtyping judgment in Ω' holds, we know that Ω is potentially satisfiable.

$$\boxed{\Gamma \to \Psi}$$

$$\frac{}{\cdot \to \cdot}\ {\to}\cdot \qquad \frac{\Gamma \to \Psi}{\Gamma, a \to \Psi, a}\ {\to}\texttt{var} \qquad \frac{\Gamma \to \Psi}{\Gamma, \widehat{\alpha} \to \Psi}\ {\to}\texttt{exvar}$$

$$\boxed{\Gamma \mid \Omega \rightsquigarrow \Omega'}$$

$$\frac{}{\cdot \mid \Omega \rightsquigarrow \Omega}\ {\rightsquigarrow}\cdot \qquad \frac{\Gamma \mid \Omega \rightsquigarrow \Omega'}{\Gamma, a \mid \Omega \rightsquigarrow \Omega'}\ {\rightsquigarrow}\texttt{var} \qquad \frac{\Gamma \to \Psi \quad \Psi \vdash \tau \quad \Gamma \mid [\tau/\widehat{\alpha}]\Omega \rightsquigarrow \Omega'}{\Gamma, \widehat{\alpha} \mid \Omega \rightsquigarrow \Omega'}\ {\rightsquigarrow}\texttt{exvar}$$

Figure 3.7: Transfer Rules

The following two lemmas generalize Rule $\rightsquigarrow$`exvar` from substituting the first existential variable to substituting any existential variable.

Lemma 3.1 (Insert). *If* $\Gamma \to \Psi$ *and* $\Psi \vdash \tau$ *and* $\Gamma, \Gamma_1 \mid [\tau/\widehat{\alpha}]\Omega \rightsquigarrow \Omega'$ *, then* $\Gamma, \widehat{\alpha}, \Gamma_1 \mid \Omega \rightsquigarrow \Omega'$.

Lemma 3.2 (Extract). *If* $\Gamma, \widehat{\alpha}, \Gamma_1 \mid \Omega \rightsquigarrow \Omega'$*, then* $\exists \tau$ *s.t.* $\Gamma \to \Psi, \Psi \vdash \tau$ *and* $\Gamma, \Gamma_1 \mid [\tau/\widehat{\alpha}]\Omega \rightsquigarrow \Omega'$.

In order to match the shape of algorithmic subtyping relation for the following proofs, we define a relation $\Psi \vdash \Omega$ for the declarative system, meaning that all the declarative judgments hold under context Ψ.

Definition 1 (Declarative Subtyping Worklist)**.**

$$\Psi \vdash \Omega := \forall (A \leq B) \in \Omega, \Psi \vdash A \leq B$$

3.3.2 Soundness

Our algorithm is sound with respect to the declarative specification. For any derivation of a list of algorithmic judgments $\Gamma \vdash \Omega$, we can find a valid transfer $\Gamma \mid \Omega \rightsquigarrow \Omega'$ such that all judgments in Ω' hold in Ψ, with $\Gamma \to \Psi$.

Theorem 3.3 (Soundness). *If* $\Gamma \vdash \Omega$ *and* $\Gamma \to \Psi$*, then there exists* Ω'*, s.t.* $\Gamma \mid \Omega \rightsquigarrow \Omega'$ *and* $\Psi \vdash \Omega'$.

The proof proceeds by induction on the derivation of $\Gamma \vdash \Omega$, finished off by appropriate applications of the insertion and extraction lemmas.

3.3.3 Completeness

Completeness of the algorithm means that any declarative derivation has an algorithmic counterpart.

Theorem 3.4 (Completeness). *If* $\Psi \vdash \Omega'$ *and* $\Gamma \to \Psi$ *and* $\Gamma \mid \Omega \rightsquigarrow \Omega'$*, then* $\Gamma \vdash \Omega$*.*

The proof proceeds by induction on the derivation of $\Psi \vdash \Omega'$. As the declarative system does not involve information propagation across judgments, the induction can focus on the subtyping derivation of the first judgment without affecting other judgments. The difficult cases correspond to the $\leq_{\texttt{a}}\texttt{instL}$ and $\leq_{\texttt{a}}\texttt{instR}$ rules. When the proof by induction on $\Psi \vdash \Omega'$ reaches the $\leq\!\!\to$ case, the first declarative judgment has a shape like $A_1 \to A_2 \leq B_1 \to B_2$. One of the possible cases for the first corresponding algorithmic judgment is $\widehat{\alpha} \leq A \to B$. However, the case analysis does not indicate that $\widehat{\alpha}$ is fresh in A and B. Thus we cannot apply Rule $\leq_{\texttt{a}}\texttt{instL}$ and make use of the induction hypothesis. The following lemma helps us out in those cases: it rules out subtypings with infinite types as solutions (e.g. $\widehat{\alpha} \leq 1 \to \widehat{\alpha}$) and guarantees that $\widehat{\alpha}$ is free in A and B.

Lemma 3.5 (Prune Transfer for Instantiation). *If* $\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2; \Omega'$ *and* $\Gamma \to \Psi$ *and* $\Gamma \mid (\widehat{\alpha} \leq A \to B; \Omega) \rightsquigarrow (A_1 \to A_2 \leq B_1 \to B_2; \Omega')$*, then* $\widehat{\alpha} \notin FV(A) \cup FV(B)$*.*

A similar lemma holds for the symmetric case $(A \to B \leq \widehat{\alpha}; \Omega)$.

3.3.4 Decidability

The third key result for our algorithm is decidability.

Theorem 3.6 (Decidability). *Given any well-formed judgment list* Ω *under* Γ*, it is decidable whether* $\Gamma \vdash \Omega$ *or not.*

We have proven this theorem by means of a lexicographic group of induction measurements $\langle |\Omega|_\forall, |\Gamma|_{\widehat{\alpha}}, |\Omega|_\to \rangle$ on the worklist Ω and algorithmic context Γ. The worklist measures $|\cdot|_\forall$ and $|\cdot|_\to$ count the number of universal quantifiers and function types respectively.

Definition 2 (Worklist Measures).

$$
\begin{array}{rcl@{\qquad}rcl}
|1|_\forall = |a|_\forall = |\widehat{\alpha}|_\forall & = & 0 & |1|_\to = |a|_\to = |\widehat{\alpha}|_\to & = & 0 \\
|A \to B|_\forall & = & |A|_\forall + |B|_\forall & |A \to B|_\to & = & |A|_\to + |B|_\to + 1 \\
|\forall x.A|_\forall & = & |A|_\forall + 1 & |\forall x.A|_\to & = & |A|_\to \\
|\Omega|_\forall & = & \sum_{A \leq B \in \Omega} |A|_\forall + |B|_\forall & |\Omega|_\to & = & \sum_{A \leq B \in \Omega} |A|_\to + |B|_\to
\end{array}
$$

The context measure $|\cdot|_{\widehat{\alpha}}$ counts the number of unsolved existential variables.

Definition 3 (Context Measure).

$$|\cdot|_{\widehat{\alpha}} = 0 \qquad |\Gamma, a|_{\widehat{\alpha}} = |\Gamma|_{\widehat{\alpha}} \qquad |\Gamma, \widehat{\alpha}|_{\widehat{\alpha}} = |\Gamma|_{\widehat{\alpha}} + 1$$

It is not difficult to see that all but two of the algorithm's rules decrease one of the three measures. The two exceptions are the Rules $\leq_{\mathtt{a}}\mathtt{instL}$ and $\leq_{\mathtt{a}}\mathtt{instR}$; both increment the number of existential variables and the number of function types without affecting the number of universal quantifiers. To handle these rules, we handle a special class of judgments, which we call *instantiation judgments* Ω_i, separately. They take the form:

Definition 4 (Ω_i).

$$\Omega_i := \cdot \mid \widehat{\alpha} \leq A; \Omega_i' \mid A \leq \widehat{\alpha}; \Omega_i' \quad \text{where } \widehat{\alpha} \notin FV(A) \cup FV(\Omega_i')$$

These instantiation judgments are these ones consumed and produced by the Rules $\leq_{\mathtt{a}}\mathtt{instL}$ and $\leq_{\mathtt{a}}\mathtt{instR}$. The following lemma handles their decidability.

Lemma 3.7 (Instantiation Decidability). *For any context* Γ *and judgment list* Ω_i, Ω*, it is decidable whether* $\Gamma \vdash \Omega_i, \Omega$ *if both of the conditions hold*

1) $\forall \Gamma', \Omega'$ *s.t.* $|\Omega'|_\forall < |\Omega_i, \Omega|_\forall$*, it is decidable whether* $\Gamma' \vdash \Omega'$*.*

2) $\forall \Gamma', \Omega'$ *s.t.* $|\Omega'|_\forall = |\Omega_i, \Omega|_\forall$ *and* $|\Gamma'|_{\widehat{\alpha}} = |\Gamma|_{\widehat{\alpha}} - |\Omega_i|$*, it is decidable whether* $\Gamma' \vdash \Omega'$*.*

In other words, for any instantiation judgment prefix Ω_i, the algorithm either reduces the number of $\forall$'s or solves one existential variable per instantiation judgment. The proof of this lemma is by induction on the measure $2 * |\Omega_i|_\to + |\Omega_i|$ of the instantiation judgment list.

In summary, the decidability theorem can be shown through a lexicographic group of induction measurements $\langle |\Omega|_\forall, |\Omega|_{\widehat{\alpha}}, |\Omega|_\to \rangle$. The critical case is that, whenever we encounter an instantiation judgment at the front of the worklist, we refer to Lemma 3.7, which reduces the number of unsolved variables by consuming that instantiation judgment, or reduces the number of $\forall$-quantifiers. Other cases are relatively straightforward.

3.4 The Choice of Abella

We have chosen the Abella (v2.0.5) proof assistant [Gacek 2008] to develop our formalization. Our development is only based on the reasoning logic of Abella, and does not make use of its specification logic. Abella is particularly helpful due to its built-in support for variable bindings, and its λ-tree syntax [Miller 2000] is a form of HOAS, which helps with

the encoding and reasoning about substitutions. For instance, the type $\forall x.x \to a$ is encoded as `all (x\ arrow x a)`, where `x\ arrow x a` is a lambda abstraction in Abella. An opening $[b/x](x \to a)$ is encoded as an application `(x\ arrow x a) b`, which can be simplified(evaluated) to `arrow b a`. Name supply and freshness conditions are controlled by the ∇-quantifier. The expression `nabla x, F` means that `x` is a unique variable in `F`, i.e. it is different from any other names occurring elsewhere. Such variables are called nominal constants. They can be of any type, in other words, every type may contain an unlimited number of such atomic nominal constants.

Encoding of the Declarative System As a concrete example, our declarative context and well-formedness rules are encoded as follows.

```
Kind ty    type.
Type i     ty.                % the unit type
Type all   (ty → ty) → ty.    % forall-quantifier
Type arrow ty → ty → ty.      % function type
Type bound ty → o.            % variable collection in contexts

Define env : olist → prop by
  env nil;
  nabla x, env (bound x :: E) := env E.

Define wft : olist → ty → prop by
  wft E i;
  nabla x, wft (E x) x := nabla x, member (bound x) (E x);
  wft E (arrow A B) := wft E A ∧ wft E B;
  wft E (all A) := nabla x, wft (bound x :: E) (A x).
```

We use the type `olist` just as normal list of `o` with two constructors, namely `nil : olist` and `(::) : o → olist → olist`, where `o` purely means "the element type of `olist`". The `member : o → olist → prop` relation is also pre-defined. The second case of the relation `wft` states rule $\mathtt{wf_d var}$. The encoding `(E x)` basically means that the context *may* contain `x`. If we write `(E x)` as `E`, then the context should not contain `x`, and both `wft E x` and `member (bound x) E` make no sense. Instead, we treat `E : ty → olist` as an *abstract structure* of a context, such as `x\ bound x :: bound a :: nil`. For the fourth case of the relation `wft`, the type $\forall x.A$ in our target language is expressed as `(all A)`, and its opening A, `(A x)`.

Encoding of the Algorithmic System In terms of the algorithmic system, notably, Abella handles the $\leq_{\mathtt{a}}\mathtt{instL}$ and $\leq_{\mathtt{a}}\mathtt{instR}$ rules in a nice way:

```
% sub_alg_list : enva → [subty_judgment] → prop
```

File(s)	SLOC	# of Theorems	Description
olist.thm, nat.thm	303	55	Basic data structures
higher.thm, order.thm	164	15	Declarative system
higher_alg.thm	618	44	Algorithmic system
trans.thm	411	46	Transfer
sound.thm	166	2	Soundness theorem
depth.thm	143	12	Definition of depth
complete.thm	626	28	Lemmas and Completeness theorem
decidable.thm	1077	53	Lemmas and Decidability theorem
Total	3627	267	(33 definitions in total)

Figure 3.8: Statistics for the proof scripts

```
Define subal : olist → olist → prop by
  subal E nil;
  subal E (subt i i :: Exp) := subal E Exp;
  % some cases omitted ...
  % <: instL
  nabla x, subal (E x) (subt x (arrow A B) :: Exp x) :=
    exists E1 E2 F, nabla x y z, append E1 (exvar x :: E2) (E x) ∧
      append E1 (exvar y :: exvar z :: E2) (F y z) ∧
      subal (F y z) (subt (arrow y z) (arrow A B) :: Exp (arrow y z));
  % <: instR is symmetric to <: instL, omitted here
  % other cases omitted ...
```

Thanks to the way Abella deals with nominal constants, the pattern `subt x (arrow A B)` implicitly states that $x \notin FV(A) \wedge x \notin FV(B)$. If the condition were not required, we would have encoded the pattern as `subt x (arrow (A x) (B x))` instead.

3.4.1 Statistics and Discussion

Some basic statistics on our proof script are shown in Figure 3.8. The proof consists of 3627 lines of code with a total of 33 definitions and 267 theorems. We have to mention that Abella provides few built-in tactics and does not support user-defined ones, and we would reduce significant lines of code if Abella provided more handy tactics. Moreover, the definition of natural numbers, the plus operation and less-than relation are defined within our proof due to Abella's lack of packages. However, the way Abella deals with name bindings is very helpful for type system formalizations and substitution-intensive formalizations, such as this one.

4 A Type-Inference Algorithm for Higher-Ranked Polymorphism

This chapter presents the first fully mechanized formalization of the metatheory for higher-ranked polymorphic type inference. Following the woirklist subtyping algorithm in the previous chapter, we address several drawbacks and extend the technique to DK's bidirectional type system [Dunfield and Krishnaswami 2013]. We chose DK's type system because it is quite elegant, well-documented and it comes with detailed manually written proofs. Furthermore, the system is adopted in practice by a few real implementations of functional languages, including PureScript and Unison. The DK type system has two variants: a declarative and an algorithmic one. The two variants have been *manually* proved to be *sound*, *complete* and *decidable*. We present a mechanical formalization in the Abella theorem prover [Gacek 2008] for DK's declarative type system using a different algorithm.

Challenges While our initial goal was to formalize both DK's declarative and algorithmic versions, we faced technical challenges with the latter, prompting us to find an alternative formulation.

The first challenge that we faced were missing details as well as a few incorrect proofs and lemmas in DK's formalization. While DK's original formalization comes with very well-written manual proofs, there are still several details missing. These complicate the task of writing a mechanically verified proof. Moreover, some proofs and lemmas are wrong and, in some cases, it is not clear to us how to fix them.

Despite the problems in DK's manual formalization, we believe that these problems do not invalidate their work and that their results are still true. In fact we have nothing but praise for their detailed and clearly written metatheory and proofs, which provided invaluable help to our own work. We expect that for most non-trivial manual proofs similar problems exist, so this should not be understood as a sign of sloppiness on their part. Instead, it should be an indicator that reinforces the arguments for mechanical formalizations: manual formalizations are error-prone due to the multiple tedious details involved in them. There are several other examples of manual formalizations that were found to have similar problems. For ex-

ample, Klein et al. [2012] mechanized formalizations in Redex for nine ICFP 2009 papers and all were found to have mistakes.

Another challenge was variable binding. Type inference algorithms typically do not rely simply on local environments but instead propagate information across judgments. While local environments are well-studied in mechanical formalizations, there is little work on how to deal with the complex forms of binding employed by type inference algorithms in theorem provers. To keep track of variable scoping, DK's algorithmic version employs input and output contexts to track information that is discovered through type inference. However, modeling output contexts in a theorem prover is non-trivial.

Due to those two challenges, our work takes a different approach by refining and extending the idea of *worklist judgments* in Chapter 3, where we mechanically formalized an algorithm for *polymorphic subtyping* [Odersky and Läufer 1996]. The algorithm eliminates output contexts compared to DK's algorithm, and therefore the problem of variable binding is solved. Unfortunately, the subtyping algorithm cannot be naively extended to support the bidirectional type system. A key innovation in the new algorithm to be introduced in this chapter is how to adapt the idea of worklist judgments to *inference judgments*, which are not needed for polymorphic subtyping, but are necessary for type inference. The idea is to use a *continuation passing style* to enable the transfer of inferred information across judgments. A further refinement to the idea of worklist judgments is the *unification between ordered contexts [Dunfield and Krishnaswami 2013; Gundry et al. 2010] and worklists*. This enables precise scope tracking of free variables in judgments. Furthermore it avoids the duplication of context information across judgments in worklists that occurs in other techniques [Abel and Pientka 2011; Reed 2009]. Despite the use of a different algorithm, we prove the same results as DK, although with significantly different proofs and proof techniques. The calculus and its metatheory have been fully formalized in the Abella theorem prover [Gacek 2008].

4.1 Overview

This section starts with a discussion on DK's declarative type system. Then it introduces several techniques that have been used in algorithmic formulations, and which have influenced our own algorithmic design. Finally we introduce the novelties of our new algorithm. In particular the support for inference judgments in worklist judgments, and a new form of worklist judgment that unifies *ordered contexts* and the worklists themselves.

Type variables	a, b		
Types	A, B, C	$::=$	$1 \mid a \mid \forall a.\ A \mid A \to B$
Monotypes	τ, σ	$::=$	$1 \mid a \mid \tau \to \sigma$
Expressions	e	$::=$	$x \mid () \mid \lambda x.\ e \mid e_1\ e_2 \mid (e : A)$
Contexts	Ψ	$::=$	$\cdot \mid \Psi, a \mid \Psi, x : A$

Figure 4.1: Syntax of Declarative System

$\boxed{\Psi \vdash A}$ Well-formed declarative type

$$\frac{}{\Psi \vdash 1}\ \mathtt{wf_d unit} \qquad \frac{a \in \Psi}{\Psi \vdash a}\ \mathtt{wf_d var} \qquad \frac{\Psi \vdash A \quad \Psi \vdash B}{\Psi \vdash A \to B}\ \mathtt{wf_d}{\to} \qquad \frac{\Psi, a \vdash A}{\Psi \vdash \forall a.A}\ \mathtt{wf_d}{\forall}$$

$\boxed{\Psi \vdash e}$ Well-formed declarative expression

$$\frac{x : A \in \Psi}{\Psi \vdash x}\ \mathtt{wf_d tmvar} \qquad \frac{}{\Psi \vdash ()}\ \mathtt{wf_d tmunit} \qquad \frac{\Psi, x : A \vdash e}{\Psi \vdash \lambda x.\ e}\ \mathtt{wf_d abs}$$

$$\frac{\Psi \vdash e_1 \quad \Psi \vdash e_2}{\Psi \vdash e_1\ e_2}\ \mathtt{wf_d app} \qquad \frac{\Psi \vdash A \quad \Psi \vdash e}{\Psi \vdash (e : A)}\ \mathtt{wf_d anno}$$

$\boxed{\Psi \vdash A \leq B}$ Declarative subtyping

$$\frac{a \in \Psi}{\Psi \vdash a \leq a}\ \leq\mathtt{Var} \qquad \frac{}{\Psi \vdash 1 \leq 1}\ \leq\mathtt{Unit} \qquad \frac{\Psi \vdash B_1 \leq A_1 \quad \Psi \vdash A_2 \leq B_2}{\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2}\ \leq{\to}$$

$$\frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \leq B}{\Psi \vdash \forall a.\ A \leq B}\ \leq\forall\mathtt{L} \qquad \frac{\Psi, b \vdash A \leq B}{\Psi \vdash A \leq \forall b.\ B}\ \leq\forall\mathtt{R}$$

Figure 4.2: Declarative Well-formedness and Subtyping

4.1.1 DK's Declarative System

Subsection 2.3.1 introduces DK's declarative subtyping and typing systems. We also duplicate the rules here for the convenience of the reader.

Overlapping Rules A problem that we found in the declarative system is that some of the rules overlap with each other. Declarative subtyping rules $\leq\forall\mathtt{L}$ and $\leq\forall\mathtt{R}$ both match the conclusion $\Psi \vdash \forall a.\ A \leq \forall a.\ B$. In such a case, choosing $\leq\forall\mathtt{R}$ first is always better, since we introduce the type variable a to the context earlier, which gives more flexibility on the choice of τ. The declarative typing rule `DeclSub` overlaps with both `Decl`$\forall$`I` and `Decl`$\to$`I`.

$\boxed{\Psi \vdash e \Leftarrow A}$ e checks against input type A.

$\boxed{\Psi \vdash e \Rightarrow A}$ e synthesizes output type A.

$\boxed{\Psi \vdash A \bullet e \Rightarrow\!\!\!\Rightarrow C}$ Applying a function of type A to e synthesizes type C.

$$\frac{(x : A) \in \Psi}{\Psi \vdash x \Rightarrow A}\ \texttt{DeclVar} \qquad \frac{\Psi \vdash e \Rightarrow A \quad \Psi \vdash A \leq B}{\Psi \vdash e \Leftarrow B}\ \texttt{DeclSub}$$

$$\frac{\Psi \vdash A \quad \Psi \vdash e \Leftarrow A}{\Psi \vdash (e : A) \Rightarrow A}\ \texttt{DeclAnno} \qquad \frac{}{\Psi \vdash () \Leftarrow 1}\ \texttt{Decl1I} \qquad \frac{}{\Psi \vdash () \Rightarrow 1}\ \texttt{Decl1I}\Rightarrow$$

$$\frac{\Psi, a \vdash e \Leftarrow A}{\Psi \vdash e \Leftarrow \forall a.\ A}\ \texttt{Decl}\forall\texttt{I} \qquad \frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \bullet e \Rightarrow\!\!\!\Rightarrow C}{\Psi \vdash \forall a.\ A \bullet e \Rightarrow\!\!\!\Rightarrow C}\ \texttt{Decl}\forall\texttt{App}$$

$$\frac{\Psi, x : A \vdash e \Leftarrow B}{\Psi \vdash \lambda x.\ e \Leftarrow A \to B}\ \texttt{Decl}\to\texttt{I} \qquad \frac{\Psi \vdash \sigma \to \tau \quad \Psi, x : \sigma \vdash e \Leftarrow \tau}{\Psi \vdash \lambda x.\ e \Rightarrow \sigma \to \tau}\ \texttt{Decl}\to\texttt{I}\Rightarrow$$

$$\frac{\Psi \vdash e_1 \Rightarrow A \quad \Psi \vdash A \bullet e_2 \Rightarrow\!\!\!\Rightarrow C}{\Psi \vdash e_1\ e_2 \Rightarrow C}\ \texttt{Decl}\to\texttt{E} \qquad \frac{\Psi \vdash e \Leftarrow A}{\Psi \vdash A \to C \bullet e \Rightarrow\!\!\!\Rightarrow C}\ \texttt{Decl}\to\texttt{App}$$

Figure 4.3: Declarative Typing

However, we argue that more specific rules are always the best choices, i.e. `Decl`$\forall$`I` and `Decl`$\to$`I` should have higher priority than `DeclSub`.

For example, $\Psi \vdash \lambda x.\ x \Leftarrow \forall a.\ a \to a$ succeeds if derived from Rule `Decl`$\forall$`I`:

$$\frac{\dfrac{\dfrac{\Psi, a, x : a \vdash x \Leftarrow a}{\Psi, a \vdash \lambda x.\ x \Leftarrow a \to a}\ \texttt{Decl}\to\texttt{I}}{\Psi \vdash \lambda x.\ x \Leftarrow \forall a.\ a \to a}\ \texttt{Decl}\forall\texttt{I}}{},$$

but fails when applied to `DeclSub`:

$$\frac{\dfrac{\Psi \vdash \sigma \to \tau \quad \Psi, x : \sigma \vdash e \Leftarrow \tau}{\Psi \vdash \lambda x.\ x \Rightarrow \sigma \to \tau}\ \texttt{Decl}\to\texttt{I}\Rightarrow \qquad \dfrac{\dfrac{\dfrac{\textit{Impossible!}}{a \notin \mathrm{FV}(\sigma)}\ ?}{\Psi, a \vdash a \leq \sigma \quad \Psi, a \vdash \tau \leq a}\ \leq\!\to}{\Psi, a \vdash \sigma \to \tau \leq a \to a}\ \leq\!\forall\texttt{R}}{\Psi \vdash \sigma \to \tau \leq \forall a.\ a \to a}\ \texttt{DeclSub}}{\Psi \vdash \lambda x.\ x \Leftarrow \forall a.\ a \to a}.$$

Rule `Decl`$\to$`I` is also better at handling higher-order types. When the lambda-expression to be inferred has a polymorphic input type, such as $\forall a.\ a \to a$, `DeclSub` may not derive

some judgments. For example, $\Psi, id : \forall a.\ a \to a \vdash \lambda f.\ f\ id\ (f\ ()) \Leftarrow (\forall a.\ a \to a) \to 1$ requires the argument of the lambda-expression to be a polymorphic type, otherwise it could not be applied to both id and $()$. If Rule `DeclSub` was chosen for derivation, the type of its argument is restricted by Rule `Decl`$\to$`I`$\Rightarrow$, which is not a polymorphic type. By contrast, Rule `Decl`$\to$`I` keeps the polymorphic argument type $\forall a.\ a \to a$, and will successfully derive the judgment.

We will come back to this topic in Section 4.3.2 and formally derive a system without overlapping rules.

4.1.2 DK's Algorithm

DK's algorithm version revolves around their notion of *algorithmic context*.

