# <Title>

*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
"**<Title>**"

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

“predicative implicit higher-rank polymorphism”

## 1.1 Contributions

In summary the contributions of this thesis are:

?? •

## 1.2 Organization

This thesis is largely based on the publications by the author [], as indicated below.

??:

---

# 2 Background

# 3 Higher-Rank Polymorphism Subtyping Algorithm

## 3.1 Introduction

Most statically typed functional languages support a form of *(implicit) parametric polymorphism* Reynolds [1983]. Traditionally, functional languages have employed variants of the Hindley-Milner Damas and Milner [1982]; Hindley [1969]; Milner [1978] type system, which supports full type-inference without any type annotations. However the Hindley-Milner type system only supports *first-order polymorphism*, where all universal quantifiers only occur at the top-level of a type. Modern functional programming languages such as Haskell go beyond Hindley-Milner and support *higher-order polymorphism*. With higher-order polymorphism there is no restriction on where universal quantifiers can occur. 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].

Unfortunately, with higher-order polymorphism full type-inference becomes undecidable Wells [1999]. To recover decidability some type annotations on polymorphic arguments are necessary. A canonical example that requires higher-order polymorphism in Haskell is:

```
hpoly = (\f :: forall a. a -> a) -> (f 1, f 'c')
```

The function `hpoly` cannot be type-checked in Hindley-Milner. The type of `hpoly` is `(forall a. a -> a) -> (Int, Cha` 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. Notably `hpoly` requires a type annotation for the first argument (`forall a. a -> a`). Despite these additional annotations, the type-inference algorithm employed by GHC Haskell Peyton Jones et al. [2007] preserves many of the desirable properties of Hindley-Milner. Like in Hindley-Milner type instantiation is *implicit*. That is, calling a polymorphic function never requires the programmer to provide the instantiations of the type parameters.

Central to type-inference with *higher-order polymorphism* is an algorithm for polymorphic subtyping. This algorithm allows us to check whether one type is more general than another, which is essential to detect valid instantiations of a polymorphic type. For example, the type

`forall a. a -> a` is more general than `Int -> Int`. A simple declarative specification for polymorphic subtyping was proposed by Odersky and Läufer Odersky and Läufer [1996]. Since then several algorithms have been proposed that implement it. Most notably, the algorithm proposed by Peyton Jones et al. Peyton Jones et al. [2007] forms the basis for the implementation of type inference in the GHC compiler. Dunfield and Krishnaswami Dunfield and Krishnaswami [2013] provided a very elegant formalization of another 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]).

Unfortunately, while many aspects of programming languages and type systems have been mechanically formalized in theorem provers, there is little work on formalizing algorithms related to type-inference. The main exceptions to the rule are mechanical formalizations of algorithm $\mathcal{W}$ and other aspects of traditional Hindler-Milner type-inference Dubois [2000]; Dubois and Menissier-Morain [1999]; Garrigue [2015]; Naraschewski and Nipkow [1999]; Urban and Nipkow [2008]. However, as far as we know, there is no mechanisation of algorithms used by modern functional languages like Haskell, and polymorphic subtyping included is no exception. This is a shame because recently there has been a lot of effort in promoting the use of theorem provers to check the meta-theory of programming languages, e.g., through well-known examples like the POPLMark challenge Aydemir et al. [2005] and the CompCert project Leroy et al. [2012]. Mechanical formalizations are especially valuable for proving the correctness of the semantics and type systems of programming languages. Type-inference algorithms are arguably among the most non-trivial aspects of the implementations of programming languages. In particular the information discovery process required by many algorithms (through unification-like or constraint-based approaches), is quite subtle and tricky to get right. 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 extended by the language designer.

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 judgements. 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.

This paper advances the state-of-the-art by formalizing an algorithm for polymorphic subtyping in the Abella theorem prover. We hope that this work encourages other researchers to use theorem provers for formalizing type-inference algorithms. In particular, we show that the problem we have identified above can be overcome by means of *worklist judgments*. These are a form of judgement that turns the complicated global propagation of unifications into a simple local

| | | | |
|---|---|---|---|
| Type variables | $a, b$ | | |
| Types | $A, B, C$ | $::=$ | $1 \mid a \mid \forall a.A \mid A \to B$ |
| Monotypes | $\tau$ | $::=$ | $1 \mid a \mid \tau_1 \to \tau_2$ |
| Contexts | $\Psi$ | $::=$ | $\cdot \mid \Psi, a$ |

Figure 3.1: Syntax of Declarative System

substitution. Moreover, we exploit several ideas in the recent inductive formulation of a type-inference algorithm by Dunfield and Krishnaswami Dunfield and Krishnaswami [2013], which turn out to be useful for mechanisation 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. While these meta-theoretical results are not new, as far as we know our work is the first to mechanically formalize them.

In summary the contributions of this paper are:

- **A mechanical formalization of a polymorphic subtyping algorithm.** We show that the algorithm is *sound*, *complete* and *decidable* in the Abella theorem prover, and make the Abella formalization available online[1].
- **Information propagation using worklist judgements:** we employ worklists judgements in our algorithmic specification of polymorphic subtyping to propagate information across judgements.

## 3.2 Overview: Polymorphic Subtyping

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

### 3.2.1 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 $A \leq B$.

[1] `https://github.com/JimmyZJX/Abella-subtyping-algorithm`

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

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

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

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

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

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

There is a very simple declarative formulation of polymorphic subtyping due to Odersky and Laüfer Odersky and Läufer [1996]. The syntax of this declarative system is shown in Figure 4.1. Types, represented 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 $\Psi$ collect a list of type variables.

In Figure 3.2, we give the well-formedness and subtyping relation for the declarative system. The cases without universal quantifiers are handled by Rules $\leq$Var, $\leq$Unit and $\leq\to$. The subtyping rule for function types ($\leq\to$) is standard, being contravariant on the argument types. Rule $\leq\forall$R says that if $A$ is a subtype of $B$ under the context extended with $a$, where $a$ is fresh in $A$, then $A \leq \forall a.B$. Intuitively, if $A$ is more general than the universally quantified type $\forall a.B$, then $A$ must instantiate to $[\tau/a]B$ for every $\tau$.

