Practical Type Inference with Levels

ANONYMOUS AUTHOR(S)

Modern functional languages rely on sophisticated type inference algorithms. However, there often exists a gap between the theoretical presentation of these algorithms and their practical implementations. Specifically, implementations employ techniques not explicitly included in formal specifications, causing undesirable consequences. First, this leads to confusion and unforeseen challenges for developers adhering to the formal specification. Moreover, theoretical guarantees established for a formal presentation may not directly translate to the implementation. This paper focuses on formalizing one such technique, known as *levels*, which is widely used in practice but whose theoretical treatment remains largely understudied. We present the first comprehensive formalization of levels and demonstrate their applicability to type inference implementations.

1 Introduction

Modern functional programming languages utilize sophisticated type inference mechanisms derived from the Hindley-Milner algorithm [Damas and Milner 1982; Hindley 1969] to support expressive type systems. These expressive types include *higher-rank polymorphism* [Dunfield and Krishnaswami 2013; Odersky and Läufer 1996; Peyton Jones et al. 2007], *impredictivity* [Emrich et al. 2020; Parreaux et al. 2024; Serrano et al. 2020], *higher-kinded types* [Xie et al. 2019], and *existential types* [Eisenberg et al. 2021; Läufer and Odersky 1992].

Most studies in type inference often involves both a formal declarative specification as well as a corresponding algorithmic type system. A central focus of such work lies in establishing *soundness* and *completeness* of the algorithmic system, demonstrating that the algorithmic system faithfully captures the properties of the declarative specification.

However, while soundness and completeness are indeed fundamental for type inference algorithms, practical implementations demand more than theoretical guarantees. A crucial aspect often overlooked in formal presentations is the actual *implementation techniques* in modern languages. These techniques are important as practical type inference systems must not only be sound, but also performant, principled, and easy-to-maintain to address practical concerns such as efficiency and code clarity.

As an example, consider the following program:

$$\textbf{let}\ f = \lambda x \rightarrow x\ \textbf{in}\ (f\ 1, f\ \mathit{True})$$

This program requires *let generalization* to infer the polymorphic type $\forall a.\ a \rightarrow a$ for f. However, standard presentations of Hindley-Milner let generalization [Damas and Milner 1982; Hindley 1969; Peyton Jones et al. 2007, 2006] involves traversing the entire typing context to decide the free type variables for $\lambda x.\ x$. This traversal can be inefficient in larger programs with numerous variables and nested let expressions. Practical implementations often use more efficient generalization strategies.

The omission of implementation techniques in the formal presentations of type inference algorithms has undesirable consequences. First, it creates a gap between the theoretical description and the practical realization of these algorithms. Consequently, developers who implement algorithms based solely on formal specifications may encounter unforeseen performance bottleneck or end up implementing additional ad-hoc checks, and only later discover more practical implementation strategies. Moreover, and perhaps more importantly, theoretical guarantees established for a formal presentation may not directly translate to the implementation. This discrepancy can undermine the reliability and predictability of type inference algorithms.

Therefore, we argue that it is essential to bridge this gap by incorporating the key implementation insights into presentations of type inference algorithms. This involves presenting the essential concepts of implementation techniques without delving into every low-level detail, as including every implementation nuance can easily lead to a cluttered and unwieldy presentation. An overly prescriptive approach is also impractical, as developers may still make varied choices regarding concrete implementation details.

To this end, this paper focuses on *levels*, a technique widely used in practical type inference implementations, but whose formal treatment remains largely understudied. Originally proposed by Rémy [1992], levels have been employed in various type checkers, particularly for OCaml and Haskell, to effectively implement features including let-generalization, escape checking of skolems in higher-rank polymorphic systems, and type regions, and more. Surprisingly, despite their prevalence, a formalism of levels remains largely absent from the presentations of those algorithms. Notable exceptions include the original formalism by Rémy [1992] and subsequent work by Kuan and MacQueen [2007], which focused only on levels for let generalization.

This paper aims to address this gap by providing the first comprehensive formalism of levels beyond let generalization, and demonstrate their broader applications within type inference implementations. The formalization provides a theoretical foundation for levels, clarifying their role and interactions, particularly when used for multiple purposes within a type inference algorithm. Moreover, we establish desirable properties including soundness and completeness, ensuring the reliability of level-based type inference. We believe this study will benefit both practitioners and researchers in the field by providing a clearer understanding of levels and their applications.

We offer the following contributions:

- We provide a declarative type system that incorporates let generalization, higher-rank polymorphism, and local datatypes within a level-based framework (§4).
- We prove that the level-based declarative system is sound and complete with respect to a non-level-based declarative system (§5). These proofs have been mechanized using the Coq proof assistant [Coq Team 2024], establishing key level-related properties and invariants.
- We present a level-based algorithmic type system, featuring a novel *polymorphic promotion* process for resolving level constraints. We prove the algorithm to be sound and complete with respect to the level-based declarative type system.
- We have implemented and evaluated the level-based type inference algorithm in the Koka compiler (§7), a strongly typed functional language with a polymorphic type-and-effect system.
- We explore language extensions and show how levels are used to support them within modern type checkers such as GHC and the OCaml type checker (§8).

The Coq proofs and the modified Koka compiler are provided as supplementary materials. Our formalism is detailed, and some rules are elided for space reasons. The complete set of rules, as well as proofs of stated theorems for the algorithmic type system are included in the appendix.

2 Overview

This section gives an overview of our work; we use Haskell-like syntax for examples.

2.1 Hindley Milner and Let Generalization

The Hindley-Milner (HM) type system [Damas and Milner 1982; Hindley 1969] provides a foundation for many modern type inference algorithms. A key feature of HM is its ability to incorporate parametric polymorphism while still being able to infer the most general type (i.e. *the principal type*) of a program without requiring user-provided annotations.

As an example, consider the following program:

$$\textbf{let}\ f = \lambda x \to x\ \textbf{in}\ (f\ 1, f\ \mathit{True}) \quad \text{-- } f : \forall a.\, a \to a$$

Here, f is applied to arguments of two different types, *Int* and *Bool* respectively. Fortunately, since the variable x in the expression $\lambda x.\ x$ is unconstrained, its type can be generalized. This allows HM to infer a polymorphic type $\forall a.\ a \to a$ for f, ensuring the program successfully type-checks.

However, generalization must be handled with care. Specifically, consider:

$$\lambda x \to \textbf{let}\ y = x\ \textbf{in}\ (y + 1, \mathit{not}\ y) \quad \text{-- error}$$

In this case, the definition of y refers to x. While it might seem that x is unconstrained within the definition of y, leading to a tempting generalization of y's type to $\forall a.\ a$, this would be incorrect! Rather, since x is defined outside of y's definition, we cannot generalize over its type.

To correctly implement generalization, the HM let generalization is formalized as follows:

$$\frac{\Psi \vdash e_1 : \tau_1 \qquad \Psi, x : \forall \overline{a}.\ \tau_1 \vdash e_2 : \tau_2 \qquad \overline{a} \notin \mathsf{ftv}\,(\Psi)}{\Psi \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\ \text{HM-LET}$$

The rule first infers the type of e_1, getting τ_1. It then generalizes τ_1 to $\forall \overline{a}.\ \tau_1$ as the type of x, and adds x to the typing context to infer the type of e_2, getting τ_2. Importantly, the side condition $\overline{a} \notin \mathsf{ftv}\,(\Psi)$ requires the generalized type variables to not appear in the free type variables of Ψ.

To illustrate the importance of the side condition in rule HM-LET, let us revisit our previous examples. In the first case, we have $\bullet \vdash \lambda x.\ x : a \to a$, allowing us to generalize the type to obtain $f : \forall a.\ a \to a$. However, in the second case, we have $x : a \vdash x : a$, and the occurrence of a in the typing context prevents generalization, resulting in $y : a$, thus correctly reject the second program.

A language implementor for the HM type system will then use the algorithmic version of this rule, which takes $\overline{a} = \mathsf{ftv}\,(\tau_1) - \mathsf{ftv}\,(\Psi)$, explicitly calculating the set of type variables to generalize.

However, implementing generalization directly this way can lead to inefficiencies. Specifically, each generalization step requires traversing the entire typing context ($\mathsf{ftv}\,(\Psi)$) to determine the free type variables. This traversal can become computationally expensive, especially when dealing with larger contexts containing numerous definitions.

