Formalization of a Polymorphic Subtyping Algorithm

Jinxu Zhao[1], Bruno C. d. S. Oliveira[1], and Tom Schrijvers[2]

[1] The University of Hong Kong, {jxzhao,bruno}@cs.hku.hk
[2] KU Leuven, tom.schrijvers@cs.kuleuven.be

Abstract. Modern functional programming languages such as Haskell support sophisticated forms of type-inference, even in the presence of *higher-order polymorphism*. Central to such advanced forms of type-inference is an algorithm for polymorphic subtyping. This paper formalizes an algorithmic specification for polymorphic subtyping in the Abella theorem prover. The algorithmic specification is shown to be *decidable*, and *sound* and *complete* with respect to Odersky and Läufer's well-known declarative formulation of polymorphic subtyping.
While the meta-theoretical results are not new, as far as we know our work is the first to mechanically formalize them. Moreover, our algorithm differs from those currently in the literature by using a novel approach based on *worklist judgements*. Worklist judgements simplify the propagation of information required by the unification process during subtyping. Furthermore they enable a simple formulation of the meta-theoretical properties, which can be easily encoded in theorem provers.

1 Introduction

Most statically typed functional languages support a form of *(implicit) parametric polymorphism* [28]. Traditionally, functional languages have employed variants of the Hindley-Milner [14, 23, 5] type system, which supports full type-inference without any type annotations. However the Hindley-Milner type system only supports *first-order polymorphism*, where all universal quantifiers only occur at the top-level of a type. Modern functional programming languages such as Haskell go beyond Hindley-Milner and support *higher-order polymorphism*. With higher-order polymorphism there is no restriction on where universal quantifiers can occur. This enables more code reuse and more expressions to type-check, and has numerous applications [15, 12, 18, 17].

Unfortunately, with higher-order polymorphism full type-inference becomes undecidable [35]. To recover decidability some type annotations on polymorphic arguments are necessary. A canonical example that requires higher-order polymorphism in Haskell is:

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

The function `hpoly` cannot be type-checked in Hindley-Milner. The type of `hpoly` is `(forall a. a -> a) -> (Int, Char)`. The single universal quantifier does

not appear at the top-level. Instead it is used to quantify a type variable `a` used in the first argument of the function. Notably `hpoly` requires a type annotation for the first argument (`forall a. a -> a`). Despite these additional annotations, the type-inference algorithm employed by GHC Haskell [16] preserves many of the desirable properties of Hindley-Milner. Like in Hindley-Milner type instantiation is *implicit*. That is, calling a polymorphic function never requires the programmer to provide the instantiations of the type parameters.

Central to type-inference with *higher-order polymorphism* is an algorithm for polymorphic subtyping. This algorithm allows us to check whether one type is more general than another, which is essential to detect valid instantiations of a polymorphic type. For example, the type `forall a. a -> a` is more general than `Int -> Int`. A simple declarative specification for polymorphic subtyping was proposed by Odersky and Läufer [26]. Since then several algorithms have been proposed that implement it. Most notably, the algorithm proposed by Peyton Jones et al. [16] forms the basis for the implementation of type inference in the GHC compiler. Dunfield and Krishnaswami [9] provided a very elegant formalization of another sound and complete algorithm, which has also inspired implementations of type-inference in some polymorphic programming languages (such as PureScript [30] or DDC [6]).

Unfortunately, while many aspects of programming languages and type systems have been mechanically formalized in theorem provers, there is little work on formalizing algorithms related to type-inference. The main exceptions to the rule are mechanical formalizations of algorithm $\mathcal{W}$ and other aspects of traditional Hindler-Milner type-inference [24, 7, 8, 33, 11]. However, as far as we know, there is no mechanisation of algorithms used by modern functional languages like Haskell, and polymorphic subtyping included is no exception. This is a shame because recently there has been a lot of effort in promoting the use of theorem provers to check the meta-theory of programming languages, e.g., through well-known examples like the POPLMark challenge [3] and the CompCert project [21]. Mechanical formalizations are especially valuable for proving the correctness of the semantics and type systems of programming languages. Type-inference algorithms are arguably among the most non-trivial aspects of the implementations of programming languages. In particular the information discovery process required by many algorithms (through unification-like or constraint-based approaches), is quite subtle and tricky to get right. Moreover, extending type-inference algorithms with new programming language features is often quite delicate. Studying the meta-theory for such extensions would be greatly aided by the existence of a mechanical formalization of the base language, which could then be extended by the language designer.

