A Mechanical Formalization of Higher-Ranked Polymorphic Type Inference

JINXU ZHAO, The University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China
TOM SCHRIJVERS, KU Leuven, Belgium

Modern functional programming languages, such as Haskell or OCaml, use sophisticated forms of type inference. While an important topic in the Programming Languages research, there is little work on the mechanization of the metatheory of type inference in theorem provers. In particular we are unaware of any complete formalization of the type inference algorithms that are the backbone of modern functional languages.

This paper presents the first full mechanical formalization of the metatheory for higher-ranked polymorphic type inference. The system that we formalize is the bidirectional type system by Dunfield and Krishnaswami (DK). The DK type system has two variants (a declarative and an algorithmic one) that have been *manually* proven *sound*, *complete* and *decidable*. We present a mechanical formalization in the Abella theorem prover of DK's declarative type system with a novel algorithmic system. We have a few reasons to use a new algorithm. Firstly, our new algorithm employs *worklist judgments*, which precisely capture the scope of variables and simplify the formalization of scoping in a theorem prover. Secondly, while DK's original formalization comes with very well-written manual proofs, there are several details missing and some incorrect proofs, which complicate the task of writing a mechanized proof. Despite the use of a different algorithm we prove the same results as DK, although with significantly different proofs and proof techniques. Since such type inference algorithms are quite subtle and have a complex metatheory, mechanical formalizations are an important advance in type-inference research.

CCS Concepts: • **Software and its engineering → Functional languages**; **Polymorphism**; **Formal language definitions**.

Additional Key Words and Phrases: type inference, higher-rank polymorphism, mechanization

ACM Reference Format:
Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. A Mechanical Formalization of Higher-Ranked Polymorphic Type Inference. *Proc. ACM Program. Lang.* 3, ICFP, Article 112 (August 2019), 29 pages. https://doi.org/10.1145/3341716

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

Authors' addresses: Jinxu Zhao, Department of Computer Science, The University of Hong Kong, Hong Kong, China, jxzhao@cs.hku.hk; Bruno C. d. S. Oliveira, Department of Computer Science, The University of Hong Kong, Hong Kong, China, bruno@cs.hku.hk; Tom Schrijvers, Department of Computer Science, KU Leuven, Leuven, Belgium, tom.schrijvers@cs.kuleuven.be.

2475-1421/2019/8-ART112
https://doi.org/10.1145/3341716

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:

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

Fig. 1. Syntax of Declarative System

- **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.

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.

2.1 DK's Declarative System

Syntax. The syntax of DK's declarative system [Dunfield and Krishnaswami 2013] is shown in Figure 1. A declarative type A is either the unit type 1, a type variable a, a universal quantification $\forall a.\ A$ or a function type $A \to B$. Nested universal quantifiers are allowed for types, but monotypes τ do not have any universal quantifier. Terms include a unit term (), variables x, lambda-functions $\lambda x.\ e$, applications $e_1\ e_2$ and annotations $(e : A)$. Contexts Ψ are sequences of type variable declarations and term variables with their types declared $x : A$.

Well-formedness. Well-formedness of types and terms is shown at the top of Figure 2. The rules are standard and simply ensure that variables in types and terms are well-scoped.

Declarative Subtyping. The bottom of Figure 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 and contravariant on the argument type. Rule $\leq\forall$R states that if A is a subtype of B in the context

$\boxed{\Psi \vdash A}$ 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

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

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

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

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

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

Fig. 2. Declarative Well-formedness and Subtyping

Ψ, a, where a is fresh in A, then $A \le \forall a.\, B$. Intuitively, if A is more general than $\forall a.\, B$ (where the universal quantifier already indicates that $\forall a.\, B$ is a general type), then A must instantiate to $[\tau/a]B$ for every τ.