To address such inefficiency, we employ the following let generalization rule LET:

$$\frac{\Psi \vdash^{n+1} e_1 : \tau_1 \qquad \Psi, x : \forall \mathsf{ftv}^{n+1}(\tau_1).\ \tau_1 \vdash^{n} e_2 : \tau_2}{\Psi \vdash^{n} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\ \text{LET} \qquad \frac{\Psi, x : \tau_3^{\leq n} \vdash^{n} e : \tau_4}{\Psi \vdash^{n} \lambda x.\ e : \tau_3 \to \tau_4}\ \text{LAM}$$

Notably, the typing judgment is now indexed by an integer n, called a *level*. This level is incremented when typing the expression e_1, effectively tracking the nesting depth of let expressions. (The concept of levels extends beyond nested lets, as we will explore later.) Moreover, each type variable is now also associated with a level. Importantly, a type variable can only be used if its level is less than or equal to the current typing level. This invariant is maintained throughout the type inference process. In particular, when typing a lambda expression (rule LAM), the type of the argument x is required to have a level at most n. As a result, the typing context Ψ in rule LET only contains variables at a level at most n, while the type τ_1 may include variables at level $n+1$. Upon exiting e_1, any variables at level $n+1$ are guaranteed to not occur in Ψ. Therefore, we can generalize those variables in τ_1.

We can see that rule LET, compared to rule HM-LET, offers a more efficient approach generalization. Specifically, rule LET calls ftv^{n+1}, which traverses the type τ_1 to identify free type variables at level $n+1$, rather than traversing the typing context. A formalism with rule LET was first introduced by Rémy [1992][1], which has inspired various practical implementations. Rémy's work focused only on let generalization. In contrast, this work demonstrates the broader applicability of levels to other type features. Moreover, we support generalization in a bidirectional type system. Furthermore,

[1] Rémy [1992] used the term *ranks* for the integer n, while modern type checkers generally refer to it as a *level*.

while the declarative specification of levels assumes an implicit mapping from variables to their levels, we additionally provide a mechanization of the level-based system, making such mapping explicit and establishing key invariants and properties for a more rigorous treatment.

2.2 Levels for Higher-Rank Polymorphism

Higher-rank polymorphism allows universal quantifiers to appear nested. Consider the following program taken from Peyton Jones et al. [2007]:

$$f :: (\forall a.\, [a] \to [a]) \to ([\mathit{Bool}], [\mathit{Char}])$$
$$f\ x = (x\ [\mathit{True}, \mathit{False}], x\ [\texttt{'a'}, \texttt{'b'}])$$

Here, f takes a polymorphic function as an argument, making f itself a rank-2 function. The argument x can thus be applied to different list types. As an example, f *reverse* is a valid application, where *reverse* takes a list and returns it in the reverse order.

However, care needs to be taken when type-checking higher-rank polymorphic programs. In particular, assuming ($g : \forall a\ b.\, a \to b \to b$), consider the following program:

$$(\lambda(f :: \forall c.\, c \to \forall d.\, d \to d) \to f\ 1)\ g \quad \text{-- error}$$

Here the function expects an argument of type ($\forall c.\, c \to \forall d.\, d \to d$), while g has type ($\forall a\ b.\, a \to b \to b$). This requires us to check a subtyping constraint ($\forall a\ b.\, a \to b \to b$) <: ($\forall c.\, c \to \forall d.\, d \to d$). However, such a subtyping relation does not hold [Dunfield and Krishnaswami 2013; Odersky and Läufer 1996]. To illustrate why, let's try to resolve the constraint. First, we skolemize c, by removing the universal quantifier and replacing the bound type variable with a fresh skolem variable c. We can then instantiate a with c. However, we need to instantiate b *before* skolemizing d. Consequently, d falls outside the scope of b, preventing the subtyping relation from holding.

In an implementation, the system will first instantiate b with a unification variable, and then skolemize d. The system must then ensure that b cannot be unified with the skolem d. This highlights a crucial aspect of higher-rank type system: the importance of managing the relative scope of unification variables and skolem variables.

Dunfield and Krishnaswami [2013] presents an elegant formalism of higher-rank polymorphism based on *ordered contexts* [Gundry et al. 2010]. This approach carefully tracks the relative scope of unification and skolem variables by imposing a strict ordering, and a unification variable can only be solved with variables preceding it in the context. This ensures well-scopedness, but maintaining such an ordered context can introduce significant overhead in practical implementations. Peyton Jones et al. [2007] ensures correctness by incorporating additional checks in the algorithmic type system. Specifically, writing σ for polymorphic types, when checking $\sigma_1 <: \forall a.\, \sigma_2$, the implementation skolemizes a, and recursively checks $\sigma_1 <: \sigma_2$, producing a substitution S from unification variables to types. The system then checks that $a \notin \mathrm{ftv}(S(\sigma_1))$ and $a \notin \mathrm{ftv}(S(\sigma_2))$, successfully preventing skolems from escaping their scope through unification variables after applying the substitution S. However, ensuring that an implementation has incorporated complete and sufficient checks (for skolem escape or beyond) can be a rather subtle matter.

In our system, we demonstrate that levels can effectively implement skolem escape checks. The key idea is to associate each skolem variable with a level. In particular, upon entering the scope of a skolem, such as $\sigma_1 <: \forall a.\, \sigma_2$, we increment the typing level, and associate the skolem a with this new level. Since unification variables in σ_1 have lower levels, they cannot be unified with skolems at higher levels, preventing skolems from escaping. Importantly, skolem escape checks are now implemented in the same framework based on levels.

Notably, levels now start serving multiple purposes. Since subtyping can also increment the level, levels no longer correspond to the nesting depth of lets. Moreover, subtyping can introduce variables with levels higher than those previously used when entering the scope of let expressions. This seems to suggest that the generalization in rule LET should be updated from $\mathrm{ftv}^{n+1}(\cdot)$ to $\mathrm{ftv}^{\geqslant n+1}(\cdot)$, to include all variables with levels greater than or equal to $n + 1$. Surprisingly, we show that in our system the generalization over $\mathrm{ftv}^{n+1}(\cdot)$ remains sound and complete. This subtle nuance stresses again the importance of a rigorous formal analysis of levels.

2.3 Type Regions

Let us now turn our attention to type regions, specifically focusing on local datatype declarations.[2] As an example, the following program declares a datatype *Tree* with a scope limited to the region following the **in** keyword:[3]

```
data Tree = Leaf Int | Node Tree Tree in
  let f x = case x of Leaf i → i; Node y z = f y + f z
  in f (Node (Leaf 2) (Leaf 3))   -- 5
```

Importantly, the type *Tree* cannot escape its declared scope. The following program will get rejected:

```
data Tree = Leaf Int | Node Tree Tree in
  Leaf 5   -- error
```

This restricted scope exemplifies the concept of type regions, similar to type declarations within local modules (as in OCaml) or type variables unpacked from existential types. To enforce this restriction, the type system must ensure that *Tree* does not appear in the return type of the expression following the declaration. This can be achieved through a straightforward syntactic check of the return type.

Interestingly, we can also leverage levels to implement this scope restriction. Specifically, when entering the scope of a type region, we increment the current typing level, and associate *Tree* with this new level. Upon exiting the scope, we check that the return type has a level less than or equal to the previous level, which effectively ensures that *Tree* does not occur free in the return type. While obtaining the level of a type might involve traversing the entire type structure, leading to a cost similar to directly searching for *Tree*, this approach highlights the versatility of a level-based framework. Checking the level of a type can also be implemented through efficient lookup mechanisms (§8).

In the work, we present a novel type system formalism combining let generalization, higher-rank polymorphism, and local datatype declarations in a unified level-based framework. This showcases the versatility of levels in type inference, enabling programmers to implement these different features through a common mechanism. Why choose this particular combination of features? Because they demonstrate the key roles levels play in modern type checkers, for generalization, subtyping and unification, and scope checking, respectively. These features also illustrate the interplay of levels when serving multiple purposes within an implementation. We demonstrate how the notion of levels can be further applied to other language extensions and how they are implemented in modern type checkers in §8.

In the rest of this paper, we begin by presenting a non-level-based declarative system, and then prove that our level-based system is sound and complete with respect to the non-level-based specification. These proofs have been mechanized to capture the subtleties of the calculus. We then present a corresponding level-based type inference algorithm.

---

[2]The datatype declarations here correspond to ML/Haskell-style *generative* datatypes [MacQueen et al. 2020, §4.3.3].

