No Unification Variable Left Behind: Fully Grounding Type Inference for the HDM System

Roger Bosman
KU Leuven, Belgium

Georgios Karachalias
Tweag, Paris, France

Tom Schrijvers
KU Leuven, Belgium

Abstract

The Hindley-Damas-Milner (HDM) system provides polymorphism, a key feature of functional programming languages such as Haskell and OCaml. It does so through a type inference algorithm, whose soundness and completeness have been well-studied and proven both manually (on paper) and mechanically (in a proof assistant). Earlier research has focused on the problem of inferring the type of a top-level expression. Yet, in practice, we also may wish to infer the type of subexpressions, either for the sake of elaboration into an explicitly-typed target language, or for reporting those types back to the programmer. One key difference between these two problems is the treatment of underconstrained types: in the former, unification variables that do not affect the overall type need not be instantiated. However, in the latter, instantiating all unification variables is essential, because unification variables are internal to the algorithm and should not leak into the output.

We present an algorithm for the HDM system that explicitly tracks *the scope* of all unification variables. In addition to solving the *subexpression type reconstruction* problem described above, it can be used as a basis for elaboration algorithms, including those that implement elaboration-based features such as type classes. The algorithm implements input and output contexts, as well as the novel concept of *full contexts*, which significantly simplifies the state-passing of traditional algorithms. The algorithm has been formalised and proven sound and complete using the Coq proof assistant.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Software and its engineering → Correctness

Keywords and phrases type inference, mechanization, let-polymorphism

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.8

Supplementary Material *Software (Source Code)*: `github.com/rogerbosman/hdm-fully-grounding` archived at `swh:1:dir:32c6b22d2de66001bf4c0ab5e255481de6561daa`

Funding This work was partly funded by KU Leuven project C14/20/079#55685055.

Acknowledgements We would like to thank Steven Keuchel for their help and insights about Coq, and their comments about a draft of this paper.

1 Introduction

Classic unification-based type inference algorithms for the Hindley–Damas–Milner (HDM) system such as algorithm $\mathcal{W}$ [7] solve the *type inference problem*. That is, they determine whether programs that lack type signatures are well-typed or not, by assigning every subterm the most general type possible (an unconstrained unification variable) and solving any type constraints that arise. Programs are well-typed if and only if all constraints can be solved.

However, depending on the setting, we would like to not only verify that a program is well-typed but also determine the type of every subterm. The canonical example of this is elaboration to System F [13, 19], but the problem arises in other settings as well. For

example, to aid development, real-world implementations of programming languages often allow developers to query the types of subterms, either via a REPL[1] like GHCi [22] or in a GUI-based editor, for example by supporting [24, 17] the Language Server Protocol [4]. We name this the *subterm type reconstruction problem*.

An important way the type inference and subterm type reconstruction problem differ is in how they treat underconstrained types (i.e. types with unconstrained parts). Consider the following program below.

$$\textbf{let } x = (\lambda f.\, \texttt{unit})\ (\lambda y.\, y)\ \textbf{in } \ldots$$

Observe that the type of y is not subject to any constraints: since $\lambda y.y$ is passed to a function that discards its argument and instead returns `unit`, it is never applied an argument, nor is its output used, either which would impose constraints. For type checking this unconstrained type is not a problem: the program is well-typed regardless of y's type. However, this situation is problematic for subterm type reconstruction, because we need to provide types for both f and y. We may only output *fully ground* types; unification variables are internal to the algorithm and should not be returned. Thus, to ground these types, we must instantiate all remaining unification variables. Generally, there are two options: (1) to generalise over the remaining variables, or (2) to default them to an arbitrary type (e.g. `Unit`).

(1) $\textbf{let } x = \Lambda a.\, (\lambda f : a \to a.\, \texttt{unit})\ (\lambda y : a.\, y)\ \textbf{in } \ldots$

(2) $\textbf{let } x = (\lambda f : \texttt{Unit} \to \texttt{Unit}.\, \texttt{unit})\ (\lambda y : \texttt{Unit}.\, y)\ \textbf{in } \ldots$

Crucially, the type of the overall expression may not determine the instantiation, as type variables may not occur in this type. Consider again the example above. Since $(\lambda f.\, \texttt{unit})\ (\lambda y.\, y)$ beta-reduces to `unit`, x's type is `Unit`. Hence, the type of y does not occur in x's type. Therefore, additional machinery is needed to keep track of unsolved unification variables and apply whichever *grounding strategy* has been chosen. While solutions to this problem are not necessarily complicated in practice, implementations are often ad hoc, making reasoning about their correctness hard.

In this paper, we address this very issue. We present algorithm $\mathcal{R}$, a *fully grounding* type inference algorithm for the HDM system. The algorithm explicitly tracks the scope of unification variables, which allows for fully grounding type inference, meaning we can infer fully ground types for all subexpressions. Since type grounding is internal to algorithm $\mathcal{R}$, its correctness proof (which we have mechanised in the Coq proof assistant [23]) carries over to the grounding strategy as well. As far as we know, we are the first to mechanically formalize a type inference algorithm for the HDM system that includes type grounding.

The algorithm utilizes in- and output contexts in the style of Dunfield and Krishnaswami [10] as well as a novel approach to unification, using a concept we dub *full contexts*. Here, contexts always contain all existing unification variables. Traditionally, inference algorithm thread through a substitution to reflect equalities found during unification in other branches of the derivation. With our approach, we avoid this threading: when an equality $\alpha := \tau$ is found, α can immediately be substituted for τ in the current context. Since the context is full, no further occurrences of α exist, and the equality can be discharged in one go.

In summary, the specific contributions of this paper are:

- This paper presents a new, *fully grounding* type inference algorithm $\mathcal{R}$ for ML-style polymorphism. The algorithm keeps track of all unification variables and their scope and uses the novel concept of full contexts to discharge all unifications in one go.

[1] Read–Eval–Print Loop

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathcal{W}} e_1 : \tau, \theta_1 \\ \theta_1\Gamma \vdash_{\mathcal{W}} e_2 : \tau_2, \theta_2 \\ a \# \theta_2\theta_1\Gamma \\ \theta_3 = \mathsf{unify}(\theta_2\tau \sim \tau_2 \to a)\end{array}}{\Gamma \vdash_{\mathcal{W}} e_1 e_2 : \theta_3 a, \theta_3\theta_2\theta_1}\ \mathcal{W}\text{-App} \qquad \frac{\Gamma \vdash_{\mathcal{W}} e : \tau, \theta \quad \overline{a} = fv(\tau) \setminus fv(\theta\Gamma)}{\Gamma \vdash_{\mathcal{W}} e : \forall \overline{a}.\tau, \theta}\ \mathcal{W}\text{-Gen}$$

$$\frac{\begin{array}{c}\Psi_{in} \vdash e_1 : [A_1]T \dashv \Psi_1 \\ \Psi_1; \{[A_1]T\} \vdash e_2 : [A_2]T_1 \dashv \Psi_2; \{[A_1']T'\} \\ \widehat{\alpha} \# \Psi_2; (A_1', A_2) \\ \Psi_2; (A_1', A_2, \widehat{\alpha}); \{\widehat{\alpha}\} \vdash T' \sim T_1 \to \widehat{\alpha} \dashv \Psi_{out}; A_{out}; \{T_{out}\}\end{array}}{\Psi_{in} \vdash e_1\ e_2 : [A_{out}]T_{out} \dashv \Psi_{out}}\ \textsc{App} \qquad \frac{\Psi_{in} \vdash e : [A]T \dashv \Psi_{out} \quad \mathsf{gen}(T, A) = S}{\Psi_{in} \vdash e : S \dashv \Psi_{out}}\ \textsc{Gen}$$

Figure 1 Application and generalisation of algorithm $\mathcal{W}$ (top) and algorithm $\mathcal{R}$ (bottom).

- We have mechanised both algorithm $\mathcal{R}$ as well as its correctness proof in the Coq proof assistant. Since algorithm $\mathcal{R}$ is fully grounding, we are – to our knowledge – the first to mechanically prove the correctness of an inference algorithm that features grounding. We admit an axiom about unification (see Section 5.3) and about the declarative specification (see Section 6.3).

2 Overview

This section describes the difference between unification-based algorithms like algorithm $\mathcal{W}$ and our algorithm $\mathcal{R}$. We first describe how algorithm $\mathcal{W}$ loses track of unconstrained type variables. We then propose our algorithm $\mathcal{R}$, which explicitly tracks the scope of unification variables, and show how this information yields fully grounding type inference.