Algorithmic Contexts $\quad \Gamma, \Delta, \Theta \quad ::= \quad \cdot \mid \Gamma, a \mid \Gamma, x : A \mid \Gamma, \widehat{\alpha} \mid \Gamma, \widehat{\alpha} = \tau \mid \Gamma, \blacktriangleright_{\widehat{\alpha}}$

In addition to the regular (universally quantified) type variables a, the algorithmic context also contains *existential* type variables $\widehat{\alpha}$. These are placeholders for monotypes τ that are still to be determined by the inference algorithm. When the existential variable is "solved", its entry in the context is replaced by the assignment $\widehat{\alpha} = \tau$. A context application on a type, denoted by $[\Gamma]A$, substitutes all solved existential type variables in Γ with their solutions on type A.

All algorithmic judgments thread an algorithmic context. They have the form $\Gamma \vdash \ldots \dashv \Delta$, where Γ is the input context and Δ is the output context: $\Gamma \vdash A \leq B \dashv \Delta$ for subtyping, $\Gamma \vdash e \Leftarrow A \dashv \Delta$ for type checking, and so on. The output context is a functional update of the input context that records newly introduced existentials and solutions.

Solutions are incrementally propagated by applying the algorithmic output context of a previous task as substitutions to the next task. This can be seen in the subsumption rule:

$$\frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A \leq [\Theta]B \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}\ \texttt{DK\_Sub}$$

The inference task yields an output context Θ which is applied as a substitution to the types A and B before performing the subtyping check to propagate any solutions of existential variables that appear in A and B.

Markers for scoping. The sequential order of entries in the algorithmic context, in combination with the threading of contexts, does not perfectly capture the scoping of all existen-

tial variables. For this reason the DK algorithm uses scope markers $\blacktriangleright_{\widehat{\alpha}}$ in a few places. An example is given in the following rule:

$$\frac{\Gamma, \blacktriangleright_{\widehat{\alpha}}, \widehat{\alpha} \vdash [\widehat{\alpha}/a]A \leq B \dashv \Delta, \blacktriangleright_{\widehat{\alpha}}, \Theta}{\Gamma \vdash \forall a.\, A \leq B \dashv \Delta}\ \texttt{DK\_}{\leq}\forall\texttt{L}$$

To indicate that the scope of $\widehat{\alpha}$ is local to the subtyping check $[\widehat{\alpha}/a]A \leq B$, the marker is pushed onto its input stack and popped from the output stack together with the subsequent part Θ, which may refer to $\widehat{\alpha}$. (Remember that later entries may refer to earlier ones, but not vice versa.) This way $\widehat{\alpha}$ does not escape its scope.

At first sight, the DK algorithm would seem to be a good basis for mechanization. After all, it comes with a careful description and extensive manual proofs. Unfortunately, we ran into several obstacles that have prompted us to formulate a different, more mechanization-friendly algorithm.

Broken Metatheory While going through the manual proofs of DK's algorithm, we found several problems. Indeed, two proofs of lemmas—Lemma 19 (Extension Equality Preservation) and Lemma 14 (Subsumption)— wrongly apply induction hypotheses in several cases. Fortunately, we have found simple workarounds that fix these proofs without affecting the appeals to these lemmas.

More seriously, we have also found a lemma that simply does not hold: Lemma 29 (Parallel Admissibility)[1]. This lemma is a cornerstone of the two metatheoretical results of the algorithm, soundness, and completeness with respect to the declarative system. In particular, both instantiation soundness (i.e. a part of subtyping soundness) and typing completeness directly require the broken lemma. Moreover, Lemma 54 (Typing Extension) also requires the broken lemma and is itself used 13 times in the proof of typing soundness and completeness. Unfortunately, we have not yet found a way to fix this problem.

In what follows, we briefly discuss the problem through counterexamples. False lemmas are found in the manual proofs of DK' two papers [Dunfield and Krishnaswami 2013] and [Dunfield and Krishnaswami 2019].

- In the first paper, Lemma 29 on page 9 of its appendix says:

  Lemma 4.1 (Parallel Admissibility of [Dunfield and Krishnaswami 2013])**.**
  If $\Gamma_L \longrightarrow \Delta_L$ *and* $\Gamma_L, \Gamma_R \longrightarrow \Delta_L, \Delta_R$ *then:*

[1] Ningning Xie found the issue with Lemma 29 in 2016 on an earlier attempt to mechanically formalize DK's algorithm. The authors acknowledged the problem after we contacted them through email. Although they briefly mentioned that it should be possible to use a weaker lemma instead they did not go into details.

1. $\Gamma_L, \widehat{\alpha}, \Gamma_R \longrightarrow \Delta_L, \widehat{\alpha}, \Delta_R$
2. *If* $\Delta_L \vdash \tau'$ *then* $\Gamma_L, \widehat{\alpha}, \Gamma_R \longrightarrow \Delta_L, \widehat{\alpha} = \tau', \Delta_R$.
3. *If* $\Gamma_L \vdash \tau$ *and* $\Delta_L \vdash \tau'$ *and* $[\Delta_L]\tau = [\Delta_L]\tau'$, *then* $\Gamma_L, \widehat{\alpha} = \tau, \Gamma_R \longrightarrow \Delta_L, \widehat{\alpha} = \tau', \Delta_R$.

We give a counter-example to this lemma:
Pick $\Gamma_L = \cdot, \Gamma_R = \widehat{\beta}, \Delta_L = \widehat{\beta}, \Delta_R = \cdot$, then both conditions $\cdot \longrightarrow \widehat{\beta}$ and $\widehat{\beta} \longrightarrow \widehat{\beta}$ hold, but the first conclusion $\widehat{\alpha}, \widehat{\beta} \longrightarrow \widehat{\beta}, \widehat{\alpha}$ does not hold.

- In the second paper, as an extended work to the first paper, Lemma 26 on page 22 of its appendix says:

  Lemma 4.2 (Parallel Admissibility of [Dunfield and Krishnaswami 2019]).
  If $\Gamma_L \longrightarrow \Delta_L$ *and* $\Gamma_L, \Gamma_R \longrightarrow \Delta_L, \Delta_R$ *then:*

  1. $\Gamma_L, \widehat{\alpha} : \kappa, \Gamma_R \longrightarrow \Delta_L, \widehat{\alpha} : \kappa, \Delta_R$
  2. *If* $\Delta_L \vdash \tau' : \kappa$ *then* $\Gamma_L, \widehat{\alpha} : \kappa, \Gamma_R \longrightarrow \Delta_L, \widehat{\alpha} : \kappa = \tau', \Delta_R$.
  3. *If* $\Gamma_L \vdash \tau : \kappa$ *and* $\Delta_L \vdash \tau' types$ *and* $[\Delta_L]\tau = [\Delta_L]\tau'$, *then* $\Gamma_L, \widehat{\alpha} : \kappa = \tau, \Gamma_R \longrightarrow \Delta_L, \widehat{\alpha} : \kappa = \tau', \Delta_R$.

  A similar counter-example is given:
  Pick $\Gamma_L = \cdot, \Gamma_R = \widehat{\beta} : \star, \Delta_L = \widehat{\beta} : \star, \Delta_R = \cdot$, then both conditions $\cdot \longrightarrow \widehat{\beta} : \star$ and $\widehat{\beta} : \star \longrightarrow \widehat{\beta} : \star$ hold, but the first conclusion $\widehat{\alpha} : \kappa, \widehat{\beta} : \star \longrightarrow \widehat{\beta} : \star, \widehat{\alpha} : \kappa$ does not hold.

Complex Scoping and Propagation Besides the problematic lemmas in DK's metatheory, there are other reasons to look for an alternative algorithmic formulation of the type system that is more amenable to mechanization. Indeed, two aspects that are particularly challenging to mechanize are the scoping of universal and existential type variables, and the propagation of the instantiation of existential type variables across judgments. DK is already quite disciplined on these accounts, in particular compared to traditional constraint-based type-inference algorithms like Algorithm $\mathcal{W}$ [Milner 1978] which features an implicit global scope for all type variables. Indeed, DK uses its explicit and ordered context Γ that tracks the relative scope of universal and existential variables and it is careful to only instantiate existential variables in a well-scoped manner.

Moreover, DK's algorithm carefully propagates instantiations by recording them into the context and threading this context through all judgments. While this works well on paper, this approach is still fairly involved and thus hard to reason about in a mechanized setting.

Indeed, the instantiations have to be recorded in the context and are applied incrementally to each remaining judgment in turn, rather than immediately to all remaining judgments at once. Also, as we have mentioned above, the stack discipline of the ordered contexts does not mesh well with the use of output contexts; explicit marker entries are needed in two places to demarcate the end of an existential variable's scope. Actually, we found a scoping issue related to the subsumption rule `DK_Sub`, which might cause existential variables to leak across judgments. In Section 4.4.1 we give a detailed discussion.

The complications of scoping and propagation are compelling reasons to look for another algorithm that is easier to reason about in a mechanized setting.

4.1.3 Judgment Lists

To avoid the problem of incrementally applying a substitution to remaining tasks, we can find inspiration in the formulation of constraint solving algorithms. For instance, the well-known unification algorithm by Martelli and Montanari [1982] decomposes the problem of unifying two terms $s \doteq t$ into a number of related unification problems between pairs of terms $s_i \doteq t_i$. These smaller problems are not tackled independently, but kept together in a set S. The algorithm itself is typically formulated as a small-step-style state transition system $S \rightarrowtail S'$ that rewrites the set of unification problems until it is in solved form or until a contradiction has been found. For instance, the variable elimination rule is written as:

$$x \doteq t, S \quad \rightarrowtail \quad x \doteq t, [t/x]S \qquad\qquad if\ x \notin t \text{ and } x \in S$$

Because the whole set is explicitly available, the variable x can be simultaneously substituted.

In the above unification problem, all variables are implicitly bound in the same global scope. Some constraint solving algorithms for Hindley-Milner type inference use similar ideas, but are more careful tracking the scopes of variables [Pottier and Rémy 2005]. However they have separate phases for constraint generation and solving. Recent unification algorithms for dependently-typed languages are also more explicit about scopes. For instance, Reed [2009] represents a unification problem as $\Delta \vdash P$ where P is a set of equations to be solved and Δ is a (modal) context. Abel and Pientka [2011] even use multiple contexts within a unification problem. Such a problem is denoted $\Delta \Vdash \mathcal{K}$ where the meta-context Δ contains all the typings of meta-variables in the constraint set $\mathcal{K}$. The latter consists of constraints like $\Psi \vdash M = N : C$ that are equipped with their individual context Ψ. While accurately tracking the scoping of regular and meta-variables, this approach is not ideal because it repeatedly copies contexts when decomposing a unification problem, and later it has to substitute solutions into these copies.

4.1.4 Single-Context Worklist Algorithm for Subtyping

As we have seen in Chapter 3, an algorithm based on *worklist judgments* is mechanized and shown to be correct with respect to DK's declarative subtyping judgment. This approach overcomes some problems in DK's algorithmic formulation by using a worklist-based judgment of the form

$$\Gamma \vdash \Omega$$

where Ω is a worklist (or sequence) of subtyping problems of the form $A \leq B$. The judgment is defined by case analysis on the first element of Ω and recursively processes the worklist until it is empty. Using both a single common ordered context Γ and a worklist Ω greatly simplifies propagating the instantiation of type variables in one subtyping problem to the remaining ones.

Unfortunately, this work does not solve all problems. In particular, it has two major limitations that prevent it from scaling to the whole DK system.

Scoping Garbage Firstly, the single common ordered context Γ does not accurately reflect the type and unification variables currently in scope. Instead, it is an overapproximation that steadily accrues variables, and only drops those unification variables that are instantiated. In other words, Γ contains "garbage" that is no longer in scope. This complicates establishing the relation with the declarative system.

No Inference Judgments Secondly, and more importantly, the approach only deals with a judgment for *checking* whether one type is the subtype of another. While this may not be so difficult to adapt to the *checking* mode of term typing $\Gamma \vdash e \Leftarrow A$, it clearly lacks the functionality to support the *inference* mode of term typing $\Gamma \vdash e \Rightarrow A$. Indeed, the latter requires not only the communication of unification variable instantiations from one typing problem to another, but also an inferred type.

4.1.5 Algorithmic Type Inference for Higher-Ranked Types: Key Ideas

Our new algorithmic type system builds on the work above, but addresses the open problems.

Scope Tracking We avoid scoping garbage by blending the ordered context and the worklist into a single syntactic sort Γ, our algorithmic worklist. This algorithmic worklist interleaves (type and term) variables with *work* like checking or inferring types of expressions. The interleaving keeps track of the variable scopes in the usual, natural way: each variable is in scope of anything in front of it in the worklist. If there is nothing in front, the variable is

no longer needed and can be popped from the worklist. This way, no garbage (i.e. variables out-of-scope) builds up.

$$\begin{array}{rrcl} \text{Algorithmic judgment chain} & \omega & ::= & A \leq B \mid e \Leftarrow A \mid e \Rightarrow_a \omega \mid A \bullet e \Rrightarrow_a \omega \\ \text{Algorithmic worklist} & \Gamma & ::= & \cdot \mid \Gamma, a \mid \Gamma, \widehat{\alpha} \mid \Gamma, x : A \mid \Gamma \Vdash \omega \end{array}$$

For example, suppose that the top judgment of the following worklist checks the identity function against $\forall a.\ a \to a$:

$$\Gamma \Vdash \lambda x.\ x \Leftarrow \forall a.\ a \to a$$

To proceed, two auxiliary variables a and x are introduced to help the type checking:

$$\Gamma, a, x : a \Vdash x \Leftarrow a$$

which will be satisfied after several steps, and the worklist becomes

$$\Gamma, a, x : a$$

Since the variable declarations $a, x : a$ are only used for a judgment already processed, they can be safely removed, leaving the remaining worklist Γ to be further reduced.

Our worklist can be seen as an all-in-one stack, containing variable declarations and subtyping/ typing judgments. The stack is an enriched form of ordered context, and it has a similar variable scoping scheme.

Inference Judgments To express the DK's inference judgments, we use a novel form of work entries in the worklist: our algorithmic judgment chains ω. In its simplest form, such a judgment chain is just a check, like a subtyping check $A \leq B$ or a term typecheck $e \Leftarrow A$. However, the non-trivial forms of chains capture an inference task together with the work that depends on its outcome. For instance, a type inference task takes the form $e \Rightarrow_a \omega$. This form expresses that we need to infer the type, say A, for expression e and use it in the chain ω by substituting it for the placeholder type variable a. Notice that such a binds a fresh type variable for the inner chain ω, which behaves similarly to the variable declarations in the context.

Take the following worklist as an example

$$\widehat{\alpha} \Vdash \underline{(\lambda x.\ x)\ () \Rightarrow_a a \leq \widehat{\alpha}}, x : \widehat{\alpha}, \widehat{\beta} \Vdash \underline{\widehat{\alpha} \leq \widehat{\beta}} \Vdash \underline{\widehat{\beta} \leq 1}$$

There are three (underlined) judgment chains in the worklist, where the first and second judgment chains (from the right) are two subtyping judgments, and the third judgment chain, $(\lambda x.\ x)\ ()\ \Rightarrow_a\ a \leq \widehat{\alpha}$, is a sequence of an inference judgment followed by a subtyping judgment.

The algorithm first analyses the two subtyping judgments and will find the best solutions $\widehat{\alpha} = \widehat{\beta} = 1$ (please refer to Figure 4.5 for detailed derivations). Then we substitute every instance of $\widehat{\alpha}$ and $\widehat{\beta}$ with 1, so the variable declarations can be safely removed from the worklist. Now we reduce the worklist to the following state

$$\cdot \Vdash \underline{(\lambda x.\ x)\ ()\ \Rightarrow_a\ a \leq 1}, x : 1$$

which has a term variable declaration as the top element. After removing the garbage term variable declaration from the worklist, we process the last remaining inference judgment $(\lambda x.\ x)\ ()\ \Rightarrow\ ?$, with the unit type 1 as its result. Finally, the last judgment becomes $1 \leq 1$, a trivial base case.

4.2 Algorithmic System

This section introduces a novel algorithmic system that implements DK's declarative specification. The new algorithm extends the idea of worklists in Chapter 3 in two ways. Firstly, unlike its worklists, the scope of variables is precisely tracked and variables do not escape their scope. This is achieved by unifying algorithmic contexts and the worklists themselves. Secondly, our algorithm also accounts for the type system (and not just subtyping). To deal with inference judgments that arise in the type system we employ a *continuation passing style* to enable the transfer of inferred information across judgments in a worklist.

4.2.1 Syntax and Well-Formedness

Figure 4.4 shows the syntax and well-formedness judgments used by the algorithm. Similarly to the declarative system the well-formedness rules are unsurprising and merely ensure well-scopedness.

Existential Variables The algorithmic system inherits the syntax of terms and types from the declarative system. It only introduces one additional feature. In order to find unknown types τ in the declarative system, the algorithmic system extends the declarative types A with *existential variables* $\widehat{\alpha}, \widehat{\beta}$. They behave like unification variables, but their scope is restricted by their position in the algorithmic worklist rather than being global. Any existential variable $\widehat{\alpha}$ should only be solved to a type that is well-formed with respect to the worklist to

$$
\begin{array}{rlll}
\text{Existential variables} & \widehat{\alpha}, \widehat{\beta} & & \\
\text{Algorithmic types} & A, B, C & ::= & 1 \mid a \mid \forall a.\, A \mid A \to B \mid \widehat{\alpha} \\
\text{Algorithmic judgment chain} & \omega & ::= & A \leq B \mid e \Leftarrow A \mid e \Rightarrow_a \omega \mid A \bullet e \Rrightarrow_a \omega \\
\text{Algorithmic worklist} & \Gamma & ::= & \cdot \mid \Gamma, a \mid \Gamma, \widehat{\alpha} \mid \Gamma, x : A \mid \Gamma \Vdash \omega
\end{array}
$$

$\boxed{\Gamma \vdash A}$ Well-formed algorithmic type

$$\frac{}{\Gamma \vdash 1}\ \texttt{wf\_unit} \qquad \frac{a \in \Gamma}{\Gamma \vdash a}\ \texttt{wf\_var} \qquad \frac{\widehat{\alpha} \in \Gamma}{\Gamma \vdash \widehat{\alpha}}\ \texttt{wf\_exvar} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \to B}\ \texttt{wf\_}\to \qquad \frac{\Gamma, a \vdash A}{\Gamma \vdash \forall a.A}\ \texttt{wf\_}\forall$$

$\boxed{\Gamma \vdash e}$ Well-formed algorithmic expression

$$\frac{x : A \in \Gamma}{\Gamma \vdash x}\ \texttt{wf\_tmvar} \qquad \frac{}{\Gamma \vdash ()}\ \texttt{wf\_tmunit} \qquad \frac{\Gamma, x : A \vdash e}{\Gamma \vdash \lambda x.\, e}\ \texttt{wf\_abs}$$

$$\frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash e_1\ e_2}\ \texttt{wf\_app} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash e}{\Gamma \vdash (e : A)}\ \texttt{wf\_anno}$$

$\boxed{\Gamma \vdash \omega}$ Well-formed algorithmic judgment

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \leq B}\ \texttt{wf}\leq \qquad \frac{\Gamma \vdash e \qquad \Gamma \vdash A}{\Gamma \vdash e \Leftarrow A}\ \texttt{wf}\Leftarrow$$

$$\frac{\Gamma \vdash e \qquad \Gamma, a \vdash \omega}{\Gamma \vdash e \Rightarrow_a \omega}\ \texttt{wf}\Rightarrow \qquad \frac{\Gamma \vdash A \qquad \Gamma, a \vdash \omega \qquad \Gamma \vdash e}{\Gamma \vdash A \bullet e \Rrightarrow_a \omega}\ \texttt{wf}\Rrightarrow$$

$\boxed{\mathrm{wf}\,\Gamma}$ Well-formed algorithmic worklist

$$\frac{}{\mathrm{wf}\,\cdot}\ \texttt{wf}\cdot \qquad \frac{\mathrm{wf}\,\Gamma}{\mathrm{wf}\,\Gamma, a}\ \texttt{wf}_\texttt{a} \qquad \frac{\mathrm{wf}\,\Gamma}{\mathrm{wf}\,\Gamma, \widehat{\alpha}}\ \texttt{wf}_{\widehat{\alpha}} \qquad \frac{\mathrm{wf}\,\Gamma \qquad \Gamma \vdash A}{\mathrm{wf}\,\Gamma, x : A}\ \texttt{wf}_\texttt{of} \qquad \frac{\mathrm{wf}\,\Gamma \qquad \Gamma \vdash \omega}{\mathrm{wf}\,\Gamma \Vdash \omega}\ \texttt{wf}_\omega$$

Figure 4.4: Extended Syntax and Well-Formedness for the Algorithmic System

which $\widehat{\alpha}$ has been added. The point is that the monotype τ, represented by the corresponding existential variable, is always well-formed under the corresponding declarative context. In other words, the position of $\widehat{\alpha}$'s reflects the well-formedness restriction of τ's.

Judgment Chains Judgment chains ω, or judgments for short, are the core components of our algorithmic type-checking. There are four kinds of judgments in our system: subtyping ($A \leq B$), checking ($e \Leftarrow A$), inference ($e \Rightarrow_a \omega$) and application inference ($A \bullet e \Rightarrow\!\!\!\Rightarrow_a \omega$). Subtyping and checking are relatively simple, since their result is only success or failure. However both inference and application inference return a type that is used in subsequent judgments. We use a continuation-passing-style encoding to accomplish this. For example, the judgment chain $e \Rightarrow_a (a \leq B)$ contains two judgments: first we want to infer the type of the expression e, and then check if that type is a subtype of B. The *unknown* type of e is represented by a type variable a, which is used as a placeholder in the second judgment to denote the type of e.

Worklist Judgments Our algorithm has a non-standard form. We combine traditional contexts and judgment(s) into a single sort, the *worklist* Γ. The worklist is an *ordered* collection of both variable bindings and judgments. The order captures the scope: only the objects that come after a variable's binding in the worklist can refer to it. For example, $[\cdot, a, x : a \Vdash x \Leftarrow a]$ is a valid worklist, but $[\cdot \Vdash \underline{x} \Leftarrow \underline{a}, x : \underline{a}, a]$ is not (the underlined symbols refer to out-of-scope variables).

Hole Notation We use the syntax $\Gamma[\Gamma_M]$ to denote the worklist $\Gamma_L, \Gamma_M, \Gamma_R$, where Γ is the worklist $\Gamma_L, \bullet, \Gamma_R$ with a hole ($\bullet$). Hole notations with the same name implicitly share the same structure Γ_L and Γ_R. A multi-hole notation splits the worklist into more parts. For example, $\Gamma[\widehat{\alpha}][\widehat{\beta}]$ means $\Gamma_1, \widehat{\alpha}, \Gamma_2, \widehat{\beta}, \Gamma_3$.

4.2.2 Algorithmic System

The algorithmic typing reduction rules, defined in Figure 4.5, have the form $\Gamma \longrightarrow \Gamma'$. The reduction process treats the worklist as a stack. In every step, it pops the first judgment from the worklist for processing and possibly pushes new judgments onto the worklist. The syntax $\Gamma \longrightarrow^* \Gamma'$ denotes multiple reduction steps.

$$\frac{}{\Gamma \longrightarrow^* \Gamma}\ \longrightarrow^*\texttt{id} \qquad \frac{\Gamma \longrightarrow \Gamma_1 \quad \Gamma_1 \longrightarrow^* \Gamma'}{\Gamma \longrightarrow^* \Gamma'}\ \longrightarrow^*\texttt{step}$$

In the case that $\Gamma \longrightarrow^* \cdot$ this corresponds to successful type checking.