[3]While we use Haskell-like syntax for illustrative purposes, Haskell does not support local datatype declarations.

$$
\begin{array}{llcll}
\text{expr} & e & ::= & i \mid x \mid D \mid \lambda x.\, e \mid \lambda x : \sigma.\, e \mid e_1\, e_2 \mid e : \sigma & \\
 & & \mid & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{data}\ T = \overline{D_i\, \overline{\sigma_j}^{\,j}}^{\,i}\ \textbf{in}\ e & (\sigma\ \text{closed}) \\
\text{polytype} & \sigma & ::= & \forall a.\, \sigma \mid \sigma_1 \rightarrow \sigma_2 \mid \tau & \\
\text{monotype} & \tau & ::= & \mathsf{Int} \mid a \mid \tau_1 \rightarrow \tau_2 \mid T & \\
\text{context} & \Psi & ::= & \bullet \mid \Psi, x : \sigma \mid \Psi, T \mid \Psi, D : \sigma &
\end{array}
$$

Fig. 1. Syntax

$\boxed{\Psi \vdash e \Rightarrow \sigma}$ *(Type Inference)*

$$
\textsc{T-Lam}\ \frac{\Psi, x : \tau \vdash e \Rightarrow \sigma}{\Psi \vdash \lambda x.\, e \Rightarrow \tau \rightarrow \sigma}
\qquad
\textsc{T-App}\ \frac{\Psi \vdash e_1 \Rightarrow \sigma \quad \sigma \triangleright \sigma_1 \rightarrow \sigma_2 \quad \Psi \vdash e_2 \Leftarrow \sigma_1}{\Psi \vdash e_1\, e_2 \Rightarrow \sigma_2}
\qquad
\textsc{T-Anno}\ \frac{\Psi \vdash \sigma \quad \Psi \vdash e \Leftarrow \sigma}{\Psi \vdash e : \sigma \Rightarrow \sigma}
$$

$$
\textsc{T-Let}\ \frac{\Psi \vdash e_1 \Rightarrow \sigma_1 \quad \Psi, x : \forall \overline{a}.\, \sigma_1 \vdash e_2 \Rightarrow \sigma_2 \quad \overline{a} \notin \mathsf{ftv}(\Psi)}{\Psi \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow \sigma_2}
\qquad
\textsc{T-Data}\ \frac{\Psi, T, \overline{D_i : \overline{\sigma_j}^{\,j} \rightarrow T}^{\,i} \vdash e \Rightarrow \sigma \quad T \notin \mathsf{fT}(\sigma) \quad \overline{\Psi \vdash \overline{\sigma_j}^{\,j}}^{\,i}}{\Psi \vdash \textbf{data}\ T = \overline{D_i\, \overline{\sigma_j}^{\,j}}^{\,i}\ \textbf{in}\ e \Rightarrow \sigma}
$$

$\boxed{\Psi \vdash e \Leftarrow \sigma}$ *(Type Checking)*

$$
\textsc{T-LamC}\ \frac{\Psi, x : \sigma_1 \vdash e \Leftarrow \sigma_2}{\Psi \vdash \lambda x.\, e \Leftarrow \sigma_1 \rightarrow \sigma_2}
\qquad
\textsc{T-Forall}\ \frac{\Psi \vdash e \Leftarrow \sigma \quad a \notin \mathsf{ftv}(\Psi)}{\Psi \vdash e \Leftarrow \forall a.\, \sigma}
\qquad
\textsc{T-Sub}\ \frac{\Psi \vdash e \Rightarrow \sigma_1 \quad \vdash \sigma_1 <: \sigma_2}{\Psi \vdash e \Leftarrow \sigma_2}
$$

$\boxed{\sigma \triangleright \sigma_1 \rightarrow \sigma_2}$ *(Matching)*

$$
\textsc{M-Forall}\ \frac{\sigma[a := \tau] \triangleright \sigma_1 \rightarrow \sigma_2}{\forall a.\, \sigma \triangleright \sigma_1 \rightarrow \sigma_2}
\qquad
\textsc{M-Func}\ \frac{}{\sigma_1 \rightarrow \sigma_2 \triangleright \sigma_1 \rightarrow \sigma_2}
$$

$\boxed{\vdash \sigma_1 <: \sigma_2}$ *(Subtyping)*

$$
\textsc{S-Refl}\ \frac{}{\vdash \sigma <: \sigma}
\qquad
\textsc{S-Func}\ \frac{\vdash \sigma_3 <: \sigma_1 \quad \vdash \sigma_2 <: \sigma_4}{\vdash \sigma_1 \rightarrow \sigma_2 <: \sigma_3 \rightarrow \sigma_4}
\qquad
\textsc{S-ForallR}\ \frac{\vdash \sigma_1 <: \sigma_2 \quad a \notin \mathsf{ftv}(\sigma_1)}{\vdash \sigma_1 <: \forall a.\, \sigma_2}
\qquad
\textsc{S-ForallL}\ \frac{\vdash \sigma_1[a := \tau] <: \sigma_2}{\vdash \forall a.\, \sigma_1 <: \sigma_2}
$$

Fig. 2. Declarative type system (selected rules)

3 Declarative Type System

This section presents a declarative higher-rank polymorphic type system without levels, similar to the one in Dunfield and Krishnaswami [2013]; Peyton Jones et al. [2007], extended with local datatype declarations. This system serves as the base system. In next section we will introduce a level-based system and then establish its soundness and completeness with respect to this system.

3.1 Syntax

Fig. 1 presents the syntax of expressions and types used in this section and §4. Expressions e include literals i, variables x, lambdas $\lambda x.\, e$, annotated lambdas $\lambda x : \sigma.\, e$, applications $e_1\, e_2$, annotated expressions $e : \sigma$, let expressions $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$, and local datatypes $\textbf{data}\ T = \overline{D_i\, \overline{\sigma_j}^{\,j}}^{\,i}\ \textbf{in}\ e$. For simplicity, we focus on datatypes without type parameters.[4] We assume type annotations (in $e : \sigma$ and $\lambda x : \sigma.\ e$) are user-provided and thus are always closed.

Polymorphic types σ include universal quantifications $\forall a.\ \sigma$, functions $\sigma_1 \rightarrow \sigma_2$, and monotypes τ. Monomorphic types τ contain no universal quantifiers, and include the integer type Int, type variable a, functions $\tau_1 \rightarrow \tau_2$, and datatype T.

Type contexts Ψ track the type of variables, the datatypes, and the types of data constructors.

3.2 Typing

Fig. 2 presents the bidirectional typing rules. For space reasons, we show only selected rules; the complete set of rules can be found in the appendix. The typing judgment has two modes: type inference $\Psi \vdash e \Rightarrow \sigma$ infers the type σ of e, while type checking $\Psi \vdash e \Leftarrow \sigma$ checks e against a given type σ.

Rule T-LAM non-deterministically guesses a monotype τ for variable x, and adds $x : \tau$ to the context to type-check the body e. Rule T-APP first infers the type of e_1, getting σ. Since σ must be a function type, the rule uses the *matching* judgment $\triangleright$ to match the type into a function type, where rule M-FORALL instantiates the universal variable a with a monotype. Once rule T-APP matches σ to the function type $\sigma_1 \rightarrow \sigma_2$, it checks the argument e_2 against the expected argument type σ_1, and returns the result type σ_2. Rule T-ANNO ensures the provided annotation is well-formed under the current typing context to exclude out-of-scope uses of type constructors and checks e against the provided annotation. Rule T-LET begins by inferring the type of e_1, getting σ_1. It then generalizes σ_1 over variables $\overline{a}$, provided that $\overline{a} \notin \mathsf{ftv}\,(\Psi)$. The rule then adds $x : \forall \overline{a}.\ \sigma_1$ to the context to type-check the let body.

Rule T-DATA introduces the type constructor and its associated data constructors into the context. It then type-checks e, obtaining the result type σ. Finally, the rule ensures that T does not escape its scope by checking $T \notin \mathsf{fT}(\sigma)$, where fT collects all type constructors in σ.

Checking. For type-checking, rule T-LAMC checks a lambda against a function type $\sigma_1 \rightarrow \sigma_2$, by adding $x : \sigma_1$ to the context and then checking the lambda body against σ_2. This rule shows the benefit of bidirectional typing, as it allows the variable x to have a potentially polymorphic type σ_1. To type-check against a polymorphic type, rule T-FORALL first ensures that $a \notin \Psi$, and then proceeds to checking the expression against σ. Lastly, rule T-SUB switches from checking to inference mode. It first infers the type of e, and then checks the subtyping relation $\vdash \sigma_1 <: \sigma_2$.

Subtyping. The bottom of Fig. 2 presents the subyping judgment. Rule S-REFL states that a type is a subtype of itself. Rule S-FUNC handles function subtyping, where subtyping is contravariant on the argument type, and covariant on the return type. Rule S-FORALLR states that σ_1 is a subtype of $\forall a.\ \sigma_2$, if σ_1 is a subtype of σ_2, provided that a does not appear free in σ_1. Lastly, rule S-FORALLL instantiates a polymorphic type on the left hand side with a monotype τ, and checks if $\sigma_1[a := \tau]$ is a subtype of σ_2.

4 Level-Based Declarative Type System

This section introduces our level-based declarative type system. The language has the same syntax given in Fig. 1. Following Rémy [1992], we assume a given mapping that maps type variables to their levels, which also tracks the levels of type constructors. (See §5 for a formalism with an explicit level context.) We assume there are infinitely many variables of every level. We write a^n or T^n to denote that a and T are of level n. We can then extend levels to types, where the level of a

[4]We foresee no fundamental challenges in supporting parameterized data types, which primarily entails incorporating higher-kinded types. Other advanced datatype features (e.g. GADTs) would require further extensions (§8).