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

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

Fig. 1. Syntax of Declarative System

This paper advances the state-of-the-art by formalizing an algorithm for polymorphic subtyping in the Abella theorem prover. We hope that this work encourages other researchers to use theorem provers for formalizing type-inference algorithms. In particular, we show that the problem we have identified above can be overcome by means of *worklist judgments*. These are a form of judgement that turns the complicated global propagation of unifications into a simple local substitution. Moreover, we exploit several ideas in the recent inductive formulation of a type-inference algorithm by Dunfield and Krishnaswami [9], which turn out to be useful for mechanisation in a theorem prover.

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

In summary the contributions of this paper are:

- **A mechanical formalization of a polymorphic subtyping algorithm.** We show that the algorithm is *sound*, *complete* and *decidable* in the Abella theorem prover, and make the Abella formalization available online[3].
- **Information propagation using worklist judgements:** we employ worklists judgements in our algorithmic specification of polymorphic subtyping to propagate information across judgements.

2 Overview: Polymorphic Subtyping

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

2.1 Declarative Polymorphic Subtyping

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

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

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

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

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

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

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

Fig. 2. Well-formedness of Declarative Types and Declarative Subtyping

There is a very simple declarative formulation of polymorphic subtyping due to Odersky and Laüfer [26]. The syntax of this declarative system is shown in Figure 1. Types, represented by A, B, C, are the unit type 1, type variables a, b, universal quantification $\forall a.A$ and function type $A \to B$. We allow nested universal quantifiers to appear in types, but not in monotypes. Contexts Ψ collect a list of type variables.

In Figure 2, we give the well-formedness and subtyping relation for the declarative system. The cases without universal quantifiers are handled by Rules $\leq\mathtt{Var}$, $\leq\mathtt{Unit}$ and $\leq{\to}$. The subtyping rule for function types ($\leq{\to}$) is standard, being contravariant on the argument types. Rule $\leq\forall\mathtt{R}$ says that if A is a subtype of B under the context extended with a, where a is fresh in A, then $A \leq \forall a.B$. Intuitively, if A is more general than the universally quantified type $\forall a.B$, then A must instantiate to $[\tau/a]B$ for every τ.

Finally, the most interesting rule is $\leq\forall\mathtt{L}$, which instantiates $\forall a.A$ to $[\tau/a]A$, and concludes the subtyping $\forall a.A \leq B$ if the instantiation is a subtype of B. Notice that τ is *guessed*, and the algorithmic system should provide the means to compute this guess. Furthermore, the guess is a *monotype*, which rules out the possibility of polymorphic (or impredicative) instantiation. The restriction to monotypes and predicative instantiation is used by both Peyton Jones et al. [16] and Dunfield and Krishnaswami's [9] algorithms, which are adopted by several practical implementations of programming languages.

2.2 Finding Solutions for Variable Instantiation

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

Dunfield and Krishnaswami's Approach An elegant algorithmic solution to computing the monotypes is presented by Dunfield and Krishnaswami [9]. Their algorithmic subtyping judgement has the form:

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

A notable difference to the declarative judgement is the presence of a so-called *output context* Φ, which refines the *input context* Ψ with solutions for existential variables found while processing the two types being compared for subtyping. Both Ψ and Φ are *ordered contexts* with the same structure. Ordered contexts are particularly useful to keep track of the correct scoping for variables, and are a notable different to older type-inference algorithms [5] that use global unification variables or constraints collected in a set.

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

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

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

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

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

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

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

The first step applies Rule $<:\!\forall\mathtt{L}$, which introduces a fresh existential variable, α, and opens the left-hand-side $\forall$-quantifier with it. Next, Rule $<:\!\!\rightarrow$ splits the judgment in two. For the first branch, Rule `InstRSolve` satisfies $1 \leq \alpha$ by solving α to 1, and stores the solution in its output context. The output context of the first branch is used as the input context of the second branch, and the judgment is updated according to current solutions. Finally, the second branch becomes a base case, and Rule $<:$`Unit` finishes the derivation, makes no change to the input context and propagates the output context back.