Algorithm $\mathcal{W}$

Unification-based algorithms like algorithm $\mathcal{W}$ derive equality constraints at application sites $e_1\ e_2$. Rule $\mathcal{W}$-App of Figure 1 describes algorithm $\mathcal{W}$ in the case of applications.

Let us apply this to the example $(\lambda f.\ \texttt{unit})\ (\lambda y.\ y)$ (shown in Section 1) under an empty context. First, we infer the type $a_1 \to \texttt{Unit}$ for $\lambda f.\ \texttt{unit}$. Then, we infer the type $a_2 \to a_2$ for $(\lambda y.\ y)$. Both steps result in empty unifiers θ_1, θ_2. Then, with a_3 fresh, we unify $a_1 \to \texttt{Unit}$ with $(a_2 \to a_2) \to a_3$, yielding $\theta = (a_3 := \texttt{Unit}, a_1 := a_2 \to a_2)$. Finally, we return $\theta(a_3)$, which equates to `Unit`. Since algorithm $\mathcal{W}$ only returns the function's result type `Unit`, it loses track of free variables that only occur in the parameter's type (i.e. a_2). As a_2 is no longer reachable, it will not be further constrained and will remain unsolved.

Algorithm $\mathcal{W}$'s generalisation logic, extracted as $\mathcal{W}$-Gen[2] in Figure 1, turns an expression's monotype into a type scheme. In our running example, since the monotype `Unit` does not contain any free variables, algorithm $\mathcal{W}$ generalises over the empty list, which simply yields the `Unit` type scheme. Observe in particular that the unsolved unification variable a_2 is not generalised over. Hence, the type for $\lambda y.\ y$ remains $a_2 \to a_2$, but we do not know in which context a_2 is defined, and whether or where it can be generalised.

[2] Normally, this logic would be incorporated as part of the rule for let expressions.

x	Variables		a	Skolem type variables
Terms	e	$::=$	$x \mid \texttt{unit} \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2$	
Monotypes	τ	$::=$	$a \mid \texttt{Unit} \mid \tau_1 \to \tau_2$	
Type Schemes	σ	$::=$	$\tau \mid \forall a.\sigma$	
Scoping/Typing Context	Γ	$::=$	$\bullet \mid \Gamma; a \mid \Gamma; x : \sigma$	

Figure 2 Syntax of the Declarative Specification.

Algorithm $\mathcal{R}$

Our algorithm solves this problem by not only inferring the type T of an expression but also a list of unification variables that are in scope for T. By "in scope" we mean those variables that are safe to generalise over. Note that this list need not be a superset or subset of the free unification variables of T. We denote unification variables as $\widehat{\alpha}$ with A denoting a list of $\widehat{\alpha}$. Furthermore, we use $[A]T$ to denote the type T having A in scope.

Algorithm $\mathcal{R}$ utilises in- and output contexts [10] as well as the notion of *full contexts* to avoid having to pass around unifiers θ. We postpone fully introducing algorithm $\mathcal{R}$ to Section 4.2. For now, we present an informal preview of the application of algorithm $\mathcal{R}$ to the same example as covered above, highlighting how algorithm $\mathcal{R}$ infers fully ground types, and showing the benefit of full contexts.

For the application $(\lambda f.\ \texttt{unit})\ (\lambda y.\ y)$, in App, we first derive the type $[\widehat{\alpha}_1](\widehat{\alpha}_1 \to \texttt{Unit})$ for $\lambda f.\ \texttt{unit}$. Then, we infer the type $[\widehat{\alpha}_2](\widehat{\alpha}_2 \to \widehat{\alpha}_2)$ for $\lambda y.\ y$. Here our notion of full contexts comes in: instead of deriving a unifier that needs to be applied to the type of $\lambda f.\ \texttt{unit}$, we instead append $[\widehat{\alpha}_1](\widehat{\alpha}_1 \to \texttt{Unit})$ to the input context of the inference on $\lambda y.\ y$, and obtain a possibly further instantiated $[A_1']T'$ from the output context (as seen in rule App in Figure 1). Here, $[A_1']T' = [\widehat{\alpha}_1](\widehat{\alpha}_1 \to \texttt{Unit})$.

With $\widehat{\alpha}_3$ fresh, we unify $[\widehat{\alpha}_1](\widehat{\alpha}_1 \to \texttt{Unit})$ with $[\widehat{\alpha}_3, \widehat{\alpha}_2]((\widehat{\alpha}_2 \to \widehat{\alpha}_2) \to \widehat{\alpha}_3)$. Again, we apply our notion of full contexts, appending all variables in scope for our to-be-unified types ($[\widehat{\alpha}_3, \widehat{\alpha}_1, \widehat{\alpha}_2]$) to the context, allowing us to retrieve a possibly further instantiated A_{out} from the output context. Here, $A_{out} = [\widehat{\alpha}_2]$. Furthermore, we append $\widehat{\alpha}_3$ once more, now occurring as a type instead of an in-scope unification variable. Since $\widehat{\alpha}_3$ enjoys any substitution occurring during unification, we obtain the possibly further instantiated T_{out} from the output context. Here, $T_{out} = \texttt{Unit}$. Finally, we return $[\widehat{\alpha}_2]\texttt{Unit}$. Observe that, even though we are dropping the argument type, we are **not dropping the variables in scope of the argument type**. Generalisation, as displayed in Gen, of type $[\widehat{\alpha}_2]\texttt{Unit}$ is (almost) trivial.

To conclude this section, we have shown that our algorithm $\mathcal{R}$ not only infers a type T, but also a list of type variables A in scope for T. This way it can infer a fully ground type for every subterm. The following sections formally introduce algorithm $\mathcal{R}$.

3 Declarative System

Before we present our algorithm, we present the declarative system that serves as its specification. The declarative system is essentially the syntax-directed system of Clement et al. [6], with two changes. First, like System F [13, 19], we explicitly track type variables in an ordered context. Consequently, we only generalise over variables that occur at the end of the context (i.e., not occurring to the left of term variable bindings). The second change is a purely syntactic one: we have extracted generalisation into a separate judgment.

$\boxed{\Gamma \Vdash_{\texttt{mono}} e : \tau}$ $\boxed{\Gamma \Vdash_{\texttt{poly}} e : \sigma}$ Term Typing

$$\frac{(x : \sigma) \in \Gamma \qquad \Gamma \Vdash \sigma \geq \tau}{\Gamma \Vdash_{\texttt{mono}} x : \tau}\ \textsc{TmVar} \qquad \frac{}{\Gamma \Vdash_{\texttt{mono}} \texttt{unit} : \texttt{Unit}}\ \textsc{TmUnit}$$

$$\frac{\Gamma \Vdash_{\texttt{ty}} \tau_1 \quad \Gamma; x : \tau_1 \Vdash_{\texttt{mono}} e : \tau_2}{\Gamma \Vdash_{\texttt{mono}} \lambda x.e : \tau_1 \to \tau_2}\ \textsc{TmAbs} \qquad \frac{\Gamma \Vdash_{\texttt{mono}} e_1 : \tau_1 \to \tau_2 \quad \Gamma \Vdash_{\texttt{mono}} e_2 : \tau_1}{\Gamma \Vdash_{\texttt{mono}} e_1\ e_2 : \tau_2}\ \textsc{TmApp}$$

$$\frac{\Gamma \Vdash_{\texttt{poly}} e_1 : \sigma \quad \Gamma; x : \sigma \Vdash_{\texttt{mono}} e_2 : \tau}{\Gamma \Vdash_{\texttt{mono}} (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : \tau}\ \textsc{TmLet} \qquad \frac{\overline{a} \# \Gamma \qquad \Gamma; \overline{a} \Vdash_{\texttt{mono}} e : \tau \quad \mathsf{gen}(\tau, \overline{a}) = \sigma}{\Gamma \Vdash_{\texttt{poly}} e : \sigma}\ \textsc{TmGen}$$

$\boxed{\Gamma \Vdash_{\texttt{ty}} \sigma}$ Type Well-formedness

$$\frac{a \in \Gamma}{\Gamma \Vdash_{\texttt{ty}} a} \qquad \frac{}{\Gamma \Vdash_{\texttt{ty}} \texttt{Unit}} \qquad \frac{\Gamma \Vdash_{\texttt{ty}} \tau_1 \qquad \Gamma \Vdash_{\texttt{ty}} \tau_2}{\Gamma \Vdash_{\texttt{ty}} \tau_1 \to \tau_2} \qquad \frac{\Gamma; a \Vdash_{\texttt{ty}} \sigma}{\Gamma \Vdash_{\texttt{ty}} \forall a.\sigma}$$

$\boxed{\texttt{wf}(\Gamma)}$ Scoping/Typing Context Well-formedness

$$\frac{\texttt{wf}(\Gamma)}{\texttt{wf}(\Gamma; \bullet)} \qquad \frac{\texttt{wf}(\Gamma) \qquad a \notin \Gamma}{\texttt{wf}(\Gamma, a)} \qquad \frac{\texttt{wf}(\Gamma) \qquad \Gamma \Vdash_{\texttt{ty}} \sigma}{\texttt{wf}(\Gamma; x : \sigma)}$$

$\boxed{\Gamma \Vdash \sigma_1 \geq \sigma_2}$ Type Subsumption

$$\frac{}{\Gamma \Vdash \tau \geq \tau} \qquad \frac{a \# \Gamma \qquad \Gamma; a \Vdash \sigma_1 \geq \sigma_2}{\Gamma \Vdash \sigma_1 \geq \forall a.\sigma_2} \qquad \frac{\Gamma \Vdash_{\texttt{ty}} \tau_1 \qquad \Gamma \Vdash [\tau_1/a]\sigma \geq \tau_2}{\Gamma \Vdash \forall a.\sigma \geq \tau_2}$$

Figure 3 Typing of the Declarative Specification.