The most interesting rule is $\le\forall\mathtt{L}$. If some instantiation of $\forall a.\, A$, $[\tau/a]A$, is a subtype of B, then $\forall a.\, A \le B$. The monotype τ we used to instantiate a is *guessed* in this declarative rule, but the algorithmic system does not guess and defers the instantiation until it can determine the monotype deterministically. The fact that τ is a monotype rules out the possibility of polymorphic (or impredicative) instantiation. However this restriction ensures that the subtyping relation remains decidable. Allowing an arbitrary type (rather than a monotype) in rule $\le\forall\mathtt{L}$ 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 $\mathtt{wf_d}\forall$, $\mathtt{wf_d abs}$, $\le\forall\mathtt{R}$, throughout the whole text.

Declarative Typing. The bidirectional type system, shown in Figure 3, has three judgments. The checking judgment $\Psi \vdash e \Leftarrow A$ checks expression e against the type A in the context Ψ. The synthesis judgment $\Psi \vdash e \Rightarrow A$ synthesizes the type A of expression e in the context Ψ. The application judgment $\Psi \vdash A \bullet e \Rrightarrow C$ 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`$\Rightarrow$ respectively check and synthesize the unit type. Rule `DeclAnno` synthesizes the annotated type A of the annotated expression $(e : A)$ and checks that e has type A. Checking an expression e against a polymorphic type $\forall a.\, A$ in the context Ψ succeeds if e checks against A in the extended context (Ψ, a). The subsumption rule `DeclSub` depends on the subtyping relation, and changes mode from checking to synthesis: if e synthesizes type A and $A \le B$ (A is more polymorphic than B), then e checks against B. If a checking problem does not match any other rules, this rule

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

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

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

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

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

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

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

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

Fig. 3. Declarative Typing

can be applied to synthesize a type instead and then check whether the synthesized type entails the checked type. Lambda abstractions are the hardest construct of the bidirectional type system to deal with. Checking $\lambda x.\ e$ against function type $A \to B$ is easy: we check the body e against B in the context extended with $x : A$. However, synthesizing a lambda-function is a lot harder, and this type system only synthesizes monotypes $\sigma \to \tau$.

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

Overlapping Rules. Some of the declarative rules overlap with each other. Declarative subtyping rules ≤∀L and ≤∀R both match the conclusion $\Psi \vdash \forall a.\ A \leq \forall a.\ B$. In such case, choosing ≤∀R first is always better, since we introduce the type variable a to the context earlier, which gives more flexibility on the choice of τ. The declarative typing rule DeclSub overlaps with both Decl∀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 4.2.

2.2 DK's Algorithm

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

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

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

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

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

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

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

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

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

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

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

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

More seriously, we have also found a lemma that simply does not hold: Lemma 29 (Parallel Admissibility)[1]. 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

[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.

$\mathcal{W}$ [Milner 1978] which features an implicit global scope for all type variables. Indeed, DK uses its explicit and ordered context Γ that tracks the relative scope of universal and existential variables and it is careful to only instantiate existential variables in a well-scoped manner.

Moreover, DK's algorithm carefully propagates instantiations by recording them into the context and threading this context through all judgments. While this works well on paper, this approach is still fairly involved and thus hard to reason about in a mechanized setting. Indeed, the instantiations have to be recorded in the context and are applied incrementally to each remaining judgment in turn, rather than immediately to all remaining judgments at once. Also, as we have mentioned above, the stack discipline of the ordered contexts does not mesh well with the use of output contexts; explicit marker entries are needed in two places to demarcate the end of an existential variable's scope. Actually we found a scoping issue related to the subsumption rule `DK_Sub`, which might cause existential variables to leak across judgments. In Section 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.

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:

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

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

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

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 Ω is a worklist (or sequence) of subtyping problems of the form $A \leq B$. The judgment is defined by case analysis on the first element of Ω and recursively processes the worklist until

it is empty. Using both a single common ordered context Γ and a worklist Ω greatly simplifies propagating the instantiation of type variables in one subtyping problem to the remaining ones.