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

2.3 The Worklist Approach

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

$$\Gamma \vdash \Omega$$

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

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

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

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

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

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

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

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

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

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

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

Fig. 3. Syntax and Well-Formedness Judgement for the Algorithmic System.

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

3 A Worklist Algorithm for Polymorphic Subtyping

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

3.1 Syntax and Well-Formedness of the Algorithmic System

Figure 3 shows the syntax and the well-formedness judgement.

Existential Variables In order to solve the unknown types τ, the algorithmic system extends the declarative syntax of types with *existential variables* α. They behave like unification variables, but are not globally defined. Instead, the ordered *algorithmic context*, inspired by Dunfield and Krishnaswami [9], defines their scope. Thus the type τ represented by the corresponding existential variable is always bound in the corresponding declarative context Ψ.

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 4. Algorithmic Subtyping

3.2 Algorithmic Subtyping

The algorithmic subtyping judgement, defined in Figure 4, has the form $\Gamma \vdash \Omega$, where Ω collects multiple subtyping judgments $A \leq B$. The algorithm treats Ω as a worklist. In every step it takes one task from the worklist for processing, possibly pushes some new tasks on the worklist, and repeats this process until the list is empty. This last and single base case is handled by Rule `a_nil`. The remaining rules all deal with the first task in the worklist. Logically we can discern 3 groups of rules.

Firstly, we have five rules that are similar to those in the declarative system, mostly just adapted to the worklist style. For instance, Rule $\leq_{\texttt{a}}\to$ consumes one judgment and pushes two to the worklist. A notable difference with the declarative Rule $\leq\forall\texttt{L}$ is that Rule $\leq_{\texttt{a}}\forall\texttt{L}$ requires no guessing of a type τ to instantiate the polymorphic type $\forall a.A$, but instead introduces an existential variable α to the context and to A. In accordance with the declarative system,

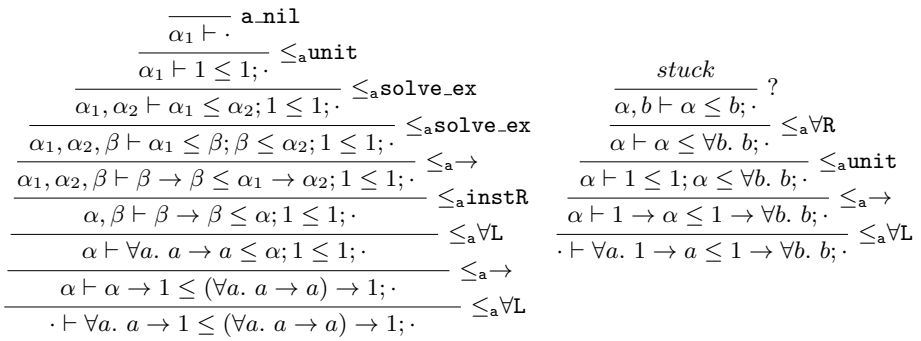

Fig. 5. Successful and Failing Derivations for the Algorithmic Subtyping Relation

where the monotype τ should be bound in the context Ψ, here α should only be solved to a monotype bound in Γ. More generally, for any algorithmic context $\Gamma[\alpha]$, the algorithmic variable α can only be solved to a monotype that is well-formed with respect to Γ_L.

Secondly, Rules $\leq_{\mathtt{a}}\mathtt{instL}$ and $\leq_{\mathtt{a}}\mathtt{instR}$ partially instantiate existential types α, to function types. The domain and range of the new function type are undetermined: they are set to two fresh existential variables α_1 and α_2. To make sure that $\alpha_1 \to \alpha_2$ has the same scope as α, the new variables α_1 and α_2 are inserted in the same position in the context where the old variable α was. To propagate the instantiation to the remainder of the worklist, α is substituted for $\alpha_1 \to \alpha_2$ in Ω. The *occurs-check* side-condition is necessary to prevent a diverging infinite instantiation. For example $1 \to \alpha \leq \alpha$ would diverge with no such check.