$\boxed{\Gamma \longrightarrow \Gamma'}$ Γ reduces to Γ'.

$$\Gamma, a \longrightarrow_1 \Gamma \qquad \Gamma, \widehat{\alpha} \longrightarrow_2 \Gamma \qquad \Gamma, x : A \longrightarrow_3 \Gamma$$

$$\begin{aligned}
\Gamma \Vdash 1 \leq 1 &\longrightarrow_4 \Gamma \\
\Gamma \Vdash a \leq a &\longrightarrow_5 \Gamma \\
\Gamma \Vdash \widehat{\alpha} \leq \widehat{\alpha} &\longrightarrow_6 \Gamma \\
\Gamma \Vdash A_1 \to A_2 \leq B_1 \to B_2 &\longrightarrow_7 \Gamma \Vdash A_2 \leq B_2 \Vdash B_1 \leq A_1 \\
\Gamma \Vdash \forall a.\, A \leq B &\longrightarrow_8 \Gamma, \widehat{\alpha} \Vdash [\widehat{\alpha}/a]A \leq B \quad \text{when } B \neq \forall a.\, B' \\
\Gamma \Vdash A \leq \forall b.\, B &\longrightarrow_9 \Gamma, b \Vdash A \leq B
\end{aligned}$$

$$\begin{aligned}
\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \leq A \to B &\longrightarrow_{10} [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2] \Vdash \widehat{\alpha}_1 \to \widehat{\alpha}_2 \leq A \to B) \\
&\qquad \text{when } \widehat{\alpha} \notin FV(A) \cup FV(B) \\
\Gamma[\widehat{\alpha}] \Vdash A \to B \leq \widehat{\alpha} &\longrightarrow_{11} [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2] \Vdash A \to B \leq \widehat{\alpha}_1 \to \widehat{\alpha}_2) \\
&\qquad \text{when } \widehat{\alpha} \notin FV(A) \cup FV(B)
\end{aligned}$$

$$\begin{aligned}
\Gamma[\widehat{\alpha}][\widehat{\beta}] \Vdash \widehat{\alpha} \leq \widehat{\beta} &\longrightarrow_{12} [\widehat{\alpha}/\widehat{\beta}](\Gamma[\widehat{\alpha}][]) \\
\Gamma[\widehat{\alpha}][\widehat{\beta}] \Vdash \widehat{\beta} \leq \widehat{\alpha} &\longrightarrow_{13} [\widehat{\alpha}/\widehat{\beta}](\Gamma[\widehat{\alpha}][]) \\
\Gamma[a][\widehat{\beta}] \Vdash a \leq \widehat{\beta} &\longrightarrow_{14} [a/\widehat{\beta}](\Gamma[a][]) \\
\Gamma[a][\widehat{\beta}] \Vdash \widehat{\beta} \leq a &\longrightarrow_{15} [a/\widehat{\beta}](\Gamma[a][]) \\
\Gamma[\widehat{\beta}] \Vdash 1 \leq \widehat{\beta} &\longrightarrow_{16} [1/\widehat{\beta}](\Gamma[]) \\
\Gamma[\widehat{\beta}] \Vdash \widehat{\beta} \leq 1 &\longrightarrow_{17} [1/\widehat{\beta}](\Gamma[])
\end{aligned}$$

$$\begin{aligned}
\Gamma \Vdash e \Leftarrow B &\longrightarrow_{18} \Gamma \Vdash e \Rightarrow_a a \leq B \quad \text{when } e \neq \lambda x.\, e' \text{ and } B \neq \forall a.\, B' \\
\Gamma \Vdash e \Leftarrow \forall a.\, A &\longrightarrow_{19} \Gamma, a \Vdash e \Leftarrow A \\
\Gamma \Vdash \lambda x.\, e \Leftarrow A \to B &\longrightarrow_{20} \Gamma, x : A \Vdash e \Leftarrow B \\
\Gamma[\widehat{\alpha}] \Vdash \lambda x.\, e \Leftarrow \widehat{\alpha} &\longrightarrow_{21} [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2], x : \widehat{\alpha}_1 \Vdash e \Leftarrow \widehat{\alpha}_2)
\end{aligned}$$

$$\begin{aligned}
\Gamma \Vdash x \Rightarrow_a \omega &\longrightarrow_{22} \Gamma \Vdash [A/a]\omega \quad \text{when } (x : A) \in \Gamma \\
\Gamma \Vdash (e : A) \Rightarrow_a \omega &\longrightarrow_{23} \Gamma \Vdash [A/a]\omega \Vdash e \Leftarrow A \\
\Gamma \Vdash () \Rightarrow_a \omega &\longrightarrow_{24} \Gamma \Vdash [1/a]\omega \\
\Gamma \Vdash \lambda x.\, e \Rightarrow_a \omega &\longrightarrow_{25} \Gamma, \widehat{\alpha}, \widehat{\beta} \Vdash [\widehat{\alpha} \to \widehat{\beta}/a]\omega, x : \widehat{\alpha} \Vdash e \Leftarrow \widehat{\beta} \\
\Gamma \Vdash e_1\, e_2 \Rightarrow_a \omega &\longrightarrow_{26} \Gamma \Vdash e_1 \Rightarrow_b (b \bullet e_2 \Rrightarrow_a \omega)
\end{aligned}$$

$$\begin{aligned}
\Gamma \Vdash \forall a.\, A \bullet e \Rrightarrow_a \omega &\longrightarrow_{27} \Gamma, \widehat{\alpha} \Vdash [\widehat{\alpha}/a]A \bullet e \Rrightarrow_a \omega \\
\Gamma \Vdash A \to C \bullet e \Rrightarrow_a \omega &\longrightarrow_{28} \Gamma \Vdash [C/a]\omega \Vdash e \Leftarrow A \\
\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \bullet e \Rrightarrow_a \omega &\longrightarrow_{29} [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2] \Vdash \widehat{\alpha}_1 \to \widehat{\alpha}_2 \bullet e \Rrightarrow_a \omega)
\end{aligned}$$

Figure 4.5: Algorithmic Typing

Please note that when a new variable is introduced in the right-hand side worklist Γ', we implicitly pick a fresh one, since the right-hand side can be seen as the premise of the reduction.

Rules 1-3 pop variable declarations that are essentially garbage. Those variables are out of scope for the remaining judgments in the worklist. All other rules concern a judgment at the front of the worklist. Logically we can discern 6 groups of rules.

1. Algorithmic subtyping We have six subtyping rules (Rules 4-9) that are similar to their declarative counterparts. For instance, Rule 7 consumes a subtyping judgment and pushes two back to the worklist. Rule 8 differs from declarative Rule $\leq\forall\mathtt{L}$ by introducing an existential variable $\widehat{\alpha}$ instead of guessing the monotype τ instantiation. Each existential variable is later solved to a monotype τ with the same context, so the final solution τ of $\widehat{\alpha}$ should be well-formed under Γ.

Worklist Variable Scoping Rules 8 and 9 involve variable declarations and demonstrate how our worklist treats variable scopes. Rule 8 introduces an existential variable $\widehat{\alpha}$ that is only visible to the judgment $[\widehat{\alpha}/a]A \leq B$. Reduction continues until all the subtyping judgments in front of $\widehat{\alpha}$ are satisfied. Finally we can safely remove $\widehat{\alpha}$ since no occurrence of $\widehat{\alpha}$ could have leaked into the left part of the worklist. Moreover, the algorithm garbage-collects the $\widehat{\alpha}$ variable at the right time: it leaves the environment immediately after being unreferenced completely for sure.

Example Consider the derivation of the subtyping judgment $(1 \to 1) \to 1 \leq (\forall a.\ 1 \to 1) \to 1$:

$$
\begin{aligned}
& \cdot \vdash (1 \to 1) \to 1 \leq (\forall a.\ 1 \to 1) \to 1 \\
\longrightarrow_7\ & \cdot \Vdash 1 \leq 1 \Vdash \forall a.\ 1 \to 1 \leq 1 \to 1 \\
\longrightarrow_8\ & \cdot \Vdash 1 \leq 1, \widehat{\alpha} \Vdash 1 \to 1 \leq 1 \to 1 \\
\longrightarrow_7\ & \cdot \Vdash 1 \leq 1, \widehat{\alpha} \Vdash 1 \leq 1 \Vdash 1 \leq 1 \\
\longrightarrow_4\ & \cdot \Vdash 1 \leq 1, \widehat{\alpha} \Vdash 1 \leq 1 \\
\longrightarrow_4\ & \cdot \Vdash 1 \leq 1, \widehat{\alpha} \\
\longrightarrow_2\ & \cdot \Vdash 1 \leq 1 \\
\longrightarrow_4\ & \cdot
\end{aligned}
$$

First, the subtyping of two function types is split into two judgments by Rule 7: a covariant subtyping on the return type and a contravariant subtyping on the argument type. Then we

apply Rule 8 to reduce the $\forall$ quantifier on the left side. The rule introduces an existential variable $\widehat{\alpha}$ to the context (even though the type $\forall a.\ 1 \to 1$ does not actually refer to the quantified type variable a). In the following 3 steps we satisfy the judgment $1 \to 1 \leq 1 \to 1$ by Rules 7, 4, and 4 (again).

Now the existential variable $\widehat{\alpha}$, introduced before but still unsolved, is at the top of the worklist and Rule 2 garbage-collects it. The process is carefully designed within the algorithmic rules: when $\widehat{\alpha}$ is introduced earlier by Rule 8, we foresee the recycling of $\widehat{\alpha}$ after all the judgments (potentially) requiring $\widehat{\alpha}$ have been processed. Finally Rule 4 reduces one of the base cases and finishes the subtyping derivation.

2. Existential decomposition. Rules 10 and 11 are algorithmic versions of Rule $\leq\!\!\to$; they both partially instantiate $\widehat{\alpha}$ to function types. The domain $\widehat{\alpha}_1$ and range $\widehat{\alpha}_2$ of the new function type are not determined: they are fresh existential variables with the same scope as $\widehat{\alpha}$. We replace $\widehat{\alpha}$ in the worklist with $\widehat{\alpha}_1, \widehat{\alpha}_2$. To propagate the instantiation to the rest of the worklist and maintain well-formedness, every reference to $\widehat{\alpha}$ is replaced by $\widehat{\alpha}_1 \to \widehat{\alpha}_2$. The *occurs-check* condition prevents divergence as usual. For example, without it $\widehat{\alpha} \leq 1 \to \widehat{\alpha}$ would diverge.

3. Solving existentials Rules 12-17 are base cases where an existential variable is solved. They all remove an existential variable and substitute the variable with its solution in the remaining worklist. Importantly the rules respect the scope of existential variables. For example, Rule 12 states that an existential variable $\widehat{\alpha}$ can be solved with another existential variable $\widehat{\beta}$ only if $\widehat{\beta}$ occurs after $\widehat{\alpha}$.

One may notice that the subtyping relation for simple types is just equivalence, which is true according to the declarative system. The DK's system works in a similar way.

4. Checking judgments. Rules 18-21 deal with checking judgments. Rule 18 is similar to `DeclSub`, but rewritten in the continuation-passing-style. The side conditions $e \neq \lambda x.\ e'$ and $B \neq \forall a.\ B'$ prevent overlap with Rules 19, 20 and 21; this is further discussed at the end of this section. Rules 19 and 20 adapt their declarative counterparts to the worklist style. Rule 21 is a special case of `Decl` $\to$ `I`, dealing with the case when the input type is an existential variable, representing a monotype *function* as in the declarative system (it must be a function type, since the expression $\lambda x.\ e$ is a function). The same instantiation technique as in Rules 10 and 11 applies. The declarative checking rule `Decl1I` does not have a direct counterpart in the algorithm, because Rules 18 and 24 can be combined to give the same result.

Rule 21 Design Choice The addition of Rule 21 is a design choice we have made to simplify the side condition of Rule 18 (which avoids overlap). It also streamlines the algorithm and the metatheory as we now treat all cases where we can see that an existential variable should be instantiated to a function type (i.e., Rules 10, 11, 21 and 29) uniformly.

The alternative would have been to omit Rule 21 and drop the condition on e in Rule 18. The modified Rule 18 would then handle $\Gamma \Vdash \lambda x.\ e \Leftarrow \widehat{\alpha}$ and yield $\Gamma \Vdash \lambda x.\ e \Rightarrow_a a \leq \widehat{\alpha}$, which would be further processed by Rule 25 to $\Gamma, \widehat{\beta}_1, \widehat{\beta}_2 \Vdash \widehat{\beta}_1 \to \widehat{\beta}_2 \leq \widehat{\alpha}, x : \widehat{\beta}_1 \Vdash e \Leftarrow \widehat{\beta}_2$. As a subtyping constraint between monotypes is simply equality, $\widehat{\beta}_1 \to \widehat{\beta}_2 \leq \widehat{\alpha}$ must end up equating $\widehat{\beta}_1 \to \widehat{\beta}_2$ with $\widehat{\alpha}$ and thus have the same effect as Rule 21, but in a more roundabout fashion.

In comparison, DK's algorithmic subsumption rule has no restriction on the expression e, and they do not have a rule that explicitly handles the case $\lambda x.\ e \Leftarrow \widehat{\alpha}$. Therefore the only way to check a lambda function against an existential variable is by applying the subsumption rule, which further breaks into type inference of a lambda function and a subtyping judgment.

5. Inference judgments. Inference judgments behave differently compared with subtyping and checking judgments: they *return* a type instead of only accepting or rejecting. For the algorithmic system, where guesses are involved, it may happen that the output type of an inference judgment refers to new existential variables, such as Rule 25. In comparison to Rule 8 and 9, where new variables are only referred to by the sub-derivation, Rule 25 introduces variables $\widehat{\alpha}, \widehat{\beta}$ that affect the remaining judgment chain. This rule is carefully designed so that the output variables are bound by earlier declarations, thus the well-formedness of the worklist is preserved, and the garbage will be collected at the correct time. By making use of the continuation-passing-style judgment chain, inner judgments always share the context with their parent judgment.

Rules 22-26 deal with type inference judgments, written in continuation-passing-style. When an inference judgment succeeds with type A, the algorithm continues to work on the inner-chain ω by assigning A to its placeholder variable a. Rule 23 infers an annotated expression by changing into checking mode, therefore another judgment chain is created. Rule 24 is a base case, where the unit type 1 is inferred and thus passed to its child judgment chain. Rule 26 infers the type of an application by firstly inferring the type of the function e_1, and then leaving the rest work to an application inference judgment, which passes a, representing the return type of the application, to the remainder of the judgment chain ω.

Rule 25 infers the type of a lambda expression by introducing $\widehat{\alpha}, \widehat{\beta}$ as the input and output types of the function, respectively. After checking the body e under the assumption $x : \widehat{\alpha}$, the return type might reflect more information than simply $\widehat{\alpha} \to \widehat{\beta}$ through propagation when

existential variables are solved or partially solved. The variable scopes are maintained during the process: the assumption of argument type $(x : \widehat{\alpha})$ is recycled after checking against the function body; the existential variables used by the function type $(\widehat{\alpha}, \widehat{\beta})$ are only visible in the remaining chain $[\widehat{\alpha} \to \widehat{\beta}/a]\omega$. The recycling process of Rule 25 differs from DK's corresponding rule significantly, and we further discuss the differences in Section 4.4.1.

6. Application inference judgments Finally, Rules 27-29 deal with application inference judgments. Rules 27 and 28 behave similarly to declarative rules `Decl`$\forall$`App` and `Decl` $\to$ `App`. Rule 29 instantiates $\widehat{\alpha}$ to the function type $\widehat{\alpha}_1 \to \widehat{\alpha}_2$, just like Rules 10, 11 and 21.

Example Figure 4.6 shows a sample derivation of the algorithm. It checks the application $(\lambda x.\, x)\ ()$ against the unit type. According to the algorithm, we apply Rule 18 (subsumption), changing to inference mode. Type inference of the application breaks into two steps by Rule 26: first we infer the type of the function, and then the application inference judgment helps to determine the return type. In the following 5 steps the type of the identity function, $\lambda x.\, x$, is inferred to be $\widehat{\alpha} \to \widehat{\alpha}$: checking the body of the lambda function (Rule 25), switching from check mode to inference mode (Rule 18), inferring the type of a term variable (Rule 22), solving a subtyping judgment between existential variables (Rule 12) and garbage collecting the term variable x (Rule 3).

After that, Rule 28 changes the application inference judgment to a check of the argument against the input type $\widehat{\alpha}$ and returns the output type $\widehat{\alpha}$. Checking $()$ against the existential variable $\widehat{\alpha}$ solves $\widehat{\alpha}$ to the unit type 1 through Rules 18, 24 and 16. Immediately after $\widehat{\alpha}$ is solved, the algorithm replaces every occurrence of $\widehat{\alpha}$ with 1. Therefore the worklist remains $1 \leq 1$, which is finished off by Rule 4. Finally, the empty worklist indicates the success of the whole derivation.

In summary, our type checking algorithm accepts $(\lambda x.\, x)\ () \Leftarrow 1$.

Non-overlapping and Deterministic Reduction An important feature of our algorithmic rules is that they are directly implementable. Indeed, although written in the form of reduction rules, they do not overlap and are thus deterministic.

Consider in particular Rules 8 and 9, which correspond to the declarative rules $\leq\forall$L and $\leq\forall$R. While those declarative rules both match the goal $\forall a.\, A \leq \forall b.\, B$, we have eliminated this overlap in the algorithm by restricting Rule 8 ($B \neq \forall a.\, B'$) and thus always applying Rule 9 to $\forall a.\, A \leq \forall b.\, B$.

Similarly, the declarative rule `DeclSub` overlaps highly with the other checking rules. Its algorithmic counterpart is Rule 18. Yet, we have avoided the overlap with other algorithmic checking rules by adding side-conditions to Rule 18, namely $e \neq \lambda x.\, e'$ and $B \neq \forall a.\, B'$.

$$
\begin{aligned}
& \cdot \Vdash (\lambda x.\, x)\ () \Leftarrow 1 \\
\longrightarrow_{18} & \cdot \Vdash (\lambda x.\, x)\ () \Rightarrow_a a \leq 1 \\
\longrightarrow_{26} & \cdot \Vdash (\lambda x.\, x) \Rightarrow_b (b \bullet () \Rrightarrow_a a \leq 1) \\
\longrightarrow_{25} & \cdot, \widehat{\alpha}, \widehat{\beta} \Vdash \widehat{\alpha} \to \widehat{\beta} \bullet () \Rrightarrow_a a \leq 1, x : \widehat{\alpha} \Vdash x \Leftarrow \widehat{\beta} \\
\longrightarrow_{18} & \cdot, \widehat{\alpha}, \widehat{\beta} \Vdash \widehat{\alpha} \to \widehat{\beta} \bullet () \Rrightarrow_a a \leq 1, x : \widehat{\alpha} \Vdash x \Rightarrow_b b \leq \widehat{\beta} \\
\longrightarrow_{22} & \cdot, \widehat{\alpha}, \widehat{\beta} \Vdash \widehat{\alpha} \to \widehat{\beta} \bullet () \Rrightarrow_a a \leq 1, x : \widehat{\alpha} \Vdash \widehat{\alpha} \leq \widehat{\beta} \\
\longrightarrow_{12} & \cdot, \widehat{\alpha} \Vdash \widehat{\alpha} \to \widehat{\alpha} \bullet () \Rrightarrow_a a \leq 1, x : \widehat{\alpha} \\
\longrightarrow_{3} & \cdot, \widehat{\alpha} \Vdash \widehat{\alpha} \to \widehat{\alpha} \bullet () \Rrightarrow_a a \leq 1 \\
\longrightarrow_{28} & \cdot, \widehat{\alpha} \Vdash \widehat{\alpha} \leq 1 \Vdash () \Leftarrow \widehat{\alpha} \\
\longrightarrow_{18} & \cdot, \widehat{\alpha} \Vdash \widehat{\alpha} \leq 1 \Vdash () \Rightarrow_a a \leq \widehat{\alpha} \\
\longrightarrow_{24} & \cdot, \widehat{\alpha} \Vdash \widehat{\alpha} \leq 1 \Vdash 1 \leq \widehat{\alpha} \\
\longrightarrow_{16} & \cdot \Vdash 1 \leq 1 \\
\longrightarrow_{4} & \cdot
\end{aligned}
$$

Figure 4.6: A Sample Derivation for Algorithmic Typing

These restrictions have not been imposed arbitrarily: we formally prove that the restricted algorithm is still complete. In Section 4.3.2 we discuss the relevant metatheory, with the help of a non-overlapping version of the declarative system.

4.3 Metatheory

This section presents the metatheory of the algorithmic system presented in the previous section. We show that three main results hold: *soundness*, *completeness* and *decidability*. These three results have been mechanically formalized and proved in the Abella theorem prover [Gacek 2008].

4.3.1 Declarative Worklist and Transfer

To aid formalizing the correspondence between the declarative and algorithmic systems, we introduce the notion of a declarative worklist Ω, defined in Figure 4.7. A declarative worklist Ω has the same structure as an algorithmic worklist Γ, but does not contain any existential variables $\widehat{\alpha}$.

Worklist instantiation. The relation $\Gamma \rightsquigarrow \Omega$ expresses that the algorithmic worklist Γ can be instantiated to the declarative worklist Ω, by appropriately instantiating all existential

Declarative worklist $\quad \Omega \quad ::= \quad \cdot \mid \Omega, a \mid \Omega, x : A \mid \Omega \Vdash \omega$

$\boxed{\Gamma \rightsquigarrow \Omega}$ $\quad$ Γ instantiates to Ω.

$$\frac{}{\Omega \rightsquigarrow \Omega}\ \rightsquigarrow\Omega \qquad \frac{\Omega \vdash \tau \qquad \Omega, [\tau/\widehat{\alpha}]\Gamma \rightsquigarrow \Omega}{\Omega, \widehat{\alpha}, \Gamma \rightsquigarrow \Omega}\ \rightsquigarrow\widehat{\alpha}$$

Figure 4.7: Declarative Worklists and Instantiation

variables $\widehat{\alpha}$ in Γ with well-scoped monotypes τ. The rules of this instantiation relation are shown in Figure 4.7 too. Rule $\rightsquigarrow\widehat{\alpha}$ replaces the first existential variable with a well-scoped monotype and repeats the process on the resulting worklist until no existential variable remains and thus the algorithmic worklist has become a declarative one. In order to maintain well-scopedness, the substitution is applied to all the judgments and term variable bindings in the scope of $\widehat{\alpha}$.

Observe that the instantiation $\Gamma \rightsquigarrow \Omega$ is not deterministic. From left to right, there are infinitely many possibilities to instantiate an existential variable and thus infinitely many declarative worklists that one can get from an algorithmic one. In the other direction, any valid monotype in Ω can be abstracted to an existential variable in Γ. Thus different Γ's can be instantiated to the same Ω.

Lemmas 4.3 and 4.4 generalize Rule $\rightsquigarrow\widehat{\alpha}$ from substituting the first existential variable to substituting any existential variable.

Lemma 4.3 (Insert). *If* $\Gamma_L, [\tau/\widehat{\alpha}]\Gamma_R \rightsquigarrow \Omega$ *and* $\Gamma_L \vdash \tau$ *, then* $\Gamma_L, \widehat{\alpha}, \Gamma_R \rightsquigarrow \Omega$.

Lemma 4.4 (Extract). *If* $\Gamma_L, \widehat{\alpha}, \Gamma_R \rightsquigarrow \Omega$ *, then there exists* τ *s.t.* $\Gamma_L \vdash \tau$ *and* $\Gamma_L, [\tau/\widehat{\alpha}]\Gamma_R \rightsquigarrow \Omega$.

Declarative transfer. Figure 4.8 defines a relation $\Omega \longrightarrow \Omega'$, which transfers all judgments in the declarative worklists to the declarative type system. This relation checks that every judgment entry in the worklist holds using a corresponding conventional declarative judgment. The typing contexts of declarative judgments are recovered using an auxiliary erasure function $\|\Omega\|$ [2]. The erasure function simply drops all judgment entries from the worklist, keeping only variable and type variable declarations.

[2] In the proof script we do not use the erasure function, for the declarative system and well-formedness judgments automatically fit the non-erased declarative worklist just as declarative contexts.

$\boxed{\|\Omega\|}$ Judgment erasure.

$$
\begin{aligned}
\|\cdot\| &= \cdot \\
\|\Omega, a\| &= \|\Omega\|, a \\
\|\Omega, x : A\| &= \|\Omega\|, x : A \\
\|\Omega \Vdash \omega\| &= \|\Omega\|
\end{aligned}
$$

$\boxed{\Omega \longrightarrow \Omega'}$ Declarative transfer.

$$
\begin{aligned}
\Omega, a &\longrightarrow \Omega \\
\Omega, x : A &\longrightarrow \Omega \\
\Omega \Vdash A \leq B &\longrightarrow \Omega && \text{when } \|\Omega\| \vdash A \leq B \\
\Omega \Vdash e \Leftarrow A &\longrightarrow \Omega && \text{when } \|\Omega\| \vdash e \Leftarrow A \\
\Omega \Vdash e \Rightarrow_a \omega &\longrightarrow \Omega \Vdash [A/a]\omega && \text{when } \|\Omega\| \vdash e \Rightarrow A \\
\Omega \Vdash A \bullet e \Rrightarrow_a \omega &\longrightarrow \Omega \Vdash [C/a]\omega && \text{when } \|\Omega\| \vdash A \bullet e \Rrightarrow C
\end{aligned}
$$

Figure 4.8: Declarative Transfer

4.3.2 Non-Overlapping Declarative System

DK's declarative system, shown in Figures 2.8 and 2.9, has a few overlapping rules. In contrast, our algorithm has removed all overlap; at most one rule applies in any given situation. This discrepancy makes it more difficult to relate the two systems.

To simplify matters, we introduce an intermediate system that is still declarative in nature, but has no overlap. This intermediate system differs only in a few rules from DK's declarative system:

1. Restrict the shape of B in the rule $\forall$L subtyping rule:

$$\frac{B \neq \forall b.\, B' \qquad \Psi \vdash \tau \qquad \Psi \vdash [\tau/a]A \leq B}{\Psi \vdash \forall a.\, A \leq B}\ \forall\mathtt{L}'$$

2. Drop the redundant rule `Decl1I`, which can be easily derived by a combination of `DeclSub`, `Decl1I`$\Rightarrow$ and $\leq$`Unit`:

$$\frac{\dfrac{}{\Psi \vdash () \Rightarrow 1}\ \mathtt{Decl1I}\Rightarrow \qquad \dfrac{}{\Psi \vdash 1 \leq 1}\ \leq\mathtt{Unit}}{\Psi \vdash () \Leftarrow 1}\ \mathtt{DeclSub}$$

3. Restrict the shapes of e and A in the subsumption rule `DeclSub`:

$$\frac{e \neq \lambda x.\, e' \qquad A \neq \forall a.\, A' \qquad \Psi \vdash e \Rightarrow A \qquad \Psi \vdash A \leq B}{\Psi \vdash e \Leftarrow B}\ \texttt{DeclSub}'$$

The resulting declarative system has no overlapping rules and more closely resembles the algorithmic system, which contains constraints of the same shape.

We have proven soundness and completeness of the non-overlapping declarative system with respect to the overlapping one to establish their equivalence. Thus the restrictions do not change the expressive power of the system. Modification (2) is relatively easy to justify, with the derivation given above: the rule is redundant and can be replaced by a combination of three other rules. Modifications (1) and (3) require inversion lemmas for the rules that overlap. Firstly, Rule $\forall\texttt{L}$ overlaps with Rule $\forall\texttt{R}$ for the judgment $\Psi \vdash \forall a.\, A \leq \forall b.\, B$. The following inversion lemma for Rule $\forall\texttt{R}$ resolves the overlap:

Lemma 4.5 (Invertibility of $\forall\texttt{R}$). *If $\Psi \vdash A \leq \forall a.\, B$ then $\Psi, a \vdash A \leq B$.*

The lemma implies that preferring Rule $\forall\texttt{R}$ does not affect the derivability of the judgment. Therefore the restriction $B \neq \forall b.\, B'$ in $\forall\texttt{L}'$ is valid.

Secondly, Rule `DeclSub` overlaps with both `Decl`$\forall$`I` and `Decl`$\rightarrow$`I`. We have proven two inversion lemmas for these overlaps:

Lemma 4.6 (Invertibility of `Decl`$\forall$`I`). *If $\Psi \vdash e \Leftarrow \forall a.\, A$ then $\Psi, a \vdash e \Leftarrow A$.*

Lemma 4.7 (Invertibility of `Decl`$\rightarrow$`I`). *If $\Psi \vdash \lambda x.\, e \Leftarrow A \to B$ then $\Psi, x : A \vdash e \Leftarrow B$.*

These lemmas express that applying the more specific rules, rather than the more general rule `DeclSub`, does not reduce the expressive power.

