Formalized Higher-Ranked Polymorphic Type Inference Algorithms

by

Jinxu Zhao
(赵锦煦)

A thesis submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy
at The University of Hong Kong

February 2021

Abstract of thesis entitled

"Formalized Higher-Ranked Polymorphic Type Inference Algorithms"

Submitted by

Jinxu Zhao

for the degree of Doctor of Philosophy

at The University of Hong Kong

in February 2021

Declaration

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

.....

Jinxu Zhao

February 2021

Acknowledgments

Contents

List of Figures

List of Tables

Part I

Prologue

1 Introduction

"predicative implicit higher-rank polymorphism"

1.1 Contributions

In summary the contributions of this thesis are:

?? •

1.2 Organization

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

??:

1.3 Introduction

Most statically typed functional languages support a form of *(implicit) parametric polymorphism* Reynolds [1983]. Traditionally, functional languages have employed variants of the Hindley-Milner Damas and Milner [1982]; Hindley [1969]; Milner [1978] type system, which supports full type-inference without any type annotations. However the Hindley-Milner type system only supports *first-order polymorphism*, where all universal quantifiers only occur at the top-level of a type. Modern functional programming languages such as Haskell go beyond Hindley-Milner and support *higher-order polymorphism*. With higher-order polymorphism there is no restriction on where universal quantifiers can occur. This enables more code reuse and more expressions to type-check, and has numerous applications Gill et al. [1993]; Jones [1995]; Lämmel and Jones [2003]; Launchbury and Peyton Jones [1995].

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

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

The function `hpoly` cannot be type-checked in Hindley-Milner. The type of `hpoly` is `(forall a. a -> a) -> (Int, 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 Peyton Jones et al. [2007] preserves many of the desirable properties of Hindley-Milner. Like in Hindley-Milner type instantiation is *implicit*. That is, calling a polymorphic function never requires the programmer to provide the instantiations of the type parameters.

Central to type-inference with *higher-order polymorphism* is an algorithm for polymorphic subtyping. This algorithm allows us to check whether one type is more general than another, which is essential to detect valid instantiations of a polymorphic type. For example, the type `forall a. a -> a` is more general than `Int -> Int`. A simple declarative specification for polymorphic subtyping was proposed by Odersky and Läufer Odersky and Läufer [1996]. Since then several algorithms have been proposed that implement it. Most notably, the algorithm proposed by Peyton Jones et al. Peyton Jones et al. [2007] forms the basis for the implementation of type inference in the GHC compiler. Dunfield and Krishnaswami Dunfield and Krishnaswami [2013] provided a very elegant formalization of another sound and complete algorithm, which has also inspired implementations of type-inference in some polymorphic programming languages (such as PureScript Freeman [2017] or DDC Disciple Development Team [2017]).

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

cess required by many algorithms (through unification-like or constraint-based approaches), is quite subtle and tricky to get right. Moreover, extending type-inference algorithms with new programming language features is often quite delicate. Studying the meta-theory for such extensions would be greatly aided by the existence of a mechanical formalization of the base language, which could then be extended by the language designer.

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

This paper advances the state-of-the-art by formalizing an algorithm for polymorphic subtyping in the Abella theorem prover. We hope that this work encourages other researchers to use theorem provers for formalizing type-inference algorithms. In particular, we show that the problem we have identified above can be overcome by means of *worklist judgments*. These are a form of judgement that turns the complicated global propagation of unifications into a simple local substitution. Moreover, we exploit several ideas in the recent inductive formulation of a type-inference algorithm by Dunfield and Krishnaswami Dunfield and Krishnaswami [2013], which turn out to be useful for mechanisation in a theorem prover.