$\boxed{\Psi \vdash^{n} e \Rightarrow \sigma}$ *(Level Type Inference)*

$$\textsc{LT-Lit}\ \frac{}{\Psi \vdash^{n} i \Rightarrow \mathsf{Int}} \qquad \textsc{LT-DCtor}\ \frac{D : \sigma \in \Psi}{\Psi \vdash^{n} D \Rightarrow \sigma} \qquad \textsc{LT-Var}\ \frac{x : \sigma \in \Psi}{\Psi \vdash^{n} x \Rightarrow \sigma} \qquad \textsc{LT-Lam}\ \frac{\Psi, x : \tau^{\leqslant n} \vdash^{n} e \Rightarrow \sigma}{\Psi \vdash^{n} \lambda x.\, e \Rightarrow \tau \to \sigma}$$

$$\textsc{LT-TLam}\ \frac{\Psi \vdash^{n} \sigma_1 \qquad \Psi, x : \sigma_1 \vdash^{n} e \Rightarrow \sigma_2}{\Psi \vdash^{n} \lambda x : \sigma_1.\, e \Rightarrow \sigma_1 \to \sigma_2} \qquad \textsc{LT-App}\ \frac{\Psi \vdash^{n} e_1 \Rightarrow \sigma \qquad \vdash^{n} \sigma \rhd \sigma_1 \to \sigma_2 \qquad \Psi \vdash^{n} e_2 \Leftarrow \sigma_1}{\Psi \vdash^{n} e_1\, e_2 \Rightarrow \sigma_2}$$

$$\textsc{LT-Anno}\ \frac{\Psi \vdash^{n} \sigma \qquad \Psi \vdash^{n} e \Leftarrow \sigma}{\Psi \vdash^{n} e : \sigma \Rightarrow \sigma} \qquad \textsc{LT-Let}\ \frac{\Psi \vdash^{n+1} e_1 \Rightarrow \sigma_1 \qquad \Psi, x : \forall \mathrm{ftv}^{n+1}(\sigma_1).\, \sigma_1 \vdash^{n} e_2 \Rightarrow \sigma_2}{\Psi \vdash^{n} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow \sigma_2}$$

$$\textsc{LT-Data}\ \frac{\Psi, T, \overline{D_i : \overline{\sigma_j}^{j} \to T}^{i} \vdash^{n+1} e \Rightarrow \sigma^{\leqslant n} \qquad T^{n+1} \qquad \overline{\Psi \vdash^{n} \overline{\sigma_j}^{j}}^{i}}{\Psi \vdash^{n} \textbf{data}\ T = \overline{D_i\, \overline{\sigma_j}^{j}}^{i}\ \textbf{in}\ e \Rightarrow \sigma}$$

$\boxed{\Psi \vdash^{n} e \Leftarrow \sigma}$ *(Level Type Checking)*

$$\textsc{LT-LamC}\ \frac{\Psi, x : \sigma_1 \vdash^{n} e \Leftarrow \sigma_2}{\Psi \vdash^{n} \lambda x.\, e \Leftarrow \sigma_1 \to \sigma_2} \qquad \textsc{LT-TLamC}\ \frac{\Psi \vdash^{n} \sigma \qquad \vdash^{n} \sigma_1 <: \sigma \qquad \Psi, x : \sigma \vdash^{n} e \Leftarrow \sigma_2}{\Psi \vdash^{n} \lambda x : \sigma.\, e \Leftarrow \sigma_1 \to \sigma_2}$$

$$\textsc{LT-Sub}\ \frac{\Psi \vdash^{n} e \Rightarrow \sigma_1 \qquad \vdash^{n} \sigma_1 <: \sigma_2}{\Psi \vdash^{n} e \Leftarrow \sigma_2} \qquad \textsc{LT-Forall}\ \frac{\Psi \vdash^{n+1} e \Leftarrow \sigma \qquad a^{n+1}}{\Psi \vdash^{n} e \Leftarrow \forall a.\, \sigma}$$

$\boxed{\vdash^{n} \sigma \rhd \sigma_1 \to \sigma_2}$ *(Matching)*

$$\textsc{LM-Forall}\ \frac{\vdash^{n} \sigma[a := \tau^{\leqslant n}] \rhd \sigma_1 \to \sigma_2}{\vdash^{n} \forall a.\, \sigma \rhd \sigma_1 \to \sigma_2} \qquad \textsc{LM-Func}\ \frac{}{\vdash^{n} \sigma_1 \to \sigma_2 \rhd \sigma_1 \to \sigma_2}$$

$\boxed{\vdash^{n} \sigma_1 <: \sigma_2}$ *(Subtyping)*

$$\textsc{LS-Refl}\ \frac{}{\vdash^{n} \sigma <: \sigma} \qquad \textsc{LS-Func}\ \frac{\vdash^{n} \sigma_3 <: \sigma_1 \qquad \vdash^{n} \sigma_2 <: \sigma_4}{\vdash^{n} \sigma_1 \to \sigma_2 <: \sigma_3 \to \sigma_4} \qquad \textsc{LS-ForallR}\ \frac{\vdash^{n+1} \sigma_1 <: \sigma_2 \qquad a^{n+1}}{\vdash^{n} \sigma_1 <: \forall a.\, \sigma_2}$$

$$\textsc{LS-ForallL}\ \frac{\vdash^{n} \sigma_1[a := \tau^{\leqslant n}] <: \sigma_2}{\vdash^{n} \forall a.\, \sigma_1 <: \sigma_2}$$

Fig. 3. Level-based declarative type system

type is the maximum level of its variables and type constructors. Therefore, closed types are always at level zero. We write $\sigma^{\leqslant n}$ to denote the type σ with the constraint that its level is at most n.

4.1 Typing

Fig. 3 presents the level-based typing rules, where both typing and subtyping are indexed by an integer level n. Rule LT-LIT and rule LT-VAR are straightforward.

Rule LT-LAM again non-deterministically guesses a type τ for the variable x, with the important constraint that τ can be at most level n, the current typing level. Rule LT-TLAM, rule LT-APP, and rule LT-ANNO are self-explanatory. Notably, the matching judgment $\rhd$ is now also associated with a level. Rule LT-APP passes the current typing level to matching, and rule LM-FORALL instantiates the polymorphic type with a type at most at level n.

Importantly, rule LT-LET increments the level to $n+1$ when typing the expression e_1. As a result, lambdas within e_1 can now guess a type at level $n+1$. After rule LT-LET finishes typing e_1, obtaining type σ_1, it generalizes all free variables at $n+1$ in σ_1, and adds $x : \forall \mathrm{ftv}^{n+1}(\sigma_1).\ \sigma_1$ to the context to type-check e_2 at level n. Compared to the previous rule T-LET for typing let expressions, this rule does not require traversing the typing context. Rule LT-DATA type-checks a local datatype declaration, where the level of T is at level $n+1$. The rule adds the type constructor and the associated data constructors to the context type-check e at level $n+1$, obtaining the result type $\sigma^{\leqslant n}$. Since σ is at most level n, it cannot contain T, ensuring that T does not escape from its scope.

Checking. Rule LT-LAMC is straightforward. Rule LT-TLAMC checks that the expected argument type is a subtype of the parameter type, and adds $x : \sigma$ to the context to check the body. Note that the subtyping relation also takes the current typing level. Rule LT-SUB checks that the inferred type is a subtype of the checked type under the current typing level. Rule LT-FORALL checks the expression against a polymorphic type. Here, we take a type variable at level $n+1$, and increment the typing level. Since the type variable is at level $n+1$, it ensures that existing types in Ψ cannot refer to a, without requiring traversing the typing context.

Subtyping. The subtyping judgment is also associated with a level. Of particular interest are rule LS-FORALLR and rule LS-FORALLL. Specifically, rule LS-FORALLR skolemizes the polymorphic type with a type variable at level $n+1$, and increments the subtyping level. Since the type σ_1 is supposed to be at most level n, this ensures that a does not occur free in σ_1, without traversing σ_1. Similar to the matching rule, Rule LS-FORALLL, instantiates the polymorphic type with a monotype at most at level n. Note that since rule LS-FORALLR can increment the subtyping level, the level used here can be greater than the typing level used when first entering subtyping.

4.2 Examples

To demonstrate how typing works, we show a few examples.

Generalization. First, consider $\mathbf{let}\, f = \lambda x.\ x\ \mathbf{in}\, f$, whose typing derivation is given as follows.

$$\dfrac{\dfrac{\bullet, x : a \vdash^{1} x \Rightarrow a \qquad a^{1}}{\bullet \vdash^{1} \lambda x.\ x \Rightarrow a \to a}\ \text{LT-LAM} \qquad \dfrac{}{f : \forall a.\ a \to a \vdash^{0} f \Rightarrow \forall a.\ a \to a}\ \text{LT-VAR}}{\bullet \vdash^{0} \mathbf{let}\, f = \lambda x.\ x\ \mathbf{in}\, f \Rightarrow \forall a.\ a \to a}\ \text{LT-LET}$$

Note that when typing $\lambda x.\ x$, we are at level 1. We assume a^{1} as a side condition, and we can assign $x : a$, since rule LT-LAM requires $x : a^{\leqslant 1}$. As a result, $\mathrm{ftv}^{n+1}(a \to a) = a$, and f gets type $\forall a.\ a \to a$.