The proofs of the above two lemmas rely on an important property of the declarative system, the *subsumption lemma*. To be able to formulate this lemma, Figure 4.9 introduces the *context subtyping relation* $\Psi \leq \Psi'$. Context Ψ subsumes context Ψ' if they bind the same variables in the same order, but the types A of the term variables x in the former are subtypes of types A' assigned to those term variables in the latter. Now we can state the subsumption lemma:

Lemma 4.8 (Subsumption). *Given $\Psi' \leq \Psi$:*

1. *If $\Psi \vdash e \Leftarrow A$ and $\Psi \vdash A \leq A'$ then $\Psi' \vdash e \Leftarrow A'$;*
2. *If $\Psi \vdash e \Rightarrow B$ then there exists B' s.t. $\Psi \vdash B' \leq B$ and $\Psi' \vdash e \Rightarrow B'$;*

$\boxed{\Psi' \leq \Psi}$

$$\frac{}{\cdot \leq \cdot}\ \texttt{CtxSubEmpty} \qquad \frac{\Psi' \leq \Psi}{\Psi', a \leq \Psi, a}\ \texttt{CtxSubTyVar}$$

$$\frac{\Psi' \leq \Psi \qquad \Psi \vdash A' \leq A}{\Psi', x : A' \leq \Psi, x : A}\ \texttt{CtxSubTmVar}$$

Figure 4.9: Context Subtyping

3. *If* $\Psi \vdash A \bullet e \Rrightarrow C$ *and* $\Psi \vdash A' \leq A$*, then there exists* C' *s.t.* $\Psi \vdash C' \leq C$ *and* $\Psi' \vdash A' \bullet e \Rrightarrow C'$.

This lemma expresses that any derivation in a context Ψ has a corresponding derivation in any context Ψ' that it subsumes.

While being written in a clear format and providing enough details, some proof is not fully accepted by the proof assistant. Specifically for the subsumption lemma, we have tried to follow DK's manual proof, but we discovered several problems in their reasoning that we have been unable to address. Fortunately we have found a different way to prove the lemma. A description of the problem and our fix are discussed in-depth as follows.

In the appendix of DK's paper [Dunfield and Krishnaswami 2013], the first two applications of induction hypotheses on page 22 are not accepted. Either of them seems to use a slightly different "induction hypothesis" than the true one. In fact, we cannot think of simple ways to fix the problem, since the induction hypothesis seems not strong enough for these two cases.

To fix the proof, we propose a new induction scheme by making use of our worklist measures. Recall that $|e|_e$ measures term size; and the judgment measure counts checking as 2, inference as 1 and application inference as 3; and $|A|_\forall$ counts the number of $\forall$'s in a type.

Proof. The proof is by a nested mutual induction on the lexicographical order of the measures

$$\langle |e|_e, |\cdot|_\Leftrightarrow, |A|_\forall + |A'|_\forall \rangle,$$

where the second measure is 2 for checking (1), 1 for inference (2) and 3 for application inference (3); and the third measure does not apply to case (2) since no A is used. Compared with DK's, our third measure that counts the degree of polymorphism fixes problems that occurred in DK's proof: in both places, our new induction hypothesis is more generalized.

All but two cases can be finished easily following the declarative typing derivation, and the proof shares a similar structure to DK's. One tricky case related to Rule $\texttt{Decl}\forall\texttt{I}$ indicates that A has the shape $\forall a.\ A_0$, thus the subtyping relation derives from either $\leq\forall\texttt{L}$ or $\leq\forall\texttt{R}$. For

each of the case, the third measure $|A|_\forall + |A'|_\forall$ decreases (the $\leq\forall\mathtt{L}$ case requires a type substitution lemma obtaining $\Psi \vdash e \Leftarrow [\tau/a]A_0$ from the typing derivation).

Another tricky case is $\mathtt{Decl}{\to}\mathtt{I}$. When the subtyping relation is derived from $\leq{\to}$, a simple application of induction hypothesis finishes the proof. When the subtyping relation is derived from $\leq\forall\mathtt{R}$, $|A'|_\forall$ decreases, and thus the induction hypothesis finishes this case. □

In short, we found a specific problem when trying to prove the subsumption lemma following DK's manual proof, yet we addressed the problem by using a slightly different induction scheme.

Three-Way Soundness and Completeness Theorems We now have three systems that can be related: DK's overlapping declarative system, our non-overlapping declarative system, and our algorithmic system. We have already established the first relation, that the two declarative systems are equivalent. In what follows, we will establish the soundness of our algorithm directly against the original overlapping declarative system. However, we have found that showing completeness of the algorithm is easier against the non-overlapping declarative system. Of course, as a corollary, it follows that our algorithm is also complete with respect to DK's declarative system.

4.3.3 Soundness

Our algorithm is sound with respect to DK's declarative system. For any worklist Γ that reduces successfully, there is a valid instantiation Ω that transfers all judgments to the declarative system.

Theorem 4.9 (Soundness). *If* $\mathrm{wf}\,\Gamma$ *and* $\Gamma \longrightarrow^* \cdot$, *then there exists* Ω *s.t.* $\Gamma \rightsquigarrow \Omega$ *and* $\Omega \longrightarrow^* \cdot$.

The proof proceeds by induction on the derivation of $\Gamma \longrightarrow^* \cdot$. Interesting cases are those involving existential variable instantiations, including Rules 10, 11, 21 and 29. Applications of Lemmas 4.3 and 4.4 help analyse the full instantiation of those existential variables. For example, when $\widehat{\alpha}$ is solved to $\widehat{\alpha}_1 \to \widehat{\alpha}_2$ in the algorithm, applying the Extract lemma gives two instantiations $\widehat{\alpha}_1 = \sigma$ and $\widehat{\alpha}_2 = \tau$. It follows that $\widehat{\alpha} = \sigma \to \tau$, which enables the induction hypothesis and finishes the corresponding case. Some immediate corollaries which show the soundness for specific judgment forms are:

Corollary 4.10 (Soundness, single judgment form). *Given* $\mathrm{wf}\,\Gamma$*:*

1. *If* $\Gamma \Vdash A \leq B \longrightarrow^* \cdot$
   then there exist A', B', Ω *s.t.* $\Gamma \Vdash A \leq B \rightsquigarrow \Omega \Vdash A' \leq B'$ *and* $\|\Omega\| \vdash A' \leq B'$*;*

2. *If* $\Gamma \Vdash e \Leftarrow A \longrightarrow^* \cdot$
   then there exist A', Ω *s.t.* $\Gamma \Vdash e \Leftarrow A \rightsquigarrow \Omega \Vdash e \Leftarrow A'$ *and* $\|\Omega\| \vdash e \Leftarrow A'$*;*

3. *If* $\Gamma \Vdash e \Rightarrow_a \omega \longrightarrow^* \cdot$ *for any* ω
   then there exists Ω, ω', A *s.t.* $\Gamma \rightsquigarrow \Omega$ *and* $\|\Omega\| \vdash e \Rightarrow A$*;*

4. *If* $\Gamma \Vdash A \bullet e \Rrightarrow_a \omega \longrightarrow^* \cdot$ *for any* ω
   then there exists Ω, ω', A', C *s.t.* $\Gamma \Vdash A \bullet e \Rrightarrow_a \omega \rightsquigarrow \Omega \Vdash A' \bullet e \Rrightarrow_a \omega'$ *and* $\|\Omega\| \vdash A' \bullet e \Rrightarrow C$.

4.3.4 Completeness

The completeness of our algorithm means that any derivation in the declarative system has an algorithmic counterpart. We explicitly relate between an algorithmic context Γ and a declarative context Ω to avoid potential confusion.

Theorem 4.11 (Completeness). *If* wf Γ *and* $\Gamma \rightsquigarrow \Omega$ *and* $\Omega \longrightarrow^* \cdot$*, then* $\Gamma \longrightarrow^* \cdot$.

We prove completeness by induction on the derivation of $\Omega \longrightarrow^* \cdot$ and use the non-overlapping declarative system. Since the declarative worklist is reduced judgment by judgment (shown in Figure 4.8), the induction always analyses the first judgment by a small step. As the algorithmic system introduces existential variables, a declarative rule may correspond to multiple algorithmic rules, and thus we analyse each of the possible cases.

Most cases are relatively easy to prove. The Insert and Extract lemmas are applied when the algorithm uses existential variables, but transferred to a monotype for the declarative system, such as Rules 6, 8, 10, 11, 12-17, 21, 25, 27 and 29.

Algorithmic Rules 10 and 11 require special treatment. When the induction reaches the $\leq\!\!\rightarrow$ case, the first judgment is of shape $A_1 \to A_2 \leq B_1 \to B_2$. One of the corresponding algorithmic judgments is $\widehat{\alpha} \leq A \to B$. However, the case analysis does not imply that $\widehat{\alpha}$ is fresh in A and B, therefore Rule 10 cannot be applied and the proof gets stuck. The following lemma helps us out in those cases: the success in declarative subtyping indicates the freshness of $\widehat{\alpha}$ in A and B in its corresponding algorithmic judgment. In other words, the declarative system does not accept infinite types. A symmetric lemma holds for $A \to B \leq \widehat{\alpha}$.

Lemma 4.12 (Prune Transfer for Instantiation). *If* $(\Gamma \Vdash \widehat{\alpha} \leq A \to B) \rightsquigarrow (\Omega \Vdash C \leq A_1 \to B_1)$ *and* $\|\Omega\| \vdash C \leq A_1 \to B_1$*, then* $\widehat{\alpha} \notin FV(A) \cup FV(B)$.

The following corollary is derived immediately from Theorem 4.11.

Corollary 4.13 (Completeness, single judgment form). *Given* wf Γ *containing no judgments:*

1. *If* $\Omega \vdash A' \leq B'$ *and* $\Gamma \Vdash A \leq B \rightsquigarrow \Omega \Vdash A' \leq B'$
   then $\Gamma \Vdash A \leq B \longrightarrow^* \cdot$;
2. *If* $\Omega \vdash e \Leftarrow A'$ *and* $\Gamma \Vdash e \Leftarrow A \rightsquigarrow \Omega \Vdash e \Leftarrow A'$
   then $\Gamma \Vdash e \Leftarrow A \longrightarrow^* \cdot$;
3. *If* $\Omega \vdash e \Rightarrow A$ *and* $\Gamma \Vdash e \Rightarrow_a 1 \leq 1 \rightsquigarrow \Omega \Vdash e \Rightarrow_a 1 \leq 1$
   then $\Gamma \Vdash e \Rightarrow_a 1 \leq 1 \longrightarrow^* \cdot$;
4. *If* $\Omega \vdash A' \bullet e \Rrightarrow C$ *and* $\Gamma \Vdash A \bullet e \Rrightarrow_a 1 \leq 1 \rightsquigarrow \Omega \Vdash A' \bullet e \Rrightarrow_a 1 \leq 1$
   then $\Gamma \Vdash A \bullet e \Rrightarrow_a 1 \leq 1 \longrightarrow^* \cdot$.

4.3.5 Decidability

Finally, we show that our algorithm is decidable:

Theorem 4.14 (Decidability). *Given* wf Γ*, it is decidable whether* $\Gamma \longrightarrow^* \cdot$ *or not.*

Our decidability proof is based on a lexicographic group of induction measures $\langle |\Gamma|_e, |\Gamma|_\Leftrightarrow, |\Gamma|_\forall, |\Gamma|_{\widehat{\alpha}}, |\Gamma|_\rightarrow + |\Gamma| \rangle$ on the worklist Γ. Formal definitions of these measures can be found in Figure 4.10. The first two, $|\cdot|_e$ and $|\cdot|_\Leftrightarrow$, measure the total size of terms and the total difficulty of judgments, respectively. In the latter, check judgments count for 2, inference judgments for 1 and function inference judgments for 3. Another two measures, $|\cdot|_\forall$ and $|\cdot|_\rightarrow$, count the total number of universal quantifiers and function types, respectively. Finally, $|\cdot|_{\widehat{\alpha}}$ counts the number of existential variables in the worklist, and $|\cdot|$ is simply the length of the worklist.

It is not difficult to see that all but two algorithmic reduction rules decrease the group of measures. (The result of Rule 29 could be directly reduced by Rule 28, which decreases the measures.) The two exceptions are Rules 10 and 11. Both rules increase the number of existential variables without decreasing the number of universal quantifiers. However, they are both immediately followed by Rule 7, which breaks the subtyping problem into two smaller problems of the form $\widehat{\alpha} \leq A$ and $A \leq \widehat{\alpha}$ which we call *instantiation judgments*.

We now show that entirely reducing these smaller problems leaves the worklist in a state with an overall smaller measure. Our starting point is a worklist Γ, Γ_i where Γ_i are instantiation judgments.

$$\Gamma_i := \cdot \mid \Gamma_i, \widehat{\alpha} \leq A \mid \Gamma_i, A \leq \widehat{\alpha} \quad \text{where } \widehat{\alpha} \notin FV(A) \cup FV(\Gamma_i)$$

Fully reducing these instantiation judgments at the top of the worklist has a twofold impact. Firstly, new entries may be pushed onto the worklist which are not instantiation judgments.

$\boxed{|\Gamma|_e, |\omega|_e, |e|_e}$ Size measure.

$$|\Gamma|_e = \sum_{\omega \in \Gamma} |\omega|_e$$

$$\begin{aligned}
|A \leq B|_e &= 0 \\
|e \Leftarrow A|_e &= |e|_e \\
|e \Rightarrow \omega|_e &= |e|_e + |\omega|_e \\
|A \bullet e \Rrightarrow_a \omega|_e &= |e|_e + |\omega|_e
\end{aligned}$$

$$\begin{aligned}
|x|_e &= |()|_e = 1 \\
|\lambda x.\, e|_e &= |e|_e + 1 \\
|e_1\ e_2|_e &= |e_1|_e + |e_2|_e + 1 \\
|e : A|_e &= |e|_e + 1
\end{aligned}$$

$\boxed{|\Gamma|_\Leftrightarrow, |\omega|_\Leftrightarrow}$ Judgment measure.

$$|\Gamma|_\Leftrightarrow = \sum_{\omega \in \Gamma} |\omega|_e$$

$$\begin{aligned}
|A \leq B|_\Leftrightarrow &= 0 \\
|e \Leftarrow A|_\Leftrightarrow &= 2 \\
|e \Rightarrow \omega|_\Leftrightarrow &= |\omega|_\Leftrightarrow + 1 \\
|A \bullet e \Rrightarrow_a \omega|_\Leftrightarrow &= |\omega|_\Leftrightarrow + 3
\end{aligned}$$

$\boxed{|\Gamma|_{\widehat{\alpha}}}$ Existential measure.

$$|\Gamma|_{\widehat{\alpha}} = \#_{\widehat{\alpha} \in \Gamma}$$

$\boxed{|\Gamma|_\forall, |\omega|_\forall, |A|_\forall}$ Polymorphism measure.

$$|\Gamma|_\Leftrightarrow = \sum_{\omega \in \Gamma} |\omega|_\forall$$

$$\begin{aligned}
|A \leq B|_\forall &= |A|_\forall + |B|_\forall \\
|e \Leftarrow A|_\forall &= |A|_\forall \\
|e \Rightarrow \omega|_\forall &= |\omega|_\forall \\
|A \bullet e \Rrightarrow_a \omega|_\forall &= |A|_\forall + |\omega|_\forall
\end{aligned}$$

$$\begin{aligned}
|1|_\forall = |a|_\forall = |\widehat{\alpha}|_\forall &= 1 \\
|A \to B|_\forall &= |A|_\forall + |B|_\forall \\
|\forall a.\, A|_\forall &= |A|_\forall + 1
\end{aligned}$$

$\boxed{|\Gamma|_\to, |\omega|_\to, |A|_\to}$ Function measure.

$$|\Gamma|_\to = \sum_{\omega \in \Gamma} |\omega|_\forall$$

$$\begin{aligned}
|A \leq B|_\forall &= |A|_\forall + |B|_\forall \\
|e \Leftarrow A|_\forall &= |A|_\forall \\
|e \Rightarrow \omega|_\forall &= |\omega|_\forall \\
|A \bullet e \Rrightarrow_a \omega|_\forall &= |A|_\forall + |\omega|_\forall
\end{aligned}$$

$$\begin{aligned}
|1|_\to = |a|_\to = |\widehat{\alpha}|_\to &= 1 \\
|A \to B|_\to &= |A|_\to + |B|_\to + 1 \\
|\forall a.\, A|_\to &= |A|_\to
\end{aligned}$$

Figure 4.10: Worklist measures

This only happens when Γ_i contains a universal quantifier that is reduced by Rule 8 or 9. The new entries then are of the form $\Gamma_\leq$:

$$\Gamma_\leq := \cdot \mid \Gamma_\leq, a \mid \Gamma_\leq, \widehat{\alpha} \mid \Gamma_\leq, A \leq B$$

Secondly, reducing the instantiation judgments may also affect the remainder of the worklist Γ, by solving existing existentials and introducing new ones. This worklist update is captured in the update judgment $\Gamma \rightharpoonup \Gamma'$ defined in Figure 4.11. For instance, an existential variable instantiation, $\Gamma_L, \widehat{\alpha}, \Gamma_R \rightharpoonup \Gamma_L, \widehat{\alpha}_1, \widehat{\alpha}_2, [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}]\Gamma_R$, can be derived as a combination of the three rules that define the update relation.

$$\boxed{\Gamma \rightharpoonup \Gamma'} \qquad \Gamma \text{ updates to } \Gamma'.$$

$$\frac{}{\Gamma \rightharpoonup \Gamma} \rightharpoonup \texttt{id} \qquad \frac{|A|_\forall = 0 \quad \Gamma_L, [A/\widehat{\alpha}]\Gamma_R \rightharpoonup \Gamma'}{\Gamma_L, \widehat{\alpha}, \Gamma_R \rightharpoonup \Gamma'} \rightharpoonup \texttt{solve} \qquad \frac{\Gamma_L, \widehat{\alpha}, \Gamma_R \rightharpoonup \Gamma'}{\Gamma_L, \Gamma_R \rightharpoonup \Gamma'} \rightharpoonup \widehat{\alpha}$$

Figure 4.11: Worklist Update

The good news is that worklist updates do not affect the three main worklist measures:

Lemma 4.15 (Measure Invariants of Worklist Extension). *If* $\Gamma \rightharpoonup \Gamma'$ *then* $|\Gamma|_e = |\Gamma'|_e$ *and* $|\Gamma|_\Leftrightarrow = |\Gamma'|_\Leftrightarrow$ *and* $|\Gamma|_\forall = |\Gamma'|_\forall$.

Moreover, we can characterize the reduction of the instantiation judgments as follows.

Lemma 4.16 (Instantiation Decidability). *For any well-formed algorithmic worklist* (Γ, Γ_i)*:*

1) If $|\Gamma_i|_\forall = 0$*, then there exists* Γ'
s.t. $(\Gamma, \Gamma_i) \longrightarrow^* \Gamma'$ *and* $|\Gamma'|_{\widehat{\alpha}} = |\Gamma|_{\widehat{\alpha}} - |\Gamma_i|$ *and* $\Gamma \rightharpoonup \Gamma'$.

2) If $|\Gamma_i|_\forall > 0$*, then there exist* $\Gamma', \Gamma_\leq$
s.t. $(\Gamma, \Gamma_i) \longrightarrow^* (\Gamma', \Gamma_\leq)$ *and* $|\Gamma_\leq|_\forall = |\Gamma_i|_\forall - 1$ *and* $\Gamma \rightharpoonup \Gamma'$.

Hence, reducing the instantiation judgment prefix Γ_i either decreases the number of universal quantifiers or eliminates one existential variable per instantiation judgment. The proof of this lemma proceeds by induction on the measure $2 * |\Gamma_i|_\rightarrow + |\Gamma_i|$ of the instantiation judgment list Γ_i.

Let us go back to the whole algorithm and summarize our findings. The decidability theorem is shown through a lexicographic group of induction measures $\langle |\Gamma|_e, |\Gamma|_\Leftrightarrow, |\Gamma|_\forall, |\Gamma|_{\widehat{\alpha}}, |\Gamma|_\rightarrow + |\Gamma| \rangle$ which is decreased by nearly all rules. In the exceptional case that the measure does not decrease immediately, we encounter an instantiation judgment at the top of the worklist. We can then make use of Lemma 4.16 to show that $|\Gamma|_{\widehat{\alpha}}$ or $|\Gamma|_\forall$ decreases when that instantiation judgment is consumed or partially reduced. Moreover, Lemma 4.15 establishes that no higher-priority measure component increases. Hence, in the exceptional case we have an overall measure decrease too.

Combining all three main results (soundness, completeness and decidability), we conclude that the declarative system is decidable by means of our algorithm.

Corollary 4.17 (Decidability of Declarative Typing). *Given* wf Ω*, it is decidable whether* $\Omega \longrightarrow^* \cdot$ *or not.*

4.3.6 Abella and Proof Statistics

We have chosen the Abella (v2.0.7-dev [3]) proof assistant [Gacek 2008] to develop our formalization. Abella is designed to help with formalizations of programming languages, due to its built-in support for variable binding and the λ-tree syntax [Miller 2000], which is a form of HOAS. Nominal variables, or ∇-quantified variables, are used as an unlimited name supply, which supports explicit freshness control and substitutions. Although Abella lacks packages, tactics and support for modules, its higher-order unification and the ease of formalizing substitution-intensive relations are very helpful.

While the algorithmic rules are in a small-step style, the proof script rewrites them into a big-step style for easier inductions. In addition, we do prove the equivalence of the two representations.

Statistics of the Proof The proof script consists of 7,977 lines of Abella code with a total of 60 definitions and 596 theorems. Figure 4.1 briefly summarizes the contents of each file. The files are linearly dependent due to the limitations of Abella.

Translation Table for the Proof Scripts In the proof scripts, we use textual relational definitions rather than the symbolic ones used in the paper. The mapping, shown in Table 4.2, should be helpful for anyone who reads the script.

4.4 Discussion

This section discusses some insights that we gained from our work and contrasts the scoping mechanisms we have employed with those in DK's algorithm. We also discuss a way to improve the precision of their scope tracking. Furthermore we discuss and sketch an extension of our algorithm with an elaboration to a target calculus, and discuss an extension of our algorithm with scoped type variables [Peyton Jones and Shields 2004].

4.4.1 Contrasting Our Scoping Mechanisms with DK's

A nice feature of our worklists is that, simply by interleaving variable declarations and judgment chains, they make the scope of variables precise. DK's algorithm deals with garbage collecting variables in a different way: through type variable or existential variable markers (as discussed in Section 4.1.2). Despite the sophistication employed in DK's algorithm to

[3] We use a development version because the developers just fixed a serious bug that accepts a simple proof of false, which also affects our proof. Specifically, our scripts compile against commit 92829a of Abella's GitHub repository.

Table 4.1: Statistics for the proof scripts

File(s)	LOC	#Thm	Description
olist.thm, nat.thm	311	57	Basic data structures
typing.thm	245	7	Declarative & algorithmic system, debug examples
decl.thm	226	33	Basic declarative properties
order.thm	235	27	The $\lvert\cdot\rvert_\forall$ measure; decl. subtyping strengthening
alg.thm	679	80	Basic algorithmic properties
trans.thm	616	53	Worklist instantiation and declarative transfer; Lemmas 4.3, 4.4
declTyping.thm	909	70	Non-overlapping declarative system; Lemmas 4.5, 4.6, 4.7, 4.8
soundness.thm	1,107	78	Soundness theorem; aux. lemmas on transfer
depth.thm	206	14	The $\lvert\cdot\rvert_\to$ measure; Lemma 4.12
dcl.thm	380	12	Non-overlapping declarative worklist
completeness.thm	1,124	61	Completeness theorem; aux. lemmas and relations
inst_decidable.thm	837	45	Other worklist measures; Lemma 4.16
decidability.thm	983	57	Decidability theorem and corollary
smallStep.thm	119	2	The equivalence between big-step and small-step
Total	7,977	596	(60 definitions in total)

Table 4.2: Translation Table for the Proof Scripts

Symbol	Abella textual relation	File (.thm)	Description
A	`ty, wft, wfta`	typing	The "type" type, decl./alg. well-formedness
e	`tm, wftm`	typing	The "term" type, (alg.) well-formedness
Ψ, Γ	`olist, wfj`	typing	`olist` is Abella's built-in list type, i.e. `[o]`
ω	`judgment, wfjg`	typing	Judgment chain and its well-formedness
$\Gamma \longrightarrow \Gamma'$	`reduction`	smallStep	Algorithmic reduction: paper version
$\Gamma \longrightarrow^* \cdot$	`judge`	typing	(`judge` Γ): a success reduction on Γ
$\Gamma \rightsquigarrow \Omega$	`tex`	trans	"transfer existential variables"
$\Omega \longrightarrow^* \cdot$	`dc, dcl`	trans, dcl	Declarative chain representation
$\Psi' \leq \Psi$	`esub`	declTyping	Declarative context subtyping
$\Gamma \rightharpoonup \Gamma'$	`jExt`	inst_decidable	"Judgment extension"
Γ_i	`iexp`	inst_decidable	instantiation judgments ($\Gamma \vdash_{wf} \Gamma_i$)
$\Gamma_{\leq}$	`subExp`	inst_decidable	subtyping judgments
$\lvert\cdot\rvert_e$	`tmSize, tmSizeJ, tmSizel`	declTyping, decidability	Size measure for term, judgment chain or context
$\lvert\cdot\rvert_{\Leftrightarrow}$	`m_judgeJ, m_judge`	decidability	Judgment measure for judgment chain or context
$\lvert\cdot\rvert_{\widehat{\alpha}}$	`nVar`	inst_decidable	Existential measure for context
$\lvert\cdot\rvert_{\forall}$	`order, orderJ, orderl`	order, inst_decidable	Polymorphism measure for type, judgment chain or context
$\lvert\cdot\rvert_{\rightarrow}$	`depth, depthJ, depthl`	order, decidability	Function measure for type, judgment chain or context

keep scoping precise, there is still a chance that unused existential variables leak their scope to an output context and accumulate indefinitely. For example, the derivation of the judgment $(\lambda x.\ x)\ () \Leftarrow 1$ is as follows

$$\dfrac{\dfrac{\dfrac{\dots x \Rightarrow \widehat{\alpha} \dots \qquad \dots \widehat{\alpha} \leq \widehat{\beta} \dots}{\Gamma, \widehat{\alpha}, \widehat{\beta}, x : \widehat{\alpha} \vdash x \Leftarrow \widehat{\beta} \dashv \Gamma_1, x : \widehat{\alpha}}}{\Gamma \vdash \lambda x.\ x \Rightarrow \widehat{\alpha} \to \widehat{\beta} \dashv \Gamma_1} \qquad \dfrac{\dots () \Leftarrow \widehat{\alpha} \dots}{\Gamma_1 \vdash \widehat{\alpha} \to \widehat{\alpha} \bullet () \Rrightarrow \widehat{\alpha} \dashv \Gamma_2}}{\Gamma \vdash (\lambda x.\ x)\ () \Rightarrow \widehat{\alpha} \dashv \Gamma_2}$$

$$\dfrac{\dfrac{}{\Gamma_2 \vdash 1 \leq 1 \dashv \Gamma_2}}{\Gamma \vdash (\lambda x.\ x)\ () \Leftarrow 1 \dashv \Gamma, \widehat{\alpha} = 1, \widehat{\beta} = \widehat{\alpha}}$$

where $\Gamma_1 := (\Gamma, \widehat{\alpha}, \widehat{\beta} = \widehat{\alpha})$ solves $\widehat{\beta}$, and $\Gamma_2 := (\Gamma, \widehat{\alpha} = 1, \widehat{\beta} = \widehat{\alpha})$ solves both $\widehat{\alpha}$ and $\widehat{\beta}$.