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

In summary the contributions of this paper are:

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

1.4 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-

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

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

As functional languages evolved the need for more expressive power has motivated language designers to look beyond Hindley-Milner. In particular one popular direction is to allow *higher-ranked polymorphism* where polymorphic types can occur anywhere in a type signature. An important challenge is that full type inference for higher-ranked polymorphism is known to be undecidable Wells [1999]. Therefore some type annotations are necessary to guide type inference. In response to this challenge several decidable type systems requiring some annotations have been proposed Dunfield and Krishnaswami [2013]; Le Botlan and Rémy [2003]; Leijen [2008]; Peyton Jones et al. [2007]; Serrano et al. [2018]; Vytiniotis et al. [2008]. Two closely related type systems that support *predicative* higher-ranked type inference were proposed by Peyton Jones et al. Peyton Jones et al. [2007] and Dunfield and Krishnaswami Dunfield and Krishnaswami [2013] (henceforth denoted as DK). These type systems are popular among language designers and their ideas have been adopted by several modern functional languages, including Haskell, PureScript Freeman [2017] and Unison Chiusano and Bjarnason [2015] among others. In those type systems type annotations are required for polymorphic arguments of functions, but other type annotations can be omitted. A canonical example (here written in Haskell) is:

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

The function `hpoly` cannot be type-checked in the Hindley-Milner type system. The type of `hpoly` is the rank-2 type: `(forall a. a -> a) -> (Int, Char)`. Notably (and unlike Hindley-Milner) the lambda argument `f` requires a *polymorphic* type annotation. This annotation is needed because the single universal quantifier does not appear at the top-level. Instead it is used to quantify a type variable `a` used in the first argument of the function. Despite these additional annotations, Peyton Jones et al. and DK's type inference algorithms preserve many of the desirable properties of Hindley-Milner. For example the applications of `f` implicitly instantiate the polymorphic type arguments of `f`.

Although type inference is important in practice and receives a lot of attention in academic research, there is little work on mechanically formalizing such advanced forms of type inference in theorem provers. The remarkable exception is work done on the formalization of certain parts of Hindley-Milner type inference Dubois [2000]; Dubois and

Menissier-Morain [1999]; Garrigue [2015]; Naraschewski and Nipkow [1999]; Urban and Nipkow [2008]. However there is still no formalization of the higher-ranked type systems that are employed by modern languages like Haskell. This is at odds with the current trend of mechanical formalizations in programming language research. In particular both the POPLMark challenge Aydemir et al. [2005] and CompCert Leroy et al. [2012] have significantly promoted the use of theorem provers to model various aspects of programming languages. Today papers in various programming language venues routinely use theorem provers to mechanically formalize: *dynamic and static semantics* and their correctness properties Aydemir et al. [2008], *compiler correctness* Leroy et al. [2012], *correctness of optimizations* Bertot et al. [2006], *program analysis* Chang et al. [2006] or proofs involving *logical relations* Abel et al. [2018]. The main argument for mechanical formalizations is a simple one. Proofs for programming languages tend to be *long*, *tedious* and *error-prone*. In such proofs it is very easy to make mistakes that may invalidate the whole development. Furthermore, readers and reviewers often do not have time to look at the proofs carefully to check their correctness. Therefore errors can go unnoticed for a long time. Mechanical formalizations provide, in principle, a natural solution for these problems. Theorem provers can automatically check and validate the proofs, which removes the burden of checking from both the person doing the proofs as well as readers or reviewers.

This paper presents the first fully mechanized formalization of the metatheory for higher-ranked polymorphic type inference. The system that we formalize is the bidirectional type system by Dunfield and Krishnaswami [2013]. We chose DK's type system because it is quite elegant, well-documented and it comes with detailed manually written proofs. Furthermore the system is adopted in practice by a few real implementations of functional languages, including PureScript and Unison. The DK type system has two variants: a declarative and an algorithmic one. The two variants have been *manually* proved to be *sound*, *complete* and *decidable*. We present a mechanical formalization in the Abella theorem prover Gacek [2008] for DK's declarative type system using a different algorithm. While our initial goal was to formalize both DK's declarative and algorithmic versions, we faced technical challenges with the latter, prompting us to find an alternative formulation.

The first challenge that we faced were missing details as well as a few incorrect proofs and lemmas in DK's formalization. While DK's original formalization comes with very well written manual proofs, there are still several details missing. These complicate the task of writing a mechanically verified proof. Moreover some proofs and lemmas are wrong and, in some cases, it is not clear to us how to fix them. Despite the problems in DK's manual formalization, we believe that these problems do not invalidate their work and that their results are still true. In fact we have nothing but praise for their detailed and clearly written metatheory and proofs, which provided invaluable help to our own work. We expect that

for most non-trivial manual proofs similar problems exist, so this should not be understood as a sign of sloppiness on their part. Instead it should be an indicator that reinforces the arguments for mechanical formalizations: manual formalizations are error-prone due to the multiple tedious details involved in them. There are several other examples of manual formalizations that were found to have similar problems. For example, Klein et al. Klein et al. [2012] mechanized formalizations in Redex for nine ICFP 2009 papers and all were found to have mistakes.