Finally, the most interesting rule is $\leq\forall$L, which instantiates $\forall a.A$ to $[\tau/a]A$, and concludes the subtyping $\forall a.A \leq B$ if the instantiation is a subtype of $B$. Notice that $\tau$ is *guessed*, and the algorithmic system should provide the means to compute this guess. Furthermore, the guess is a *monotype*, which rules out the possibility of polymorphic (or impredicative) instantiation. The restriction to monotypes and predicative instantiation is used by both Peyton Jones et al. Peyton Jones et al. [2007] and Dunfield and Krishnaswami's Dunfield and Krishnaswami [2013] algorithms, which are adopted by several practical implementations of programming languages.

### 3.2.2 Finding Solutions for Variable Instantiation

The declarative system specifies the behavior of subtyping relations, but is not directly implementable: the rule $\leq\forall$L requires guessing a monotype $\tau$. 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 $\leq\!\!\rightarrow$ splits one judgment into two, and the (partial) solutions found for existential variables when processing the first judgment should be transfered to the second judgement.

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

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

A notable difference to the declarative judgement is the presence of a so-called *output context* $\Phi$, which refines the *input context* $\Psi$ with solutions for existential variables found while processing the two types being compared for subtyping. Both $\Psi$ and $\Phi$ 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 different 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 judgements in Dunfield and Krishnaswami's approach. For example, the algorithmic rule corresponding to $\leq\!\!\rightarrow$ in their approach is:

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

The information gathered by the output context when comparing the input types of the functions for subtyping is transfered to the second judgement 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 $\forall a.a \rightarrow a$ is a subtype of $1 \rightarrow 1$, the declarative system will guess the proper $\tau = 1$ for Rule $\leq\forall\mathtt{L}$:

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

Dunfield and Krishnaswami introduce an *existential variable*—denoted with $\alpha, \beta$—whenever a monotype $\tau$ needs to be guessed. Below is a sample derivation of their algorithm; we omit the full set of algorithmic rules due to lack of space:

$$\dfrac{\dfrac{\dfrac{}{\alpha \vdash 1 \leq \alpha \dashv \alpha = 1}\,\texttt{InstRSolve} \qquad \dfrac{}{\alpha = 1 \vdash 1 \leq 1 \vdash \alpha = 1}\,<:\texttt{Unit}}{\alpha \vdash \alpha \to \alpha \leq 1 \to 1 \dashv \alpha = 1}\,<:\to}{\cdot \vdash \forall a.a \to a \leq 1 \to 1 \dashv \cdot}\,<:\forall\texttt{L}$$

The first step applies Rule $<:\forall\texttt{L}$, which introduces a fresh existential variable, $\alpha$, and opens the left-hand-side $\forall$-quantifier with it. Next, Rule $<:\to$ splits the judgment in two. For the first branch, Rule `InstRSolve` satisfies $1 \leq \alpha$ by solving $\alpha$ 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 judgement is still fairly complicated and poses challenges when porting their approach to a theorem prover.

### 3.2.3 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 $\Omega$ is a list of judgments $A \leq B$ instead of a single one. This judgement form, which we call *worklist judgement*, 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 become simpler. The transfer of information across judgements is simplified because all judgements share the input context. Moreover the order of the judgements in the list allows in-

formation discovered when processing the earlier judgements to be easily transfered to the later judgements. 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:

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

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 judgement to $\alpha \vdash \alpha \to \alpha \leq 1 \to 1; \cdot$. Then, we split the judgment in two for the function types and the derivation comes to $\alpha \vdash 1 \leq \alpha; \alpha \leq 1; \cdot$. When the first judgment is solved with $\alpha = 1$, we immediately remove $\alpha$ 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.

## 3.3 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 judgement that facilitates the propagation of information.

### 3.3.1 Syntax and Well-Formedness of the Algorithmic System

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

$$
\begin{array}{lll}
\text{Type variables} & a, b & \\
\text{Existential variables} & \alpha, \beta & \\
\\
\text{Algorithmic types} & A, B, C & ::= \quad 1 \mid a \mid \alpha \mid \forall a.A \mid A \to B \\
\text{Algorithmic context} & \Gamma & ::= \quad \cdot \mid \Gamma, a \mid \Gamma, \alpha \\
\text{Algorithmic judgments} & \Omega & ::= \quad \cdot \mid A \leq B; \Omega
\end{array}
$$

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

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

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

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

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

Worklist Judgements The form of our algorithmic judgements is non-standard. Our algorithm keeps track of an explicit list of outstanding work: the list $\Omega$ of (reified) *algorithmic judgements* 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.

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 $\Gamma$ is the context $\Gamma_L, \bullet, \Gamma_R$ with a hole ($\bullet$). Hole notations with the same name implicitly share the same $\Gamma_L$ and $\Gamma_R$. A multi-hole notation like $\Gamma[\alpha][\beta]$ means $\Gamma_1, \alpha, \Gamma_2, \beta, \Gamma_3$.

### 3.3.2 Algorithmic Subtyping

The algorithmic subtyping judgement, defined in Figure 4.5, has the form $\Gamma \vdash \Omega$, where $\Omega$ collects multiple subtyping judgments $A \leq B$. The algorithm treats $\Omega$ 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`.

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

$$\frac{}{\Gamma \vdash \cdot}\ \texttt{a\_nil}$$

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

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

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

$$\frac{\alpha \notin FV(A) \cup FV(B) \quad \Gamma[\alpha_1, \alpha_2] \vdash \alpha_1 \to \alpha_2 \leq A \to B; [\alpha_1 \to \alpha_2/\alpha]\Omega}{\Gamma[\alpha] \vdash \alpha \leq A \to B; \Omega} \leq_{\texttt{a}}\texttt{instL}$$

$$\frac{\alpha \notin FV(A) \cup FV(B) \quad \Gamma[\alpha_1, \alpha_2] \vdash A \to B \leq \alpha_1 \to \alpha_2; [\alpha_1 \to \alpha_2/\alpha]\Omega}{\Gamma[\alpha] \vdash A \to B \leq \alpha; \Omega} \leq_{\texttt{a}}\texttt{instR}$$