Notably, using type variables from different levels in the typing derivation can yield different types of the same expression. Specifically, consider the following derivation:

$$\dfrac{\dfrac{\bullet, x : b \vdash^{1} x \Rightarrow b \qquad b^{0}}{\bullet \vdash^{1} \lambda x.\ x \Rightarrow b \to b}\ \text{LT-LAM} \qquad \dfrac{}{f : b \to b \vdash^{0} f \Rightarrow b \to b}\ \text{LT-VAR}}{\bullet \vdash^{0} \mathbf{let}\, f = \lambda x.\ x\ \mathbf{in}\, f \Rightarrow b \to b}\ \text{LT-LET}$$

Here we use b^{0} as the type of x, and thus the type of f is not generalized. The same type can be derived in the non-level-based system, since rule T-LET may not generalize all free variables in σ_1.

Subtyping. As another example to demonstrate how the typing of a let binding and subtyping could both increment the level, consider typing $(\mathbf{let}\, x = (\lambda f : \sigma_1.\ f\ 2)\ g\ \mathbf{in}\, x)$, where

$$\sigma_1 = (\forall a\, b.\ a \to b \to b) \qquad \sigma_2 = (\forall c.\ c \to \forall d.\ d \to d) \qquad \Psi = g : \sigma_2$$

We give the derivation below, where some intermediate subderivations are omitted:

$$
\dfrac{
  \dfrac{
    \dfrac{
      \Psi, f : \sigma_1 \vdash^1 f \Rightarrow \sigma_1 \quad \vdash^1 \sigma_1 \triangleright \mathsf{Int} \to b_1 \to b_1 \quad b_1{}^1
    }{
      \Psi, f : \sigma_1 \vdash^1 f\, 2 \Rightarrow b_1 \to b_1
    }\,\textsc{lt-app}
  }{
    \Psi \vdash^1 (\lambda f : \sigma_1.\, f\, 2) \Rightarrow \sigma_1 \to b_1 \to b_1
  }\,\textsc{lt-tlam}
  \quad
  \dfrac{
    \Psi \vdash^1 g \Rightarrow \sigma_2 \quad
    \dfrac{
      \dfrac{
        \dfrac{}{\vdash^3 a \to \forall d.\, d \to d <: a \to b \to b}\,\textsc{ls-forallL}
      }{
        \vdash^3 \sigma_2 <: a \to b \to b
      }
      \quad a^2 \quad b^3
    }{
      \vdash^1 \sigma_2 <: \sigma_1
    }\,\textsc{ls-forallR}
  }{
    \Psi \vdash^1 g \Leftarrow \sigma_1
  }\,\textsc{lt-sub}
}{
  \dfrac{\Psi \vdash^1 (\lambda f : \sigma_1.\, f\, 2)\, g \Rightarrow b_1 \to b_1}{\vdots}
}\,\textsc{lt-app}
$$

$$
\dfrac{
  \vdots \qquad \Psi, x : \forall b_1.\, b_1 \to b_1 \vdash^0 x \Rightarrow \forall b_1.\, b_1 \to b_1
}{
  \Psi \vdash^0 \mathbf{let}\, x = (\lambda f : \sigma_1.\, f\, 2)\, g \,\mathbf{in}\, x \Rightarrow \forall b_1.\, b_1 \to b_1
}\,\textsc{lt-let}
$$

There are a few notable things. First, $\vdash^1 \sigma_2 <: \sigma_1$ holds, as we first skolemize σ_1 with a^2 and b^3. Then, $\vdash^3 \sigma_2 <: a \to b \to b$ holds, as subtyping is now at level 3, and we can instantiate c with a^2 and d with b^3 respectively. On the other hand, $\vdash^1 \sigma_1 <: \sigma_2$ does not hold, as shown in the derivation on the right. At the top of the derivation, we would need to instantiate b with a type at most at level 2. However, when d is skolemized later, it will get a level 3.

$$
\dfrac{
  \dfrac{
    \dfrac{\text{not hold}}{\vdash^2 \forall b.\, c \to b \to b <: c \to \forall d.\, d \to d}
  }{
    \vdash^2 \sigma_1 <: c \to \forall d.\, d \to d
  }\,\textsc{ls-forallL}
}{
  \vdash^1 \sigma_1 <: \sigma_2
}\,\textsc{ls-forallR}
$$

Second, note that the type of the entire let binding, $(\lambda f : \sigma_1.\, f\, 2)\, g$, has type $(b_1 \to b_1)^{\leqslant 1}$, given $b_1{}^1$, while the subtyping derivation used level 3. Thus, generalization of $b_1 \to b_1$ only needs to consider free variables at level 1, but not variables of higher levels, corresponding to rule LT-LET.

5 Coq Mechanization

In this section, we establish the soundness and completeness of the level-based declarative type system with respect to the non-level-based system. We begin by outlining the Coq mechanization of the type system, which explicitly encodes level contexts to reason about level-related properties. We then present soundness and completeness.

5.1 Coq Representation

We have assumed an implicit mapping that maps type variables to their levels, which also tracks the levels of type constructors. To facilitate mechanization, we now make level contexts explicit:

$$
\text{level context} \qquad \Delta ::= \bullet \mid \Delta, a^n \mid \Delta, T^n
$$

Level contexts Δ track the levels of both type variables (a^n) and type constructors (T^n). As before, we extend levels to types and contexts, writing $\Delta \vdash \sigma : n$ to denote that the level of σ is n, and $\Delta \vdash \Psi : n$ to denote that the level of Ψ is n.

The typing judgments now incorporate a level context Δ, taking the form $\Delta; \Psi \vdash^n e \Rightarrow \sigma$ and $\Delta; \Psi \vdash^n e \Leftarrow \sigma$. Judgments including matching and subtyping are similarly extended with the level context Δ. These rules are largely unchanged, except for the explicit level handling. For example, rule LCT-FORALL adds a^{n+1} into the context. We assume distinct variables in Δ, which is enforced in our mechanization with the locally nameless representation [Charguéraud 2012]. Thus rule LCT-LET uses $\mathrm{ftv}_\Delta^{n+1}(\sigma)$ to generalize $n+1$ level variables within σ according to the level information in Δ.

$$
\textsc{lct-forall}\quad
\dfrac{\Delta, a^{n+1}; \Psi \vdash^{n+1} e \Leftarrow \sigma}{\Delta; \Psi \vdash^n e \Leftarrow \forall a.\, \sigma}
\qquad
\textsc{lct-let}\quad
\dfrac{\Delta; \Psi \vdash^{n+1} e_1 \Rightarrow \sigma_1 \qquad \Delta; \Psi, x : \forall \mathrm{ftv}_\Delta^{n+1}(\sigma_1).\, \sigma_1 \vdash^n e_2 \Rightarrow \sigma_2}{\Delta; \Psi \vdash^n \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2 \Rightarrow \sigma_2}
$$

The level context allows us to reason explicitly about level-related properties. As an example, we prove that at typing level n, if all type variables and type constructors appearing in type context Ψ have a level no greater than the current typing level ($\Delta \vdash \Psi \leqslant n$), then the inferred type also has a level of at most n ($\Delta \vdash \sigma \leqslant n$):

Lemma 5.1 (Level of inference mode). *If $\Delta; \Psi \vdash^n e \Rightarrow \sigma$, and $\Delta \vdash \Psi \leqslant n$, then $\Delta \vdash \sigma \leqslant n$.*

In typing, $\Delta \vdash \Psi \leqslant n$ will be maintained as an invariant; at the top level, typing starts with level 0 and an empty context, and $\Delta \vdash \bullet \leqslant 0$ holds trivially. This justifies the use of $\text{ftv}_{\Delta}^{n+1}$ in rule LT-LET.

By explicitly reasoning about these invariants, we can now formally establish the equivalence between the level-based and non-level-based versions of our type system.

5.2 Soundness and Completeness

Soundness. The soundness theorem follows directly. Notably, by maintaining the invariant that all type variables and constructors have a level no greater than the current typing level, introducing a type variable (e.g. rule LCT-FORALL) or type constructor at level $n + 1$ ensures freshness relative to the current context. This allows us to establish soundness:

Theorem 5.2 (Soundness of level typing). *Given $\Delta \vdash \Psi \leqslant n$,*
(Inference) if $\Delta; \Psi \vdash^n e \Rightarrow \sigma$, then $\Psi \vdash e \Rightarrow \sigma$.
(Checking) if $\Delta; \Psi \vdash^n e \Leftarrow \sigma$ where $\Delta \vdash \sigma \leqslant n$, then $\Psi \vdash e \Leftarrow \sigma$.

In other words, if an expression e has type σ under level n in level-based declarative typing system, then e also has type σ in the non-level-based declarative type system.