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

Due to those two challenges, our work takes a different approach by refining and extending the idea of *worklist judgments* Zhao et al. [2018], proposed recently to mechanically formalize an algorithm for *polymorphic subtyping* Odersky and Läufer [1996]. A key innovation in our work is how to adapt the idea of worklist judgments to *inference judgments*, which are not needed for polymorphic subtyping, but are necessary for type-inference. The idea is to use a *continuation passing style* to enable the transfer of inferred information across judgments. A further refinement to the idea of worklist judgments is the *unification between ordered contexts Dunfield and Krishnaswami [2013]; Gundry et al. [2010] and worklists*. This enables precise scope tracking of free variables in judgments. Furthermore it avoids the duplication of context information across judgments in worklists that occurs in other techniques Abel and Pientka [2011]; Reed [2009]. Despite the use of a different algorithm we prove the same results as DK, although with significantly different proofs and proof techniques. The calculus and its metatheory have been fully formalized in the Abella theorem prover Gacek [2008].

In summary, the contributions of this paper are:

- **A fully mechanized formalization of type inference with higher-ranked types:** Our work presents the first fully mechanized formalization for type inference of higher ranked types. The formalization is done in the Abella theorem prover Gacek [2008] and it is available online at `https://github.com/JimmyZJX/TypingFormalization`.
- **A new algorithm for DK's type system:** Our work proposes a novel algorithm that implements DK's declarative bidirectional type system. We prove *soundness*, *completeness* and *decidability*.

- **Worklists with inference judgments:** One technical contribution is the support for inference judgments using worklists. The idea is to use a continuation passing style to enable the transfer of inferred information across judgments.
- **Unification of worklists and contexts:** Another technical contribution is the unification between ordered contexts and worklists. This enables precise scope tracking of variables in judgments, and avoids the duplication of context information across judgments in worklists.

2 Background

2.1 Hindley-Milner Type System

2.1.1 Declarative System

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

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

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

For example,

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

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

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

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

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

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

Figure 2.1: HM Syntax

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

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

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

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

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

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

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

Figure 2.2: HM Type System

must be the same as the input type of the function, and the type of the whole application is τ_2.

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

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

2.1.2 Algorithmic System and Principality

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

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

A generalization on σ, the synthesized type of e_1, is added to rule `HM-Let`, since it is the source place where a polymorphic type is generated. However, a too generalized type might reject applications due to its shape, therefore, an instantiation procedure is added to eliminate all the universal quantifiers on rule `HM-Var`. We omit rules `HM-Unit`, `HM-Abs`, and `HM-App` for the syntax-directed system $\Psi \vdash^{S}_{HM}$. The following property shows that the new system is (almost) equivalent to the original declarative system.

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

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

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

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

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

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

Figure 2.3: Syntax of Odersky-Läufer System

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

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

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

2.2 Odersky-Läufer Bidirectional Type System

2.2.1 Higher-Ranked Types

2.2.2 Declarative System

2.2.3 Relating to HM

2.3 Dunfield's Bidirectional Type System

2.4 MLSub

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

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

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

Figure 2.5: Subtyping of the Odersky-Läufer System

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

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

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

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

Figure 2.6: Typing of the Odersky-Läufer System

Bibliography

[Citing pages are listed after each reference.]

Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2018. POPLMark Reloaded: Mechanizing Proofs by Logical Relations. *Submitted to the Journal of functional programming* (2018). [cited on page 6]

Andreas Abel and Brigitte Pientka. 2011. Higher-order dynamic pattern unification for dependent types and records. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 10–26. [cited on page 7]

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. [cited on page 6]

Brian E Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized metatheory for the masses: The POPLmark challenge. In *The 18th International Conference on Theorem Proving in Higher Order Logics*. [cited on pages 3 and 6]

Yves Bertot, Benjamin Grégoire, and Xavier Leroy. 2006. A Structured Approach to Proving Compiler Optimizations Based on Dataflow Analysis. In *Proceedings of the 2004 International Conference on Types for Proofs and Programs (TYPES'04)*. [cited on page 6]

Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula. 2006. A Framework for Certified Program Analysis and Its Applications to Mobile-code Safety. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*. [cited on page 6]

Paul Chiusano and Runar Bjarnason. 2015. Unison. `http://unisonweb.org` [cited on page 5]

Luis Damas and Robin Milner. 1982. Principal Type-schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*. [cited on pages 2 and 5]

Disciple Development Team. 2017. The Disciplined Disciple Compiler. `http://disciple.ouroborus.net/` [cited on page 3]

Catherine Dubois. 2000. Proving ML type soundness within Coq. *Theorem Proving in Higher Order Logics* (2000), 126–144. [cited on pages 3 and 5]

Catherine Dubois and Valerie Menissier-Morain. 1999. Certification of a type inference tool for ML: Damas–Milner within Coq. *Journal of Automated Reasoning* 23, 3 (1999), 319–346. [cited on pages 3 and 5]

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-rank Polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. [cited on pages 3, 4, 5, 6, and 7]

Phil Freeman. 2017. PureScript. `http://www.purescript.org/` [cited on pages 3 and 5]

Andrew Gacek. 2008. The Abella Interactive Theorem Prover (System Description). In *Proceedings of IJCAR 2008 (Lecture Notes in Artificial Intelligence)*. [cited on pages 6 and 7]

Jacques Garrigue. 2015. A certified implementation of ML with structural polymorphism and recursive types. *Mathematical Structures in Computer Science* 25, 4 (2015), 867–891. [cited on pages 3 and 6]

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. [cited on page 2]

Adam Gundry, Conor McBride, and James McKinna. 2010. Type Inference in Context. In *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming (MSFP '10)*. [cited on page 7]

Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society* 146 (1969), 29–60. [cited on pages 2 and 5]

Mark P. Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming (Lecture Notes in Computer Science 925)*. [cited on page 2]

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run Your Research: On the Effectiveness of Lightweight Mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. 285–296. [cited on page 7]

Ralf Lämmel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '03)*. [cited on page 2]

John Launchbury and Simon L. Peyton Jones. 1995. State in Haskell. *LISP and Symbolic Computation* 8, 4 (1995), 293–341. [cited on page 2]

Didier Le Botlan and Didier Rémy. 2003. MLF: Raising ML to the Power of System F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. [cited on page 5]

Daan Leijen. 2008. HMF: Simple Type Inference for First-class Polymorphism. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. [cited on page 5]

Xavier Leroy et al. 2012. The CompCert verified compiler. *Documentation and user' s manual. INRIA Paris-Rocquencourt* (2012). [cited on pages 3 and 6]

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375. [cited on pages 2, 5, and 11]

Wolfgang Naraschewski and Tobias Nipkow. 1999. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning* 23, 3 (1999), 299–318. [cited on pages 3 and 6]

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. [cited on pages 3 and 7]

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of functional programming* 17, 1 (2007), 1–82. [cited on pages 3 and 5]

Jason Reed. 2009. Higher-order Constraint Simplification in Dependent Type Theory. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP '09)*. [cited on page 7]

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Proceedings of the IFIP 9th World Computer Congress*. [cited on pages 2 and 5]

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded Impredicative Polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. [cited on page 5]

Christian Urban and Tobias Nipkow. 2008. Nominal verification of algorithm W. *From Semantics to Computer Science. Essays in Honour of Gilles Kahn* (2008), 363–382. [cited on pages 3 and 6]

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: First-class Polymorphism for Haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. [cited on page 5]

Joe B Wells. 1999. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic* 98, 1-3 (1999), 111–156. [cited on pages 2 and 5]

Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. Formalization of a Polymorphic Subtyping Algorithm. In *ITP (Lecture Notes in Computer Science, Vol. 10895)*. Springer, 604–622. [cited on page 7]

Part II

Technical Appendix

A Proof Experience with Abella