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

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

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

Figure 3.4: Algorithmic Subtyping

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 $\leq_{\texttt{a}}\to$ consumes one judgment and pushes two to the worklist. A notable difference with the declarative Rule $\leq\forall\texttt{L}$ is that Rule $\leq_{\texttt{a}}\forall\texttt{L}$ requires no guessing of a type $\tau$ to instantiate the polymorphic type $\forall a.A$, but instead introduces an existential variable $\alpha$ to the context and to $A$. In accordance with the declarative system, where the monotype $\tau$ should be bound in the context $\Psi$, here $\alpha$ should only be solved to a monotype bound in $\Gamma$. More generally, for any algorithmic context $\Gamma[\alpha]$, the algorithmic variable $\alpha$ can only be solved to a monotype that is well-formed with respect to $\Gamma_L$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha_1 \vdash \cdot}\,\texttt{a\_nil}}{\alpha_1 \vdash 1 \leq 1; \cdot}\,\leq_{\texttt{a}}\texttt{unit}}{\alpha_1, \alpha_2 \vdash \alpha_1 \leq \alpha_2; 1 \leq 1; \cdot}\,\leq_{\texttt{a}}\texttt{solve\_ex}}{\alpha_1, \alpha_2, \beta \vdash \alpha_1 \leq \beta; \beta \leq \alpha_2; 1 \leq 1; \cdot}\,\leq_{\texttt{a}}\texttt{solve\_ex}}{\alpha_1, \alpha_2, \beta \vdash \beta \to \beta \leq \alpha_1 \to \alpha_2; 1 \leq 1; \cdot}\,\leq_{\texttt{a}}\to}{\alpha, \beta \vdash \beta \to \beta \leq \alpha; 1 \leq 1; \cdot}\,\leq_{\texttt{a}}\texttt{instR}}{\alpha \vdash \forall a.\, a \to a \leq \alpha; 1 \leq 1; \cdot}\,\leq_{\texttt{a}}\forall\texttt{L}}{\alpha \vdash \alpha \to 1 \leq (\forall a.\, a \to a) \to 1; \cdot}\,\leq_{\texttt{a}}\to}{\cdot \vdash \forall a.\, a \to 1 \leq (\forall a.\, a \to a) \to 1; \cdot}\,\leq_{\texttt{a}}\forall\texttt{L}$$

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

Figure 3.5: Successful and Failing Derivations for the Algorithmic Subtyping Relation

Secondly, Rules $\leq_{\texttt{a}}\texttt{instL}$ and $\leq_{\texttt{a}}\texttt{instR}$ partially instantiate existential types $\alpha$, to function types. The domain and range of the new function type are undetermined: they are set to two fresh existential variables $\alpha_1$ and $\alpha_2$. To make sure that $\alpha_1 \to \alpha_2$ has the same scope as $\alpha$, the new variables $\alpha_1$ and $\alpha_2$ are inserted in the same position in the context where the old variable $\alpha$ was. To propagate the instantiation to the remainder of the worklist, $\alpha$ is substituted for $\alpha_1 \to \alpha_2$ in $\Omega$. The *occurs-check* side-condition is necessary to prevent a diverging infinite instantiation. For example $1 \to \alpha \leq \alpha$ 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.

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 the left of 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 $\alpha$. Variable $\alpha$ will be solved later to some monotype that is well-formed within the context before $\alpha$. 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 $\alpha$ should be solved to a monotype of the form $\sigma \to \tau$. Rule `instR` implements this, but avoids guessing $\sigma$ and $\tau$ by "splitting" $\alpha$ into two existential variables, $\alpha_1$ and $\alpha_2$, which will be solved to some $\sigma$ and $\tau$ later. Step 5 applies Rule $\leq_{\mathtt{a}}\!\!\to$ again. Notice that after the split, $\beta$ appears in two judgments. When the first $\beta$ is solved during any step of derivation, the next $\beta$ 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, $\beta$ is solved in terms of $\alpha_1$, and in step 7, $\alpha_2$ is solved in terms of $\alpha_1$. Additionally, in step 6, when $\beta$ is solved, the substitution $[\alpha_1/\beta]$ is propagated to the rest of the judgment list, and thus the second judgment becomes $\alpha_1 \leq \alpha_2$. Steps 8 and 9 trivially finish the derivation. Notice that $\alpha_1$ 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 $\tau$ first, then $b$ is introduced to the context, so that $b \notin FV(\tau)$. As a result, we cannot find $\tau$ such that $\tau \leq b$. The right of Figure 3.5 shows the algorithmic derivation, which also fails due to the scoping—$\alpha$ is introduced earlier than $b$, thus it cannot be solved to $b$.

## 3.4 Metatheory

This section presents the 3 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.

### 3.4.1 Transfer to the Declarative System

To state the correctness of the algorithmic subtyping rules, Figure 3.6 introduces two *transfer* judgements to relate the declarative and the algorithmic system. The first judgement, transfer of contexts $\Gamma \to \Psi$, removes existential variables from the algorithmic context $\Gamma$ to obtain a declarative context $\Psi$. The second judgement, transfer of the judgement list $\Gamma \mid \Omega \rightsquigarrow \Omega'$, replaces all occurrences of existential variables in $\Omega$ by well-scoped mono-types. Notice that this judgment is not decidable, i.e. a pair of $\Gamma$ and $\Omega$ may be related with multiple $\Omega'$. However, if there exists some substitution that transforms $\Omega$ to $\Omega'$, and each subtyping judgment in $\Omega'$ holds, we know that $\Omega$ is potentially satisfiable.

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

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

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

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

Figure 3.6: 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* $\Gamma, \Gamma_1 \mid [\tau/\alpha]\Omega \rightsquigarrow \Omega'$ *, then* $\Gamma, \alpha, \Gamma_1 \mid \Omega \rightsquigarrow \Omega'$.

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

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 $\Psi$.

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

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

### 3.4.2 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 $\Omega'$ hold in $\Psi$, with $\Gamma \to \Psi$.

**Theorem 3.3** (Soundness). *If* $\Gamma \vdash \Omega$ *and* $\Gamma \to \Psi$*, then there exists* $\Omega'$*, 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.

### 3.4.3 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 $\leq_{\mathtt{a}}\mathtt{instL}$ and $\leq_{\mathtt{a}}\mathtt{instR}$ 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 possibile cases for the first corresponding algorithmic judgement is $\alpha \leq A \to B$. However, the case analysis does not indicate that $\alpha$ is fresh in $A$ and $B$. Thus we cannot apply Rule $\leq_{\mathtt{a}}\mathtt{instL}$ 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. $\alpha \leq 1 \to \alpha$) and guarantees that $\alpha$ 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* $\Gamma \mid (\alpha \leq A \to B; \Omega) \rightsquigarrow (A_1 \to A_2 \leq B_1 \to B_2; \Omega')$*, then* $\alpha \notin FV(A) \cup FV(B)$*.*

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

### 3.4.4 Decidability

The third key result for our algorithm is decidability.