Level-related properties. Proving completeness is more subtle, as it requires us to show that for any non-level-based typing $\Psi \vdash e \Leftrightarrow \sigma$, there *exists* a level context Δ such that $\Delta; \Psi \vdash^n e \Leftrightarrow \sigma$. However, constructing such a Δ presents a few challenges. First, when typing an application $e_1\, e_2$, we need to provide the same level context Δ to derive $\Delta; \Psi \vdash^n e_1 \Rightarrow \sigma$ and $\Delta; \Psi \vdash^n e_2 \Leftarrow \sigma_1$. However, the induction hypothesis provides two distinct level context, Δ_1 and Δ_2, for $\Delta_1; \Psi \vdash^n e_1 \Rightarrow \sigma$ and $\Delta_2; \Psi \vdash^n e_2 \Leftarrow \sigma_1$ respectively. Moreover, rule LCT-LET and rule LCT-FORALL require finding a Δ such that type context Ψ has a level no greater than the current typing level. Additionally, for generalization, we must ensure that any fresh variables satisfying $\overline{a} \notin \text{ftv}(\Psi)$ (as in rule T-LET) must be assigned level $n + 1$ in Δ, while other variables should not.

To address these challenges, we introduce auxiliary definitions that help with the union of two level contexts. Specifically,

Definition 5.3 (Level compatibility). *We say that Δ_1 and Δ_2 are compatible at level n, defined as*

$$\Delta_1 \oslash^n \Delta_2 \triangleq \forall a^{n_1} (or\ T^{n_1}) \in \Delta_1, n_1 \leqslant n \Longrightarrow \exists n_2, n_2 \leqslant n \wedge a^{n_2} (or\ T^{n_2}) \in \Delta_2.$$

Definition 5.4 (Level matching). *We say that Δ_1 and Δ_2 match at level n, defined as*

$$\Delta_1 \varobslash^n \Delta_2 \triangleq \forall a^{n_1} (or\ T^{n_1}) \in \Delta_1, n_1 > n \Longrightarrow a^{n_1} (or\ T^{n_1}) \in \Delta_2.$$

Intuitively, these definitions capture the observations that levels of type variables and constructors can be adjusted with respect to a typing level n. Specifically, compatibility states if a type variable or constructor in Δ_1 has a level no greater than n, its level in Δ_2 remains no greater than n, though the exact levels may differ. Level matching enforces that a type variable or constructor with a level above n in Δ_1 retains the same level in Δ_2.

Combining these two definitions, we can define level consistency between two level contexts:

Definition 5.5 (Consistency). $\Delta_1 \otimes^n \Delta_2 \triangleq \Delta_1 \oslash^n \Delta_2 \wedge \Delta_1 \varobslash^n \Delta_2$.

$$
\begin{array}{llcl}
\text{polytype} & \sigma & ::= & \forall a.\ \sigma \mid \sigma_1 \to \sigma_2 \mid \tau \\
\text{monotype} & \tau & ::= & \mathsf{Int} \mid a \mid T \mid \tau_1 \to \tau_2 \mid \hat{\alpha} \\
\text{term context} & \Sigma & ::= & \bullet \mid \Sigma, x : \sigma \mid \Sigma, D : \sigma \\
\text{algorithmic context} & \Gamma, \Theta, \Delta & ::= & \bullet \mid \Gamma, T^n \mid \Gamma, a^n \mid \Gamma, \hat{\alpha}^n \mid \Gamma, \hat{\alpha}^n = \tau \\
\text{complete context} & \Omega & ::= & \bullet \mid \Gamma, T^n \mid \Gamma, a^n \mid \Gamma, \hat{\alpha}^n = \tau
\end{array}
$$

$\boxed{[\Gamma]\sigma}$ *(Context Application)*

$$
\begin{array}{lll}
[\Gamma]\mathsf{Int} = \mathsf{Int} & [\Gamma](\sigma_1 \to \sigma_2) = [\Gamma]\sigma_1 \to [\Gamma]\sigma_2 & [\Gamma]\hat{\alpha} = \hat{\alpha} \quad \text{if } \hat{\alpha} \notin \Gamma \text{ or } \hat{\alpha}^n \in \Gamma \\
[\Gamma]a \ = a & [\Gamma](\forall a.\ \sigma) \quad = \forall a.\ [\Gamma]\sigma & [\Gamma]\hat{\alpha} = [\Gamma]\tau \quad \text{if } \hat{\alpha}^n = \tau \in \Gamma \\
[\Gamma]T \ = T & &
\end{array}
$$

$\boxed{\Gamma \vdash^n \sigma}$ *(Type Well-Formedness)*

$$
\frac{}{\Gamma \vdash^n \mathsf{Int}}
\qquad
\frac{a^m \in \Gamma \quad m \leqslant n}{\Gamma \vdash^n a}
\qquad
\frac{T^m \in \Gamma \quad m \leqslant n}{\Gamma \vdash^n T}
\qquad
\frac{\Gamma \vdash^n \sigma_1 \quad \Gamma \vdash^n \sigma_2}{\Gamma \vdash^n \sigma_1 \to \sigma_2}
\qquad
\frac{\hat{\alpha}^m \in \Gamma \quad m \leqslant n}{\Gamma \vdash^n \hat{\alpha}}
\qquad
\frac{\hat{\alpha}^m = \tau \in \Gamma \quad m \leqslant n}{\Gamma \vdash^n \hat{\alpha}}
\qquad
\frac{\Gamma, a^n \vdash^n \sigma}{\Gamma \vdash^n \forall a.\ \sigma}
$$

Fig. 4. Syntax of the algorithmic system, context application, and well-formedness of types

Lemma 5.6 (Consistency preserves typing). *Given $\Delta \vdash \Psi \leqslant n$, and $\Delta_1 \otimes^n \Delta_2$, (1) if $\Delta_1; \Psi \vdash^n e \Rightarrow \sigma$, then $\Delta_2; \Psi \vdash^n e \Rightarrow \sigma$; and (2) if $\Delta_1; \Psi \vdash^n e \Leftarrow \sigma$ where $\Delta \vdash \sigma \leqslant n$, then $\Delta_2; \Psi \vdash^n e \Leftarrow \sigma$.*

With these definitions and properties, we can now rename variables and adjust their levels in the level contexts when needed to resolve the challenges.

Completeness. We prove completeness:

Theorem 5.7 (Completeness of level typing).

(Inference) If $\Psi \vdash e \Rightarrow \sigma$, then there exists Δ and n, such that $\Delta \vdash \Psi \leqslant n$ and $\Delta; \Psi \vdash^n e \Rightarrow \sigma$.
(Checking) If $\Psi \vdash e \Leftarrow \sigma$, then there exist Δ and n, such that $\Delta \vdash \Psi \leqslant n$ and $\Delta \vdash \sigma \leqslant n$ and $\Delta; \Psi \vdash^n e \Leftarrow \sigma$.

With that, we concluded the equivalence between the level and non-level version of the type system.

6 Algorithmic Type System with Levels

This section first presents the algorithmic type system with levels, and then shows that the algorithmic system is sound and complete with respect to the level-based declarative type system.

Fig. 4 presents the syntax of the algorithmic system. Monomorphic types are extended with unification variables $\hat{\alpha}$, representing unknown types that will be inferred.

We have two typing contexts: a term context Σ that maps local variables $(x : \sigma)$ and data constructors $(D : \sigma)$ to their types, and an algorithmic context Γ that tracks levels of type constructors (T^n), type variables (a^n), and unification variables $(\hat{\alpha}^n)$. Additionally, Γ records the solutions for unification variables $(\hat{\alpha}^n = \tau)$, with the invariant that τ has a level no greater than n. A complete context Ω is an algorithmic context in which all unification variables have been solved. Notably, the contexts are not ordered, unlike Dunfield and Krishnaswami [2013]. Since contexts contain solutions for unification variables, we use $[\Gamma]\sigma$ to denote the type obtained by applying Γ as a substitution to σ.