3.1 Syntax

Figure 2 displays the syntax of the declarative system. The terms and types are as given by Damas and Milner [7]. Terms consist of term variables, `unit` values, lambda abstractions, applications, and let-bindings. Type schemes are in Skolem normal form, consisting of a number of quantifiers in front of a monotype. Finally, contexts Γ track the scope of type and term variables that are in scope of an expression.

3.2 Typing

Figure 3 displays the typing rules of our declarative system. As stated, we have extracted the generalisation logic in a separate judgment, giving rise to both a monomorphic typing judgment $\Gamma \Vdash_{\texttt{mono}} e : \tau$, and a polymorphic judgment $\Gamma \Vdash_{\texttt{poly}} e : \sigma$, the latter of which is exclusively used in the typing rule for let-bindings TmLet. Rule TmGen uses the auxiliary function $\mathsf{gen}(\tau, \overline{a})$, which generalises the passed τ over the passed $\overline{a}$ in the usual way. The type- and context well-formedness judgments $\Gamma \vdash_{\texttt{ty}} \sigma$ and $\texttt{wf}(\Gamma)$ are standard. Finally, rule TmVar uses type subsumption [7, 6] to instantiate a type scheme. Since subsumption is only used in this manner, we could have given it the signature $\Gamma \Vdash \sigma \geq \tau$ and omitted the middle rule. Yet, the advantage of the subsumption rules in Figure 3 is that subsumption

x Variables a Skolem type variables $\widehat{\alpha}$ Existential type variables

$$
\begin{array}{rlcl}
\text{Terms} & e & ::= & x \mid \texttt{unit} \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
\text{Monotypes} & T & ::= & a \mid \widehat{\alpha} \mid \texttt{Unit} \mid T_1 \to T_2 \\
\text{Type Schemes} & S & ::= & T \mid \forall a.S \\
\text{Local Existential Context} & A & ::= & \bullet \mid A, \widehat{\alpha} \\
\text{Scoping/Typing Context} & \Psi & ::= & \bullet \mid \Psi; a \mid \Psi; A \mid \Psi; x : S \mid \Psi; \{[A]S\} \\
\text{Type Equalities} & E & ::= & \bullet \mid T_1 \sim T_2, E \\
& \{S\} & \doteq & \{[\bullet]S\}
\end{array}
$$

Figure 4 Syntax of Algorithm $\mathcal{R}$.

proofs can be done in multiple parts and combined using transitivity.

4 Algorithmic System

We now introduce algorithm $\mathcal{R}$. We discuss its syntax, rules and unification algorithm.

4.1 Syntax

Figure 4 displays the syntax used by algorithm $\mathcal{R}$. Observe that we now have two kinds of type variables: like our declarative system we have (Skolem) type variables representing types generalised over by a type scheme. We have added unification variables $\widehat{\alpha}$, which we refer to as existential type variables. Like Skolem type variables they are placeholders which can be substituted for other types. Accordingly, monotypes T may now also take the form of an existential type variable.

Contexts Ψ differ from their declarative counterparts in two significant ways. First, besides Skolem type variables, contexts also track the scope of existential type variables, similar to [10, 30]. However, unlike Skolem type variables, they are not simply appended as individual variables, but instead come in a list-like structure A. As unification may both split and solve existential type variables, reasoning about ranges of existential type variables traditionally [10] requires adding markers to the context. By putting them in a list we obtain the same reasoning power, without having to add explicit markers.

Secondly, types with their list of existential variables in scope may live in the context as an *invisible* object $\{[A]S\}$. These invisible objects, when combined with input and output contexts, are the essence behind *full contexts*, which we already introduced in Section 2. These allow us to append As and Ss on the context in branches of the inference algorithm that normally would not have them in scope. Invisible objects are invisible to membership $\in$, but visible to both substitution and fresh variable generation $\#$.

4.2 Inference algorithm

Figure 5 shows the rules of algorithm $\mathcal{R}$. Its main judgments feature in and output contexts, where the output context consists of the input context subjected to all unifications made in the derivation, which means sequences A may shrink or grow and substitutions may be made, but their basic structure is the same.

Rule Var looks up a variable in the context, and instantiates polytype S to $[A]T$ using instantiation, discussed below. Rule Unit is trivial.

$\boxed{\Psi_{in} \vdash e : [A]T \dashv \Psi_{out}}$ Type Inference

$$\frac{(x : S) \in \Psi \qquad \Psi \vdash S \geq [A]T}{\Psi \vdash x : [A]T \dashv \Psi}\ \textsc{Var} \qquad \frac{}{\Psi \vdash \texttt{unit} : [\bullet]\texttt{Unit} \dashv \Psi}\ \textsc{Unit}$$

$$\frac{\widehat{\alpha}\#\Psi_{in} \quad \Psi_{in}; \widehat{\alpha}; x : \widehat{\alpha} \vdash e : [A_2]T_2 \dashv \Psi_{out}; A_1; x : T_1}{\Psi_{in} \vdash \lambda x.e : [A_1, A_2](T_1 \to T_2) \dashv \Psi_{out}}\ \textsc{Abs} \qquad \frac{\Psi_{in} \vdash e_1 : S \dashv \Psi \quad \Psi; x : S \vdash e_2 : [A]T \dashv \Psi_{out}; x : S'}{\Psi_{in} \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : [A]T \dashv \Psi_{out}}\ \textsc{Let}$$

$$\frac{\Psi_{in} \vdash e_1 : [A_1]T \dashv \Psi_1 \qquad \Psi_1; \{[A_1]T\} \vdash e_2 : [A_2]T_1 \dashv \Psi_2; \{[A_1']T'\} \quad \widehat{\alpha}\#\Psi_2; (A_1', A_2) \qquad \Psi_2; (A_1', A_2, \widehat{\alpha}); \{\widehat{\alpha}\} \vdash T' \sim T_1 \to \widehat{\alpha} \dashv \Psi_{out}; A_{out}; \{T_{out}\}}{\Psi_{in} \vdash e_1\ e_2 : [A_{out}]T_{out} \dashv \Psi_{out}}\ \textsc{App}$$

$\boxed{\Psi_{in} \vdash e : S \dashv \Psi_{out}}$ Generalization

$$\frac{\Psi_{in} \vdash e : [A]T \dashv \Psi_{out} \qquad \mathsf{gen}(T, A) = S}{\Psi_{in} \vdash e : S \dashv \Psi_{out}}\ \textsc{Gen}$$

$\boxed{\Psi \vdash_{\texttt{ty}} S}$ Type Well-formedness

$$\frac{a \in \Psi}{\Psi \vdash_{\texttt{ty}} a} \qquad \frac{\widehat{\alpha} \in \Psi}{\Psi \vdash_{\texttt{ty}} \widehat{\alpha}} \qquad \frac{}{\Psi \vdash_{\texttt{ty}} \texttt{Unit}} \qquad \frac{\Psi \vdash_{\texttt{ty}} T_1 \qquad \Psi \vdash_{\texttt{ty}} T_2}{\Psi \vdash_{\texttt{ty}} T_1 \to T_2} \qquad \frac{\Psi; a \vdash_{\texttt{ty}} S}{\Psi \vdash_{\texttt{ty}} \forall a.S}$$

$\boxed{\texttt{wf}(\Psi)}$ Scoping/Typing Context Well-formedness

$$\frac{\texttt{wf}(\Psi)}{\texttt{wf}(\Psi; \bullet)} \qquad \frac{\texttt{wf}(\Psi) \qquad A\#\Psi}{\texttt{wf}(\Psi, A)} \qquad \frac{\texttt{wf}(\Psi) \qquad \Psi \vdash_{\texttt{ty}} S}{\texttt{wf}(\Psi; x : S)} \qquad \frac{\texttt{wf}(\Psi; A) \qquad \Psi; A \vdash_{\texttt{ty}} T}{\texttt{wf}(\Psi; \{[A]T\})}$$

$\boxed{\Psi \vdash S \geq [A]T}$ Polymorphic Type Instantiation

$$\frac{}{\Psi \vdash T \geq [\bullet]T}\ \textsc{InstMono} \qquad \frac{\widehat{\alpha}\#\Psi \qquad \Psi; (\widehat{\alpha}) \vdash [\widehat{\alpha}/a]S \geq [A]T}{\Psi \vdash \forall a.S \geq [(\widehat{\alpha}), A]T}\ \textsc{InstPoly}$$

Figure 5 Typing of Algorithm $\mathcal{R}$.