If the reader is not familiar with DK's algorithm, he/she might be confused about the inconsistent types across judgment. As an example, $(\lambda x.\ x)\ ()$ synthesizes $\widehat{\alpha}$, but the second premise of the subsumption rule uses 1 for the result. This is because a context application $[\Gamma, \widehat{\alpha} = 1, \widehat{\beta} = \widehat{\alpha}]\widehat{\alpha} = 1$ happens between the premises.

The existential variables $\widehat{\alpha}$ and $\widehat{\beta}$ are clearly not used after the subsumption rule, but according to the algorithm, they are kept in the context until some parent judgment recycles a block of variables, or to the very end of a type inference task. In that sense, DK's algorithm does not control the scoping of variables precisely.

Two rules we may blame for not garbage collecting correctly are the inference rule for lambda functions and an application inference rule:

$$\dfrac{\Gamma, \widehat{\alpha}, \widehat{\beta}, x : \widehat{\alpha} \vdash e \Leftarrow \widehat{\beta} \dashv \Delta, x : \widehat{\alpha}, \Theta}{\Gamma \vdash \lambda x.\ e \Rightarrow \widehat{\alpha} \to \widehat{\beta} \dashv \Delta}\ \texttt{DK\_}{\to}\texttt{I}{\Rightarrow} \qquad \dfrac{\Gamma, \widehat{\alpha} \vdash [\widehat{\alpha}/a]A \bullet e \Rrightarrow C \dashv \Delta}{\Gamma \vdash \forall a.\ A \bullet e \Rrightarrow C \dashv \Delta}\ \texttt{DK\_}\forall\texttt{App}$$

In contrast, Rule 25 of our algorithm collects the existential variables right after the second judgment chain, and Rule 27 collects one existential variable similarly:

$$\Gamma \Vdash \lambda x.\ e \Rightarrow_a \omega \ \longrightarrow_{25} \Gamma, \widehat{\alpha}, \widehat{\beta} \Vdash [\widehat{\alpha} \to \widehat{\beta}/a]\omega, x : \widehat{\alpha} \Vdash e \Leftarrow \widehat{\beta}$$

$$\Gamma \Vdash \forall a.\ A \bullet e \Rrightarrow_a \omega \ \longrightarrow_{27} \Gamma, \widehat{\alpha} \Vdash [\widehat{\alpha}/a]A \bullet e \Rrightarrow_a \omega$$

It seems impossible to achieve a similar outcome in DK's system by only modifying these two rules. Taking $\texttt{DK\_}{\to}\texttt{I}{\Rightarrow}$ as an example, the declaration or solution for $\widehat{\alpha}$ and $\widehat{\beta}$ may be

referred to by subsequent judgments. Therefore leaving $\widehat{\alpha}$ and $\widehat{\beta}$ in the output context is the only choice, when the subsequent judgments cannot be consulted.

To fix the problem, one possible modification is on the algorithmic subsumption rule, as garbage collection for a checking judgment is always safe:

$$\frac{\Gamma, \blacktriangleright_{\widehat{\alpha}} \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A \leq [\Theta]B \dashv \Delta, \blacktriangleright_{\widehat{\alpha}}, \Delta'}{\Gamma \vdash e \Leftarrow B \dashv \Delta} \texttt{DK\_Sub}$$

Here we employ the markers in a way they were originally not intended for. We create a dummy fresh existential $\widehat{\alpha}$ and add a marker to the input context of the inference judgment. After the subtyping judgment is processed we look for the marker and drop everything afterwards. We pick this rule because it is the only one where a checking judgment calls an inference judgment. That is the point where our algorithm recycles variables—right after a judgment chain is satisfied. After applying this patch, to the best of our knowledge, DK's algorithm behaves equivalently to our algorithm in terms of variable scoping. However, this exploits markers in a way they were not intended to be used and seems ad-hoc.

4.4.2 Elaboration

Type-inference algorithms are often extended with an associated elaboration. For example, for languages with implicit polymorphism, it is common to have an elaboration to a variant of System F [Reynolds 1983], which recovers type information and explicit type applications. Therefore a natural question is whether our algorithm can also accommodate such elaboration. While our algorithmic reduction does not elaborate to System F, we believe that it is not difficult to extend the algorithm with a (type-directed) elaboration. We explain the rough idea as follows.

Since the judgment form of our algorithmic worklist contains a collection of judgments, elaboration expressions are also generated as a list of equal length to the number of judgments (*not judgment chains*) in the worklist. As usual, subtyping judgments translate to coercions (denoted by f and represented by System F functions), all three other types of judgments translate to terms in System F (denoted by t).

Let Φ be the elaboration list, containing translated type coercions and terms:

$$\Phi ::= \cdot \mid \Phi, f \mid \Phi, t$$

Then the form of our algorithmic judgment becomes:

$$\Gamma \hookrightarrow \Phi$$

We take Rule 18 as an example, rewriting small-step reduction in a relational style,

$$\frac{\Gamma \Vdash e \Rightarrow_a a \leq B \hookrightarrow \Phi, f, t}{\Gamma \Vdash e \Leftarrow B \hookrightarrow \Phi, ft}\ \texttt{Translation\_18}$$

As is shown in the conclusion of the rule, a checking judgment at the top of the worklist corresponds to a top element for elaboration. The premise shows that one judgment chain may relate to more than one elaboration elements, and that the outer judgment, being processed before inner ones, elaborates to the top element in the elaboration list.

Moreover, existential variables need special treatment, since they may be solved at any point, or be recycled if not solved within their scopes. If an existential variable is solved, we not only propagate the solution to the other judgments, but also replace occurrences in the elaboration list. If an existential variable is recycled, meaning that it is not constrained, we can pick any well-formed monotype as its solution. The unit type 1, as the simplest type in the system, is a good choice.

4.4.3 Lexically-Scoped Type Variables

We have further extended the type system with support for lexically-scoped type variables [Peyton Jones and Shields 2004]. Our Abella formalization for this extension proves all the metatheory we discuss in Section 4.3.

From a practical point of view, this extension allows the implementation of a function to refer to type variables from its type signature. For example,

$$(\lambda x.\ \lambda y.\ (x : a)) : \forall a\ b.\ a \to b \to a$$

has an annotation $(x : a)$ that refers to the type variable a in the type signature. This is not a surprising feature, since the declarative system already accepts similar programs

$$\frac{\Psi \vdash \forall a.\ A \qquad \dfrac{\Psi, a \vdash e \Leftarrow A}{\Psi \vdash e \Leftarrow \forall a.\ A}\ \texttt{Decl}\forall\texttt{I}}{\Psi \vdash (e : \forall a.\ A) \Rightarrow \forall a.\ A}\ \texttt{DeclAnno}$$

The main issue is the well-formedness condition. Normally $\Psi \vdash (e : A)$ follows from $\Psi \vdash e$ and $\Psi \vdash A$. However, when $A = \forall a.\ A'$, the type variable a is not in scope at e,

therefore $\Psi \vdash e$ is not derivable. To address the problem, we add a new syntactic form that explicitly binds a type variable simultaneously in a function and its annotation.

$$\text{Expressions} \qquad e \quad ::= \qquad \ldots \mid \Lambda a.\, e : A$$

This new type-lambda syntax $\Lambda a.\, e : A$ actually annotates its body e with $\forall a.\, A$, while making a visible inside the body of the function. The well-formedness judgments are extended accordingly:

$$\frac{\Psi, a \vdash e \quad \Psi, a \vdash A}{\Psi \vdash \Lambda a.\, e : A}\ \mathtt{wf_d\Lambda} \qquad \frac{\Gamma, a \vdash e \quad \Gamma, a \vdash A}{\Gamma \vdash \Lambda a.\, e : A}\ \mathtt{wf\_\Lambda}$$

Corresponding rules are introduced for both the declarative system and the algorithmic system:

$$\frac{\Psi, a \vdash A \quad \Psi, a \vdash e \Leftarrow A}{\Psi \vdash \Lambda a.\, e : A \Rightarrow \forall a.\, A}\ \mathtt{Decl\Lambda}$$

$$\Gamma \Vdash \Lambda a.\, e : A \Rightarrow_b \omega \ \longrightarrow_{30} \Gamma \Vdash [(\forall a.\, A)/b]\omega, a \Vdash e \Leftarrow A$$

In practice, programmers would not write the syntax $\Lambda a.\, e : A$ directly. The ScopedTypeVariables extension of Haskell is effective only when the type signature is explicitly universally quantified (which the compiler translates into an expression similar to $\Lambda a.\, e : A$); otherwise the program means the normal syntax $e : \forall a.\, A$ and may not later refer to the type variable a.

We have proven all three desired properties for the extended system, namely soundness, completeness and decidability.

5 Higher-Rank Polymorphism with Object-Oriented Subtyping

5.1 Overview

In this chapter, we present a worklist algorithm that further supports subtyping, by introducing the top and bottom types. Such type inference algorithms are known to be hard to design, due to the complication of unifying variables under subtyping inequality rather than equality. Therefore, algorithmic existential variables that appear in subtyping judgments might have more solutions than before, and the eager substitution rules miss some possibilities. Our new *backtracking* algorithm extends the existing one by some overlapping rules that non-deterministically try different sorts of solutions instead of a single sort, improving the rate of success guessing.

Formalization of the algorithm in the Abella theorem prover shows soundness of the algorithm with respect to our declarative specification. Although the backtracking algorithm is still incomplete in some corner cases, the proof script indicates that the only source of incompleteness comes from higher-ranked subtyping relations. Following that discovery, we formally proved partial completeness of our subtyping algorithm under the rank-1 restriction, and is at least comparable to local type inference algorithms.

5.1.1 Type Inference in Presence of Subtyping

In all previous chapters, "subtyping" refers to a relationship that compares the degree of polymorphism between two types. In addition to that, we also include the top and bottom types. Both of the types have practical uses, especially in object-oriented programming languages. The top type, $\top$, is the super type of any type, i.e. any type is more general than $\top$ and thus can be considered as an instance of type $\top$. In typical object-oriented programming languages, the `Object` class, as the base class of any class, is the $\top$ type. In contrast, the bottom type, $\bot$, is dual to $\top$. An instance of $\bot$ can be casted to a value of any type, which is usually impossible, except when that is a `null pointer` value (`(void *)` in C++, for example). Another practical use for bottom types is for exceptions. The type Exception $\to \bot$ given to the `raise` function may pass type checkers naturally. The type Exception $\to \forall a.\ a$ is also a

reasonable choice, which in fact reveals that $\bot$ behaves almost identically as $\forall a.\ a$. From the theoretical point of view, both of them represent "falsity".

A number of systems and algorithms are proposed for type inference with subtyping. Since subtyping constraints cannot be easily reduced in presence of subtyping, it is natural to extend the syntax of types so that unsolved constraints are carried with them, and the idea is studied by previous work [Eifrig et al. 1995b; Trifonov and Smith 1996]. For example, the following function

```
select p v d = if (p v) then v else d
```

is inferred to have the type

$$(\alpha \to \text{bool}) \to \alpha \to \beta \to \gamma \mid \alpha \leq \gamma, \beta \leq \gamma$$

However, this representation usually contains a large set of constraints, which might confuse programmers. Besides, subtyping comparison between constraint types is also not easy.

MLsub MLsub [Dolan and Mycroft 2017], as introduced in Section 2.4, is based on a type system with lattices and polar types. Inspired by [Pottier 1998], they separated input and output types into polar types, which greatly simplifies the subtyping judgment. With the further help of their biunification algorithm, constraints are no longer carried with types, instead they are expressed directly on types. For example, the `select` function defined above has type

$$(\alpha \to \text{bool}) \to \alpha \to \beta \to \alpha \sqcup \beta$$

in MLsub. The $\sqcup$ symbol in the output type indicates that any of the types might be returned when executed. On the other hand, the $\sqcap$ symbol might occur on an input type, when that argument is used as different types, meaning that the type must be able to convert to all these types.

Nevertheless, there are still drawbacks in practice. Firstly, the type inference algorithm produces types that have comparable sizes to previous systems with constraint types. Although a simplification algorithm is provided, it is not guaranteed to produce the most simple form.

Secondly, MLsub always infers an expression and returns its principle type, but some of the types are not easy to understand. For example, the `twice` function defined as

```
twice f x = f (f x)
```

has type $(\alpha \sqcup \beta \to \alpha) \to \beta \to \alpha$, where normally people might expect $(\alpha \to \alpha) \to \alpha \to \alpha$.

Thirdly, MLsub tries to infer on almost every possible expression, even those that might just be a mistake made by the programmer. For instance, the function

```
positive f x = if x > 0 then f x else f 0
```

always checks the positivity of `x` before applying to `f`. However, a slight mistake is made in the following definition

```
positive' f x = if x > 0 then f x else f
```

then the algorithm might still accept the program by giving the type:

$$\alpha \sqcap (\beta \to \alpha) \to \beta \sqcap \mathsf{int} \to \alpha$$

Type inference algorithms, as logical tools that help people program, should accept good programs and reject bad ones. Typically, such a function is considered to be badly-written, but unfortunately, MLsub accepts it without giving any warnings.

5.1.2 Judgment List and Eager Substitution

The judgment list algorithm we discussed in the last chapter treats subtyping judgments mainly as equality unification. However, the algorithm may lose some solutions when top and bottom types are introduced. For example, the judgment $\widehat{\alpha} \leq 1$ has the best solution $\widehat{\alpha} := 1$ in the previous system, but now $\widehat{\alpha} := \bot$ should also be allowed. As a result, several subtyping judgments cannot be reduced easily with a single eager substitution.

Similar incompleteness also affects the instantiation and existential variable solving rules

$$\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \leq A \to B \longrightarrow_{10} [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2] \Vdash \widehat{\alpha}_1 \to \widehat{\alpha}_2 \leq A \to B)$$
$$\Gamma[\widehat{\alpha}][\widehat{\beta}] \Vdash \widehat{\alpha} \leq \widehat{\beta} \longrightarrow_{12} [\widehat{\alpha}/\widehat{\beta}](\Gamma[\widehat{\alpha}][])$$

The instantiation judgment $\widehat{\alpha} \leq A \to B$ immediately split $\widehat{\alpha}$ into $\widehat{\alpha}_1 \to \widehat{\alpha}_2$, which is no longer the case — $\widehat{\alpha}$ can also be $\bot$.

The rule that solves one existential variable against another, $\widehat{\alpha} \leq \widehat{\beta}$, is even more problematic, because the shapes of the instantiations of both existential variables are unknown, there are infinitely many possibilities if no additional constraints present. For example, if $\widehat{\alpha}$ is finally instantiated to Int $\to$ Int, then $\widehat{\beta}$ may be solved to the same type as $\widehat{\alpha}$, or other types that satisfy the subtyping relationship, including Int $\to \top$, $\bot \to$ Int or $\bot \to \top$. What's worse, given only this single subtyping judgment $\widehat{\alpha} \leq \widehat{\beta}$, we have no assumption on both existential variables.

5.1.3 Our Solution: Backtracking Algorithm

After examining the examples where the worklist algorithm behaves incompletely, we propose some improvements to the worklist algorithm to accept some simple cases. We have observed that subtyping judgments like $\widehat{\alpha} \leq A$ and $A \leq \widehat{\alpha}$ have trivial solutions $\widehat{\alpha} := \bot$ and $\widehat{\alpha} := \top$ respectively, thus we can try these simple solutions before the more complicated analysis.

For example, for the judgment list $\Gamma \Vdash \widehat{\alpha} \leq A \to B$, we first assign $\widehat{\alpha} := \bot$ and continue to check other judgments. If the judgment reduction succeeds, then we know that $\widehat{\alpha} := \bot$ is a possible solution. If the reduction failed, we continue to try the other possibility according to the previous instantiation rule by splitting $\widehat{\alpha}$ into two existential variables $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$. Note that for the reduced judgment $\widehat{\alpha}_1 \to \widehat{\alpha}_2 \leq A \to B$, $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ might be solved to $\top$ or $\bot$.

For the type system, there are also interesting changes to both the declarative system and algorithmic system. Two algorithmic typing rules also adopt the backtracking approach:

- Checking judgment $e \Leftarrow \widehat{\alpha}$ now has a trivial solution $\widehat{\alpha} := \top$, because $e \Leftarrow \top$ should be accepted for any (well-formed) expression.
- The introduction of top and bottom types come with another new declarative rule $\bot \bullet e \implies \bot$. Therefore the application inference judgment $\widehat{\alpha} \bullet e \implies_a \omega$ can be satisfied by $\widehat{\alpha} := \bot$, with output $\bot$ applied to the rest of the judgment chain ω.

By applying the above modifications, the algorithm remains sound w.r.t our new declarative system, and at the same time, we address most issues regarding the top and bottom types, obtaining a more complete algorithm. Furthermore, we formally verified that the subtyping algorithm is complete under the rank-1 restriction. However, for rank-2 judgments like $\widehat{\alpha} \leq \widehat{\beta}$, completeness is impossible for any algorithm that reduces eagerly.

Following the formal statements, we conclude that the backtracking algorithm is complete in most cases other than those where higher-ranked subtyping judgments are involved. In those cases, we only consider the equality case and ignore any other possibilities, and we believe that this is a practical tradeoff. In another perspective, our algorithm does not try to solve completely when parametric polymorphism and subtyping polymorphism occur at the same time.

5.2 Declarative System

Syntax The syntax of the declarative system, shown in Figure 5.1, is similar to the previous systems by having a primitive type 1, type variables a, polymorphic types $\forall a.\ A$ and function types $A \to B$. Additionally, top and bottom types are introduced to the type system.

$$
\begin{array}{rlll}
\text{Type variables} & a, b & & \\
\text{Types} & A, B, C & ::= & 1 \mid \top \mid \bot \mid a \mid \forall a.\, A \mid A \to B \\
\text{Monotypes} & \tau & ::= & 1 \mid \top \mid \bot \mid a \mid \tau_1 \to \tau_2 \\
\text{Expressions} & e & ::= & x \mid () \mid \lambda x.\, e \mid e_1\, e_2 \mid (e : A) \\
\text{Context} & \Psi & ::= & \cdot \mid \Psi, a \mid \Psi, x : A
\end{array}
$$

Figure 5.1: Declarative Syntax

$\boxed{\Psi \vdash A \leq B}$

$$
\frac{a \in \Psi}{\Psi \vdash a \leq a}\ \leq\texttt{Var} \qquad \frac{}{\Psi \vdash 1 \leq 1}\ \leq\texttt{Unit} \qquad \frac{\Psi \vdash B_1 \leq A_1 \quad \Psi \vdash A_2 \leq B_2}{\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2}\ \leq\to
$$

$$
\frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \leq B}{\Psi \vdash \forall a.\, A \leq B}\ \leq\forall\texttt{L} \qquad \frac{\Psi, b \vdash A \leq B}{\Psi \vdash A \leq \forall b.\, B}\ \leq\forall\texttt{R}
$$

$$
\frac{}{\boldsymbol{A} \leq \top}\ \leq\textbf{Top} \qquad \frac{}{\bot \leq \boldsymbol{A}}\ \leq\textbf{Bot}
$$

Figure 5.2: Declarative Subtyping

The well-formedness formalization of the system is standard and almost identical to the previous systems, therefore we omit the formal definitions.

Declarative Subtyping Shown in Figure 5.2, the declarative subtyping extends the polymorphic subtyping relation originally proposed by Odersky and Läufer [1996] by adding rules $\leq$Top and $\leq$Bot, defining the properties of the $\top$ and $\bot$ types, respectively. Although the new rules seem quite simple, they may increase the uncertainty of polymorphic instantiations. For example, the subtyping judgment

$$\forall a.\, a \to a \leq \bot \to \top$$

accepts any well-formed instantiation on the polymorphic type $\forall a.\, a \to a$.

Declarative Typing The declarative typing rules, shown in Figure 5.3, extend DK's higher-rank type system in order to support the top and bottom types. Rule Decl$\top$ allows any well-formed expression to check against $\top$. Rule Decl$\bot$App returns the $\bot$ type when a function of $\bot$ type is applied to any argument. All other rules remain exactly the same as our previous work.

$\boxed{\Psi \vdash e \Leftarrow A}$ e checks against input type A.

$\boxed{\Psi \vdash e \Rightarrow A}$ e synthesizes output type A.

$\boxed{\Psi \vdash A \bullet e \Rightarrow\!\!\!\Rightarrow C}$ Applying a function of type A to e synthesizes type C.

$$\frac{(x : A) \in \Psi}{\Psi \vdash x \Rightarrow A}\ \texttt{DeclVar} \qquad \frac{\Psi \vdash e \Rightarrow A \quad \Psi \vdash A \leq B}{\Psi \vdash e \Leftarrow B}\ \texttt{DeclSub}$$

$$\frac{\Psi \vdash A \quad \Psi \vdash e \Leftarrow A}{\Psi \vdash (e : A) \Rightarrow A}\ \texttt{DeclAnno} \qquad \frac{}{\Psi \vdash () \Rightarrow 1}\ \texttt{Decl1I}\Rightarrow$$

$$\frac{}{\Psi \vdash () \Leftarrow 1}\ \texttt{Decl1I} \qquad \frac{\Psi \vdash e}{\Psi \vdash e \Leftarrow \top}\ \texttt{Decl}\top \qquad \frac{\Psi \vdash e}{\Psi \vdash \bot \bullet e \Rightarrow\!\!\!\Rightarrow \bot}\ \texttt{Decl}\bot\texttt{App}$$

$$\frac{\Psi, a \vdash e \Leftarrow A}{\Psi \vdash e \Leftarrow \forall a.\ A}\ \texttt{Decl}\forall\texttt{I} \qquad \frac{\Psi \vdash \tau \quad \Psi \vdash [\tau/a]A \bullet e \Rightarrow\!\!\!\Rightarrow C}{\Psi \vdash \forall a.\ A \bullet e \Rightarrow\!\!\!\Rightarrow C}\ \texttt{Decl}\forall\texttt{App}$$

$$\frac{\Psi, x : A \vdash e \Leftarrow B}{\Psi \vdash \lambda x.\ e \Leftarrow A \to B}\ \texttt{Decl}{\to}\texttt{I} \qquad \frac{\Psi \vdash \sigma \to \tau \quad \Psi, x : \sigma \vdash e \Leftarrow \tau}{\Psi \vdash \lambda x.\ e \Rightarrow \sigma \to \tau}\ \texttt{Decl}{\to}\texttt{I}\Rightarrow$$

$$\frac{\Psi \vdash e_1 \Rightarrow A \quad \Psi \vdash A \bullet e_2 \Rightarrow\!\!\!\Rightarrow C}{\Psi \vdash e_1\ e_2 \Rightarrow C}\ \texttt{Decl}{\to}\texttt{E} \qquad \frac{\Psi \vdash e \Leftarrow A}{\Psi \vdash A \to C \bullet e \Rightarrow\!\!\!\Rightarrow C}\ \texttt{Decl}{\to}\texttt{App}$$

Figure 5.3: Declarative Typing

It's worth mentioning that the design of the two new rules is driven by the subsumption property described in Section 5.4.1. They maintain the property in presence of a more powerful declarative subtyping, and we will discuss further later in that part.

5.3 Backtracking Algorithm

5.3.1 Syntax

Existential variables	$\widehat{\alpha}, \widehat{\beta}$		
Types	A, B, C	$::=$	$1 \mid \top \mid \bot \mid a \mid \forall a.\ A \mid A \to B \mid \widehat{\alpha}$
Algorithmic judgment chain	ω	$::=$	$A \leq B \mid e \Leftarrow A \mid e \Rightarrow_a \omega \mid A \bullet e \Rightarrow\!\!\!\Rightarrow_a \omega$
Algorithmic worklist	Γ	$::=$	$\cdot \mid \Gamma, a \mid \Gamma, \widehat{\alpha} \mid \Gamma \Vdash \omega$

Figure 5.4: Algorithmic Syntax

The algorithmic syntax is shown in Figure 5.4. Compared with the declarative system, existential variables $\widehat{\alpha}, \widehat{\beta}$ are used as placeholders for unsolved mono-types. The judgment chain ω and worklist context Γ are defined in the same way as the ICFP work.

The well-formedness relation is almost the same as that of the ICFP work. The hole notation is also inherited.

THE SOLVE NOTATION We define a set of auxiliary substitution functions in the form $\{\widehat{\alpha} := \tau_A\}$ to improve readability of our algorithmic definitions. Basically it refers to a global substitution when solving an existential variable.