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

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

Example We illustrate the subtyping rules through a sample derivation in the left of Figure 5, which shows that that $\forall a.\ a \to 1 \leq (\forall a.\ a \to a) \to 1$. Thus the derivation starts with an empty context and a judgment list with only one element.

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

hand-side $\forall$ of the first judgment is opened according to rule $\leq_{\mathtt{a}}\forall\mathtt{L}$ with a fresh existential variable. In step 4, the first judgment has an arrow on the left hand side, but the right-hand-side type is an existential variable. It is obvious that α should be solved to a monotype of the form $\sigma \to \tau$. Rule `instR` implements this, but avoids guessing σ and τ by "splitting" α into two existential variables, α_1 and α_2, which will be solved to some σ and τ later. Step 5 applies Rule $\leq_{\mathtt{a}}\to$ again. Notice that after the split, β appears in two judgments. When the first β is solved during any step of derivation, the next β will be substituted by that solution. This propagation mechanism ensures the consistent solution of the variables, while keeping the context as simple as possible. Steps 6 and 7 solve existential variables. The existential variable that is right-most in the context is always solved in terms of the other. Therefore in step 6, β is solved in terms of α_1, and in step 7, α_2 is solved in terms of α_1. Additionally, in step 6, when β is solved, the substitution $[\alpha_1/\beta]$ is propagated to the rest of the judgment list, and thus the second judgment becomes $\alpha_1 \leq \alpha_2$. Steps 8 and 9 trivially finish the derivation. Notice that α_1 is not instantiated at the end. This means that any well-scoped instantiation is fine.

A Failing Derivation We illustrate the role of ordered contexts through another example: $\forall a.\ 1 \to a \leq 1 \to \forall b.\ b$. From the declarative perspective, a should be instantiated to some τ first, then b is introduced to the context, so that $b \notin FV(\tau)$. As a result, we cannot find τ such that $\tau \leq b$. The right of Figure 5 shows the algorithmic derivation, which also fails due to the scoping—α is introduced earlier than b, thus it cannot be solved to b.

4 Metatheory

This section presents the 3 main meta-theoretical results that we have proved in Abella. The first two are soundness and completeness of our algorithm with respect to Odersky and Läufer's declarative subtyping. The third result is our algorithm's decidability.

4.1 Transfer to the Declarative System

To state the correctness of the algorithmic subtyping rules, Figure 6 introduces two *transfer* judgements to relate the declarative and the algorithmic system. The first judgement, transfer of contexts $\Gamma \to \Psi$, removes existential variables from the algorithmic context Γ to obtain a declarative context Ψ. The second judgement, transfer of the judgement list $\Gamma \mid \Omega \rightsquigarrow \Omega'$, replaces all occurrences of existential variables in Ω by well-scoped mono-types. Notice that this judgment is not decidable, i.e. a pair of Γ and Ω may be related with multiple Ω'. However, if there exists some substitution that transforms Ω to Ω', and each subtyping judgment in Ω' holds, we know that Ω is potentially satisfiable.

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

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

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

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

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

Fig. 6. Transfer Rules

Lemma 1 (Insert). *If* $\Gamma \to \Psi$ *and* $\Psi \vdash \tau$ *and* $\Gamma, \Gamma_1 \mid [\tau/\alpha]\Omega \rightsquigarrow \Omega'$ *, then* $\Gamma, \alpha, \Gamma_1 \mid \Omega \rightsquigarrow \Omega'$*.*

Lemma 2 (Extract). *If* $\Gamma, \alpha, \Gamma_1 \mid \Omega \rightsquigarrow \Omega'$*, then* $\exists \tau$ *s.t.* $\Gamma \to \Psi, \Psi \vdash \tau$ *and* $\Gamma, \Gamma_1 \mid [\tau/\alpha]\Omega \rightsquigarrow \Omega'$*.*

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

Definition 1 (Declarative Subtyping Worklist).