While Abs may visually look different from conventional abstraction typing rules, it follows the same approach, with added machinery to derive the list of existential type variables in scope. Term variable x is assigned a fresh existential variable $\widehat{\alpha}$; this assignment is added to the context as well as (the singleton list) $\widehat{\alpha}$. We utilize full contexts to let $[\widehat{\alpha}]\widehat{\alpha}$ enjoy any unifications made during the recursive inference by appending them to the input context and obtaining the possibly further instantiated $[A_1]T_1$ from the output context.

Rule App, as already discussed in Section 2, first infers a type $[A_1]T$ for e_1. Inference proceeds on e_2, with the input environment extended with $[A_1]T$, by using an invisible object. By usage of this invisible object we ensure that we can safely extend the context with $[A_1]T$, because it does not bring either A_1 or T into scope. We now unify e_1's type with a function consisting of e_2's type as argument, and fresh variable $\widehat{\alpha}$ as result. We do so under an environment extend with all existential variables in scope for both types being unified, as well as $\widehat{\alpha}$, occurring as a type, instead of an in-scope variable. For this second occurrence of $\widehat{\alpha}$ we again use an invisible object, which avoids us bringing $\widehat{\alpha}$ into scope twice. We obtain

the results from the unification's output.

Rule Gen, as already discussed in Section 2, is (almost) trivial: based on the recursive, monomorphic inference, we generalise T over A. Note that we do not derive a list of variables in scope of S: since we generalise over all existential variables in scope, this list would always be empty. Finally, we have rule Let, which first infers a polytype using Gen. Inference proceeds on e_2, on which the output is based.

Type Instantiation

Type instantiation is of form $\Psi \vdash S \geq [A]T$, where context Ψ and polytype S are inputs, and the monomorphic instance T and list in scope A are outputs. Essentially, type instantiation takes a type of form $\overline{\forall a}^i.T$, removes all quantifiers, and generates a fresh existential type variable $\widehat{\alpha}^i$ for each Skolem type variable a^i, and returns $[\overline{\widehat{\alpha}}^i]([\overline{\widehat{\alpha}}^i/\overline{a}^i]T)$. For example, the `fst` projection of pairs instantiates to $\bullet \vdash \forall a_1.\forall a_2.(a_1, a_2) \rightarrow a_1 \geq [\widehat{\alpha}_1, \widehat{\alpha}_2](\widehat{\alpha}_1, \widehat{\alpha}_2) \rightarrow \widehat{\alpha}_1$.

Well-formedness

Type well-formedness for the algorithmic system is a moderate extension of the declarative one, adding a single rule that checks if existential type variables $\widehat{\alpha}$ are in the context Ψ. Observe that, since objects are invisible to set membership $\in$, $\{[\widehat{\alpha}]\texttt{Unit}\} \not\vdash_{\texttt{ty}} \widehat{\alpha}$.

Contexts are well-formed iff all contained existential type variables are unique and all contained types are well-formed w.r.t. the context to their left, with any A enclosed in an invisible object temporarily added to the context. The notation $A\#\Psi$ ensures not only that A is fresh w.r.t. Ψ, but also that all $\widehat{\alpha}$ in A are fresh w.r.t. each other. Since objects *are* visible to freshness $\#$, context $\{[\widehat{\alpha}]\texttt{Unit}\}; \widehat{\alpha}$ is ill-formed. Another interesting detail is that, while contexts Ψ may contain Skolem type variables a (and this is used to verify the well-formedness of types), well-formed contexts may not contain any Skolem type variables.

4.3 Unification

Figure 6 displays our unification algorithm. The judgment $\Psi_{in} \vdash E \dashv \Psi_{out}$ unifies a list of constraints E of form $T_1 \sim T_2$ under input context Ψ_{in} and produces an output context Ψ_{out}. It can be viewed as the transitive closure of the single-step unification judgment $\Psi_1 \vdash E_1 \longrightarrow \Psi_2 \vdash E_2$, restricted to those sequences that end in $\bullet$.

Hole Notation

We use the syntax $\Psi[\widehat{\alpha}]$ to denote the context $\Psi_L; (A_L, \widehat{\alpha}, A_R); \Psi_R$, where $\Psi[]$ is the context $\Psi_L; (A_L, A_R); \Psi_R$. A multi-hole notation splits the context into more parts. For example, $\Psi[\widehat{\alpha}_1][\widehat{\alpha}_2]$ means $\Psi_1; (A_1, \widehat{\alpha}_1, A_2, \widehat{\alpha}_2, A_3); \Psi_2$ or $\Psi_1; (A_1, \widehat{\alpha}_1, A_2); \Psi_2; (A_3, \widehat{\alpha}_2, A_4); \Psi_3$. Note that hole notation does not split invisible objects.

Single-step Unification

The single-step unification algorithm essentially is a subset of Zhao et al.'s [30], taking only the cases that apply. Rules 1 and 2 simply discharge already-solved constraints. Rule 3 splits constraints on function types. Rules 7 and 8 deal with constraints on two existential type variables. Since our contexts are ordered, we avoid existential type variables escaping their scope by always substituting away the rightmost variable. Rules 9 and 10 solve constraints with an existential variable on one side, and `Unit` on the other.

$\boxed{\Psi_{in} \vdash E \dashv \Psi_{out}}$ Unification Algorithm

$$\frac{}{\Psi \vdash \bullet \dashv \Psi}\ \textsc{SolNil} \qquad \frac{\Psi_{in} \vdash T_1 \sim T_2, E \longrightarrow \Psi \vdash E \qquad \Psi \vdash E \dashv \Psi_{out}}{\Psi_{in} \vdash T_1 \sim T_2, E \dashv \Psi_{out}}\ \textsc{SolCons}$$

$\boxed{\Psi_1 \vdash E_1 \longrightarrow \Psi_2 \vdash E_2}$ Unification Algorithm (Single-step)

$$\begin{array}{rcl}
\Psi \vdash \texttt{Unit} \sim \texttt{Unit}, E & \longrightarrow_1 & \Psi \vdash E \\
\Psi \vdash \widehat{\alpha} \sim \widehat{\alpha}, E & \longrightarrow_2 & \Psi \vdash E \\
\Psi \vdash (T_1 \to T_2) \sim (T_3 \to T_4), E & \longrightarrow_3 & \Psi \vdash T_1 \sim T_3, T_2 \sim T_4, E \\
\Psi[\widehat{\alpha}] \vdash \widehat{\alpha} \sim (T_1 \to T_2), E & \longrightarrow_4 & [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Psi[\widehat{\alpha}_1, \widehat{\alpha}_2] \vdash (\widehat{\alpha}_1 \to \widehat{\alpha}_2) \sim (T_1 \to T_2), E) \\
 & & \quad \text{where } \widehat{\alpha} \notin fv(T_1 \to T_2) \text{ and } \widehat{\alpha}_1, \widehat{\alpha}_2 \# \Psi[\widehat{\alpha}] \\
\Psi[\widehat{\alpha}] \vdash (T_1 \to T_2) \sim \widehat{\alpha}, E & \longrightarrow_5 & [\widehat{\alpha}_1 \to \widehat{\alpha}_2/\widehat{\alpha}](\Psi[\widehat{\alpha}_1, \widehat{\alpha}_2] \vdash (T_1 \to T_2) \sim (\widehat{\alpha}_1 \to \widehat{\alpha}_2), E) \\
 & & \quad \text{where } \widehat{\alpha} \notin fv(T_1 \to T_2) \text{ and } \widehat{\alpha}_1, \widehat{\alpha}_2 \# \Psi[\widehat{\alpha}] \\
\Psi[\widehat{\alpha}_1][\widehat{\alpha}_2] \vdash \widehat{\alpha}_1 \sim \widehat{\alpha}_2, E & \longrightarrow_7 & [\widehat{\alpha}_1/\widehat{\alpha}_2](\Psi[\widehat{\alpha}_1][] \vdash E) \\
\Psi[\widehat{\alpha}_1][\widehat{\alpha}_2] \vdash \widehat{\alpha}_2 \sim \widehat{\alpha}_1, E & \longrightarrow_8 & [\widehat{\alpha}_1/\widehat{\alpha}_2](\Psi[\widehat{\alpha}_1][] \vdash E) \\
\Psi[\widehat{\alpha}] \vdash \widehat{\alpha} \sim \texttt{Unit}, E & \longrightarrow_9 & [\texttt{Unit}/\widehat{\alpha}](\Psi \vdash E) \\
\Psi[\widehat{\alpha}] \vdash \texttt{Unit} \sim \widehat{\alpha}, E & \longrightarrow_{10} & [\texttt{Unit}/\widehat{\alpha}](\Psi \vdash E)
\end{array}$$

Figure 6 Unification Algorithm.