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

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

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

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 Γ, our algorithmic worklist. This algorithmic worklist interleaves (type and term) variables with *work* like checking or inferring types of expressions. The interleaving keeps track of the variable scopes in the usual, natural way: each variable is in scope of anything in front of it in the worklist. If there is nothing in front, the variable is no longer needed and can be popped from the worklist. This way, no garbage (i.e. variables out-of-scope) builds up.

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

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

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

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

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

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

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

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

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

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

placeholder type variable a. Notice that such a binds a fresh type variable for the inner chain ω, which behaves similarly to the variable declarations in the context.

Take the following worklist as an example

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

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

The algorithm first analyses the two subtyping judgments and will find the best solutions $\widehat{\alpha} = \widehat{\beta} = 1$ (please refer to Figure 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.

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.

3.1 Syntax and Well-Formedness

Figure 4 shows the syntax and well-formedness judgments used by the algorithm. Similarly to the declarative system the well-formedness rules are unsurprising and merely ensure well-scopedness.

Existential Variables. The algorithmic system inherits the syntax of terms and types from the declarative system. It only introduces one additional feature. In order to find unknown types τ in the declarative system, the algorithmic system extends the declarative types A with *existential variables* $\widehat{\alpha}, \widehat{\beta}$. They behave like unification variables, but their scope is restricted by their position in the algorithmic worklist rather than being global. Any existential variable $\widehat{\alpha}$ should only be solved to a type that is well-formed with respect to the worklist to which $\widehat{\alpha}$ has been added. The point is that the monotype τ, represented by the corresponding existential variable, is always well-formed under the corresponding declarative context. In other words, the position of $\widehat{\alpha}$'s reflects the well-formedness restriction of τ's.

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

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

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

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

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

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

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

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

$$
\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \le B}\ \mathtt{wf}{\le} \qquad \frac{\Gamma \vdash e \qquad \Gamma \vdash A}{\Gamma \vdash e \Leftarrow A}\ \mathtt{wf}{\Leftarrow}
$$

$$
\frac{\Gamma \vdash e \qquad \Gamma, a \vdash \omega}{\Gamma \vdash e \Rightarrow_a \omega}\ \mathtt{wf}{\Rightarrow} \qquad \frac{\Gamma \vdash A \qquad \Gamma, a \vdash \omega \qquad \Gamma \vdash e}{\Gamma \vdash A \bullet e \Rightarrow\!\!\!\Rightarrow_a \omega}\ \mathtt{wf}{\Rightarrow\!\!\!\Rightarrow}
$$

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

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

Fig. 4. Extended Syntax and Well-Formedness for the Algorithmic System

is a subtype of B. The *unknown* type of e is represented by a type variable a, which is used as a placeholder in the second judgment to denote the type of e.

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

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

3.2 Algorithmic System

The algorithmic typing reduction rules, defined in Figure 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

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

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

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

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

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

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

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

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

Fig. 5. Algorithmic Typing

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 Γ', we implicitly pick a fresh one, since the right-hand side can be seen as the premise of the reduction.

Rules 1-3 pop variable declarations that are essentially garbage. 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 $\leq\forall L$ by introducing an existential variable $\widehat{\alpha}$ instead of guessing the monotype τ instantiation. Each existential variable is later solved to a monotype τ with the same context, so the final solution τ of $\widehat{\alpha}$ should be well-formed under Γ.

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

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

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

First, the subtyping of two function types is split into two judgments by Rule 7: a covariant subtyping on the return type and a contravariant subtyping on the argument type. Then we apply Rule 8 to reduce the $\forall$ quantifier on the left side. The rule introduces an existential variable $\widehat{\alpha}$ to the context (even though the type $\forall a.\ 1 \to 1$ does not actually refer to the quantified type variable a). In the following 3 steps we satisfy the judgment $1 \to 1 \leq 1 \to 1$ by Rules 7, 4 and 4.