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

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

**Definition 2** (Worklist Measures)**.**

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

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

**Definition 3** (Context Measure).

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

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 judgements, which we call *instantiation judgements* $\Omega_i$, separately. They take the form:

**Definition 4** ($\Omega_i$).

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

These instantiation judgements 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* $\Gamma$ *and judgment list* $\Omega_i, \Omega$*, 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* $|\Gamma'|_\alpha = |\Gamma|_\alpha - |\Omega_i|$*, it is decidable whether* $\Gamma' \vdash \Omega'$.

In other words, for any instantiation judgment prefix $\Omega_i$, 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 $\langle |\Omega|_\forall, |\Omega|_\alpha, |\Omega|_\to \rangle$. 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.

## 3.5 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 $\lambda$-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 $\nabla$-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 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:

| 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 3.7: Statistics for the proof scripts

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

### 3.5.1 Statistics and Discussion

Some basic statistics on our proof script are shown in Figure 3.7. 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.

## 3.6 Related Work

Type Inference for Polymorphic Subtyping Higher-order polymorphism is a practical and important programming language feature. Due to the undecidability of type-inference for System F Wells [1999], different decidable partial type-inference approaches were developed. The subtyping relation of this paper, originally proposed by Odersky and Laüfer Odersky and Läufer [1996], is *predicative* ($\forall$'s only instantiate to monotypes), which is considered a reasonable and practical trade-off. There is also work on partial impredicative type-inference algorithms Le Botlan and Rémy [2003]; Leijen [2008]; Vytiniotis et al. [2008]. However, unlike the predicative subtyping relation for System F, the subtyping for impredicative System F is undecidable Tiuryn and Urzyczyn [1996]. Therefore such algorithms have to navigate through the design space to impose restrictions that allow for a decidable algorithm. As a result such algorithms tend to be more complex, and are less adopted in practice.

Gundry et al. Gundry et al. [2010] revisited the Hindley-Milner type system. They make use of ordered contexts on the unification during type inference, and their algorithm works differently from algorithm $\mathcal{W}$. Dunfield and Krishnaswami Dunfield and Krishnaswami [2013] adopted a similar idea on ordered contexts and presented an algorithmic approach for predicative polymorphic subtyping that tracks the (partial) solutions of existential variables in the algorithmic context—this denotes a delayed substitution that is incrementally applied to outstanding work as it is encountered. Their algorithm comes with 40 pages of manual proofs on the soundness, completeness and decidability. We have tried to mechanize these proofs directly, but have not been successful yet because most proof assistants do not naturally support output contexts and their more complex ordered contexts. Their theorems have statements that are more complex than those in the worklist approach. One of the reasons for the added complexity is that, when the constraints are not strict enough, the algorithm may not instantiate all existential variables. However in order to match the declarative judgement, all the unsolved variables should be properly assigned. For example, their generalized completeness theorem is:

**Theorem 3.8** (Generalized Completeness of Subtyping Dunfield and Krishnaswami [2013])**.** *If* $\Psi \longrightarrow \Phi$ *and* $\Psi \vdash A$ *and* $\Psi \vdash B$ *and* $[\Phi]\Psi \vdash [\Phi]A \leq [\Phi]B$ *then there exist* $\Delta$ *and* $\Phi'$ *such that* $\Delta \longrightarrow \Phi'$ *and* $\Phi \longrightarrow \Phi'$ *and* $\Psi \vdash [\Psi]A <: [\Psi]B \dashv \Delta$.

Here, the auxiliary relation $\Psi \longrightarrow \Psi'$ extends a context $\Psi$ to a context $\Psi'$. This is used to extend the algorithm's input and output contexts $\Psi$ and $\Delta$, with with possibly unassigned existential variables, to a complete (i.e., fully-assigned) contexts $\Phi$ and $\Phi'$ suitable for the declarative specification.

While we are faced with a similar gap between algorithm and specification, which we tackle with our transfer relations $\Gamma \rightarrow \Psi$, our completeness statement is much shorter because our

algorithm does not return an output context which needs to be transferred. Moreover, we have cleanly encapsulated any substitutions to the worklist in the worklist transfer judgement $\Gamma \mid \Omega \rightsquigarrow \Omega'$.

Peyton Jones et al. Peyton Jones et al. [2007] developed a higher-rank predicative bidirectional type system. They enriched their subtyping relations with deep skolemisation, while other relations remain similar to ours. Their algorithm is unification-based with a structure similar to algorithm $\mathcal{W}$'s.

Unification Algorithms Our algorithm works similarly to some unification algorithms that use a set of unification constraints and single-step simplification transitions. Some work Abel and Pientka [2011]; Reed [2009] adopts this idea in dependently typed inference and reconstruction. These approaches collect a set of constraints and nondeterministically process one of them at a time. Those approaches consider various forms of constraints, including term unification, context unification and solution for meta-variables. In contrast, our algorithm is presented in a simpler form, using ordered (worklist) judgements, which is sufficient for the subtyping problem.

Formalizations of Type-Inference Algorithms in Theorem Provers The well-known POPLMark challenge Aydemir et al. [2005] has encouraged the development of new proof assistant features for facilitating the development and verification of type systems. As a result, many theorem provers and packages now provide methods for dealing with variable binding Aydemir et al. [2008]; Chlipala [2008]; Urban [2008], and more and more type system designers choose to formalize their proofs with these tools. Yet, difficulties with mechanising algorithmic aspects, like unification and constraint solving, have received very little attention. Moreover, while most type system judgements only feature local (input) contexts, which have a simple binding/scoping structure, many traditional type-inference algorithms require more complex binding structures with output contexts.

Naraschewski and Nipkow Naraschewski and Nipkow [1999] published the first formal verification of algorithm $\mathcal{W}$ in Isabelle/HOL Nipkow et al. [2002]. The treatment of new variables is a little tricky in their formalization, while most other parts follow the structure of Damas's manual proof closely. Following Naraschewski and Nipkow other researchers Dubois [2000]; Dubois and Menissier-Morain [1999] prove a similar result in Coq The Coq development team [2017]. Nominal techniques Urban [2008] in Isabelle/HOL have also been used for a similar verification Urban and Nipkow [2008]. Moreover, Garrigue Garrigue [2015] mechanized a type inference algorithm for Core ML extended with structural polymorphism and recursion.