Finally, rules 4 and 5 solve constraints with an existential variable $\widehat{\alpha}$ on one side, and a function type $T_1 \to T_2$ on the other. Because our contexts are ordered, and both T_1 and T_2 may contain existential variables to the left of $\widehat{\alpha}$, we do not directly unify $\widehat{\alpha} := T_1 \to T_2$, but instead split $\widehat{\alpha}$ into a function type $\widehat{\alpha}_1 \to \widehat{\alpha}_2$, where $\widehat{\alpha}_1$ and $\widehat{\alpha}_2$ are fresh w.r.t. the context. This way, rules 7 and 8 may correctly determine which existential variable to eliminate. Because of our notion of *full* contexts, after substitution we can discharge the fact that $\widehat{\alpha} := \widehat{\alpha}_1 \to \widehat{\alpha}_2$, since no other occurrences of $\widehat{\alpha}$ exist. Finally, to ensure termination, we require $\widehat{\alpha}$ does not occur in $T_1 \to T_2$.

5 Metatheory

To reason about how declarative and algorithmic derivations relate, we first need a way of converting between them. We do so through context instantiation, which takes an algorithmic context and converts it to a declarative one. However, this instantiation leaves us with a problem: what to do about invisible objects? To make reasoning about the declarative system easier, we extend declarative contexts Γ with a rule for objects $\Gamma; \{[\overline{a}]\sigma\}$, and assert we can rewrite these away.

▶ **Definition 1.** $\Gamma_1 \equiv_{a,x} \Gamma_2 \triangleq (\forall a, a \in \Gamma_1 \iff a \in \Gamma_2) \ \wedge \ (\forall (x : \sigma), (x : \sigma) \in \Gamma_1 \iff (x : \sigma) \in \Gamma_2)$

▶ **Lemma 2.** *If* $\Gamma_1 \Vdash_{\texttt{mono}} e : \tau$ *and* $\Gamma_1 \equiv_{a,x} \Gamma_2$, *then* $\Gamma_2 \Vdash_{\texttt{mono}} e : \tau$.

5.1 Context instantiation

Figure 7 shows simplified context instantiation rules, which implicitly coerce Ψs to Γs and allow for the appending of a and A. They are meant to convey the intuition; their actual full definition can be found in the supplementary materials.

$\boxed{\Psi \rightsquigarrow \Gamma}$ Context instantiation

$$\frac{}{\Gamma \rightsquigarrow \Gamma} \qquad \frac{\Gamma; \overline{a} \vdash_{\mathtt{ty}} \tau \quad \Gamma; \overline{a}; [\tau/\widehat{\alpha}]A; \Psi \rightsquigarrow \Gamma'}{\Gamma; (\widehat{\alpha}; A); \Psi \rightsquigarrow \Gamma'} \qquad \frac{\Gamma; \overline{a}_1; \overline{a}_2 \vdash_{\mathtt{ty}} \tau \quad \Gamma; \{[\overline{a}_1; \overline{a}_2; A][\tau/\widehat{\alpha}]T\}; \Psi \rightsquigarrow \Gamma'}{\Gamma; \{[\overline{a}_1; \widehat{\alpha}; A]T\}; \Psi \rightsquigarrow \Gamma'}$$

Figure 7 Context instantiation.

For existential type variables outside invisible objects, we choose a sequence of Skolem type variables $\overline{a}$ and a declarative type τ that is well-typed w.r.t. the *already-instantiated* context Γ to its left as well as the chosen sequence $\overline{a}$. We proceed by replacing $\widehat{\alpha}$ by $\overline{a}$, and substituting τ for $\widehat{\alpha}$ in the remaining, still-to-be instantiated Ψ to its right. For $\widehat{\alpha}$'s in invisible objects the logic is similar, but the generated sequences A and substitutions stay local to the object itself.

5.2 Soundness

Using context instantiation, we can formulate the soundness of the algorithmic system. We want to show that, for every closed algorithmic derivation, *any* instantiation leads to a valid derivation in the declarative system.

▶ **Theorem 3** (Soundness of the algorithmic system). *If* $\bullet \vdash e : [A]T \dashv \bullet$ *then for all* $A; \{T\} \rightsquigarrow \{\tau\}$ *we have that* $\bullet \Vdash_{\mathtt{mono}} e : \tau$.

This formulation is too weak to prove directly. Instead, we prove a more general variant, from which soundness follows.

▶ **Lemma 4.** *Given* $\mathtt{wf}(\Psi_{in})$*:*
1. *If* $\Psi_{in} \vdash e : [A]T \dashv \Psi_{out}$ *then for all* $\Psi_{out}; A; \{T\} \rightsquigarrow \Gamma; \{\tau\}$ *we have that* $\Gamma \Vdash_{\mathtt{mono}} e : \tau$.
2. *If* $\Psi_{in} \vdash e : S \dashv \Psi_{out}$ *then for all* $\Psi_{out}; \{S\} \rightsquigarrow \Gamma; \{\sigma\}$ *we have that* $\Gamma \Vdash_{\mathtt{poly}} e : \sigma$.

The proof proceeds by mutual induction on the monomorphic and polymorphic algorithmic typing judgments. As the given instantiation instantiates the *output* context, we reason backwards through the algorithm. As a consequence, for rules App and Gen that have multiple recursive hypotheses, to invoke the induction hypotheses the second time we must produce an instantiation of the intermediate context from the instantiation of the output context. To allow for this, we have proven several lemmas about the backwards preservation of instantiation.

▶ **Lemma 5.** *Both typing judgments and unification preserve instantiation. That is:*
1. *If* $\Psi_{in} \vdash e : [A]T \dashv \Psi_{out}$ *and* $\Psi_{out} \rightsquigarrow \Gamma$*, then* $\Psi_{in} \rightsquigarrow \Gamma$.
2. *If* $\Psi_{in} \vdash e : S \dashv \Psi_{out}$ *and* $\Psi_{out} \rightsquigarrow \Gamma$*, then* $\Psi_{in} \rightsquigarrow \Gamma$.
3. *If* $\Psi_{in} \vdash E \dashv \Psi_{out}$ *and* $\Psi_{out} \rightsquigarrow \Gamma$*, then* $\Psi_{in} \rightsquigarrow \Gamma$.

5.3 Completeness

Completeness states that, for any declarative derivation, there exists an algorithmic derivation that instantiates to it.

▶ **Theorem 6** (Completeness of the algorithmic system). *For each declarative derivation there exists an algorithmic derivation that instantiates to it. That is,*
1. *If* $\bullet \Vdash_{\mathtt{mono}} e : \tau$ *then there exists* A T *such that* $A; \{T\} \rightsquigarrow \{\tau\}$ *and* $\bullet \vdash e : [A]T \dashv \bullet$.

2. *If* $\bullet \Vdash_{\texttt{poly}} e : \sigma$ *then there exists* σ' *such that* $\bullet \vdash e : \sigma' \dashv \bullet$ *and* $\Vdash \sigma' \geq \sigma$.

Observe that (2) from Theorem 6 asserts that a polytype σ' *not containing any existential type variables* is inferred. In other words, σ' is fully ground. Again, we proceed by proving a more general lemma.