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

4.2 Soundness

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

Theorem 1 (Soundness). *If* $\Gamma \vdash \Omega$ *and* $\Gamma \to \Psi$*, then there exists* Ω'*, s.t.* $\Gamma \mid \Omega \rightsquigarrow \Omega'$ *and* $\Psi \vdash \Omega'$*.*

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

4.3 Completeness

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

Theorem 2 (Completeness). *If* $\Psi \vdash \Omega'$ *and* $\Gamma \to \Psi$ *and* $\Gamma \mid \Omega \rightsquigarrow \Omega'$*, then* $\Gamma \vdash \Omega$*.*

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

Lemma 3 (Prune Transfer for Instantiation). *If $\Psi \vdash A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2; \Omega'$ and $\Gamma \rightarrow \Psi$ and $\Gamma \mid (\alpha \leq A \rightarrow B; \Omega) \rightsquigarrow (A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2; \Omega')$, then $\alpha \notin FV(A) \cup FV(B)$.*

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

4.4 Decidability

The third key result for our algorithm is decidability.

Theorem 3 (Decidability). *Given any well-formed judgment list Ω under Γ, it is decidable whether $\Gamma \vdash \Omega$ or not.*

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

Definition 2 (Worklist Measures).

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

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

Definition 3 (Context Measure).

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

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

Definition 4 (Ω_i)**.**

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

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

Lemma 4 (Instantiation Decidability). *For any context Γ and judgment list Ω_i, Ω, it is decidable whether $\Gamma \vdash \Omega_i, \Omega$ if both of the conditions hold*

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

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

In summary, the decidability theorem can be shown through a lexicographic group of induction measurements $\langle |\Omega|_\forall, |\Omega|_\alpha, |\Omega|_\rightarrow \rangle$. The critical case is that, whenever we encounter an instantiation judgment at the front of the worklist, we refer to Lemma 4, which reduces the number of unsolved variables by consuming that instantiation judgment, or reduces the number of $\forall$-quantifiers. Other cases are relatively straightforward.

5 The Choice of Abella

We have chosen the Abella (v2.0.5) proof assistant [10] to develop our formalization. Our development is only based on the reasoning logic of Abella, and does not make use of its specification logic. Abella is particularly helpful due to its built-in support for variable bindings, and its λ-tree syntax [22] is a form of HOAS, which helps with the encoding and reasoning about substitutions. For instance, the type $\forall x.x \rightarrow a$ is encoded as `all (x\ arrow x a)`, where `x\ arrow x a` is a lambda abstraction in Abella. An opening $[b/x](x \rightarrow a)$ is encoded as an application `(x\ arrow x a) b`, which can be simplified(evaluated) to `arrow b a`. Name supply and freshness conditions are controlled by the ∇-quantifier. The expression `nabla x, F` means that `x` is a unique variable in `F`, i.e. it is different from any other names occurring elsewhere. Such variables are called nominal constants. They can be of any type, in other words, every type may contain unlimited number of such atomic nominal constants.