## 3.7 Conclusion and Future Work

In this paper we have shown how to mechanise an algorithmic subtyping relation for higher-order polymorphism, together with its proofs of soundness, completeness and decidability, in the Abella proof assistant. In ongoing work we are extending our mechanisation with a bidirectional type inference algorithm. The main difficulty there is communicating the instantiations of existential variables from the subtyping algorithm to the type inference. To make this possible we are exploring a continuation passing style formulation, which generalises the worklist approach. Another possible extension is to have the algorithm return an explicit witness for the subtyping as part of type-directed elaboration into System F.

# 4 Higher-Rank Polymorphism Worklist Algorithm

A Mechanical Formalization of Higher-Ranked Polymorphic Type Inference

## 4.1 Introduction

Modern functional programming languages, such as Haskell or OCaml, use sophisticated forms of type inference. The type systems of these languages are descendants of Hindley-Milner Damas and Milner [1982]; Hindley [1969]; Milner [1978], which was revolutionary at the time in allowing type-inference to proceed without any type annotation. The traditional Hindley-Milner type system supports top-level *implicit (parametric) polymorphism* Reynolds [1983]. With implicit polymorphism, type arguments of polymorphic functions are automatically instantiated. Thus implicit polymorphism and the absence of type annotations mean that the Hindley-Milner type system strikes a great balance between expressive power and usability.

As functional languages evolved the need for more expressive power has motivated language designers to look beyond Hindley-Milner. In particular one popular direction is to allow *higher-ranked polymorphism* where polymorphic types can occur anywhere in a type signature. An important challenge is that full type inference for higher-ranked polymorphism is known to be undecidable Wells [1999]. Therefore 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]. 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. In those type systems type annotations are required for polymorphic arguments of functions, but other type annotations can be omitted. A canonical example (here written in Haskell) is:

```
hpoly = \(f :: forall a. a -> a) -> (f 1, f 'c')
```

The function `hpoly` cannot be type-checked in the Hindley-Milner type system. The type of `hpoly` is the rank-2 type: `(forall a. a -> a) -> (Int, Char)`. 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. Despite these additional annotations, Peyton Jones et al. and DK's type inference algorithms preserve many of the desirable properties of Hindley-Milner. For example the applications of `f` implicitly instantiate the polymorphic type arguments of `f`.

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]. 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. 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.

This paper presents the first fully mechanized formalization of the metatheory for higher-ranked polymorphic type inference. The system that we formalize is the bidirectional type system by 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. 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 example, Klein et al. 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* Zhao et al. [2018], proposed recently to mechanically formalize an algorithm for *polymorphic subtyping* Odersky and Läufer [1996]. A key innovation in our work 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].

In summary, the contributions of this paper are:

- **A fully mechanized formalization of type inference with higher-ranked types:** Our work presents the first fully mechanized formalization for type inference of higher ranked types. The formalization is done in the Abella theorem prover Gacek [2008] and it is available online at `https://github.com/JimmyZJX/TypingFormalization`.
- **A new algorithm for DK's type system:** Our work proposes a novel algorithm that implements DK's declarative bidirectional type system. We prove *soundness*, *completeness* and *decidability*.
- **Worklists with inference judgments:** One technical contribution is the support for inference judgments using worklists. The idea is to use a continuation passing style to enable the transfer of inferred information across judgments.
- **Unification of worklists and contexts:** Another technical contribution is the unification between ordered contexts and worklists. This enables precise scope tracking of variables in judgments, and avoids the duplication of context information across judgments in worklists.

## 4.2 Overview

This section starts by introducing DK's declarative type system. Then it discusses 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 the unifies *ordered contexts* and the worklists themselves.

### 4.2.1 DK's Declarative System

Syntax. The syntax of DK's declarative system Dunfield and Krishnaswami [2013] is shown in Figure 4.1. 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 $\tau$ do not have any universal quantifier. Terms include a unit term $()$, variables $x$, lambda-functions $\lambda x.\ e$, applications $e_1\ e_2$ and annotations $(e : A)$. Contexts $\Psi$ are sequences of type variable declarations and term variables with their types declared $x : A$.

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

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

Well-formedness Well-formedness of types and terms is shown at the top of Figure 4.2. The rules are standard and simply ensure that variables in types and terms are well-scoped.

Declarative Subtyping The bottom of Figure 4.2 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 $\leq$`Var`, $\leq$`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

$\boxed{\Psi \vdash e \Leftarrow A}$ $e$ checks against input type $A$.
$\boxed{\Psi \vdash e \Rightarrow A}$ $e$ synthesizes output type $A$.
$\boxed{\Psi \vdash A \bullet e \Rrightarrow C}$ Applying a function of type $A$ to $e$ synthesizes type $C$.

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

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

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

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

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

Figure 4.3: Declarative Typing

and contravariant on the argument type. Rule $\leq\forall\texttt{R}$ states that if $A$ is a subtype of $B$ in the context $\Psi, a$, 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 $\tau$.

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 $\tau$ 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 $\tau$ 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 4.3, has three judgments. The checking judgment $\Psi \vdash e \Leftarrow A$ checks expression $e$ against the type $A$ in the context $\Psi$. The synthesis judgment $\Psi \vdash e \Rightarrow A$ synthesizes the type $A$ of expression $e$ in the context $\Psi$. The

application judgment $\Psi \vdash A \bullet e \Rrightarrow C$ 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⇒` 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 $A$. Checking an expression $e$ against a polymorphic type $\forall a.\ A$ in the context $\Psi$ succeeds if $e$ checks against $A$ in the extended context $(\Psi, a)$. 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 $\tau$ 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.

Overlapping Rules Some of the declarative 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 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 $\tau$. The declarative typing rule `DeclSub` overlaps with both `Decl∀I` and `Decl→I`. However, we argue that more specific rules are always the best choices, i.e. `Decl∀I` and `Decl→I` should have higher priority than `DeclSub`. We will come back to this topic in Section 4.4.2.

### 4.2.2 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 $\tau$ 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 $\Gamma$ with their solutions on type $A$.

All algorithmic judgments thread an algorithmic context. They have the form $\Gamma \vdash \ldots \dashv \Delta$, where $\Gamma$ is the input context and $\Delta$ 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 $\Theta$ 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 existential 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:

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

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 $\Theta$, 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 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. Fortu-

nately, 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]. See Appendix for a detailed explanation and counterexample. 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.