▶ **Lemma 7.** *Given* $\texttt{wf}(\Psi_{in})$*:*

1. *If* $\Gamma \Vdash_{\texttt{mono}} e : \tau$, $\Gamma' \leqq_{a,x} \Gamma$, *and* $\Psi_{in}; A_{in} \rightsquigarrow \Gamma'$, *then there exists* T A Ψ_{out} Γ'' $\overline{a}$ *s.t.* $\Gamma' = \Gamma''; \overline{a}$, $\Psi_{in} \vdash e : [A]T \dashv \Psi_{out}$ *and* $\Psi_{out}; A; \{T\}; A_{in} \rightsquigarrow \Gamma''; \{\tau\}; \overline{a}$.
2. *If* $\Gamma \Vdash_{\texttt{poly}} e : \sigma$, $\Gamma' \leqq_{a,x} \Gamma$, *and* $\Psi_{in}; A_{in} \rightsquigarrow \Gamma'$, *then there exists* S σ' Ψ_{out} Γ'' $\overline{a}$ *s.t.* $\Gamma' = \Gamma''; \overline{a}$, $\Psi_{in} \vdash e : S \dashv \Psi_{out}$, $\Psi_{out}; \{S\}; A_{in} \rightsquigarrow \Gamma''; \{\sigma'\}; \overline{a}$, *and* $\Gamma' \Vdash \sigma' \geq \sigma$.

Here, $\Gamma_1 \leqq_{a,x} \Gamma_2$ iff two conditions hold. First, the contexts must contain the same type variables in the same order. Second, their term bindings $(x : S)$ must (1) bind the same names in the same order to (2) types that are related by subsumption under Γ_1.

Finally, we admit the following property about unification:

▶ **Axiom 8.** *If a unifier exists, unification succeeds. That is, if* $\theta T_1 = \theta T_2$ *and* $\Psi_{in} \rightsquigarrow \Gamma$ *then there exists* Ψ_{out} *such that* $\Psi_{in} \vdash T_1 \sim T_2 \dashv \Psi_{out}$ *and* $\Psi_{out} \rightsquigarrow \Gamma$.

5.4 Decidability

In our algorithm there is only one part of which decidability is not obvious: unification. Hence, we prove its decidability here.

▶ **Theorem 9** (Decidability of unification). *Given* $\forall T_1 T_2.\ T_1 \sim T_2 \in E \implies (\Psi_{in} \vdash_{\texttt{ty}} T_1 \wedge \Psi_{in} \vdash_{\texttt{ty}} T_2)$*, it is decidable whether there exists a* Ψ_{out} *such that* $\Psi_{in} \vdash E \dashv \Psi_{out}$.

The proof proceeds by induction on the lexicographic measure $\langle |\Psi_{in}|_{\widehat{\alpha}}, |E| + 2 * |E|_{\rightarrow} \rangle$, representing the number of existential type variables in Ψ_{in} and the length and number of function arrows in E, respectively. All rules directly reduce this measure, except for rules 4 and 5. For these, we need an additional lemma, from which these cases follow. Let us categorize lists of constraints where one side is an existential type variable that does not occur in the rest of the list as E_i, and assert that we can solve any head of pattern E_i without increasing the length of the tail.

$$
\begin{array}{rcll}
E_i & ::= & \bullet & \\
 & | & \widehat{\alpha} \sim T, E_i & \text{with } \widehat{\alpha} \notin E_i \\
 & | & T \sim \widehat{\alpha}, E_i & \text{with } \widehat{\alpha} \notin E_i
\end{array}
$$

▶ **Lemma 10** (Solving E_i). *For all* Ψ_{in} E_i E *there exist* Ψ_{out} E' *such that* $\Psi_{in} \vdash E_i + E \longrightarrow^* \Psi_{out} \vdash E'$ *and* $|\Psi_{out}|_{\widehat{\alpha}} = |\Psi_{in}|_{\widehat{\alpha}} - |E_i|$.

Proof. By induction on $\langle |E_i| + 2 * |E_i|_{\rightarrow} \rangle$. Rules 1, 9 and 10 do not apply. The rest directly reduce the measure, except for (again) rules 4 and 5. We consider rule 4, where $E_i = \widehat{\alpha} \sim (T_1 \rightarrow T_2), E_i'$. It must be immediately followed by rule 3, which gives us $\Psi_{in}[\widehat{\alpha}] \vdash \widehat{\alpha} \sim (T_1 \rightarrow T_2), E_i, E \longrightarrow^* [\widehat{\alpha}_1 \rightarrow \widehat{\alpha}_2]\Psi_{in}[\widehat{\alpha}_1, \widehat{\alpha}_2] \vdash \widehat{\alpha}_1 \sim T_1, \widehat{\alpha}_2 \sim T_2, E_i, E$. Because we know $\widehat{\alpha} \notin E_i'$, we know any substitution of $\widehat{\alpha}$ on E_i does not increase $|E_i|_{\rightarrow}$. Even though we have added an existential variable, we end up with a decreased measure because we have eliminated an arrow, which counts for two. ◀

6 Mechanization

We have mechanised both the declarative specification presented in Section 3 as well as the algorithmic system presented in Section 4 in the Coq proof assistant [23]. Furthermore, we have proven the algorithmic system sound and correct w.r.t. the declarative speciation following the approach described in Section 5. The mechanization is implemented by generating definitions with Ott [20] and its backend [29] for the locally nameless representation [2, 5, 15]. To reason about the locally nameless representation, we have generated many useful lemmas with LNgen [1]. The mechanisation consists of ±700 handwritten lines of Ott DSL, ±10 000 lines of handwritten Coq code, ±900 lines of Coq code generated by Ott, and ±6 800 lines of Coq code generated by LNgen.

We start this section with a discussion of these tools and the locally nameless representation. Then, we discuss the major points of difference between what is presented in the paper and the formalization. The mechanisation as well as an exhaustive list of the delta between the paper and the mechanization are available in the supplementary material, as well as at `https://github.com/rogerbosman/hdm-fully-grounding`.

6.1 Ott

Our mechanization uses the Sewel et al.'s Ott [20] DSL to express both the syntax and inference rules in this paper and generate corresponding (LaTeX and) Coq definitions, as well as boilerplate definitions such as substitutions and free variable functions. As Sewell et al. already argue the general benefits of Ott, here we focus only on the aspects that we found particularly useful.

Typically, manually written LaTeX specifications make notational liberties that do not translate well to Coq. For example, we have taken such a liberty in the environment instantiation judgment as discussed in Section 5.1. Ott rejects such ill-typed definitions. Hence, it forces well-typed formulations that can be translated to Coq, but are more verbose in LaTeX. As a compromise, we have stuck to the Ott-generated LaTeX during the development and have manually produced a cleaned-up version for this paper.[3]

A clear advantage of the Ott-generated outputs is that they both have the same single source of truth. Thus, the LaTeX output can be used to reason about the Coq output. Another substantial advantage is that Ott takes care of generating boilerplate definitions such as free variable functions and substitutions.

6.2 The locally nameless representation

Formalizations that contain abstraction must represent variables in some way. Typically, variables are either referred to by explicit name – which suffers from the lack of built-in α-equivalence, and have issues such as shadowing – or a nameless representation such as De Bruijn indices [8], which are sensitive to the context in which they are defined, requiring *shifting* operations whenever such changes occur.

The locally nameless representation combines the two approaches: it uses a named representation for free variables, and a nameless representation for locally bound variables. As a consequence, each alpha-equivalence class of closed lambda terms has a unique representation. At the same time, terms are less sensitive to changes in their context. For example, the lambda expression $\lambda x.\, x\ y$ is represented as $\lambda.\, 0y$, because x is locally bound, while y is

[3] We describe the difference in Section 6.4.

free. This implies a well-formedness condition, namely that every nameless variable has a corresponding abstraction, in other words, that nameless variables are not free. This condition is called *locally closed*.

A locally bound variable can be converted to a named, free variable through *opening*, where any reference to the outermost abstraction is replaced by a named variable. We use e^x to denote opening term e with name x. It's dual is *closing*. We use ${}^{\backslash x}e$ to denote closing e w.r.t. x. Our mechanization uses the locally nameless representation for both the declarative and algorithmic term variables x and for the Skolem type variables a. Since existential type variables $\widehat{\alpha}$ do not have a matching abstraction, they are always free, and thus use the named representation.

Cofinite Quantification

To preserve the locally closed property, whenever we go under a binder, we have to open the term with some named variable quantified over in some way. There are several ways to go about this. One way would be to use existential quantification, where we assert that there exists some name not in the free variables of the term being opened. Consider rule ∀WF-Ex below, which applies this principle to the well-formedness of declarative type schemes.

$$\frac{\exists a.(a \notin fv(\sigma) \cup fv(\Gamma) \qquad \Gamma; a \vdash_{\mathtt{ty}} \sigma^a)}{\Gamma \vdash_{\mathtt{ty}} \forall.\sigma}\ \forall\text{WF-Ex} \qquad \frac{\exists L.\forall a.a \notin L.\ \Gamma; a \vdash_{\mathtt{ty}} \sigma^a}{\Gamma \vdash_{\mathtt{ty}} \forall.\sigma}\ \forall\text{WF-Cof}$$

As described by Aydemir et al. [2], existential quantification is weak as an elimination form. For example, since eliminating this rule only gives well-formedness for one particular name, renaming lemmas are required for deriving well-formedness over any other name.