A detailed definition of the notation is shown as follows:

$$\begin{aligned}
\{\widehat{\alpha} := \tau\}\ (\Gamma_L, \widehat{\alpha}, \Gamma_R) &= \Gamma_L, [\tau/\widehat{\alpha}]\Gamma_R && \tau = 1, \top, \bot \text{ or } a \\
\{\widehat{\alpha} := \widehat{\beta}\}\ (\Gamma_L, \widehat{\alpha}, \Gamma_R) &= \Gamma_L, [\widehat{\beta}/\widehat{\alpha}]\Gamma_R && \widehat{\beta} \in \Gamma_L \\
\{\widehat{\alpha} := \widehat{\alpha}_1 \to \widehat{\alpha}_2\}\ (\Gamma_L, \widehat{\alpha}, \Gamma_R) &= \Gamma_L, \widehat{\alpha}_1, \widehat{\alpha}_2, [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}]\Gamma_R
\end{aligned}$$

5.3.2 ALGORITHMIC SUBTYPING

Figure 5.5 describes the algorithmic rules for subtyping. The relation is stated in a small-step "reduction" form, i.e. in each step, the worklist is analysed from the right-hand-side and reduced according to the top judgment. The overall procedure succeeds iff the worklist eventually reduces to $\cdot$ (the empty worklist).

We categorize them into 7 groups according to their behavior:

1. Rules 1-3 are basic garbage collection rules. Given that the worklist Γ is well-formed, no reference of a variable should occur before its declaration. Therefore removing the declaration in the top position does not break well-formedness.

   An existential variable that is unsolved in the top position indicates that it is not constrained, thus picking any well-formed mono-type as its solution is acceptable. In our algorithmic formalization, we simply drop the existential variable.

2. Rules 4-10 directly correspond to the declarative subtyping rules. With no top-level existential variables, there is nothing to guess immediately, and thus the algorithm behaves just like the declarative system.

3. Rule 11 is a base case in the algorithmic system. The declarative reflexivity property suggests that any solution is acceptable, thus the judgment holds without any constraint.

4. Rules 12-13 are important rules that require backtracking techniques for implementation. These rules *overlap* with all the remaining rules when solving an existential

$\boxed{\Gamma \longrightarrow \Gamma'}$ Γ reduces to Γ'.

$$
\begin{array}{rcl}
\Gamma, a & \longrightarrow_1 & \Gamma \\
\Gamma, \widehat{\alpha} & \longrightarrow_2 & \Gamma \\
\Gamma, x : A & \longrightarrow_3 & \Gamma \\
\\
\Gamma \Vdash 1 \leq 1 & \longrightarrow_4 & \Gamma \\
\Gamma \Vdash a \leq a & \longrightarrow_5 & \Gamma \\
\Gamma \Vdash A_1 \to A_2 \leq B_1 \to B_2 & \longrightarrow_6 & \Gamma \Vdash A_2 \leq B_2 \Vdash B_1 \leq A_1 \\
\Gamma \Vdash \forall a.A \leq B & \longrightarrow_7 & \Gamma, \widehat{\alpha} \Vdash [\widehat{\alpha}/a]A \leq B \\
\Gamma \Vdash A \leq \forall b.B & \longrightarrow_8 & \Gamma, b \Vdash A \leq B \\
\Gamma \Vdash A \leq \top & \longrightarrow_9 & \Gamma \\
\Gamma \Vdash \bot \leq B & \longrightarrow_{10} & \Gamma \\
\\
\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \leq \widehat{\alpha} & \longrightarrow_{11} & \Gamma \\
\\
\Gamma[\widehat{\alpha}] \Vdash A \leq \widehat{\alpha} & \longrightarrow_{12} & \{\widehat{\alpha} := \top\}\, \Gamma[\widehat{\alpha}] \\
\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \leq A & \longrightarrow_{13} & \{\widehat{\alpha} := \bot\}\, \Gamma[\widehat{\alpha}] \\
\\
\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \leq A \to B & \longrightarrow_{14} & \{\widehat{\alpha} := \widehat{\alpha}_1 \to \widehat{\alpha}_2\}\, (\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \leq A \to B) \\
 & & \quad \text{when } \widehat{\alpha} \notin FV(A \to B) \\
\Gamma[\widehat{\alpha}] \Vdash A \to B \leq \widehat{\alpha} & \longrightarrow_{15} & \{\widehat{\alpha} := \widehat{\alpha}_1 \to \widehat{\alpha}_2\}\, (\Gamma[\widehat{\alpha}] \Vdash A \to B \leq \widehat{\alpha}) \\
 & & \quad \text{when } \widehat{\alpha} \notin FV(A \to B) \\
\\
\Gamma[a][\widehat{\beta}] \Vdash a \leq \widehat{\beta} & \longrightarrow_{16} & \{\widehat{\beta} := a\}\, \Gamma[a][\widehat{\beta}] \\
\Gamma[a][\widehat{\beta}] \Vdash \widehat{\beta} \leq a & \longrightarrow_{17} & \{\widehat{\beta} := a\}\, \Gamma[a][\widehat{\beta}] \\
\Gamma[\widehat{\beta}] \Vdash 1 \leq \widehat{\beta} & \longrightarrow_{18} & \{\widehat{\beta} := 1\}\, \Gamma[\widehat{\beta}] \\
\Gamma[\widehat{\beta}] \Vdash \widehat{\beta} \leq 1 & \longrightarrow_{19} & \{\widehat{\beta} := 1\}\, \Gamma[\widehat{\beta}] \\
\\
\Gamma[\widehat{\alpha}][\widehat{\beta}] \Vdash \widehat{\alpha} \leq \widehat{\beta} & \longrightarrow_{20} & \{\widehat{\beta} := \widehat{\alpha}\}\, \Gamma[\widehat{\alpha}][\widehat{\beta}] \\
\Gamma[\widehat{\alpha}][\widehat{\beta}] \Vdash \widehat{\beta} \leq \widehat{\alpha} & \longrightarrow_{21} & \{\widehat{\beta} := \widehat{\alpha}\}\, \Gamma[\widehat{\alpha}][\widehat{\beta}]
\end{array}
$$

Figure 5.5: Algorithmic Garbage Collection and Subtyping

variable. In other words, they simply try if $\top$ or $\bot$ satisfies the constraints in parallel with other possibilities.

5. Rules 14-15 compares an existential variable $\widehat{\alpha}$ with a function type, resulting in solving the $\widehat{\alpha}$ by $\widehat{\alpha}_1 \to \widehat{\alpha}_2$. The freshness condition rules out the possibility when there is a cyclic dependency. For example, the judgment

$$\widehat{\alpha} \leq 1 \to \widehat{\alpha}$$

is satisfied with either of these solutions to $\widehat{\alpha}$:

$$\bot, 1 \to \bot, \top \to \bot, 1 \to 1 \to \bot, \ldots$$

However, we argue that comparing $\widehat{\alpha}$ with a function type that contains $\widehat{\alpha}$ itself is hardly useful in practice, and most of the solutions are meaningless. Therefore, in our algorithm, only the $\bot$ solution is considered with rule 13. The condition of rule 14 rejects the judgment for further analysis and thus does not produce more solutions. This is one source of incompleteness of our algorithm with respect to the declarative specification.

6. Rules 16-19 solve existential variables against a type variable or the unit type. For instance, the judgment

$$\widehat{\alpha} \leq 1$$

only has two solutions: $\widehat{\alpha} = 1$ or $\widehat{\alpha} = \bot$. In similar cases, one of the solutions is produced by rule 12 or 13, and the other one is given by one of the rules in this group. Additional well-formedness checks are performed when type variables are encountered; a solution to an existential variable must be well-formed in the context before the existential variable is defined.

7. Rules 20-21 deal with subtyping judgments that compare two different existential variables. The only difference between them is the variable order. Similar to the type variable case in the previous group, existential variables must solve to another one defined earlier. With rules 12, 13, 20 and 21, a judgment like

$$\widehat{\alpha} \leq \widehat{\beta}$$

could possibly give any of the following solutions:

$$\widehat{\alpha} = \widehat{\beta} \text{ (or } \widehat{\beta} = \widehat{\alpha}) \text{ or } \widehat{\alpha} = \bot \text{ or } \widehat{\beta} = \top$$

Those are good attempts, but unfortunately, they do not cover the complete set of possibilities. The following example worklist

$$\widehat{\alpha}, \widehat{\beta} \Vdash \widehat{\beta} \leq 1 \to 1 \Vdash \widehat{\alpha} \leq \widehat{\beta}$$

has a solution $\widehat{\alpha} = 1 \to \top, \widehat{\beta} = 1 \to 1$ missed by our algorithm. Similar situations happen when the judgments are specifically ordered; if $\widehat{\beta} \leq 1 \to 1$ is the top-most judgment, the algorithm will not miss this solution.

Although such treatment for existential variable solving is incomplete in theory, a large number of practical unification is simply equality, and the algorithm completely handles these programs. For other programs that exploit complex guessing involving subtyping, the programmer may put type annotations when the type inference algorithm does not find the optimal solution.

5.3.3 Algorithmic Typing

$\boxed{\Gamma \longrightarrow \Gamma'}$ Γ reduces to Γ' (continued).

$$\begin{aligned}
\Gamma \Vdash e \Leftarrow B &\longrightarrow_{22} \Gamma \Vdash e \Rightarrow_a a \leq B \quad \text{when } e \neq \lambda x.\, e' \text{ and } B \neq \forall a.\, B' \\
\Gamma \Vdash e \Leftarrow \forall a.\, A &\longrightarrow_{23} \Gamma, a \Vdash e \Leftarrow A \\
\Gamma \Vdash \lambda x.\, e \Leftarrow A \to B &\longrightarrow_{24} \Gamma, x : A \Vdash e \Leftarrow B \\
\Gamma[\widehat{\alpha}] \Vdash \lambda x.\, e \Leftarrow \widehat{\alpha} &\longrightarrow_{25} \{\widehat{\alpha} := \widehat{\alpha}_1 \to \widehat{\alpha}_2\}\, \Gamma, x : \widehat{\alpha}_1 \Vdash e \Leftarrow \widehat{\alpha}_2 \\
\Gamma \Vdash e \Leftarrow \top &\longrightarrow_{26} \Gamma \\
\Gamma[\widehat{\alpha}] \Vdash e \Leftarrow \widehat{\alpha} &\longrightarrow_{27} \{\widehat{\alpha} := \top\}\, \Gamma[\widehat{\alpha}]
\end{aligned}$$

$$\begin{aligned}
\Gamma \Vdash x \Rightarrow_a \omega &\longrightarrow_{28} \Gamma \Vdash [A/a]\omega \quad \text{when } (x : A) \in \Gamma \\
\Gamma \Vdash (e : A) \Rightarrow_a \omega &\longrightarrow_{29} \Gamma \Vdash [A/a]\omega \Vdash e \Leftarrow A \\
\Gamma \Vdash () \Rightarrow_a \omega &\longrightarrow_{30} \Gamma \Vdash [1/a]\omega \\
\Gamma \Vdash \lambda x.\, e \Rightarrow_a \omega &\longrightarrow_{31} \Gamma, \widehat{\alpha}, \widehat{\beta} \Vdash [\widehat{\alpha} \to \widehat{\beta}/a]\omega, x : \widehat{\alpha} \Vdash e \Leftarrow \widehat{\beta} \\
\Gamma \Vdash e_1\, e_2 \Rightarrow_a \omega &\longrightarrow_{32} \Gamma \Vdash e_1 \Rightarrow_b (b \bullet e_2 \Rrightarrow_a \omega)
\end{aligned}$$

$$\begin{aligned}
\Gamma \Vdash \forall a.\, A \bullet e \Rrightarrow_a \omega &\longrightarrow_{33} \Gamma, \widehat{\alpha} \Vdash [\widehat{\alpha}/a]A \bullet e \Rrightarrow_a \omega \\
\Gamma \Vdash A \to C \bullet e \Rrightarrow_a \omega &\longrightarrow_{34} \Gamma \Vdash [C/a]\omega \Vdash e \Leftarrow A \\
\Gamma \Vdash \bot \bullet e \Rrightarrow_a \omega &\longrightarrow_{35} \Gamma \Vdash [\bot/a]\omega \\
\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \bullet e \Rrightarrow_a \omega &\longrightarrow_{36} \{\widehat{\alpha} := \widehat{\alpha}_1 \to \widehat{\alpha}_2\}\, \Gamma \Vdash \widehat{\alpha}_1 \to \widehat{\alpha}_2 \bullet e \Rrightarrow_a \omega \\
\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \bullet e \Rrightarrow_a \omega &\longrightarrow_{37} \{\widehat{\alpha} := \bot\}\, \Gamma[\widehat{\alpha}] \Vdash [\bot/a]\omega
\end{aligned}$$

Figure 5.6: Algorithmic Typing

The algorithmic typing rules are split into three groups, according to the category of the top-most judgment.

1. Checking mode. Rules 22-27 reduce top-level checking judgments. Rules 22, 23, 24 and 26 directly reflect how the declarative system behaves. Rule 25 splits $\widehat{\alpha}$ into $\widehat{\alpha}_1 \to \widehat{\alpha}_2$ to mimic the same logic as rule 24, since a lambda expression must be of a function type. Rule 27 is another backtracking rule, which overlaps with all other checking rules. Solving $\widehat{\alpha}$ to $\top$ in such case reflects the declarative rule `Decl⊤` when the unknown type is $\top$. Furthermore, we eliminate the algorithmic counterpart of declarative rule `Decl1I`, because a combination of Rules 22 and 30 already accepts the judgment $() \Leftarrow 1$.

2. Inference mode. Rules 28-32 reduce inference judgments. Like the work presented in Chapter 4, the encoding of the return type is by an explicit substitution on the binder of a judgment chain. Rules 28, 29, 30 and 32 correspond to the declarative system straightforwardly. Rule 31 guesses an unannotated lambda with a mono function type $\widehat{\alpha} \to \widehat{\beta}$. Writing in a slightly different way may help improve readability:

$$\Gamma \Vdash \lambda x.\, e \Rightarrow_a \omega \;\longrightarrow_{31'} \Gamma, \widehat{\alpha}, \widehat{\beta} \Vdash [\widehat{\alpha} \to \widehat{\beta}/a]\omega \Vdash \lambda x.\, e \Leftarrow \widehat{\alpha} \to \widehat{\beta}$$

   Rule 32 illustrates how the judgment chain works, with a continuation-passing-style encoding of the type inference task.

3. Application inference mode. Rules 33-37 reduce application inference judgments. Each of these judgments accepts an input function type and an argument expression and produces the expected return type. Rules 33, 34 and 35 are direct translations from the declarative rules. Rule 33, specifically, enables implicit parametric polymorphism via existential variable solving. Rules 36 and 37 deal with cases where a single existential variable $\widehat{\alpha}$ behaves as a function type. Rule 36 splits $\widehat{\alpha}$ into a function unknown type $\widehat{\alpha}_1 \to \widehat{\alpha}_2$. Rule 37 tries the solution $\widehat{\alpha} = \bot$ and returns the $\bot$ type.

5.4 Metatheory

In this section we present several properties formally verified. For the declarative system, the typing subsumption and subtyping transitivity lemmas are discussed in detail. The algorithmic system is proven to be sound with respect to the declarative system via a transfer relation. A partial completeness theorem is shown under the rank-1 restriction. We then briefly describe the challenges we face when proving termination. Lastly, proof statistics of Abella are discussed.

5.4.1 Declarative Properties

The Typing Subsumption Lemma. An important desired property for a type system is *checking subsumption*, which basically says that any expression can check against any super type of its actual type. Since our bidirectional type system defines the checking mode, inference mode and application inference mode mutually, we formalize the generalized *typing subsumption*.

First of all, we give the definition of worklist subtyping, which is used to further generalize the typing subsumption lemma. This is necessary because rules like `Decl→I` will push the argument type A into the context, thus when checking against a super type of $A \to B$, say $C \to D$, will cause the bind of x in the context to a subtype of A (since $C \leq A$).

Definition 5 (Worklist Subtyping)**.** Worklist subtyping compares the type of variables bound in the worklist. $\Psi <: \Psi'$ iff each binding in Ψ is converted to one with a super type.

$$\frac{}{\cdot <: \cdot} <: \texttt{nil} \qquad \frac{\Psi <: \Psi'}{\Psi, a <: \Psi', a} <: \texttt{ty}$$

$$\frac{\Psi' \vdash A \leq B \qquad \Psi <: \Psi'}{\Psi, x : A <: \Psi', x : B} <: \texttt{of} \qquad \frac{\Psi <: \Psi'}{\Psi \Vdash \omega <: \Psi' \Vdash \omega} <: \omega$$

A basic property of worklist subtyping is that they acts similarly when dealing with subtyping between well-formed types.

Lemma 5.1 (Worklist Subtyping Equivalence)**.**
Given $\Psi <: \Psi'$, $\Psi \vdash A \leq B \iff \Psi' \vdash A \leq B$.

Finally, we give the statement of typing subsumption lemma, which is generalized by the worklist subtyping relation.

Lemma 5.2 (Typing Subsumption)**.** *Given* $\Psi <: \Psi'$,

1) If $\Psi' \vdash e \Leftarrow A$ *and* $\Psi' \vdash A \leq B$, *then* $\Psi \vdash e \Leftarrow B$;

2) If $\Psi' \vdash e \Rightarrow A$, *then* $\exists B$ *s.t.* $\Psi' \vdash B \leq A$ *and* $\Psi \vdash e \Rightarrow B$.

3) If $\Psi' \vdash C \bullet e \Rrightarrow A$ *and* $\Psi' \vdash D \leq C$, *then* $\exists B$ *s.t.* $\Psi' \vdash B \leq A$ *and* $\Psi \vdash D \bullet e \Rrightarrow B$.

Proof. By induction on the following size measure (lexicographical order on a 3-tuple):

- Checking ($e \Leftarrow A$): $\langle |e|, 1, |A|_\forall + |B|_\forall \rangle$
- Inference ($e \Rightarrow A$): $\langle |e|, 0, 0 \rangle$

- Application inference ($A \bullet e \Rrightarrow C$): $\langle |e|, 2, |C|_\forall + |D|_\forall \rangle$

Most of the cases are straightforward. When rule $\leq\forall\mathtt{L}$ is applied for the subtyping predicate like $\Psi' \vdash A \leq B$, a mono-type substitution is performed on $\forall a.\ A$, resulting in $[\tau/a]A$. Since τ is a mono-type, the result type reduces the number of $\forall$'s, and thus reduces the size measure. □

Interestingly, the two new declarative rules $\mathtt{Decl}\top$ and $\mathtt{Decl}\bot\mathtt{App}$ are discovered when we were trying to prove the property instead of before exploring the meta-theory. Given the typing and subtyping judgments $\Psi \vdash e \Leftarrow A$ and $\Psi \vdash A \leq \top$, we should derive $\Psi \vdash e \Leftarrow \top$ from the lemma, therefore Rule $\mathtt{Decl}\top$ is required, saying that any expression can be checked against the top type. Similarly, the most general type $\bot$, being able to convert to any type due to Rule $\leq\mathtt{Bot}$, can be converted to any function type, or simply the most general one $\top \to \bot$, which accepts any input and returns the $\bot$ type, resulting in the derivation $\Psi \vdash \bot \bullet e \Rrightarrow \bot$. From the lemma we can also derive that by $\Psi \vdash C \bullet e \Rrightarrow A$, $\Psi \vdash \bot \leq C$ and $\Psi \vdash \bot \leq A$.

With the addition of Rules $\mathtt{Decl}\top$ and $\mathtt{Decl}\bot\mathtt{App}$, we can prove the typing subsumption lemma. To the best of the authors' knowledge, they are the minimal set of rules that make the lemma hold.

Transitivity of Subtyping The transitivity lemma for declarative subtyping is a commonly expected property. The proof depends on the following subtyping derivation size relation and an auxiliary lemma.

Definition 6 (Subtyping Derivation Size)**.**

$$
\begin{aligned}
|1 \leq 1| &= 0 \\
|a \leq a| &= 0 \\
|A \leq \top| &= 0 \\
|\bot \leq B| &= 0 \\
|A_1 \to A_2 \leq B_1 \to B_2| &= |B_1 \leq A_1| + |A_2 \leq B_2| + 1 \\
|\forall a.\ A \leq B| &= |[\tau/a]A \leq B| + 1 \\
|A \leq \forall a.\ B| &= |A \leq B| + 1
\end{aligned}
$$

Lemma 5.3 (Monotype Subtyping Substitution)**.** *If* $\Psi \vdash \tau$ *and* $\Psi, a, \Psi_R \vdash A \leq B$*, then* $\Psi, [\tau/a]\Psi_R \vdash [\tau/a]A \leq [\tau/a]B$.

Proof. A routine induction on the subtyping relation $\Psi, a, \Psi_R \vdash A \leq B$ finishes the proof. □

$$\text{Declarative worklist} \qquad \Omega \quad ::= \quad \cdot \mid \Omega, a \mid \Omega, x : A \mid \Omega \Vdash \omega$$

$$\boxed{\Gamma \rightsquigarrow \Omega} \qquad\qquad \Gamma \text{ instantiates to } \Omega.$$

$$\frac{}{\Omega \rightsquigarrow \Omega}\ \rightsquigarrow\Omega \qquad \frac{\Omega \vdash \tau \qquad \Omega, [\tau/\widehat{\alpha}]\Gamma \rightsquigarrow \Omega}{\Omega, \widehat{\alpha}, \Gamma \rightsquigarrow \Omega}\ \rightsquigarrow\widehat{\alpha}$$

Figure 5.7: Declarative Worklists and Instantiation

Corollary 5.4 (Monotype Subtyping Substitution for Type Variables). *If* $\Psi \vdash \tau$ *and* $\Psi, a \vdash A \leq B$*, then* $\Psi \vdash [\tau/a]A \leq [\tau/a]B$.

The above lemma and corollary reveal the fact that a type variable occurred in the subtyping relation represents an *arbitrary* well-formed monotype. And it also explains the difference in treatment of polymorphic types between Rules $\leq\forall\mathtt{L}$ and $\leq\forall\mathtt{R}$: Rule $\leq\forall\mathtt{R}$ is in fact equivalent to:

$$\frac{\forall\tau \text{ s.t. } \Psi \vdash \tau \Longrightarrow \Psi \vdash A \leq [\tau/b]B}{\Psi \vdash A \leq \forall b.\ B}\ \leq\forall\mathtt{R}'$$

Finally, with the size measure defined and required lemma proven, we can obtain the transitivity lemma for declarative subtyping.

Lemma 5.5 (Subtyping Transitivity). *If* $\Psi \vdash A \leq B$ *and* $\Psi \vdash B \leq C$ *then* $\Psi \vdash A \leq C$.

Proof. Induction on the lexicographical order defined by $\langle |B|_\forall, |A \leq B| + |B \leq C| \rangle$. Most cases preserve the first element of the size measures $|B|_\forall$, and are relatively easy to prove. The difficult case is when B is a polymorphic type, when the conditions are $\Psi \vdash A \leq \forall a.\ B$ and $\Psi \vdash \forall a.\ B \leq C$. They are derived through rules $\leq\forall\mathtt{L}$ and $\leq\forall\mathtt{R}$, respectively. Therefore, we have $\Psi, a \vdash A \leq B$ and $\Psi \vdash [\tau/a]B \leq C$. To exploit the induction hypothesis, the contexts should be unified. By Corollary 5.4, $\Psi \vdash A \leq [\tau/a]B$. Notice that the freshness condition is implicit for rule $\leq\forall\mathtt{L}$. Clearly, $|[\tau/a]B|_\forall < |\forall a.\ B|_\forall$, i.e. the first size measure decreases. By induction hypothesis we get $\Psi \vdash A \leq C$ and finish this case. □

5.4.2 Transfer

Follow the approach of Section 4.3, the transfer relation and the declarative instantiation relation are defined in Figure 5.7.

Similarly, Lemmas 5.6 and 5.7 generalizing Rule $\rightsquigarrow\widehat{\alpha}$ hold as well.

Lemma 5.6 (Insert). *If* $\Gamma_L, [\tau/\widehat{\alpha}]\Gamma_R \rightsquigarrow \Omega$ *and* $\Gamma_L \vdash \tau$ *, then* $\Gamma_L, \widehat{\alpha}, \Gamma_R \rightsquigarrow \Omega$.

Lemma 5.7 (Extract). *If* $\Gamma_L, \widehat{\alpha}, \Gamma_R \rightsquigarrow \Omega$*, then there exists* τ *s.t.* $\Gamma_L \vdash \tau$ *and* $\Gamma_L, [\tau/\widehat{\alpha}]\Gamma_R \rightsquigarrow \Omega$.

$\boxed{\|\Omega\|}$ Judgment erasure.

$$
\begin{aligned}
\|\cdot\| &= \cdot \\
\|\Omega, a\| &= \|\Omega\|, a \\
\|\Omega, x : A\| &= \|\Omega\|, x : A \\
\|\Omega \Vdash \omega\| &= \|\Omega\|
\end{aligned}
$$

$\boxed{\Omega \longrightarrow \Omega'}$ Declarative transfer.

$$
\begin{aligned}
\Omega, a &\longrightarrow \Omega \\
\Omega, x : A &\longrightarrow \Omega \\
\Omega \Vdash A \leq B &\longrightarrow \Omega && \text{when } \|\Omega\| \vdash A \leq B \\
\Omega \Vdash e \Leftarrow A &\longrightarrow \Omega && \text{when } \|\Omega\| \vdash e \Leftarrow A \\
\Omega \Vdash e \Rightarrow_a \omega &\longrightarrow \Omega \Vdash [A/a]\omega && \text{when } \|\Omega\| \vdash e \Rightarrow A \\
\Omega \Vdash A \bullet e \Rrightarrow_a \omega &\longrightarrow \Omega \Vdash [C/a]\omega && \text{when } \|\Omega\| \vdash A \bullet e \Rrightarrow C
\end{aligned}
$$

Figure 5.8: Declarative Transfer

Figure 5.8 defines a relation $\Omega \longrightarrow \Omega'$, checking that every judgment entry in the worklist holds using a corresponding declarative judgment.