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 $\Gamma$ 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 4.5.1 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.

[1] Ningning Xie found the issue with Lemma 29 in 2016 while collaborating with the second author 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.

### 4.2.3 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:

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

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 $\Delta$ 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 $\Delta$ 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 $\Psi$. 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.

### 4.2.4 Single-Context Worklist Algorithm for Subtyping

In recent work, Zhao et al. [2018] have shown how to mechanize a variant of DK's subtyping algorithm and shown it correct with respect to DK's declarative subtyping judgment. This approach overcomes some problems in DK's formulation by using a *worklist*-based judgment of the form

$$\Gamma \vdash \Omega$$

where $\Omega$ 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 $\Omega$ and recursively processes the worklist until

it is empty. Using both a single common ordered context $\Gamma$ and a worklist $\Omega$ 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 $\Gamma$ 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, $\Gamma$ 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.

### 4.2.5 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 $\Gamma$, 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 $\Gamma$ 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 $\omega$. 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 $\omega$ by substituting it for the placeholder type variable $a$. Notice that such $a$ binds a fresh type variable for the inner chain $\omega$, 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.

## 4.3 Algorithmic System

This section introduces a novel algorithmic system that implements DK's declarative specification. The new algorithm extends the idea of worklists proposed by Zhao et al. [2018] in two ways. Firstly, unlike Zhao et al. [2018]'s 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.

### 4.3.1 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 $\tau$ 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 which $\widehat{\alpha}$ has been added. The point is that the monotype $\tau$, 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 $\tau$'s.

Judgment Chains Judgment chains $\omega$, 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

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

$\boxed{\Gamma \vdash A}$ 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

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* $\Gamma$. 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 $\Gamma$ is the worklist $\Gamma_L, \bullet, \Gamma_R$ with a hole ($\bullet$). Hole notations with the same name implicitly share the same structure $\Gamma_L$ and $\Gamma_R$. A multi-hole notation splits the worklist into more parts. For example, $\Gamma[\widehat{\alpha}][\widehat{\beta}]$ means $\Gamma_1, \widehat{\alpha}, \Gamma_2, \widehat{\beta}, \Gamma_3$.

### 4.3.2 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.

Please note that when a new variable is introduced in the right-hand side worklist $\Gamma'$, 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. That is variables that 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 $\leq\forall\mathrm{L}$ by introducing an existential variable $\widehat{\alpha}$ instead of guessing the monotype $\tau$ instantiation. Each existential variable is later solved to a monotype $\tau$ with the same context, so the final solution $\tau$ of $\widehat{\alpha}$ should be well-formed under $\Gamma$.

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

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

$$\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 \\
\\
\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) \\
\\
\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[]) \\
\\
\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) \\
\\
\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) \\
\\
\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

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.

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` $\rightarrow$ `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 $\Gamma, \widehat{\beta}_1, \widehat{\beta}_2 \Vdash \widehat{\beta}_1 \rightarrow \widehat{\beta}_2 \leq \widehat{\alpha}, x : \widehat{\beta}_1 \Vdash e \Leftarrow \widehat{\beta}_2$. As a subtyping constraint between monotypes is simply equality, $\widehat{\beta}_1 \rightarrow \widehat{\beta}_2 \leq \widehat{\alpha}$ must end up equating $\widehat{\beta}_1 \rightarrow \widehat{\beta}_2$ 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 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 $\omega$ 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 $\omega$.

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 4.5.1.

**6. Application inference judgments** Finally, Rules 27-29 deal with application inference judgments. Rules 27 and 28 behaves similarly to declarative rules `Decl∀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 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

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

Figure 4.6: A Sample Derivation for Algorithmic Typing

$\widehat{\alpha}$ 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 $\leq\forall\mathtt{L}$ and $\leq\forall\mathtt{R}$. 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 $B \neq \forall a.\ B'$.

These restrictions have not been imposed arbitrarily: we formally prove that the restricted algorithm is still complete. In Section 4.4.2 we discuss the relevant metatheory, with the help of a non-overlapping version of the declarative system.

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

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

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

Figure 4.7: Declarative Worklists and Instantiation

## 4.4 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].

### 4.4.1 Declarative Worklist and Transfer

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

Worklist instantiation. The relation $\Gamma \rightsquigarrow \Omega$ expresses that the algorithmic worklist $\Gamma$ can be instantiated to the declarative worklist $\Omega$, by appropriately instantiating all existential variables $\widehat{\alpha}$ in $\Gamma$ with well-scoped monotypes $\tau$. 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 $\Omega$ can be abstracted to an existential variable in $\Gamma$. Thus different $\Gamma$'s can be instantiated to the same $\Omega$.

Lemmas 4.1 and 4.2 generalize Rule $\rightsquigarrow\widehat{\alpha}$ from substituting the first existential variable to substituting any existential variable.

**Lemma 4.1** (Insert). *If* $\Gamma_L, [\tau/\widehat{\alpha}]\Gamma_R \rightsquigarrow \Omega$ *and* $\Gamma_L \vdash \tau$ *, then* $\Gamma_L, \widehat{\alpha}, \Gamma_R \rightsquigarrow \Omega$.

$\boxed{\|\Omega\|}$ 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

**Lemma 4.2** (Extract). *If* $\Gamma_L, \widehat{\alpha}, \Gamma_R \rightsquigarrow \Omega$ *, then there exists* $\tau$ *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.

### 4.4.2 Non-Overlapping Declarative System

DK's declarative system, shown in Figures 4.2 and 4.3, 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:

[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.

1. Restrict the shape of $B$ in the rule $\forall\mathtt{L}$ 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`:

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

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 $\forall\mathtt{L}$ overlaps with Rule $\forall\mathtt{R}$ for the judgment $\Psi \vdash \forall a.\, A \leq \forall b.\, B$. The following inversion lemma for Rule $\forall\mathtt{R}$ resolves the overlap:

**Lemma 4.3** (Invertibility of $\forall\mathtt{R}$). *If* $\Psi \vdash A \leq \forall a.\, B$ *then* $\Psi, a \vdash A \leq B$.