Universal quantification suffers from the opposite problem: it can be cumbersome to prove the well-formedness of *any* variable satisfying the freshness constaints. In particular, sometimes we want to exclude more variables than just those in $fv(\sigma) \cup fv(\Gamma)$.

Cofinite quantification, as displayed by rule ∀WF-Cof above, offers exactly this. Here, we quantify universally over any name not in some existentially quantified set L. This elimination form is much stronger than with existential quantification, because we know well-formedness to hold for any $a \notin L$, instead of just one, avoiding, in general, the need for renaming lemmas. Yet, as an introduction form, it is much easier to use than with universal quantification, because it allows us to exclude finitely many names, instead of just the fixed set of free variables. While cofinite quantification is not free of quirks (particularly the control flow of quantification), which we describe below, in general, it strikes the best balance.

Ott's Locally Nameless Backend & LNgen

One drawback of cofinite quantification is that implementation details of the variable representation leak to the LaTeX inference rules. Here, Ott's locally nameless backend [29] comes in handy: it automatically converts inference rules as specified in Sections 3 and 4 to those that use a (cofinitely quantified) locally nameless presentation *for Coq only*. The LaTeX definitions render as the original specification.

By default, Ott's locally nameless backend generates definitions for opening terms, but not for closing them. Weirich's Ott fork [27] adds the generation of these closing definitions.

The opening and closing operations are subject to various laws. One of these, which will become relevant later, is the following.

▶ **Proposition 11** (Subsitution as Open and Close). *Subsitution can be defined in terms of* open *and* close. *That is,* $[T/a]S = ({}^{\backslash a}S)^T$.

Proposition 11 as well as many others are automatically generated and proven by LNgen [1], which bases itself on the Ott specification. Our mechanization uses these laws extensively.

6.3 Quirks of the locally nameless representation

As with any variable representation, some quirks arise. We cover three here.

Generalisation

First is the gen function used in Gen in Figure 5, and its definition[4] is displayed below.

$$\begin{aligned}
&\mathsf{gen}(S, \quad \bullet \quad, _) = S \\
&\mathsf{gen}(S, (A; \widehat{\alpha}), L) = \textbf{let } S' = \mathsf{gen}(S, A, L),\ a \# fv(S') \cup L \\
&\qquad\qquad\qquad\qquad\quad \textbf{in } \forall.^{\backslash a}([a/\widehat{\alpha}]\ S')
\end{aligned}$$

Since variable closing closes nameless Skolem type variables a only, we first substitute in a freshly generated one, only to close it away immediately after. While it would be possible to manually define a closing operation that replaces (named) existential type variables with unnamed Skolem type variables, we would lose the ability to reason over them with the laws generated by LNgen. While we cannot completely avoid having to manually replicate some of these in some instances, here we can avoid doing so. Fortunately, because of these same LN-generated laws, reasoning about this is straightforward. If we open the generalised term with some T, we get $({}^{\backslash a}([a/\widehat{\alpha}]\ S'))^{T}$. By Proposition 11, this can be rewritten into $[T/a][a/\widehat{\alpha}]\ S'$, which simplifies to $[T/\widehat{\alpha}]\ S'$.

Lists of variables

Rule TmGen in Figure 2 quantifies over a list of variables $\overline{a}$. Quantifying cofinitely over and opening with a list of type variables instead of a singular variable requires additional machinery and is not supported by Ott. Attempts at patching the generated definitions manually were unsuccessful (we discuss this again in Section 8). As a consequence, the list of variables $\overline{a}$ is quantified existentially, which is why we used an axiom in our proof of the weakening lemma for declarative typing judgments.

Control Flow

When inducting over typing derivations, we have existentially quantified sets of variables L, and universally quantified variables fresh w.r.t. L. Sets L flow downwards from the induction hypothesis to the conclusion. Yet, variables flow upwards from the conclusion to the induction hypothesis. Consider the abstraction case for completeness, which essentially consists of proving the following implication.

$$\begin{aligned}
&(\exists L. \forall x. x \notin L \implies \exists\ \Psi_{out}\ A\ T_2.\ \Psi_{in}; [\widehat{\alpha}]; x : \widehat{\alpha} \vdash e^x : [A_2]T_2 \dashv \Psi_{out}; A_1; x : T_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \Psi_{out}; A_1; x : T_1; A_2; \{T_2\} \rightsquigarrow \Gamma; x : \tau_1; \{\tau_2\}) \\
&\implies \Psi_{in} \vdash \lambda.e : [A_1; A_2]T_1 \to T_2 \dashv \Psi_{out} \wedge \Psi_{out}; A_1; A_2; \{T_1 \to T_2\} \rightsquigarrow \Gamma; \{\tau_1 \to \tau_2\}
\end{aligned}$$

[4] Observe that gen is parametrised with a third argument, unspecified in Gen, which is included in the set w.r.t. fresh variables are generated, i.e. $a \# fv(S') \cup L$. Since fresh variables are immediately closed away, the generalised term is not affected by a choice for L. It is helpful proving the commutativity of generalisation with for example substitution of existential type variables.

There is a problem here. Since we only obtain the term variable to open e with after applying the Abs constructor in the right branch of the conclusion, we do not have access to it in the left branch of the conclusion. Since the IH existentially quantifies objects that occur in both branches of the conclusion, we cannot simply apply the IH twice, once per branch. While the IH can probably be strengthened to shift the $\forall x$ to each of its two branches, we found it easier to apply the IH to a sufficiently fresh variable before splitting the conclusion. This leaves us with a typing derivation opened with a different term than required. However, this can be remedied straightforwardly with the following renaming lemma.

▶ **Lemma 12.** $\Psi_{in} \vdash e : [A]T \dashv \Psi_{out} \implies [y/x]\Psi_{in} \vdash [y/x]e : [A]T \dashv [y/x]\Psi_{out}$

6.4 Delta between the paper and the mechanization

We cover the two most important differences between the system as presented in this paper and the mechanization.

Unification

To facilitate easier reasoning over unification, the mechanisation's single-step unification judgment rules do not apply the substitution directly, but instead output the substitution as a third output, giving unification the form $\Psi_1 \vdash E_1 \longrightarrow \Psi_2 \vdash E_2, \gamma$, where γ has form $[\overline{T}/\overline{\widehat{\alpha}}]$. Note that single steps return either the empty list, or a singleton list. The auxiliary judgment $\Psi_{in} \vdash E \dashv \Psi_{out}, \gamma$ takes the substitution generated by the single-step judgment, applies it to the step's result (yielding the same result as the paper's single-step judgment), and then combines it with the inductive result. Finally, $\Psi_{in} \vdash E \dashv \Psi_{out}$ is defined in terms of this auxiliary judgment by simply discarding the substitution.

Context Instantiation

The instantiation as presented in Section 5.1 contains notation that is not properly translatable to an inductive type. We present instantiation in this manner to obtain a simpler overview of the logic of context instantiation. The instantiation in the mechanization can be obtained by applying the following three transformations.

First, instead of concatenating the already-processed Γ with the yet-to-be processed Ψ, we define instantiation inductively on Ψ, where we pattern match on the different heads of Ψ, process the tail, and then add the processed head. This means that when generating a substitution for an existential type variable $\widehat{\alpha}$ we do not have access to the yet-to-be processed Ψ, since now $\widehat{\alpha}$ is at the head. Therefore, we flip the control flow, instead deriving a substitution θ of form $[\overline{\tau}/\overline{\widehat{\alpha}}]$, and apply it to any bound type later. This yields a signature of $\Gamma \rightsquigarrow \Psi, \theta$.

Then we split out the instantiation of A's in a dedicated judgment, $A \rightsquigarrow \overline{a}, \theta$. Finally, to make it easier to reason about instantiation, we add a context Γ_{in}, θ_{in} such that the following holds.