5.4.3 Soundness

Our algorithm is sound with respect to the declarative system. For any worklist Γ that reduces successfully, there is a valid instantiation Ω that transfers all judgments to the declarative system.

Theorem 5.8 (Soundness). *If* wf Γ *and* $\Gamma \longrightarrow^* \cdot$*, then there exists* Ω *s.t.* $\Gamma \rightsquigarrow \Omega$ *and* $\Omega \longrightarrow^* \cdot$.

Soundness is a basic desired property of a type inference algorithm, which ensures that the algorithm is always producing valid declarative derivations when the judgments are accepted.

5.4.4 Partial Completeness of Subtyping: Rank-1 Restriction

The algorithm is incomplete due to the subtyping rules 14, 15, 20 and 21. However, subtyping is complete with respect to the declarative system in a rank-1 setting.

Declarative Rank-1 Restriction Rank-1 types are also named type schemes in Hindley-Milner type system.

$$\text{Declarative Type Schemes} \qquad \sigma \quad ::= \quad \forall a.\, \sigma \mid \tau$$

In other words, the universal quantifiers only appear in the top level of all polymorphic types.

For declarative subtyping, a judgment must be of form $\sigma_1 \leq \sigma_2$.

5.4.5 Algorithmic Rank-1 Restriction (Partial Completeness)

The algorithmic mono-types and type schemes are defined as following:

$$\begin{aligned} \text{Algorithmic Mono-types} \qquad & \tau_A \quad ::= \quad 1 \mid \top \mid \bot \mid a \mid A \to B \mid \widehat{\alpha} \\ \text{Algorithmic Type Schemes} \qquad & \sigma_A \quad ::= \quad \forall a.\, \sigma_A \mid \tau_A \end{aligned}$$

Starting from the declarative judgment $\sigma_1 \leq \sigma_2$, the algorithmic derivation might involve different other kinds of judgments. The following derivation, as an example, shows how a rank-1 judgment derives.

$$\begin{aligned} & \cdot \Vdash \forall a.\, a \to a \leq \forall b.\, (b \to b) \to (b \to b) \\ \longrightarrow_8 \; & b \Vdash \forall a.\, a \to a \leq (b \to b) \to (b \to b) \\ \longrightarrow_7 \; & b, \widehat{\alpha} \Vdash \widehat{\alpha} \to \widehat{\alpha} \leq (b \to b) \to (b \to b) \\ \longrightarrow_6 \; & b, \widehat{\alpha} \Vdash \widehat{\alpha} \leq b \to b \Vdash b \to b \leq \widehat{\alpha} \\ \longrightarrow \; & \cdots \end{aligned}$$

In this derivation, we begin from a judgment of the form $\sigma \leq \sigma$. After rule 8 is applied, the judgment becomes $\sigma \leq \tau$, since the right-hand-side polymorphic type is reduced to a declarative mono-type. Then, rule 7 introduces existential variables to the left-hand-side, resulting in a judgment like $\tau_A \leq \tau$, or $\sigma_A \leq \tau$ in a more general case. Finally, rule 6 breaks a judgment between functions into two sub-judgments, which swaps the positions of the argument types and creates a judgment like $\tau \leq \tau_A$. Notice that σ_A is not possible to occur to the right because the function type may not contain any polymorphic types as its argument type.

After a detailed analysis on the judgments derivations, we found that the only possible judgments that a rank-1 declarative subtyping judgment might step to belong to the following two categories:

$$\sigma_A \leq \sigma \quad \text{or} \quad \tau \leq \sigma_A$$

All the possible judgment types shown above fall into these categories. For example, $\tau_A \leq \tau$ is a special form of $\sigma_A \leq \sigma$, and $\tau \leq \tau_A$ belongs to $\tau \leq \sigma_A$.

An interesting observation is that $\widehat{\alpha} \leq \widehat{\beta}$ does not belong to either category, neither does $\widehat{\alpha} \leq A \to B$ when $\widehat{\alpha} \in \mathrm{FV}(A \to B)$. Therefore, in the rank-1 setting, both cases of incompleteness never occur, and our algorithm is complete.

Theorem 5.9 (Completeness of Rank-1 Subtyping). *Given* $\Psi \vdash \sigma_1 \leq \sigma_2$,

- *If* $\Gamma \Vdash \sigma_A \leq \sigma \rightsquigarrow \Psi \Vdash \sigma_1 \leq \sigma_2$
  then $\Gamma \Vdash \sigma_A \leq \sigma \longrightarrow^* \cdot$;
- *If* $\Gamma \Vdash \tau \leq \sigma_A \rightsquigarrow \Psi \Vdash \sigma_1 \leq \sigma_2$
  then $\Gamma \Vdash \tau \leq \sigma_A \longrightarrow^* \cdot$.

5.4.6 Termination

The measure used in Chapter 4 no longer works because subtyping judgments like

$$\widehat{\alpha} \leq \bot \to \top$$

cause $\widehat{\alpha}$ to split into $\widehat{\alpha}_1 \to \widehat{\alpha}_2$, without solving any part of it, resulting in an increased number of existential variables and possibly increased complexity of the worklist through the size-increasing substitution $\{\widehat{\alpha} := \widehat{\alpha}_1 \to \widehat{\alpha}_2\}$.

We have performed a large set of tests on generated subtyping judgments that are consist of algorithmic monotypes, and all judgments terminated within a reasonable number of derivation depth. Unfortunately, we have not yet find any formal proof for the termination statement.

5.4.7 Formalization in the Abella Proof Assistant

We have chosen the Abella (v2.0.7-dev [1]) proof assistant [Gacek 2008] to develop our formalization. Equipped with HOAS, Abella eases the formalization and proof tasks a lot compared with various libraries in Coq. Additionally, our algorithm heavily uses eager substitutions, and Abella greatly simplifies relevant proofs thanks to its built-in substitution representation and higher-order unification algorithms.

Statistics of the Proof The proof script consists of 7,301 lines of Abella code with a total of 48 definitions and 592 theorems. Figure 5.1 briefly summarizes the contents of each file. The files are linearly dependent due to the limitations of Abella.

[1] We use a forked version `https://github.com/JimmyZJX/abella` by only enhancing the Abella prover with a handy "applys" tactic.

Table 5.1: Statistics for the proof scripts

File(s)	LOC	#Thm	Description
olist.thm, nat.thm	311	57	Basic data structures
typing.thm	273	7	Declarative & algorithmic system, debug examples
decl.thm	241	33	Basic declarative properties
order.thm	274	27	The $\lvert \cdot \rvert_\forall$ measure; decl. subtyping strengthening
alg.thm	699	82	Basic algorithmic properties
trans.thm	635	53	Worklist instantiation and declarative transfer; Lemmas 4.3, 4.4
declTyping.thm	1,087	76	Non-overlapping declarative system; Lemmas 4.5, 4.6, 4.7, 4.8
soundness.thm	1,206	81	Soundness theorem; aux. lemmas on transfer
dcl.thm	417	12	Non-overlapping declarative worklist
scheme.thm	1,113	98	Type scheme (rank-1 restriction)
completeness.thm	1,045	63	Completeness theorem; aux. lemmas and relations
Total	7,301	592	(48 definitions in total)

5.5 Discussion

5.5.1 A Complete Algorithm Under Monotype Guessing Restrictions

The incompleteness of the algorithm is mainly due to incomplete guesses for existential variable instantiations when dealing with subtyping. For example, when reducing the algorithmic subtyping judgment

$$\widehat{\alpha} \leq \widehat{\alpha} \to 1$$

we only consider the possibility $\widehat{\alpha} := \bot$ and stop reducing because the judgment involves a self-reference. In fact, there are infinitely many valid solutions just in our type system, including $\widehat{\alpha} := \top \to \bot$ and $\widehat{\alpha} := \top \to 1$. The problem can be solved by introducing recursive types, so that recursive constraints can be solved. However, the other type of incompleteness that comes from

$$\widehat{\alpha} \leq \widehat{\beta}$$

is still not addressed.

We propose an alternative approach that requires little change to the type system by restricting the declarative system. If the declarative system did not treat the bottom and top

types as monotypes in the first place, the algorithm would not need to guess those types. Formally, if monotypes are defined as follows,

$$\text{Monotypes'} \qquad \tau' \quad ::= \quad 1 \mid a \mid \tau_1 \to \tau_2$$

then subtyping relation between monotypes is simply equality.

The incomplete examples mentioned above can be addressed now, judgments like $\widehat{\alpha} \leq \widehat{\alpha} \to 1$ can no longer instantiate $\widehat{\alpha}$ to a function type, otherwise it will infinitely loop without emitting any valid solution; judgments like $\widehat{\alpha} \leq \widehat{\beta}$ can now be safely treated as $\widehat{\alpha} = \widehat{\beta}$, because the instantiations of $\widehat{\alpha}$ and $\widehat{\beta}$ must be equal to each other. Formally speaking, we proved the following lemma

Lemma 5.10 (Subtyping between Monotypes is Equality)**.**
If $\Psi \vdash \tau_1' \leq \tau_2'$*, then* $\tau_1' = \tau_2'$*.*

which also holds for the system in Chapter 4, yet it does not hold for the system with subtyping when monotypes are not restricted.

On the meantime, algorithmic subtyping rules 12 and 13, and typing rules 27 and 37 should be removed from the algorithm under the monotype restriction, since these rules solve existential variables to $\top$ or $\bot$. Other than that, the algorithm remains unchanged. Note that after removing these rules, the algorithm no longer has overlapping rules, therefore backtracking is not needed any more.

Following the proof techniques in Chapter 4, both *soundness* and *completeness* theorems are successfully proven and also mechanized. Given that the algorithm reduces judgments in a similar way to the one in the previous chapter, the result is not so surprising. However, it still reveals two facts: firstly, if we do not try to guess any type involving subtyping relation (the top and bottom types in our case), the algorithm should remain simple and complete; secondly, although the current algorithm for the system with subtyping is incomplete, all source of incompleteness comes from subtyping relations.

5.5.2 Lazy Substitution and Non-terminating Loops

In this subsection, we propose a possible way to fix the incompleteness issue for the subtyping part of the algorithm and discuss its impact to the existing formalization.

As we have analysed before, one major source of incompleteness comes from rules 20 and 21, where two different existential variables are compared. A natural idea is to delay the solving procedure until all the corresponding constraints are collected. Constraint collection can be implemented by the following new worklist definition, where the existential variables

also come with a set of upper bounds and a set of lower bounds. Both of the bounds consist of algorithmic mono-types.

$$\text{Algorithmic worklist} \qquad \Gamma \quad ::= \quad \cdot \mid \Gamma, a \mid \Gamma, \overline{\tau_A} \leq \widehat{\alpha} \leq \overline{\tau_A} \mid \Gamma \Vdash \omega$$

Differences of Algorithmic Rules. In presence of the bound collections, the algorithm behaves slightly differently on variable solving rules. Firstly, rules that eagerly solve existential variables to the top or bottom type are discarded (Rules 12 and 13). Next, Rules 20 and 21 add the corresponding existential variable as part of a bound rather than performing a global substitution.

$$\Gamma[\widehat{\alpha}][L \leq \widehat{\beta} \leq U] \Vdash \widehat{\alpha} \leq \widehat{\beta} \longrightarrow_{20'} \Gamma[\widehat{\alpha}][L \cup \{\widehat{\alpha}\} \leq \widehat{\beta} \leq U]$$
$$\Gamma[\widehat{\alpha}][L \leq \widehat{\beta} \leq U] \Vdash \widehat{\beta} \leq \widehat{\alpha} \longrightarrow_{21'} \Gamma[\widehat{\alpha}][L \leq \widehat{\beta} \leq U \cup \{\widehat{\alpha}\}]$$

Instantiation rules (Rules 14 and 15) also behave lazily in order not to lose completeness.

$$\Gamma[L \leq \widehat{\alpha} \leq U] \Vdash \widehat{\alpha} \leq A \to B \longrightarrow_{14'} \Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2, L \leq \widehat{\alpha} \leq U \cup \{\widehat{\alpha}_1 \to \widehat{\alpha}_2\}] \Vdash \widehat{\alpha}_1 \to \widehat{\alpha}_2 \leq A \to B$$
$$\Gamma[L \leq \widehat{\alpha} \leq U] \Vdash A \to B \leq \widehat{\alpha} \longrightarrow_{15'} \Gamma[\widehat{\alpha}_1, \widehat{\alpha}_2, L \cup \{\widehat{\alpha}_1 \to \widehat{\alpha}_2\} \leq \widehat{\alpha} \leq U] \Vdash A \to B \leq \widehat{\alpha}_1 \to \widehat{\alpha}_2$$

These new rules differ from the original ones: the two fresh existential variables representing a function type $\widehat{\alpha}_1 \to \widehat{\alpha}_2$ now acts as a bridge between $\widehat{\alpha}$ and $A \to B$. The algorithm breaks the judgment $\widehat{\alpha} \leq \widehat{\alpha}_1 \to \widehat{\alpha}_2$ into two sub-problems: $\widehat{\alpha} \leq \widehat{\alpha}_1 \to \widehat{\alpha}_2$ and $\widehat{\alpha}_1 \to \widehat{\alpha}_2 \leq A \to B$. And it is not difficult to prove that the problems before and after such transformation are equivalent to each other.

The change of existential variable declaration and solving mechanisms also result in the change of the algorithmic rule for garbage collecting them. When $\widehat{\alpha}$ is at the top of the worklist and thus going to be recycled, we need to further check if it can actually be solved, by ensuring that lower bounds are indeed subtypes of upper bounds: any type that is a subtype of $\widehat{\alpha}$ should be a subtype of any super type of $\widehat{\alpha}$.

$$\Gamma, \{l_i\}^n \leq \widehat{\alpha} \leq \{u_j\}^m \longrightarrow_{2'} \Gamma \Vdash_{1 \leq i \leq n, 1 \leq j \leq m} l_i \leq u_j$$

The notion $\{l_i\}^n$ indicates that the bound collection is of length n; the notion $\Vdash_{1\leq i\leq n, 1\leq j\leq m}$ $l_i \leq u_j$ is similar to the syntax of list-comprehension, which creates a list of judgments

$$\Vdash l_1 \leq u_1 \cdots \Vdash l_1 \leq u_m \cdots \Vdash l_2 \leq u_1 \cdots \Vdash l_n \leq u_m$$

According to the transitivity of subtyping, this rule is sound: any valid instantiation of $\widehat{\alpha}$, denoted $\tau_{\widehat{\alpha}}$, indicates that the instantiation of $_i \leq \tau_{\widehat{\alpha}}$ and $\tau_{\widehat{\alpha}} \leq$ holds, therefore the instantiation of $l_i \leq u_j$ holds for any $1 \leq i \leq n, 1 \leq j \leq m$.

On the other hand, such check is sufficient. In other words, we can always find a proper solution for $\widehat{\alpha}$ if any lower bound is a subtype of any upper bound. This is supported by the following property on declarative monotypes:

Lemma 5.11 (Sufficient Bound Check for Monotypes).
Given collections of monotypes $\{l_i\}^n$ *and* $\{u_j\}^m$,
if $\Psi \vdash l_i \leq u_j$ *holds for any* $1 \leq i \leq n, 1 \leq j \leq m$,
then there exists τ *s.t.* $\Psi \vdash l_i \leq \tau$ *and* $\Psi \vdash \tau \leq u_i$ *for any* $1 \leq i \leq n, 1 \leq j \leq m$.

Impact of the New Algorithm. The new algorithm addresses the incompleteness issue throughly by re-designing in a lazy-substitution style. However, the algorithm might loop forever due to the new version of instantiation rules. For example, the following derivation results in infinite instantiations:

$$\begin{aligned}
&\widehat{\alpha}, \widehat{\beta} \Vdash \widehat{\alpha} \leq 1 \to \widehat{\beta} \Vdash \widehat{\beta} \leq 1 \to \widehat{\alpha} \\
\longrightarrow\ &\widehat{\alpha}, \widehat{\beta}_1, \widehat{\beta}_2, \{\} \leq \widehat{\beta} \leq \{\widehat{\beta}_1 \to \widehat{\beta}_2\} \Vdash \widehat{\alpha} \leq 1 \to \widehat{\beta} \Vdash \widehat{\beta}_1 \to \widehat{\beta}_2 \leq 1 \to \widehat{\alpha} \\
\longrightarrow\ &\cdots \\
\longrightarrow\ &\widehat{\alpha}, \{1\} \leq \widehat{\beta}_1 \leq \{\}, \{\} \leq \widehat{\beta}_2 \leq \{\widehat{\alpha}\}, \{\} \leq \widehat{\beta} \leq \{\widehat{\beta}_1 \to \widehat{\beta}_2\} \Vdash \widehat{\alpha} \leq 1 \to \widehat{\beta} \\
\longrightarrow\ &\cdots
\end{aligned}$$

A further analyse on the problem suggests that the *loop dependencies* are not detected and reduced smartly by the lazy algorithm: it insist too much on not missing any possible solutions. By loop dependency we refer to the following pattern $\widehat{\alpha} \leq A \to B$ where $\widehat{\alpha} \in A \to B$ In the above example, one can derive (by transitivity of subtyping) $\widehat{\alpha} \leq 1 \to \widehat{\beta} \leq 1 \to 1 \to \widehat{\alpha}$ which forms a (indirect) loop dependency on $\widehat{\alpha}$.

By performing experimental implementations and randomly generated tests, we observe that the algorithm accepts more valid relations when there are no loop dependencies, but may be trapped in infinite loops otherwise. We have also tried to formalize how a loop detection

procedure might be used to help prevent the non-termination problem, but most of them turn out to be rather complicated and hard to mechanically formalize.

Part III

Related Work

6 Related Work

Throughout the thesis, we have already discussed much of the closest related work. In this section we summarize the key differences and novelties, and discuss some other related work.

6.1 Higher-Ranked Polymorphic Type Inference Algorithms

6.1.1 Predicative Algorithms

Higher-ranked polymorphism is a convenient and practical feature of programming languages. Since full type-inference for System F is undecidable [Wells 1999], various decidable partial type-inference algorithms were developed. The declarative formulation of subtyping in Chapters 3 and 4 (and later extended in Chapter 5), originally proposed by Odersky and Läufer [1996], is *predicative*: $\forall$'s only instantiate to monotypes. The monotype restriction on instantiation is considered reasonable and practical for most programs, except for those that require sophisticated forms of higher-order polymorphism. In those cases, type annotations may guide the type system to accept lambda functions of higher-ranked types.

In addition to OL's type system, the bidirectional system proposed by DK [Dunfield and Krishnaswami 2013] accepts even better type annotations, which also allow polymorphic types. Such annotations also improve readability of the program, and are not much of a burden in practice. DK's algorithm is shown to be sound, complete and decidable in 70 pages of manual proofs. Though carefully written, some of the proofs are incorrect (see discussion in Section 4.1.2 and 4.3.2), which creates difficulties when formalizing them in a proof assistant. In their follow-up work Dunfield and Krishnaswami [2019] enrich the bidirectional higher-rank system with existentials and indexed types. With a more complex declarative system, they developed a proof of over 150 pages. It is even more difficult to argue its correctness for every single detail within such a big development. Unfortunately, we find that their Lemma 26 (Parallel Admissibility) appears to have the same issue as lemma 29 in [Dunfield and Krishnaswami 2013]: the conclusion is false. We also discuss the issue in more detail in Section 4.1.2.

Peyton Jones et al. [2007] developed another higher-rank predicative bidirectional type system. Their subtyping relation is enriched with *deep skolemisation*, which is more general than ours and allows more valid relations. For example,

$$\forall a.\, \forall b.\, a \to b \to b \leq \forall a.\, a \to (\forall b.\, b \to b)$$

this subtyping relation does not hold in OL's subtyping relation, because the instantiation of b in the left-hand-side type happens strictly before the introduction of the variable b in the right-hand-side type. Deep skolemisation can be achieved through a pre-processing that extracts all the $\forall$'s in the return position of a function type to the top level. After such pre-processing, the right-hand-side type becomes $\forall a.\, \forall b.\, a \to b \to b$ and the subtyping relation holds.

In terms of handling applications where type parameters are implicit, they do not use the application inference judgment as DK's type system. Instead, they employ a complicated mechanism for implicit instantiation taken care by the unification process for the algorithm. A manual proof is given, showing that the algorithm is sound and complete with respect to their declarative specification.

In a more recent work, Xie and Oliveira [2018] proposed a variant of a bidirectional type inference system for a predicative system with higher-ranked types. Type information flows from arguments to functions with an additional *application* mode. This variant allows more higher-order typed programs to be inferred without additional annotations. Following the new mode, the let-generalization of the Hindley-Milner system is well supported as syntactic sugar. The formalization includes some mechanized proofs for the declarative type system, but all proofs regarding the algorithmic type system are manual.

6.1.2 Impredicative Algorithms

Impredicative System F allows instantiation with polymorphic types, but unfortunately its subtyping system is already undecidable [Tiuryn and Urzyczyn 1996]. Works on partial impredicative type-inference algorithms navigate a variety of design tradeoffs for a decidable algorithm. Generally speaking, such algorithms tend to be more complicated, and thus less adopted in real-world programming languages.

ML$^{\mathrm{F}}$ [Le Botlan and Rémy 2003, 2009; Rémy and Yakobowski 2008] extends types of System F with a form of bounded quantification and proposes an impredicative system. The type inference algorithm always infers principle types given proper type annotations. Through the technique called "monomorphic abstraction of polymorphic types", polymorphic instantiations are expressed by constraints on type variables of type schemes. An annotation is only needed when an argument of a lambda function is used polymorphically. Moreover,

their type system is robust against a wide range of program transformations, including let-expansion, let-reduction and η-expansion. However, the extended type structure of ML^F introduces non-compatible types with System F, and it complicates the metatheory and implementation of the type system.

HMF [Leijen 2008] takes a slightly different approach by not extending types with bounds, therefore programmers still work with plain System F types. The algorithm works similarly compared to ML^F, except that it does not output richer types. Instead, when there are ambiguity and no principal type can be inferred, being a conservative algorithm, HMF prefers predicative instantiations to maintain backward the compatibility of HM. Soon after HMF, HML [Leijen 2009] is proposed as an extension of HMF with flexible types. The system eliminates rigid quantifications ($\alpha = \sigma$) of ML^F and only keeps the flexible quantifications ($\alpha \geq \sigma$). Furthermore, a variant "Rigid HML" is presented by restricting let-bindings to System F types only, so that type annotations can still stay inside System F. Similar to HMF, they both require type annotations at higher-ranked types or ambiguous implicit instantiations.

FPH [Vytiniotis et al. 2008] employ a box notion $\boxed{\sigma}$ to encapsulate polymorphic instantiations internally in the algorithm. A type inside a box indicates that the instantiation is impredicative. Therefore an inferred type with a box type in it means that incomparable System F types may be produced, and that is rejected before entering the environment. Although more annotations might be required compared with other approaches, the algorithm of FPH is much simpler, thanks to the simple syntax and subtyping relation of box types.

Guarded Impredicative Polymorphism [Serrano et al. 2018] was recently proposed as an improvement on GHC's type inference algorithm with impredicative instantiation. They make use of local information in n-ary applications to infer polymorphic instantiations with a relatively simple specification and unification algorithm. Although not all impredicative instantiations can be handled well, their algorithm is already quite useful in practice.

A recent follow-up work, the Quick Look [Serrano et al. 2020], takes a more conservative approach. The algorithm will try its best to infer impredicative instantiations, and only use the instantiation when it is the best one. In order to do so, the algorithm also needs to analyse all the argument types during a function call, and make use of the *guardedness* property to decide the principality of the inferred instantiation. When Quick Look cannot give the best instantiation, it will instead try predicative inference as HM. This conservative approach makes sure that the algorithm does not infer bad types and leads the subsequent type checking in wrong directions. In the meantime, they treat the function arrow ($\rightarrow$) as invariant like normal type constructors. This change significantly restricts the subtyping relation and thus simplifies guesses for implicit parameters. Manual η-expansions are required to regain the co- and contravariance of function types.

FreezeML [Emrich et al. 2020] is another recent work that extends the ML type system with impredicative instantiations. A special syntax of expression, the explicit freezing $\lceil x \rceil$, is added to guide the type system. Freezed variables are prevented to be instantiated by the type system, therefore variables of polymorphic types may force the type inference algorithm to instantiate impredicatively. In combination with the let-generalization rule, programmers may also encode explicit generalization and explicit instantiation. This work provides a nice means for programmers to control how type inference algorithm deals with impredicative instantiations.

6.2 Type Inference Algorithms with Subtyping

Type systems in presence of subtyping usually encounter constraints that are not simply equalities as in HM. Therefore constraint solvers used in HM, where unifications are based on equality, cannot be easily extended to support subtyping. Instead, constraints are usually collected as subtyping relations and may delay resolving as the constraints accumulate. Eifrig et al. [1995a,c] proposed systems that are based on *constraint types*, i.e. types expressed together with a set of constraints $\tau \mid \{\overline{\tau_1 \leq \tau_2}\}$. Their type checking algorithm checks at each step whether each constraint in the closure of constraint set is *consistent*, which is a set of rules that prevent obvious contradictions appear, such as Int $\leq$ Bool. Our attempt in Section 5.5.2 is similar to this idea, where we exhaustively check if every subtyping relation derived from the bounds is valid. However, our algorithm tries to solve the bounds into concrete types, and it turns out that it never terminates in some complex cases.

Constraint types improve the expressiveness of the type system, yet the type inference algorithm can be quite slow. The size of the constraint is linear in the program size, and the closure can grow to cubic size. Pottier [1998] proposed three methods to simplify constraints in his Ph.D. thesis, aiming at improving the efficiency of type inference algorithms and improving the readability of the resulting types. By *canonization*, constraints are converted to canonical forms with the introduction of new meta-variables; *garbage collection* eliminates constraints that do not affect the type; finally, *minimization* shares nodes within the constraint graph.