The lemma implies that preferring Rule $\forall\mathtt{R}$ does not affect the derivability of the judgment. Therefore the restriction $B \neq \forall b.\, B'$ in $\forall\mathtt{L}'$ 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 4.4** (Invertibility of `Decl`$\forall$`I`). *If* $\Psi \vdash e \Leftarrow \forall a.\, A$ *then* $\Psi, a \vdash e \Leftarrow A$.

**Lemma 4.5** (Invertibility of `Decl`$\rightarrow$`I`). *If* $\Psi \vdash \lambda x.\, e \Leftarrow A \to B$ *then* $\Psi, x : A \vdash e \Leftarrow B$.

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

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

Figure 4.9: Context Subtyping

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 $\Psi$ subsumes context $\Psi'$ 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 4.6** (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* $\Psi' \vdash e \Rightarrow B'$*;*
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 $\Psi$ has a corresponding derivation in any context $\Psi'$ that it subsumes.

We have tried to follow DK's manual proof of this lemma, 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. The details of this issue can be found in Appendix.

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.

### 4.4.3 Soundness

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

**Theorem 4.7** (Soundness). *If* wf $\Gamma$ *and* $\Gamma \longrightarrow^* \cdot$, *then there exists* $\Omega$ *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 4.1 and 4.2 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 4.8** (Soundness, single judgment form). *Given* wf $\Gamma$*:*

1. *If* $\Gamma \Vdash A \leq B \longrightarrow^* \cdot$
   *then there exist* $A', B', \Omega$ *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* $A', \Omega$ *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* $\omega$
   *then there exists* $\Omega, \omega', A$ *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* $\omega$
   *then there exists* $\Omega, \omega', A', C$ *s.t.* $\Gamma \Vdash A \bullet e \Rrightarrow_a \omega \rightsquigarrow \Omega \Vdash A' \bullet e \Rrightarrow_a \omega'$ *and* $\|\Omega\| \vdash A' \bullet e \Rrightarrow C$.

### 4.4.4 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 $\Gamma$ and a declarative context $\Omega$ to avoid potential confusion.

**Theorem 4.9** (Completeness). *If* wf $\Gamma$ *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 4.10** (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 4.9.

**Corollary 4.11** (Completeness, single judgment form). *Given* wf $\Gamma$ *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* $\Gamma \Vdash A \bullet e \Rrightarrow_a 1 \leq 1 \longrightarrow^* \cdot$.

### 4.4.5 Decidability

Finally, we show that our algorithm is decidable:

**Theorem 4.12** (Decidability). *Given* wf $\Gamma$, *it is decidable whether* $\Gamma \longrightarrow^* \cdot$ *or not.*

Our decidability proof is based on a lexicographic group of induction measures $\langle |\Gamma|_e, |\Gamma|_\Leftrightarrow, |\Gamma|_\forall, |\Gamma|_{\widehat{\alpha}}, |\Gamma|_\to + |\Gamma| \rangle$ on the worklist $\Gamma$. Formal definitions of these measures can be found in the Appendix. The first two, $|\cdot|_e$ and $|\cdot|_\Leftrightarrow$, measure the total size of terms and the

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

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

Figure 4.10: Worklist Update

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 $\Gamma, \Gamma_i$ where $\Gamma_i$ 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. This only happens when $\Gamma_i$ 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 $\Gamma$, by solving existing existentials and introducing new ones. This worklist update is captured in the update judgment $\Gamma \rightharpoonup \Gamma'$ defined in Figure 4.10. 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.

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

**Lemma 4.13** (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 4.14** (Instantiation Decidability). *For any well-formed algorithmic worklist* $(\Gamma, \Gamma_i)$*:*

*1) If* $|\Gamma_i|_\forall = 0$*, then there exists* $\Gamma'$
*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 $\Gamma_i$ 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 $\Gamma_i$.

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 4.14 to show that $|\Gamma|_{\widehat{\alpha}}$ or $|\Gamma|_\forall$ decreases when that instantiation judgment is consumed or partially reduced. Moreover, Lemma 4.13 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 4.15** (Decidability of Declarative Typing). *Given* wf $\Omega$*, it is decidable whether* $\Omega \longrightarrow^* \cdot$ *or not.*

### 4.4.6 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 $\lambda$-tree syntax Miller [2000], which is a form of HOAS. Nominal variables, or $\nabla$-quantified variables, are used as an unlimited name supply, which supports explicit freshness control and substitutions. Although Abella lacks packages, tactics and support

[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) | SLOC | #Theorems | 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 4.1, 4.2 |
| declTyping.thm | 909 | 70 | Non-overlapping declarative system; Lemmas 4.3, 4.4, 4.5, 4.6 |
| soundness.thm | 1,107 | 78 | Soundness theorem; aux. lemmas on transfer |
| depth.thm | 206 | 14 | The $\lvert\cdot\rvert_\rightarrow$ measure; Lemma 4.10 |
| 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 4.14 |
| 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) |

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 limitations of Abella.

## 4.5 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].

### 4.5.1 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 4.2.2). Despite the sophistication employed in DK's algorithm to 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

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

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

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 `DK_→I⇒` 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.

### 4.5.2 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 $\Phi$ 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.

### 4.5.3 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 4.4.

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

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

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 simulatenously 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:

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

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

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

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

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.

## 4.6 Related Work

Throughout the paper we have already discussed much of the closest related work. In this section we summarize the key differences and novelties, and we discuss some other related work.

Predicative Higher-Ranked Polymorphism Type Inference 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 system of this paper, proposed by Dunfield and Krishnaswami [2013], is *predicative*: ∀'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, the bidirectional system accepts guidance through type annotations, which 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 Appendix ), 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 Appendix.

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. In comparison to DK's system, they do not use the application inference judgment, resulting in 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 a syntactic sugar. The formalization includes some mechanized proofs for the declarative type system, but all proofs regarding the algorithmic type system are manual.

Impredicative Higher-Ranked Polymorphism Type Inference 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 Le Botlan and Rémy [2003]; Leijen [2008]; Vytiniotis et al. [2008] navigate a

variety of design tradeoffs for a decidable algorithm. As a result, such algorithms tend to be more complicated, and thus less adopted in practice. Recent work proposed *Guarded Impredicative Polymorphism* Serrano et al. [2018], 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.