Well-formedness of types $\Gamma \vdash^n \sigma$ denotes that σ is well-typed under the algorithmic context Γ at level n. It checks that all type variables and unification variables are bound in the context, and that the levels of those variables are no greater than the typing level.

$$\boxed{\Gamma \mid \Sigma \vdash^n e \Longrightarrow \sigma \dashv \Delta} \qquad \left(\begin{array}{c}\textit{Algorithmic Level Type Inference} \\ \textit{Inputs: } \Gamma, \Sigma, n, e\textit{; Outputs: } \sigma, \Delta\end{array}\right)$$

$$\textsc{at-lit}\ \frac{}{\Gamma \mid \Sigma \vdash^n i \Longrightarrow \mathsf{Int} \dashv \Gamma} \qquad \textsc{at-dctor}\ \frac{D : \sigma \in \Sigma}{\Gamma \mid \Sigma \vdash^n D \Longrightarrow \sigma \dashv \Gamma} \qquad \textsc{at-var}\ \frac{x : \sigma \in \Sigma}{\Gamma \mid \Sigma \vdash^n x \Longrightarrow \sigma \dashv \Gamma}$$

$$\textsc{at-lam}\ \frac{\Gamma, \hat{\alpha}^n \mid \Sigma, x : \hat{\alpha} \vdash^n e \Longrightarrow \sigma \dashv \Delta}{\Gamma \mid \Sigma \vdash^n \lambda x.\, e \Longrightarrow \hat{\alpha} \to \sigma \dashv \Delta} \qquad \textsc{at-tlam}\ \frac{\Gamma \vdash^n \sigma_1 \qquad \Gamma \mid \Sigma, x : \sigma_1 \vdash^n e \Longrightarrow \sigma_2 \dashv \Delta}{\Gamma \mid \Sigma \vdash^n \lambda x : \sigma_1.\, e \Longrightarrow \sigma_1 \to \sigma_2 \dashv \Delta} \qquad \textsc{at-anno}\ \frac{\Gamma \vdash^n \sigma \qquad \Gamma \mid \Sigma \vdash^n e \Longleftarrow \sigma \dashv \Delta}{\Gamma \mid \Sigma \vdash^n e : \sigma \Longrightarrow \sigma \dashv \Delta}$$

$$\textsc{at-app}\ \frac{\Gamma \mid \Sigma \vdash^n e_1 \Longrightarrow \sigma \dashv \Theta_1 \qquad \Theta_1 \vdash^n [\Theta_1]\sigma \rhd \sigma_1 \to \sigma_2 \dashv \Theta_2 \qquad \Theta_2 \mid \Sigma \vdash^n e_2 \Longleftarrow [\Theta_2]\sigma_1 \dashv \Delta}{\Gamma \mid \Sigma \vdash^n e_1\, e_2 \Longrightarrow \sigma_2 \dashv \Delta}$$

$$\textsc{at-let}\ \frac{\Gamma \mid \Sigma \vdash^{n+1} e_1 \Longrightarrow \sigma_1 \dashv \Theta \qquad \mathrm{ftv}_{\Theta}^{n+1}([\Theta]\sigma_1) = \overline{\hat{\alpha}} \qquad \Theta \mid \Sigma, x : \forall \overline{a}.\, (([\Theta]\sigma_1)[\overline{\hat{\alpha}} := \overline{a}]) \vdash^n e_2 \Longrightarrow \sigma_2 \dashv \Delta}{\Gamma \mid \Sigma \vdash^n \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Longrightarrow \sigma_2 \dashv \Delta}$$

$$\textsc{at-data}\ \frac{\overline{\Gamma \vdash^n \overline{\sigma_j}^{\,j}}^{\,i} \qquad \Gamma, T^{n+1} \mid \Sigma, \overline{D_i : \overline{\sigma_j}^{\,j} \to T}^{\,i} \vdash^{n+1} e \Longrightarrow \sigma \dashv \Delta \qquad \Delta \vdash^n \sigma}{\Gamma \mid \Sigma \vdash^n \mathbf{data}\ T = \overline{D_i\, \overline{\sigma_j}^{\,j}}^{\,i}\ \mathbf{in}\ e \Longrightarrow \sigma \dashv \Delta_{\setminus T}}$$

$$\boxed{\Gamma \mid \Sigma \vdash^n e \Longleftarrow \sigma \dashv \Delta} \qquad \left(\begin{array}{c}\textit{Algorithmic Level Type Checking} \\ \textit{Inputs: } \Gamma, \Sigma, n, e, \sigma\textit{; Output: } \Delta\end{array}\right)$$

$$\textsc{at-lamC}\ \frac{\Gamma \mid \Sigma, x : \sigma_1 \vdash^n e \Longleftarrow \sigma_2 \dashv \Delta}{\Gamma \mid \Sigma \vdash^n \lambda x.\, e \Longleftarrow \sigma_1 \to \sigma_2 \dashv \Delta} \qquad \textsc{at-tlamC}\ \frac{\Gamma \vdash^n \sigma_1 <: \sigma \dashv \Theta \qquad \begin{array}{c}\Gamma \vdash^n \sigma \\ \Theta \mid \Sigma, x : \sigma \vdash^n e \Longleftarrow [\Theta]\sigma_2 \dashv \Delta_2\end{array}}{\Gamma \mid \Sigma \vdash^n \lambda x : \sigma.\, e \Longleftarrow \sigma_1 \to \sigma_2 \dashv \Delta_2}$$

$$\textsc{at-sub}\ \frac{\Gamma \mid \Sigma \vdash^n e \Longrightarrow \sigma_1 \dashv \Theta \qquad \Theta \vdash^n [\Theta]\sigma_1 <: [\Theta]\sigma_2 \dashv \Delta}{\Gamma \mid \Sigma \vdash^n e \Longleftarrow \sigma_2 \dashv \Delta} \qquad \textsc{at-forall}\ \frac{\Gamma, a^{n+1} \mid \Sigma \vdash^{n+1} e \Longleftarrow \sigma \dashv \Delta}{\Gamma \mid \Sigma \vdash^n e \Longleftarrow \forall a.\, \sigma \dashv \Delta}$$

$$\boxed{\Gamma \vdash^n \sigma \rhd \sigma_1 \to \sigma_2 \dashv \Delta} \qquad \left(\begin{array}{c}\textit{Algorithmic Matching} \\ \textit{Inputs: } \Gamma, n, \sigma\textit{; Outputs: } \sigma_1, \sigma_2, \Delta\end{array}\right)$$

$$\textsc{am-forall}\ \frac{\Gamma, \hat{\alpha}^n \vdash^n \sigma[a := \hat{\alpha}] \rhd \sigma_1 \to \sigma_2 \dashv \Delta}{\Gamma \vdash^n \forall a.\, \sigma \rhd \sigma_1 \to \sigma_2 \dashv \Delta} \qquad \textsc{am-func}\ \frac{}{\Gamma \vdash^n \sigma_1 \to \sigma_2 \rhd \sigma_1 \to \sigma_2 \dashv \Gamma}$$

$$\textsc{am-uvar}\ \frac{}{\Gamma, \hat{\alpha}^m, \Gamma' \vdash^n \hat{\alpha} \rhd \hat{\alpha}_1 \to \hat{\alpha}_2 \dashv \Gamma, \hat{\alpha}^m = \hat{\alpha}_1 \to \hat{\alpha}_2, \Gamma', \hat{\alpha}_1^m, \hat{\alpha}_2^m}$$

Fig. 5. Algorithmic typing

6.1 Algorithmic Typing

Fig. 5 presents the algorithmic typing rules. The typing judgment $\Gamma \mid \Sigma \vdash^n e \Longrightarrow \sigma \dashv \Delta$ (and $\Gamma \mid \Sigma \vdash^n e \Longleftarrow \sigma \dashv \Delta$) reads: under the algorithmic context Γ and context Σ, at typing level n, expression e infers (or checks against) type σ, updating the algorithmic context to Δ. Intuitively, the algorithmic context is threaded through algorithmic judgments and accumulates information.

Rule AT-LIT, rule AT-DCTOR, and rule AT-VAR are straightforward, and all return the algorithmic context unchanged. Rule AT-LAM, instead of guessing a monotype for x as in the declarative system,

$\boxed{\Gamma \vdash^{n} \sigma_1 <: \sigma_2 \dashv \Delta}$ (*Algorithmic Subtyping* *Inputs:* Γ, n, σ_1, σ_2; *Output:* Δ)

$$\textsc{as-var}\ \frac{}{\Gamma \vdash^{n} a <: a \dashv \Gamma} \qquad \textsc{as-int}\ \frac{}{\Gamma \vdash^{n} \mathsf{Int} <: \mathsf{Int} \dashv \Gamma} \qquad \textsc{as-tyctor}\ \frac{}{\Gamma \vdash^{n} T <: T \dashv \Gamma} \qquad \textsc{as-uvar}\ \frac{}{\Gamma \vdash^{n} \hat{\alpha} <: \hat{\alpha} \dashv \Gamma}$$

$$\textsc{as-func}\ \frac{\Gamma \vdash^{n} \sigma_3 <: \sigma_1 \dashv \Theta \qquad \Theta \vdash^{n} [\Theta]\sigma_2 <: [\Theta]\sigma_4 \dashv \Delta}{\Gamma \vdash^{n} \sigma_1 \to \sigma_2 <: \sigma_3 \to \sigma_4 \dashv \Delta} \qquad \textsc{as-forallR}\ \frac{\Gamma, a^{n+1} \vdash^{n+1} \sigma_1 <: \sigma_2 \dashv \Delta}{\Gamma \vdash^{n} \sigma_1 <: \forall a.\, \sigma_2 \dashv \Delta} \qquad \textsc{as-forallL}\ \frac{\Gamma, \hat{\alpha}^{n} \vdash^{n} \sigma_1[a := \hat{\alpha}] <: \sigma_2 \dashv \Delta}{\Gamma \vdash^{n} \forall a.\, \sigma_1 <: \sigma_2 \dashv \Delta}$$

$$\textsc{as-solveL}\ \frac{\hat{\alpha} \notin \mathsf{ftv}(\sigma) \qquad \Gamma, \hat{\alpha}^{m}, \Gamma' \vdash \sigma \rightsquigarrow^{-}_{m} \tau \dashv \Delta, \hat{\alpha}^{m}, \Delta'}{\Gamma, \hat{\alpha}^{m}, \Gamma' \vdash^{n} \hat{\alpha} <: \sigma \dashv \Delta, \hat{\alpha}^{m} = \tau, \Delta'} \qquad \textsc{as-solveR}\ \frac{\hat{\alpha} \notin \mathsf{ftv}(\sigma) \qquad \Gamma, \hat{\alpha}^{m}, \Gamma' \vdash \sigma \rightsquigarrow^{+}_{m} \tau \dashv \Delta, \hat{\alpha}^{m}, \Delta'}{\Gamma, \hat{\alpha}^{m}, \Gamma' \vdash^{n} \sigma <: \hat{\alpha} \dashv \Delta, \hat{\alpha}^{m} = \tau, \Delta'}$$