▶ **Theorem 13** (Splitting and merging context instantiation). *Context instantiation judgments can be split and merged. That is:*

- $\Gamma_{in}, \theta_{in} \vdash \Psi_1; \Psi_2 \rightsquigarrow \Gamma, \theta \implies \exists\, \Gamma_1\ \Gamma_2\ \theta_1\ \theta_2,\ \Gamma = \Gamma_1; \Gamma_2 \wedge \theta = \theta_2; \theta_1$
  $\wedge\ \Gamma_{in}, \theta_{in} \vdash \Psi_1 \rightsquigarrow \Gamma_1, \theta_1 \wedge \Gamma_{in}; \Gamma_1, \theta_1; \theta_{in} \vdash \Psi_2 \rightsquigarrow \Gamma_2, \theta_2.$
- $\Gamma_{in}, \theta_{in} \vdash \Psi_1 \rightsquigarrow \Gamma_1, \theta_1 \wedge \Gamma_{in}; \Gamma_1, \theta_1; \theta_{in} \vdash \Psi_2 \rightsquigarrow \Gamma_2, \theta_2 \implies$
  $\Gamma_{in}, \theta_{in} \vdash \Psi_1; \Psi_2 \rightsquigarrow \Gamma_1; \Gamma_2, \theta_2; \theta_2.$

7 Related Work

The algorithm presented in this paper extends a long line of work on the inference of the HDM system [9, 16, 12, 14]. Yet, a surprisingly small amount of work addresses the issue of underconstrained type variables.

Pottier [18] gives an (not formalised) elaboration algorithm which inspects the accumulated constraints to determine the list of variables in scope of types. In the appendix, they identify the problem of potentially unnecessary quantification. They address this with a non-deterministic specification that "magically" chooses which variables to abstract over.

Vytiniotis et al. advocate [25] removing the generalisation of lets altogether, citing unwanted interactions and needless complexity in context of generalising types with constraints arising from, for example, type classes or GADTs [28]. They observe that removing let generalisation would not be a significant restriction, since most programs do not utilize this functionality. Yet, removing let generalisation would not address the problem of underconstrained types: they would still need to be dealt with, only now by defaulting, since generalisation is no longer an option.

Zhao et al. mechanised [30] an algorithm for Dunfield and Krishnaswami's [10] type system featuring higher-rank polymorphism. However, since these systems are bidirectional, it is left to the programmer to decide which type variable should be generalised over where, if at all. Yet, we have taken a great deal of inspiration from both these works, adopting the in- and output contexts from Dunfield and Krishnaswami, and manner of tracking existential type variables and approach to unification from Zhao et al.

Zhao et al. rewrote Dunfield and Krishnaswami's algorithmic system, citing the lack of support by their proof assistant of choice (Abella [11]) as one of their reasons. Since we are not using any built-in variable binding support (like what is supported by Abella), we did not encounter such limitations. Thus, we were able to maintain the tree-like structure of Dunfield and Krishnaswami instead of the flatter, list-based approach of Zhao et al.

8 Conclusion

In this paper we have presented algorithm $\mathcal{R}$: the first mechanically verified, fully grounding type inference algorithm for the HDM system. The contribution features the novel approach to unification by using full contexts, in which the current context always represents the *entire* context. The algorithm lays the foundation for formalizing algorithms that require determining types for every subterm.

While any variable representation will have its quirks, the quirks of locally nameless as discussed in Section 6.3 make us wonder if a fully nameless representation would be easier to work with. Our design choice of a separate judgment for generalisation did not turn out well. This approach requires mutual induction on the monomorphic and polymorphic typing judgments, which is a nuisance. Furthermore, Coq not being able to generate this mutual induction scheme is what left us unable to manually patch the inference rule for generalisation to quantify the list of variables $\overline{a}$ cofinitely, as discussed in Section 6.3.

One particularly interesting future area of work is the extension of the algorithm with elaboration to an explicitly typed language like System F, potentially extended with elaboration-based features such as Go's structural subtyping system [21] or type classes [26], whose coherence has been proven on paper in a bidirectional setting [3], but – as far as we know – not yet in the HDM system. Since formalizing these algorithms requires reasoning about the scope of existential variables, our work should serve as a solid starting point.

References

1 Brian Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Technical report, Department of Computer and Information Science, University of Pennsylvania, 2010.

2 Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 3–15. ACM, 2008. doi:10.1145/1328438.1328443.

3 Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. Coherence of type class resolution. *Proc. ACM Program. Lang.*, 3(ICFP):91:1–91:28, 2019. doi:10.1145/3341695.

4 Hendrik Bünder. Decoupling language and editor - the impact of the language server protocol on textual domain-specific languages. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20-22, 2019*, pages 129–140. SciTePress, 2019. doi:10.5220/0007556301310142.

5 Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi:10.1007/s10817-011-9225-2.

6 Dominique Clément, Joëlle Despeyroux, Th. Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ml. In William L. Scherlis, John H. Williams, and Richard P. Gabriel, editors, *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, Cambridge, Massachusetts, USA, August 4-6, 1986*, pages 13–27. ACM, 1986. doi:10.1145/319838.319847.

7 Luís Damas and Robin Milner. Principal type-schemes for functional programs. In Richard A. DeMillo, editor, *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, pages 207–212. ACM Press, 1982. doi:10.1145/582153.582176.

8 N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.

9 Catherine Dubois and Valérie Ménissier-Morain. Certification of a type inference tool for ML: damas-milner within coq. *J. Autom. Reason.*, 23(3-4):319–346, 1999. doi:10.1023/A:1006285817788.

10 Jana Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 429–442. ACM, 2013. doi:10.1145/2500365.2500582.

11 Andrew Gacek. The abella interactive theorem prover (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 154–161. Springer, 2008. doi:10.1007/978-3-540-71070-7_13.

12 Jacques Garrigue. A certified implementation of ML with structural polymorphism and recursive types. *Math. Struct. Comput. Sci.*, 25(4):867–891, 2015. doi:10.1017/S0960129513000066.

13 Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Éditeur inconnu, 1972.

14 Adam Gundry, Conor McBride, and James McKinna. Type inference in context. In Venanzio Capretta and James Chapman, editors, *Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010*, pages 43–54. ACM, 2010. doi:10.1145/1863597.1863608.

15 Conor McBride and James McKinna. Functional pearl: i am not a number-i am a free variable. In Henrik Nilsson, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell,*

Haskell 2004, Snowbird, UT, USA, September 22-22, 2004, pages 1–9. ACM, 2004. doi:10.1145/1017472.1017477.

16 Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in isabelle/hol. *J. Autom. Reason.*, 23(3-4):299–318, 1999. doi:10.1023/A:1006277616879.

17 Andrey Popp, Rusty Key, Louis Roché, Oleksiy Golovko, Rudi Grinberg, Sacha Ayoun, cannorin, Ulugbek Abdullaev, Thibaut Mattio, and Max Lantas. ocaml-lsp-server 1.15.1-5.0 – opam, January 2023. URL: https://opam.ocaml.org/packages/ocaml-lsp-server/ocaml-lsp-server.1.15.1-5.0/.

18 François Pottier. Hindley-milner elaboration in applicative style: functional pearl. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 203–212. ACM, 2014. doi:10.1145/2628136.2628145.

19 John C. Reynolds. Towards a theory of type structure. In Bernard J. Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974. doi:10.1007/3-540-06859-7_148.

20 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Tom Ridge, Susmit Sarkar, and Rok Strnisa. Ott: effective tool support for the working semanticist. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 1–12. ACM, 2007. doi:10.1145/1291151.1291155.

21 Martin Sulzmann and Stefan Wehr. A dictionary-passing translation of featherweight go. In Hakjoo Oh, editor, *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings*, volume 13008 of *Lecture Notes in Computer Science*, pages 102–120. Springer, 2021. doi:10.1007/978-3-030-89051-3_7.

22 GHC Team. Using GHCi - GHC User's Guide 9.4.4. URL: https://downloads.haskell.org/ghc/9.4.4/docs/users_guide/index.html.

23 The Coq Development Team. The coq proof assistant, September 2022. doi:10.5281/zenodo.7313584.

24 The Haskell IDE Team. haskell-language-server documentation. URL: https://haskell-language-server.readthedocs.io/en/latest/.

25 Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. Let should not be generalized. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 39–50, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1708016.1708023.

26 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989. doi:10.1145/75277.75283.

27 Stephanie Weirich. Github repository: sweirich/ott, April 2022. URL: https://github.com/sweirich/ott/tree/aa65f53ea0587223662aaad9c48cb0770549f018.

28 Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In Alex Aiken and Greg Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 224–235. ACM, 2003. doi:10.1145/604131.604150.

29 Francesco Zappa Nardelli. A locally-nameless backend for ott, March 2009. URL: https://fzn.fr/projects/ln_ott/.

30 Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. A mechanical formalization of higher-ranked polymorphic type inference. *Proc. ACM Program. Lang.*, 3(ICFP):112:1–112:29, 2019. doi:10.1145/3341716.