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 $\leq\!\!\rightarrow$; 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 $\widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2$. The *occurs-check* condition prevents divergence as usual. For example, without it $\widehat{\alpha} \leq 1 \rightarrow \widehat{\alpha}$ 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 ω by assigning A to its placeholder variable a. Rule 23 infers an annotated expression by changing into checking mode, therefore another judgment chain is created. Rule 24 is a base case, where the unit type 1 is inferred and thus passed to its child judgment chain. Rule 26 infers the type of an application by firstly inferring the type of the function e_1, and then leaving the rest work to an application inference judgment, which passes a, representing the return type of the application, to the remainder of the judgment chain ω.

Rule 25 infers the type of a lambda expression by introducing $\widehat{\alpha}, \widehat{\beta}$ as the input and output types of the function, respectively. After checking the body e under the assumption $x : \widehat{\alpha}$, the return type might reflect more information than simply $\widehat{\alpha} \to \widehat{\beta}$ through propagation when existential variables are solved or partially solved. The variable scopes are maintained during the process: the assumption of argument type $(x : \widehat{\alpha})$ is recycled after checking against the function body; the existential variables used by the function type $(\widehat{\alpha}, \widehat{\beta})$ are only visible in the remaining chain $[\widehat{\alpha} \to \widehat{\beta}/a]\omega$. The recycling process of Rule 25 differs from DK's corresponding rule significantly, and we further discuss the differences in Section 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 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 $\widehat{\alpha}$ 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 $1 \le 1$, 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 $\le\forall\mathsf{L}$ and $\le\forall\mathsf{R}$. While those declarative rules both match the goal $\forall a.\ A \le \forall b.\ B$, we have eliminated this overlap in the algorithm by restricting Rule 8 ($B \neq \forall a.\ B'$) and thus always applying Rule 9 to $\forall a.\ A \le \forall b.\ B$.

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 4.2 we discuss the relevant metatheory, with the help of a non-overlapping version of the declarative system.

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

Fig. 6. A Sample Derivation for Algorithmic Typing

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

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

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

Fig. 7. Declarative Worklists and Instantiation

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].

4.1 Declarative Worklist and Transfer

To aid formalizing the correspondence between the declarative and algorithmic systems, we introduce the notion of a declarative worklist Ω, defined in Figure 7. A declarative worklist Ω has the same structure as an algorithmic worklist Γ, but does not contain any existential variables $\widehat{\alpha}$.

Worklist instantiation. The relation $\Gamma \rightsquigarrow \Omega$ expresses that the algorithmic worklist Γ can be instantiated to the declarative worklist Ω, by appropriately instantiating all existential variables $\widehat{\alpha}$ in Γ with well-scoped monotypes τ. The rules of this instantiation relation are shown in Figure 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 $\widehat{\alpha}$.

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

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

Fig. 8. Declarative Transfer

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

Lemmas 4.1 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* $\Gamma_L, \widehat{\alpha}, \Gamma_R \rightsquigarrow \Omega$.

Lemma 4.2 (Extract). *If* $\Gamma_L, \widehat{\alpha}, \Gamma_R \rightsquigarrow \Omega$ *, then there exists* τ *s.t.* $\Gamma_L \vdash \tau$ *and* $\Gamma_L, [\tau/\widehat{\alpha}]\Gamma_R \rightsquigarrow \Omega$.

Declarative transfer. Figure 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.

4.2 Non-Overlapping Declarative System

DK's declarative system, shown in Figures 2 and 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:

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

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

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

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

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

Fig. 9. Context Subtyping

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

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

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

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

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

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

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

The lemma implies that preferring Rule $\forall$`R` does not affect the derivability of the judgment. Therefore the restriction $B \neq \forall b.\, B'$ in $\forall$`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* $\Psi \vdash \lambda x.\, e \Leftarrow A \rightarrow B$ *then* $\Psi, x : A \vdash e \Leftarrow B$.