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

```
Kind ty    type.
Type i     ty.                % the unit type
Type all   (ty → ty) → ty.    % forall-quantifier
```

```
Type arrow ty → ty → ty.     % function type
Type bound ty → o.           % variable collection in contexts

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

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

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

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

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

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

5.1 Statistics and Discussion

Some basic statistics on our proof script are shown in Figure 7. The proof consists of 3627 lines of code with a total of 33 definitions and 267 theorems. We have to mention that Abella provides few built-in tactics and does not support user-defined ones, and we would reduce significant lines of code if Abella provided more handy tactics. Moreover, the definition of natural numbers, the plus

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

Fig. 7. Statistics for the proof scripts

operation and less-than relation are defined within our proof due to Abella's lack of packages. However, the way Abella deals with name bindings is very helpful for type system formalizations and substitution-intensive formalizations, such as this one.

6 Related Work

Type Inference for Polymorphic Subtyping Higher-order polymorphism is a practical and important programming language feature. Due to the undecidability of type-inference for System F [35], different decidable partial type-inference approaches were developed. The subtyping relation of this paper, originally proposed by Odersky and Laüfer [26], is *predicative* ($\forall$'s only instantiate to monotypes), which is considered a reasonable and practical trade-off. There is also work on partial impredicative type-inference algorithms [19, 20, 34]. However, unlike the predicative subtyping relation for System F, the subtyping for impredicative System F is undecidable [31]. Therefore such algorithms have to navigate through the design space to impose restrictions that allow for a decidable algorithm. As a result such algorithms tend to be more complex, and are less adopted in practice.

Gundry et al. [13] revisited the Hindley-Milner type system. They make use of ordered contexts on the unification during type inference, and their algorithm works differently from algorithm $\mathcal{W}$. Dunfield and Krishnaswami [9] adopted a similar idea on ordered contexts and presented an algorithmic approach for predicative polymorphic subtyping that tracks the (partial) solutions of existential variables in the algorithmic context—this denotes a delayed substitution that is incrementally applied to outstanding work as it is encountered. Their algorithm comes with 40 pages of manual proofs on the soundness, completeness and decidability. We have tried to mechanize these proofs directly, but have not been successful yet because most proof assistants do not naturally support output contexts and their more complex ordered contexts. Their theorems have statements that are more complex than those in the worklist approach. One of the reasons for the added complexity is that, when the constraints are not strict

enough, the algorithm may not instantiate all existential variables. However in order to match the declarative judgement, all the unsolved variables should be properly assigned. For example, their generalized completeness theorem is:

Theorem 4 (Generalized Completeness of Subtyping [9]).
If $\Psi \longrightarrow \Phi$ *and* $\Psi \vdash A$ *and* $\Psi \vdash B$ *and* $[\Phi]\Psi \vdash [\Phi]A \leq [\Phi]B$ *then there exist* Δ *and* Φ' *such that* $\Delta \longrightarrow \Phi'$ *and* $\Phi \longrightarrow \Phi'$ *and* $\Psi \vdash [\Psi]A <: [\Psi]B \dashv \Delta$.

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

While we are faced with a similar gap between algorithm and specification, which we tackle with our transfer relations $\Gamma \to \Psi$, our completeness statement is much shorter because our algorithm does not return an output context which needs to be transferred. Moreover, we have cleanly encapsulated any substitutions to the worklist in the worklist transfer judgement $\Gamma \mid \Omega \rightsquigarrow \Omega'$.

Peyton Jones et al. [16] developed a higher-rank predicative bidirectional type system. They enriched their subtyping relations with deep skolemisation, while other relations remain similar to ours. Their algorithm is unification-based with a structure similar to algorithm $\mathcal{W}$'s.

Unification Algorithms Our algorithm works similarly to some unification algorithms that use a set of unification constraints and single-step simplification transitions. Some work [27, 1] adopts this idea in dependently typed inference and reconstruction. These approaches collect a set of constraints and nondeterministically process one of them at a time. Those approaches consider various forms of constraints, including term unification, context unification and solution for meta-variables. In contrast, our algorithm is presented in a simpler form, using ordered (worklist) judgements, which is sufficient for the subtyping problem.

Formalizations of Type-Inference Algorithms in Theorem Provers The well-known POPLMark challenge [3] has encouraged the development of new proof assistant features for facilitating the development and verification of type systems. As a result, many theorem provers and packages now provide methods for dealing with variable binding [2, 32, 4], and more and more type system designers choose to formalize their proofs with these tools. Yet, difficulties with mechanising algorithmic aspects, like unification and constraint solving, have received very little attention. Moreover, while most type system judgements only feature local (input) contexts, which have a simple binding/scoping structure, many traditional type-inference algorithms require more complex binding structures with output contexts.