Mechanical Formalization of Polymorphic Subtyping In all previous work on type inference for higher-ranked polymorphism (predicative and impredicative) discussed above, proofs and metatheory for the algorithmic aspects are manual. The only partial effort on mechanizing algorithmic aspects of type inference for higher-ranked types is the Abella formalization of *polymorphic subtyping* by Zhao et al. [2018]. The judgment form of worklist $\Gamma \vdash \Omega$ used in the formalization simplifies the propagation of existential variable instantiations. However, the approach has two main drawbacks: it does not collect unused variable declarations effectively; and the simple form of judgment cannot handle inference modes, which output types. The new worklist introduced in this paper inherits the simplicity of propagating instantiations, but overcomes both of the issues by mixing judgments with declarations and using the continuation-passing-style judgment chains. Furthermore, we formalize the complete bidirectional type system by Dunfield and Krishnaswami [2013], whereas Zhao et al. only formalize the subtyping relation.

Mechanical Formalizations of Other Type-Inference Algorithms 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.

Ordered Contexts in Type Inference Gundry et al. 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 flow 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.

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.

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 scopes 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 in Section 4.2.3.

## 4.7 Conclusion

In this paper we have provided the first mechanized formalization of type inference for higher-ranked polymorphism. This contribution is made possible by a new type inference algorithm for DK's declarative type system that incorporates two novel mechanization-friendly ideas. Firstly, we merge the traditional type context with the recently proposed concept of judgment chains, to accurately track variable scopes and to easily propagate substitutions. Secondly, we use a continuation-passing style to return types from the synthesis mode to subsequent tasks.

We leave extending our algorithm with elaboration to future work, as well as investigating whether the problems we have found in DK's manual proofs for their algorithm can be addressed.

# 5 Higher-Rank Polymorphism with Object-Oriented Subtyping

## 5.1 Introduction and Motivation

## 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. 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 exception. The type Exception $\to \bot$ given to the `raise` function may pass type checkers natually. The type Exception $\to \forall a.\ a$ is also a reasonable choice, which in fact reveals that $\bot$ behaves almost identical to $\forall a.\ a$. In the theoretical point of view, both of them are considered "falsity".

The well-formedness for 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 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 on polymorphic instantiations. For example, the subtyping judgement

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

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

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

Figure 5.1: Declarative Syntax

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

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

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

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

Figure 5.2: Declarative Subtyping

Declarative Typing The declarative typing rules, shown in Figure 5.3, extends 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.

It's worth mentioning that the design of the two new rules are driven by the subsumption property described in Section 5.4.1. They maintains 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

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 $\omega$ and worklist context $\Gamma$ are defined in the same way as the ICFP work.

The well-formedness relation is almost the same as that of the ICFP 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

### 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).

Jimmy: TODO: $\{\widehat{\alpha} := \tau\}$ Notation: Hole notation + solve notation

We categorize them into 7 groups according to their behavior:

1. Rules 1-3 are basic garbage collection rules. Given that the worklist $\Gamma$ 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 are nothing to guess immediately, and thus the algorithm behaves just like the declarative system.

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

Figure 5.4: Algorithmic Syntax

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 requires backtracking techniques for implementation. These rules *overlaps* with all the remaining rules when solving an existential 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 judgement 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 have 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 check is 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 deals with subtyping judgments that compares 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 does 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. Other programs that exploit complex guessing involving subtyping, the programmer may put annotation when the type inference algorithm does not find the optimal solution.

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

$$
\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

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

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

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

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

Figure 5.6: Algorithmic Subtyping

# Part II

# Related and Future Work

# 6 Related Work

# 7 Future Work

# Part III

# Epilogue

# 8 Conclusion

# 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 28]

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 24, 29, 36, and 62]

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 24, 28, and 61]

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 8, 24, 28, and 61]

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 28]

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 28]

Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (01 Oct 2012), 363–408. [cited on page 62]

Paul Chiusano and Runar Bjarnason. 2015. Unison. `http://unisonweb.org` [cited on page 27]

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 pages 24 and 61]

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 7, 11, 27, and 61]

Disciple Development Team. 2017. The Disciplined Disciple Compiler. `http://disciple.ouroborus.net/` [cited on page 8]

Catherine Dubois. 2000. Proving ML type soundness within Coq. *Theorem Proving in Higher Order Logics* (2000), 126–144. [cited on pages 8, 24, 28, and 61]

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 8, 24, 28, and 61]

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 8, 9, 10, 11, 14, 23, 27, 28, 29, 30, 60, and 61]

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 page 60]

Phil Freeman. 2017. PureScript. `http://www.purescript.org/` [cited on pages 8 and 27]

Andrew Gacek. 2008. The Abella Interactive Theorem Prover (System Description). In *Proceedings of IJCAR 2008 (Lecture Notes in Artificial Intelligence)*. [cited on pages 20, 29, 30, 47, and 54]

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 8, 24, 28, and 62]

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 7]

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 23, 29, and 62]

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 7, 27, and 61]

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 7]

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 page 29]

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 7]

John Launchbury and Simon L. Peyton Jones. 1995. State in Haskell. *LISP and Symbolic Computation* 8, 4 (1995), 293–341. [cited on page 7]

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 23, 27, and 60]

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 23, 27, and 60]

Xavier Leroy et al. 2012. The CompCert verified compiler. *Documentation and user' s manual. INRIA Paris-Rocquencourt* (2012). [cited on pages 8 and 28]

Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.* 4, 2 (April 1982), 258–282. [cited on page 36]

Dale Miller. 2000. Abstract Syntax for Variable Binders: An Overview. In *CL 2000: Computational Logic (Lecture Notes in Artificial Intelligence)*. [cited on pages 21 and 54]

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375. [cited on pages 7, 27, 35, and 61]

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 8, 24, 28, and 61]

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 pages 24 and 61]

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 8, 10, 23, 29, 31, and 65]

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 56 and 58]

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of functional programming* 17, 1 (2007), 1–82. [cited on pages 7, 8, 10, 24, 27, 32, and 60]

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 36 and 62]

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 24, 29, 36, and 62]

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Proceedings of the IFIP 9th World Computer Congress*. [cited on pages 7, 27, and 57]

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 27 and 61]

The Coq development team. 2017. The Coq proof assistant. `https://coq.inria.fr/` [cited on pages 24 and 61]

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 23, 32, and 60]

Christian Urban. 2008. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning* 40, 4 (2008), 327–356. [cited on pages 24, 61, and 62]

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 8, 24, 28, and 62]

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 23, 27, and 60]

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 7, 23, 27, and 60]

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 60]

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 pages 29, 36, 39, and 61]

# Part IV

# Technical Appendix