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

The proofs of the above two lemmas rely on an important property of the declarative system, the *subsumption lemma*. To be able to formulate this lemma, Figure 9 introduces the *context subtyping relation* $\Psi \leq \Psi'$. Context Ψ subsumes context Ψ' if they bind the same variables in the same order, but the types A of the term variables x in the former are subtypes of types A' assigned to those term variables in the latter. Now we can state the subsumption lemma:

Lemma 4.6 (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 $\Psi \vdash A \bullet e \Longrightarrow C$ *and* $\Psi \vdash A' \leq A$*, then there exists* C' *s.t.* $\Psi \vdash C' \leq C$ *and* $\Psi' \vdash A' \bullet e \Longrightarrow C'$.

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

4.3 Soundness

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

Theorem 4.7 (Soundness). *If* $\text{wf}\,\Gamma$ *and* $\Gamma \longrightarrow^* \cdot$, *then there exists* Ω *s.t.* $\Gamma \rightsquigarrow \Omega$ *and* $\Omega \longrightarrow^* \cdot$.

The proof proceeds by induction on the derivation of $\Gamma \longrightarrow^* \cdot$. Interesting cases are those involving existential variable instantiations, including Rules 10, 11, 21 and 29. Applications of Lemmas 4.1 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* $\text{wf}\,\Gamma$*:*

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

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

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

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

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 Γ and a declarative context Ω to avoid potential confusion.

Theorem 4.9 (Completeness). *If* $\text{wf}\,\Gamma$ *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 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 $A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2$. One of the corresponding algorithmic judgments is $\widehat{\alpha} \leq A \rightarrow B$. 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 $A \rightarrow B \leq \widehat{\alpha}$.

Lemma 4.10 (Prune Transfer for Instantiation). *If* $(\Gamma \Vdash \widehat{\alpha} \leq A \rightarrow B) \rightsquigarrow (\Omega \Vdash C \leq A_1 \rightarrow B_1)$ *and* $\|\Omega\| \vdash C \leq A_1 \rightarrow B_1$, *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 Γ *containing no judgments:*

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

4.5 Decidability

Finally, we show that our algorithm is decidable:

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

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

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

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

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

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

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

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

Fig. 10. Worklist Update

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

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

Secondly, reducing the instantiation judgments may also affect the remainder of the worklist Γ, by solving existing existentials and introducing new ones. This worklist update is captured in the update judgment $\Gamma \rightharpoonup \Gamma'$ defined in Figure 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* (Γ, Γ_i)*:*

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

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

Let us go back to the whole algorithm and summarize our findings. The decidability theorem is shown through a lexicographic group of induction measures $\langle |\Gamma|_e, |\Gamma|_\Leftrightarrow, |\Gamma|_\forall, |\Gamma|_{\widehat{\alpha}}, |\Gamma|_\to + |\Gamma| \rangle$ 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 Ω*, it is decidable whether* $\Omega \longrightarrow^* \cdot$ *or not.*

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

[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 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 variable binding and the λ-tree syntax [Miller 2000], which is a form of HOAS. Nominal variables, or ∇-quantified variables, are used as an unlimited name supply, which supports explicit freshness control and substitutions. Although Abella lacks packages, tactics and support for modules, its higher-order unification and the ease of formalizing substitution-intensive relations are very helpful.

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

Statistics of the Proof. The proof script consists of 7,977 lines of Abella code with a total of 60 definitions and 596 theorems. Figure 1 briefly summarizes the contents of each file. The files are linearly dependent due to limitations of Abella.

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].