Inspired by the simplification strategies of Pottier's, MLsub [Dolan and Mycroft 2017] suggest that the data flow on the constraint graph can reflect directly on types extended by a richer type system, where lattices operations are used to represent constraints imposed on type variables. Polar types distinguish between input types and output types, and pose different restrictions on them. As a result, constraints can easily be transformed into canonical forms, and the bi-unification algorithm can solve them by simple substitutions. One can also view the MLsub system as a different way to encode the constraint types: instead of a set of

constraints that is stated along with the type, types themselves now contain subtyping constraints with the help of lattice operations. Similarly, the size of constraints may grow with the size of program and affect the readability. Therefore, various simplification algorithms should be used in real applications. A more recent work, the Simple-sub [Parreaux 2020], further simplifies the algorithm of MLsub and is implemented in 500 lines of code. While being equivalent to MLsub, it is a more efficient variant.

6.3 Techniques Used in Type Inference Algorithms

6.3.1 Ordered Contexts in Type Inference

Gundry et al. [2010] revisit algorithm $\mathcal{W}$ and propose a new unification algorithm with the help of ordered contexts. Similar to DK's algorithm, information of meta-variables flows from input contexts to output contexts. Not surprisingly, its information increase relation has a similar role to DK's context extension. Our algorithm, in contrast, eliminates output contexts and solution records ($\widehat{\alpha} = \tau$), simplifying the information propagation process through immediate substitution by collecting all the judgments in a single worklist.

6.3.2 The Essence of ML Type Inference

Constraint-based type inference is adopted by Pottier and Rémy [2005] for ML type systems, which do not employ higher-ranked polymorphism. An interesting feature of their algorithm is that it keeps precise scoping of variables, similarly to our approach. Their algorithm is divided into constraint generation and solving phases (which are typical of constraint-based algorithms). Furthermore an intermediate language is used to describe constraints and their constraint solver utilizes a stack to track the state of the solving process. In contrast, our algorithm has a single phase, where the judgment chains themselves act as constraints, thus no separate constraint language is needed.

6.3.3 Lists of Judgments in Unification

Some work [Abel and Pientka 2011; Reed 2009] adopts a similar idea to this paper in work on unification for dependently typed languages. Similarly to our work the algorithms need to be very careful about scoping, since the order of variable declarations is fundamental in a dependently typed setting. Their algorithms simplify a collection of unification constraints progressively in a single-step style. In comparison, our algorithm mixes variable declarations with judgments, resulting in a simpler judgment form, while processing them in a similar way. One important difference is that contexts are duplicated in their unification judgments,

which complicates the unification process, since the information of each local context needs to be synchronized. Instead we make use of the nature of ordered context to control the scope of unification variables. While their algorithms focus only on unification, our algorithm also deals with other types of judgments like synthesis. A detailed discussion is given in Section 4.1.3.

6.4 Mechanical Formalization of Polymorphic Type Systems

Since the publication of the POPLMark challenge [Aydemir et al. 2005], many theorem provers and packages provide new methods for dealing with variable binding [Aydemir et al. 2008; Chlipala 2008; Urban 2008]. More and more type systems are formalized with these tools. However, mechanizing certain algorithmic aspects, like unification and constraint solving, has received very little attention and is still challenging. Moreover, while most tools support local (input) contexts in a neat way, many practical type-inference algorithms require more complex binding structures with output contexts or various forms of constraint solving procedures.

Algorithm $\mathcal{W}$, as one of the classic type inference algorithms for polymorphic type systems, has been manually proven to be sound and complete with respect to the Hindley-Milner type system [Damas and Milner 1982; Hindley 1969; Milner 1978]. After around 15 years, the algorithm was formally verified by Naraschewski and Nipkow [1999] in Isabelle/HOL [Nipkow et al. 2002]. The treatment of new variables was tricky at that time, while the overall structure follows the structure of Damas's manual proof closely. Later on, other researchers [Dubois 2000; Dubois and Menissier-Morain 1999] formalized algorithm $\mathcal{W}$ in Coq [The Coq development team 2017]. Nominal techniques [Urban 2008] in Isabelle/HOL have been developed to help programming language formalizations, and are used for a similar verification [Urban and Nipkow 2008]. Moreover, Garrigue [Garrigue 2015] mechanized a type inference algorithm, with the help of locally nameless [Charguéraud 2012], for Core ML extended with structural polymorphism and recursion.

Part IV

Epilogue

7 Conclusion and Future Work

7.1 Conclusion

In this thesis, we proposed new bidirectional type inference algorithms for predicative higher-ranked implicit parametric polymorphic systems. We showed how worklists ease the design of algorithms and also mechanical formalizations. By collecting all judgments in a single worklist, the algorithm performs unification with a bird's-eye view of all the judgments, therefore propagation between judgments is as simple as a global substitution. Compared with classical HM unification procedure, Our algorithm does not need separated relations (or fixpoints) for unification. From the formalization point of view, eager substitutions are also easier to state and reason in a proof assistant. Therefore, we obtained fully formalized properties for all our developments relatively easily. Overall, we developed the following systems and/or type inference algorithms:

- We developed a worklist algorithm for OL's higher-ranked subtyping system. The algorithm operates on a worklist of subtyping judgments and a single context where a variable declaration is shared across the worklist. Compared with DK's algorithm, our approach avoids the use of output contexts, which complicates the scoping of variables and is hard to formalize in a proof assistant. Eager substitutions that solve existential variables are directly applied to the worklist, therefore passing the partial information to the rest judgments. We proved *soundness*, *completeness* and *decidability* of the algorithm in the Abella theorem prover.

- We developed a worklist algorithm for DK's higher-ranked bidirectional type system. In order to properly encode judgments that output types, such as the type inference judgment, continuation-passing-style *judgment chains* are developed. Compared with the previous work that uses a single context, we further unify the worklist with variable declarations. Such unification results in a much more accurate track of variable scopings, and the algorithm will garbage-collect variables as soon as they are not referred to anymore. Unlike using output contexts as DK's algorithm, designing rules for worklist context is less likely to contain bugs in terms of variable scoping. Once

again, based on eager substitutions, the algorithm is easy to formalize. We showed *soundness*, *completeness* and *decidability* in the Abella theorem prover.

- We developed a backtracking-based algorithm for a higher-ranked bidirectional type system with object-oriented subtyping. With the introduction of the top and bottom types and relevant subtyping relations, meta-variable instantiations are no longer deterministic like the HM system. The backtracking-based algorithm preserves most characteristics of the previous worklist context, and it "tries" obvious solutions in parallel with detailed analysis as previous work. We proved that the algorithm is always *sound*, and subtyping is complete under the rank-1 restriction.

7.2 Future Work

In this section, we discuss several interesting possibilities to explore in the future.

Termination Theorem for Type Inference with Subtyping The algorithm we presented and formalized in Chapter 5 enjoys several important machine-verified properties, including soundness and completeness under a rank-1 restriction. However, we have not yet found a proper termination measure for the algorithmic subtyping rules. What complicates the problem is that common measures are not decreasing all the time. For example, subtyping between function types may increase the number of judgments; and instantiation judgments that split existential variables may increase the number of unsolved existential variables. We are mostly convinced by the fact that a large set of tests performed on randomly generated subtyping judgments do terminate. And we hope that a clever termination argument may be proposed in the near future.

Practical Impredicative Type Inference Impredicative type inference algorithms are sometimes helpful when dealing with higher-ranked types. Simple forms of impredicativity can be unambiguous and easy to understand by programmers, such as

```
head ids
```

where `head` :: $\forall p.\ [p] \to p$ and `ids` :: $[\forall a.\ a \to a]$. The application (`head ids`) should have type $\forall a.\ a \to a$, because p is instantiated to $\forall a.\ a \to a$ and that is the only choice we can make.

Moreover, even though there are programs that have incomparable System F types, impredicative system may offer means like type annotation to manually pick the right one. For example,

```
single id
```

where `single` :: $\forall a.\ p \to [p]$ and `id` :: $\forall a.\ a \to a$. This term has two incomparable types $[\forall a.\ a \to a]$ and $\forall a.\ [a \to a]$. One might expect that a type annotation might help pick the right case, like (`(single id)` :: $[\forall a.\ a \to a]$), which requires a system that allows impredicative instantiation to do so. There are many impredicative systems developed out of various techniques Emrich et al. [2020]; Le Botlan and Rémy [2003]; Leijen [2008, 2009]; Serrano et al. [2020, 2018]; Vytiniotis et al. [2008], and we leave the extension of impredicativity to our algorithms as future work.

Optimization All the three type inference algorithms described in this thesis are based on eager global substitution on the worklists. The benefit is to reduce the difficulty for mechanical formalizations in theorem provers, especially in Abella. Additionally, eager substitutions represent simple equivalent transformations on the state of worklist, alleviating the complication of variable scoping and reasoning for correctness. However, a naive implementation of our algorithm will be very inefficient, since the worklist is iterated so often whenever an existential variable is solved or partially solved.

Compared with mature algorithms like the algorithm $\mathcal{W}$ [Milner 1978] and the OutsideIn(X) [Vytiniotis et al. 2011] type inference algorithm, which produces substitutions once all the constraints are collected and reduced, our algorithms manipulate the judgments themselves on-the-fly. Efficient implementations will model meta-variables as mutable references and apply the results if they are solved by the constraint solver. DK's algorithm [Dunfield and Krishnaswami 2013] requires output contexts, which are likely to be implemented in a state monad or other stateful approaches. Solved existential variable is encoded in their algorithmic context different from ours. Instead of removing the declaration of the variable and propagate its solution to all the rest judgments, they extend the entry in the context with its solution, $\widehat{\alpha} = \tau$, and pass it to the output context. Such entry is easy to implement with an efficient approach using mutable references. We can also adopt such an idea and reformulate our eager substitution with lazy ones by keeping track of solutions to existential variables, and only perform substitution when the judgment is just about to be reduced.

Unfortunately, lazy substitution only solves part of the problem. By using ordered contexts, the scoping of variables is precisely captured by the relative positions in which they are declared in the context. Some algorithmic rules are designed to use the position informa-

tion, where swapping positions might lead to unsound implementation. For example, the algorithmic system of Chapter 5 include these rules:

$$\Gamma[a][\widehat{\beta}] \Vdash a \leq \widehat{\beta} \longrightarrow_{16} \{\widehat{\beta} := a\}\, \Gamma[a][\widehat{\beta}]$$
$$\Gamma[a][\widehat{\beta}] \Vdash \widehat{\beta} \leq a \longrightarrow_{17} \{\widehat{\beta} := a\}\, \Gamma[a][\widehat{\beta}]$$

The hole notation $\Gamma[a][\widehat{\beta}]$ is used to ensure that a is declared before $\widehat{\beta}$, therefore it is fine for the monotype represented by $\widehat{\beta}$ to be a. In a naive implementation, frequent iteration is required to look up the relative positions of variable declarations.

Moreover, there are also reduction rules that *remove* and/or *insert* variables from(to) the middle of the ordered context, as occurred in the following rules:

$$\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \leq A \to B \longrightarrow_{14} \{\widehat{\alpha} := \widehat{\alpha}_1 \to \widehat{\alpha}_2\}\, (\Gamma[\widehat{\alpha}] \Vdash \widehat{\alpha} \leq A \to B) \quad \text{when } \widehat{\alpha} \notin FV(A \to B)$$
$$\Gamma[\widehat{\alpha}] \Vdash A \to B \leq \widehat{\alpha} \longrightarrow_{15} \{\widehat{\alpha} := \widehat{\alpha}_1 \to \widehat{\alpha}_2\}\, (\Gamma[\widehat{\alpha}] \Vdash A \to B \leq \widehat{\alpha}) \quad \text{when } \widehat{\alpha} \notin FV(A \to B)$$

To sum up, we plan to propose an implementation-friendly version of the algorithm that does not require many eager substitutions, and we also hope that there will be suitable data structures or algorithms which optimizes the complexity of those time-consuming operations on the ordered context.

Class Hierarchy of Nominal Subtyping Nowadays, most mainstream object-oriented programming languages support nominal subtyping, which means that the subtyping relation is defined by the programmers explicitly through language constructs such as inheritance. For example, we may define a Pos class to represent positive integers, which inherits the Int class. The subtyping relation Pos $\leq$ Int automatically holds. An algorithmic subtyping judgment $\widehat{\alpha} \leq$ Int now might have several solutions: Int, Pos, or $\bot$. In order to calculate a *range* of solutions instead of just simple types, lazy algorithms that operate on bounds fit the task better. We plan to explore extensions of the algorithm described in Section 5.5.2. The algorithm progressively collects upper and lower bounds of existential variables, and finally checks if valid solutions are satisfying all the bounds.

Nonetheless, we encounter termination problems when trying to adopt the lazy approach. Experiments indicate that loop dependency in combination with the instantiation rule ($\widehat{\alpha} \leq A \to B$) might cause infinite loops. As a practical compromise, we may impose restrictions on existential variables so that they are not instantiated to function types when they are likely to be a class type. We also hope that better treatment on loop dependencies can be developed,

especially under assumptions of real-world programming tasks. For future work, we aim at a terminating and sound algorithm for practical object-oriented type inference.

Type Inference with Recursive Types Among all the algorithms we proposed in this thesis, we reject subtyping judgments like

$$\widehat{\alpha} \leq A \to B \text{ where } \widehat{\alpha} \in \text{FV}(A) \cup \text{FV}(B)$$

because no HM monotype satisfies a subtyping relation like

$$\tau \leq \tau \to \text{Int}$$

and unfolding such algorithmic judgment might cause an infinite loop. However, recursive types may satisfy such judgment, for example,

$$\mu x.\ x \to \text{Int} \leq (\mu x.\ x \to \text{Int}) \to Int$$

holds since the unfolding of $\mu x.\ x \to \text{Int}$ is $(\mu x.\ x \to \text{Int}) \to \text{Int}$. In other words, if we extend our type system by including recursive types, we would accept such judgment instead of rejecting it. The MLsub [Dolan and Mycroft 2017] already accepts recursive types, and it deals with recursive constraints in the way we described above. It is an interesting question if adding recursive types may improve the expressiveness of our system and remain complete at the same time.

Bounded Quantification and F-Bounded Quantification Bounded quantification [Cardelli and Wegner 1985] and F-bounded quantification [Canning et al. 1989] are techniques commonly seen in object-oriented programming languages. Bounded quantification allows constraints imposed on universally quantified variables:

$$\forall(a \leq \text{Int}).\ a \to a \to \text{String}$$

F-Bounded quantification further extends the constraints so that recursive references of the variables are also allowed. Although the subtyping is proven to be undecidable in presence of both F-bounded quantification and variance on generics, Kennedy and Pierce [2007] proposed three forms of restrictions that we can explore further. Combining bidirectional type checking with bounded quantifications may further improve the experience for object-oriented programmers.

Bibliography

[Citing pages are listed after each reference.]

Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2018. POPLMark Reloaded: Mechanizing Proofs by Logical Relations. *Submitted to the Journal of functional programming* (2018). [cited on page 9]

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 43, 49, and 104]

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. [cited on pages 9 and 105]

Brian E Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized metatheory for the masses: The POPLmark challenge. In *The 18th International Conference on Theorem Proving in Higher Order Logics*. [cited on pages 9 and 105]

Yves Bertot, Benjamin Grégoire, and Xavier Leroy. 2006. A Structured Approach to Proving Compiler Optimizations Based on Dataflow Analysis. In *Proceedings of the 2004 International Conference on Types for Proofs and Programs (TYPES'04)*. [cited on page 9]

Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. 1989. F-Bounded Polymorphism for Object-Oriented Programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (Imperial College, London, United Kingdom) *(FPCA '89)*. Association for Computing Machinery, New York, NY, USA, 273–280. https://doi.org/10.1145/99370.99392 [cited on page 111]

Luca Cardelli. 1988. A semantics of multiple inheritance. *Information and Computation* 76, 2 (1988), 138–164. https://doi.org/10.1016/0890-5401(88)90007-7 [cited on page 8]

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (Dec. 1985), 471–523. https://doi.org/10.1145/6041.6042 [cited on page 111]

Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula. 2006. A Framework for Certified Program Analysis and Its Applications to Mobile-code Safety. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*. [cited on page 9]

Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (01 Oct 2012), 363–408. [cited on page 105]

Paul Chiusano and Runar Bjarnason. 2015. Unison. http://unisonweb.org [cited on page 7]

Adam Chlipala. 2008. Parametric Higher-order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. [cited on page 105]

Alonzo Church. 1932. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics* 33, 2 (1932), 346–366. http://www.jstor.org/stable/1968337 [cited on page 4]

Luis Damas and Robin Milner. 1982. Principal Type-schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*. [cited on pages 5, 13, 30, and 105]

N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0 [cited on page 10]

Disciple Development Team. 2017. The Disciplined Disciple Compiler. http://disciple.ouroborus.net/ [cited on page 8]

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New

York, NY, USA, 60–72. https://doi.org/10.1145/3009837.3009882 [cited on pages 23, 24, 78, 103, and 111]

Catherine Dubois. 2000. Proving ML type soundness within Coq. *Theorem Proving in Higher Order Logics* (2000), 126–144. [cited on pages 9 and 105]

Catherine Dubois and Valerie Menissier-Morain. 1999. Certification of a type inference tool for ML: Damas–Milner within Coq. *Journal of Automated Reasoning* 23, 3 (1999), 319–346. [cited on pages 9 and 105]

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-rank Polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. [cited on pages 7, 9, 20, 28, 29, 32, 42, 43, 47, 64, 100, and 109]

Joshua Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and Complete Bidirectional Typechecking for Higher-rank Polymorphism with Existentials and Indexed Types. *Proc. ACM Program. Lang.* 3, POPL, Article 9 (Jan. 2019), 28 pages. [cited on pages 47, 48, and 100]

Jonathan Eifrig, Scott Smith, and Valery Trifonov. 1995a. Sound Polymorphic Type Inference for Objects. *SIGPLAN Not.* 30, 10 (Oct. 1995), 169–184. https://doi.org/10.1145/217839.217858 [cited on page 103]

Jonathan Eifrig, Scott Smith, and Valery Trifonov. 1995b. Type Inference for Recursively Constrained Types and its Application to OOP. *Electronic Notes in Theoretical Computer Science* 1 (1995), 132 – 153. https://doi.org/10.1016/S1571-0661(04)80008-2 MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference. [cited on page 78]

Jonathan Eifrig, Scott Smith, and Valery Trifonov. 1995c. Type Inference for Recursively Constrained Types and its Application to OOP. *Electronic Notes in Theoretical Computer Science* 1 (1995), 132–153. https://doi.org/10.1016/S1571-0661(04)80008-2 MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference. [cited on page 103]

Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. 2020. FreezeML: Complete and Easy Type Inference for First-Class Polymorphism. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New

York, NY, USA, 423–437. https://doi.org/10.1145/3385412.3386003 [cited on pages 6, 103, and 109]

Phil Freeman. 2017. PureScript. http://www.purescript.org/ [cited on pages 7 and 8]

Andrew Gacek. 2008. The Abella Interactive Theorem Prover (System Description). In *Proceedings of IJCAR 2008 (Lecture Notes in Artificial Intelligence)*. [cited on pages 10, 39, 42, 43, 60, 70, and 93]

Jacques Garrigue. 2015. A certified implementation of ML with structural polymorphism and recursive types. *Mathematical Structures in Computer Science* 25, 4 (2015), 867–891. [cited on pages 9 and 105]

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 (FPCA '93)*. [cited on page 6]

Adam Gundry, Conor McBride, and James McKinna. 2010. Type Inference in Context. In *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming (MSFP '10)*. [cited on pages 43 and 104]

Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society* 146 (1969), 29–60. [cited on pages 5, 13, and 105]

Mark P. Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming (Lecture Notes in Computer Science 925)*. [cited on page 6]

Andrew Kennedy and Benjamin C. Pierce. 2007. On Decidability of Nominal Subtyping with Variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)* (international workshop on foundations and developments of object-oriented languages (fool/wood) ed.). https://www.microsoft.com/en-us/research/publication/on-decidability-of-nominal-subtyping-with-variance/ [cited on page 111]

A. J. Kfoury and J. Tiuryn. 1992. Type Reconstruction in Finite Rank Fragments of the Second-Order λ-Calculus. *Inf. Comput.* 98, 2 (June 1992), 228–257. https://doi.org/10.1016/0890-5401(92)90020-G [cited on page 17]

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler.

2012. Run Your Research: On the Effectiveness of Lightweight Mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. 285–296. [cited on pages 9 and 43]

Ralf Lämmel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '03)*. [cited on page 6]

John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. Association for Computing Machinery, New York, NY, USA, 24–35. `https://doi.org/10.1145/178243.178246` [cited on page 6]

John Launchbury and Simon L. Peyton Jones. 1995. State in Haskell. *LISP and Symbolic Computation* 8, 4 (1995), 293–341. [cited on page 6]

Didier Le Botlan and Didier Rémy. 2003. MLF: Raising ML to the Power of System F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. [cited on pages 6, 7, 101, and 109]

Didier Le Botlan and Didier Rémy. 2009. Recasting MLF. *Information and Computation* 207, 6 (2009), 726–785. `https://doi.org/10.1016/j.ic.2008.12.006` [cited on page 101]

Daan Leijen. 2008. HMF: Simple Type Inference for First-class Polymorphism. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. [cited on pages 6, 7, 102, and 109]

Daan Leijen. 2009. Flexible Types: Robust Type Inference for First-Class Polymorphism. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 66–77. `https://doi.org/10.1145/1480881.1480891` [cited on pages 102 and 109]

Xavier Leroy et al. 2012. The CompCert verified compiler. *Documentation and user' s manual. INRIA Paris-Rocquencourt* (2012). [cited on page 9]

Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.* 4, 2 (April 1982), 258–282. [cited on page 49]

Dale Miller. 2000. Abstract Syntax for Variable Binders: An Overview. In *CL 2000: Computational Logic (Lecture Notes in Artificial Intelligence)*. [cited on pages 39 and 70]

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375. [cited on pages 5, 13, 16, 48, 105, and 109]

John C. Mitchell. 1984. Coercion and Type Inference. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Salt Lake City, Utah, USA) *(POPL '84)*. Association for Computing Machinery, New York, NY, USA, 175–185. `https://doi.org/10.1145/800017.800529` [cited on page 8]

Wolfgang Naraschewski and Tobias Nipkow. 1999. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning* 23, 3 (1999), 299–318. [cited on pages 9 and 105]

Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media. [cited on page 105]

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. [cited on pages 6, 7, 16, 18, 21, 28, 43, 81, and 100]

Lionel Parreaux. 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 124 (Aug. 2020), 28 pages. `https://doi.org/10.1145/3409006` [cited on page 104]

Simon Peyton Jones and Mark Shields. 2004. Lexically-scoped type variables. (2004). `http://research.microsoft.com/en-us/um/people/simonpj/papers/scoped-tyvars/` Draft. [cited on pages 70 and 75]

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of functional programming* 17, 1 (2007), 1–82. [cited on pages 7, 22, and 100]

Benjamin C Pierce. 2002. *Types and programming languages*. [cited on page 4]

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. `https://doi.org/10.1145/345099.345100` [cited on page 7]

Andrew M. Pitts. 2003. Nominal logic, a first order theory of names and binding. *Information and Computation* 186, 2 (2003), 165–193. https://doi.org/10.1016/S0890-5401(03)00138-X Theoretical Aspects of Computer Software (TACS 2001). [cited on page 10]

François Pottier. 1998. *Type inference in the presence of subtyping: from theory to practice*. Ph.D. Dissertation. INRIA. [cited on pages 78 and 103]

François Pottier and Didier Rémy. 2005. *Advanced Topics in Types and Programming Languages*. The MIT Press, Chapter The Essence of ML Type Inference, 387–489. [cited on pages 49 and 104]

Jason Reed. 2009. Higher-order Constraint Simplification in Dependent Type Theory. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP '09)*. [cited on pages 43, 49, and 104]

Didier Rémy and Boris Yakobowski. 2008. From ML to MLF: Graphic Type Constraints with Efficient Type Inference. *SIGPLAN Not.* 43, 9 (Sept. 2008), 63–74. https://doi.org/10.1145/1411203.1411216 [cited on page 101]

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Proceedings of the IFIP 9th World Computer Congress*. [cited on pages 4 and 74]

J C Reynolds. 1985. Three Approaches to Type Structure. In *Proc. of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT) Berlin, March 25-29, 1985 on Mathematical Foundations of Software Development, Vol. 1: Colloquium on Trees in Algebra and Programming (CAAP'85)* (Berlin, Germany). Springer-Verlag, Berlin, Heidelberg, 97–138. [cited on page 8]

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, 102, and 109]

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 (PLDI 2018)*. [cited on pages 6, 7, 102, and 109]

The Coq development team. 2017. The Coq proof assistant. https://coq.inria.fr/ [cited on page 105]

Jerzy Tiuryn and Pawel Urzyczyn. 1996. The subtyping problem for second-order types is undecidable. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. [cited on pages 5, 22, and 101]

Valery Trifonov and Scott Smith. 1996. Subtyping constrained types. In *Static Analysis*, Radhia Cousot and David A. Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 349–365. [cited on page 78]

Christian Urban. 2008. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning* 40, 4 (2008), 327–356. [cited on page 105]

Christian Urban and Tobias Nipkow. 2008. Nominal verification of algorithm W. *From Semantics to Computer Science. Essays in Honour of Gilles Kahn* (2008), 363–382. [cited on pages 9 and 105]

Dimitrios Vytiniotis, Simon Peyton jones, Tom Schrijvers, and Martin Sulzmann. 2011. Outsidein(x) Modular Type Inference with Local Assumptions. *J. Funct. Program.* 21, 4–5 (Sept. 2011), 333–412. `https://doi.org/10.1017/S0956796811000098` [cited on page 109]

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: First-class Polymorphism for Haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. [cited on pages 6, 7, 102, and 109]

Joe B Wells. 1999. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic* 98, 1-3 (1999), 111–156. [cited on pages 6, 15, 16, and 100]

Ningning Xie and Bruno C. d. S. Oliveira. 2018. Let Arguments Go First. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 272–299. [cited on page 101]

Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. Formalization of a Polymorphic Subtyping Algorithm. In *ITP (Lecture Notes in Computer Science, Vol. 10895)*. Springer, 604–622. [cited on page 12]

Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. A Mechanical Formalization of Higher-Ranked Polymorphic Type Inference. *Proc. ACM Program. Lang.* 3, ICFP, Article 112 (July 2019), 29 pages. `https://doi.org/10.1145/3341716` [cited on page 12]