Naraschewski and Nipkow [24] published the first formal verification of algorithm $\mathcal{W}$ in Isabelle/HOL [25]. The treatment of new variables is a little tricky in their formalization, while most other parts follow the structure of Damas's manual proof closely. Following Naraschewski and Nipkow other researchers [7, 8] prove a similar result in Coq [29]. Nominal techniques [32] in Isabelle/HOL have

also been used for a similar verification [33]. Moreover, Garrigue [11] mechanized a type inference algorithm for Core ML extended with structural polymorphism and recursion.

7 Conclusion and Future Work

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

Acknowledgement

We sincerely thank the anonymous reviewers for their insightful comments. This work has been sponsored by the Hong Kong Research Grant Council projects number 17210617 and 17258816, and by the Research Foundation - Flanders.

References

1. Abel, A., Pientka, B.: Higher-order dynamic pattern unification for dependent types and records. In: International Conference on Typed Lambda Calculi and Applications. pp. 10–26. Springer (2011)
2. Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '08 (2008)
3. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The POPLmark challenge. In: The 18th International Conference on Theorem Proving in Higher Order Logics (2005)
4. Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. ICFP '08 (2008)
5. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '82 (1982)
6. Disciple Development Team: The Disciplined Disciple Compiler (2017), http://disciple.ouroborus.net/
7. Dubois, C.: Proving ML type soundness within Coq. Theorem Proving in Higher Order Logics pp. 126–144 (2000)
8. Dubois, C., Menissier-Morain, V.: Certification of a type inference tool for ML: Damas–Milner within Coq. Journal of Automated Reasoning **23**(3), 319–346 (1999)

9. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. ICFP '13 (2013)
10. Gacek, A.: The Abella interactive theorem prover (system description). In: Proceedings of IJCAR 2008. Lecture Notes in Artificial Intelligence (2008)
11. Garrigue, J.: A certified implementation of ML with structural polymorphism and recursive types. Mathematical Structures in Computer Science **25**(4), 867–891 (2015)
12. Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture. FPCA '93 (1993)
13. Gundry, A., McBride, C., McKinna, J.: Type inference in context. In: Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming. MSFP '10 (2010)
14. Hindley, R.: The principal type-scheme of an object in combinatory logic. Transactions of the american mathematical society **146**, 29–60 (1969)
15. Jones, M.P.: Functional programming with overloading and higher-order polymorphism. In: Advanced Functional Programming. Lecture Notes in Computer Science 925 (1995)
16. Jones, S.P., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. Journal of functional programming **17**(1), 1–82 (2007)
17. Lämmel, R., Jones, S.P.: Scrap your boilerplate: A practical design pattern for generic programming. In: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation. TLDI '03 (2003)
18. Launchbury, J., Peyton Jones, S.L.: State in Haskell. LISP and Symbolic Computation **8**(4), 293–341 (1995)
19. Le Botlan, D., Rémy, D.: MLF: Raising ML to the power of system F. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming. ICFP '03 (2003)
20. Leijen, D.: HMF: Simple type inference for first-class polymorphism. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. ICFP '08 (2008)
21. Leroy, X., et al.: The CompCert verified compiler. Documentation and users manual. INRIA Paris-Rocquencourt (2012)
22. Miller, D.: Abstract syntax for variable binders: An overview. In: CL 2000: Computational Logic. Lecture Notes in Artificial Intelligence (2000)
23. Milner, R.: A theory of type polymorphism in programming. Journal of computer and system sciences **17**(3), 348–375 (1978)
24. Naraschewski, W., Nipkow, T.: Type inference verified: Algorithm W in Isabelle/HOL. Journal of Automated Reasoning **23**(3), 299–318 (1999)
25. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
26. Odersky, M., Läufer, K.: Putting type annotations to work. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '96 (1996)
27. Reed, J.: 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 (2009)
28. Reynolds, J.C.: Types, abstraction and parametric polymorphism. Information Processing pp. 513–523 (1983)

29. The Coq development team: The Coq proof assistant (2017), https://coq.inria.fr/
30. The PureScript development team: PureScript (2017), http://www.purescript.org/
31. Tiuryn, J., Urzyczyn, P.: The subtyping problem for second-order types is undecidable. In: Proceedings 11th Annual IEEE Symposium on Logic in Computer Science (1996)
32. Urban, C.: Nominal techniques in Isabelle/HOL. Journal of Automated Reasoning **40**(4), 327–356 (2008)
33. Urban, C., Nipkow, T.: Nominal verification of algorithm W. From Semantics to Computer Science. Essays in Honour of Gilles Kahn pp. 363–382 (2008)
34. Vytiniotis, D., Weirich, S., Peyton Jones, S.: FPH: First-class polymorphism for Haskell. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. ICFP '08 (2008)
35. Wells, J.B.: Typability and type checking in system F are equivalent and undecidable. Annals of Pure and Applied Logic **98**(1-3), 111–156 (1999)