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 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

$$\dfrac{\dfrac{\dfrac{\dfrac{\ldots x \Rightarrow \widehat{\alpha} \ldots \qquad \ldots \widehat{\alpha} \leq \widehat{\beta} \ldots}{\Gamma, \widehat{\alpha}, \widehat{\beta}, x : \widehat{\alpha} \vdash x \Leftarrow \widehat{\beta} \dashv \Gamma_1, x : \widehat{\alpha}}}{\Gamma \vdash \lambda x.\ x \Rightarrow \widehat{\alpha} \to \widehat{\beta} \dashv \Gamma_1} \qquad \dfrac{\ldots () \Leftarrow \widehat{\alpha} \ldots}{\Gamma_1 \vdash \widehat{\alpha} \to \widehat{\alpha} \bullet () \Rrightarrow \widehat{\alpha} \dashv \Gamma_2}}{\Gamma \vdash (\lambda x.\ x)\ () \Rightarrow \widehat{\alpha} \dashv \Gamma_2} \qquad \dfrac{}{\Gamma_2 \vdash 1 \leq 1 \dashv \Gamma_2}}{\Gamma \vdash (\lambda x.\ x)\ () \Leftarrow 1 \dashv \Gamma, \widehat{\alpha} = 1, \widehat{\beta} = \widehat{\alpha}}$$

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

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

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

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

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

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

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

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

It seems impossible to achieve a similar outcome in DK's system by only modifying these two rules. Taking $\texttt{DK\_}{\to}\texttt{I}\Rightarrow$ as an example, the declaration or solution for $\widehat{\alpha}$ and $\widehat{\beta}$ may be referred to by subsequent judgments. Therefore leaving $\widehat{\alpha}$ and $\widehat{\beta}$ in the output context is the only choice, when the subsequent judgments cannot be consulted.

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

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

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.

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

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

Then the form of our algorithmic judgment becomes:

$$\Gamma \hookrightarrow \Phi$$

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

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

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

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

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.

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

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

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

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

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

derivable. To address the problem, we add a new syntactic form that explicitly binds a type variable 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:

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

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

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

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

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

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

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*: $\forall$'s only instantiate to monotypes. The monotype restriction on instantiation is considered reasonable and practical for most programs, except for those that require sophisticated forms of higher-order polymorphism. In those cases, 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 2.3.

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.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their insightful comments. Ningning Xie found the issue with Lemma 29 in DK's formalization that we reported on this article. This work has been sponsored by the Hong Kong Research Grant Council projects number 17210617 and 17258816, and by the Research Foundation - Flanders.

REFERENCES

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).

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.

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)*.

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*.

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)*.

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)*.

Arthur Charguéraud. 2012. The Locally Nameless Representation. *Journal of Automated Reasoning* 49, 3 (01 Oct 2012), 363–408.

Paul Chiusano and Runar Bjarnason. 2015. Unison. http://unisonweb.org

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)*.

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)*.

Catherine Dubois. 2000. Proving ML type soundness within Coq. *Theorem Proving in Higher Order Logics* (2000), 126–144.

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.

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)*.

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.

Phil Freeman. 2017. PureScript. http://www.purescript.org/

Andrew Gacek. 2008. The Abella Interactive Theorem Prover (System Description). In *Proceedings of IJCAR 2008 (Lecture Notes in Artificial Intelligence)*.

Jacques Garrigue. 2015. A certified implementation of ML with structural polymorphism and recursive types. *Mathematical Structures in Computer Science* 25, 4 (2015), 867–891.

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)*.

Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society* 146 (1969), 29–60.

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 (POPL '12)*. 285–296.

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)*.

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)*.

Xavier Leroy et al. 2012. The CompCert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt* (2012).

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

Dale Miller. 2000. Abstract Syntax for Variable Binders: An Overview. In *CL 2000: Computational Logic (Lecture Notes in Artificial Intelligence)*.

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375.

Wolfgang Naraschewski and Tobias Nipkow. 1999. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning* 23, 3 (1999), 299–318.

Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.

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)*.

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.

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.

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.

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)*.

John C Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. *Information Processing* (1983), 513–523.

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)*.

The Coq development team. 2017. The Coq proof assistant. https://coq.inria.fr/

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*.

Christian Urban. 2008. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning* 40, 4 (2008), 327–356.

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.

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)*.

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.

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.

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.