Fig. 6. Algorithmic subtyping

$\boxed{\Gamma \vdash \sigma \rightsquigarrow^{\pm}_{m} \tau \dashv \Delta}$ $\boxed{\text{polarity} \quad \pm ::= + \mid -}$ (*Polymorphic Promotion* *Inputs:* Γ, σ, $\pm$, m; *Outputs:* τ, Δ)

$$\textsc{pr-int}\ \frac{}{\Gamma \vdash \mathsf{Int} \rightsquigarrow^{\pm}_{m} \mathsf{Int} \dashv \Gamma} \qquad \textsc{pr-tyctor}\ \frac{T^{m_1} \in \Gamma \qquad m_1 \leqslant m_2}{\Gamma \vdash T \rightsquigarrow^{\pm}_{m_2} T \dashv \Gamma} \qquad \textsc{pr-sk}\ \frac{a^{m_1} \in \Gamma \qquad m_1 \leqslant m_2}{\Gamma \vdash a \rightsquigarrow^{\pm}_{m_2} a \dashv \Gamma}$$

$$\textsc{pr-uvar}\ \frac{\hat{\alpha}^{m_1} \in \Gamma \qquad m_1 \leqslant m_2}{\Gamma \vdash \hat{\alpha} \rightsquigarrow^{\pm}_{m_2} \hat{\alpha} \dashv \Gamma} \qquad \textsc{pr-uvarPr}\ \frac{m_1 > m_2}{\Gamma, \hat{\alpha}_1^{m_1}, \Gamma' \vdash \hat{\alpha}_1 \rightsquigarrow^{\pm}_{m_2} \hat{\alpha}_2 \dashv \Gamma, \hat{\alpha}_1^{m_1} = \hat{\alpha}_2, \Gamma', \hat{\alpha}_2^{m_2}}$$

$$\textsc{pr-func}\ \frac{\Gamma \vdash \sigma_1 \rightsquigarrow^{\mp}_{m} \tau_1 \dashv \Theta \qquad \Theta \vdash [\Theta]\sigma_2 \rightsquigarrow^{\pm}_{m} \tau_2 \dashv \Delta}{\Gamma \vdash \sigma_1 \to \sigma_2 \rightsquigarrow^{\pm}_{m} \tau_1 \to \tau_2 \dashv \Delta} \qquad \textsc{pr-forallPos}\ \frac{\Gamma, \hat{\alpha}^{m} \vdash \sigma[a := \hat{\alpha}] \rightsquigarrow^{+}_{m} \tau \dashv \Delta}{\Gamma \vdash \forall a.\, \sigma \rightsquigarrow^{+}_{m} \tau \dashv \Delta} \qquad \textsc{pr-forallNeg}\ \frac{\Gamma, a^{m+1} \vdash \sigma \rightsquigarrow^{-}_{m} \tau \dashv \Delta}{\Gamma \vdash \forall a.\, \sigma \rightsquigarrow^{-}_{m} \tau \dashv \Delta}$$

Fig. 7. Polymorphic promotion

creates a new unification variable $\hat{\alpha}^{n}$ of the current typing level in the algorithmic context, and adds $x : \hat{\alpha}$ to the context. We assume that new unification variables introduced to the context are always fresh. By assigning $\hat{\alpha}^{n}$, we constrain its solution to a type with a level no greater than n, thus effectively ensuring that x gets a type of a level no greater than n. The rule then proceeds to type-check the lambda body, updating the algorithmic context accordingly. Rule AT-TLAM simply adds $x : \sigma_1$ to the context to type-check the body. Rule AT-ANNO checks the expression against the provided type annotation.

Rule AT-APP first infers the type of e_1, obtaining σ and updating the algorithmic context to Θ_1. Next, the rule applies the matching judgment, given at the bottom of Fig. 5, to instantiate $[\Theta_1]\sigma$ to a function type. In the algorithmic system, the matching judgment takes both the typing level and the algorithmic context. There are three rules. Rule AM-FORALL instantiates a polymorphic type with a new unification variable of the given level n, and matches the body. Rule AM-FUNC directly returns the input function. Lastly, rule AM-UVAR handles unification variables. In this case, the variable must be unsolved, at some level m. Since the variable's solution must be a function, we create two new unification variables $\hat{\alpha}_1^{m}$ and $\hat{\alpha}_2^{m}$, both at the level as m, and set $\hat{\alpha}^{m} = \hat{\alpha}_1 \to \hat{\alpha}_2$. Once matching $[\Theta_1]\sigma$ in rule AT-APP returns $\sigma_1 \to \sigma_2$, the rule checks the argument with the expected type $[\Theta_2]\sigma_1$, resulting in the final algorithmic context Δ.

Rule AT-LET type-checks let expressions. The rule begins by incrementing the typing level to type-check e_1, obtaining σ_1. Then, it collects the unsolved unification variables $\mathsf{ftv}^{n+1}_{\Theta}$ at level $n+1$ within $[\Theta]\sigma_1$ (using the level information in Θ), resulting in a set of unification variables $\overline{\hat{\alpha}}$. Next, it generalizes these unification variables by substituting them with fresh type variables $\overline{a}$ within $[\Theta]\sigma_1$, obtaining $\forall\overline{a}.\ (([\Theta]\sigma_1)[\overline{\hat{\alpha}} := \overline{a}])$. The unification variables $\overline{\hat{\alpha}}$ will no longer be useful and may be removed from the algorithmic context, although this is not strictly required. Finally, it adds x of the generalized type to the context, and type-checks the let body at level n. Lastly, rule AT-DATA adds T^{n+1} to the algorithmic context and associated data constructors to the context to type-check e under $n+1$, obtaining σ. It then checks that σ is well-typed under level n. The returned algorithmic context is $\Delta_{\setminus T}$, which removes all the occurrences of T^{n+1} from the output context and whole complete definition can be found in the appendix.

Type checking. For checking, we maintain the invariant that the type used for checking is fully substituted by the current algorithmic context. Rule AT-LAMC is self-explanatory. Rule AT-TLAMC checks that σ_1 is a subtype of σ, where the subtyping judgment takes the algorithmic context and returns a new Θ. Since Θ may contain new solutions for unification variables, we apply it to σ_2 when checking the lambda body. Rule AT-SUB first infers the type of e, obtaining σ_1 and a new Θ. The rule then applies the context to the types for subtyping, and thus the input types to subtyping are also fully substituted. Lastly, rule AT-FORALL adds the type variable a^{n+1} to the algorithmic context and increments the typing level to check e.

6.2 Subtyping

Fig. 6 presents the algorithmic subtyping rules. The judgment $\Gamma \vdash^{n} \sigma_1 <: \sigma_2 \dashv \Delta$ reads: under the algorithmic context Γ and at level n, type σ_1 is a subtype of σ_2, updating the algorithmic context to Δ. We maintain the invariant that the input types σ_1 and σ_2 are fully substituted under Γ, and thus rules AS-SOLVEL and AS-SOLVER only deal with cases with unsolved unification variables.

The first four rules are straightforward. In rule AS-FUNC, subtyping is contravariant over function argument types, and covariant over return types. Note that the rule applies the context Θ to σ_2 and σ_4, as Θ may contain new information about unification variables. Rule AS-FORALLR skolemizes the polymorphic type with a new type variable a^{n+1}, and increments the subtyping level. Rule AS-FORALLL instantiates the polymorphic type with a new unification variable of the current level.

Of particular interest are the last two rules, which involve unification variables. Rule AS-SOLVEL requires $\hat{\alpha}$ to be a subtype of σ, while rule AS-SOLVER requires σ to be a subtype of $\hat{\alpha}$. In both cases, the rule performs occurs-check ($\hat{\alpha} \notin \mathsf{ftv}\,(\sigma)$). Then, it uses the *promotion* judgment to promote σ to a monotype τ. This process is discussed below. The result monotype τ is guaranteed to be well-typed at the promotion level m. Therefore, we set $\hat{\alpha}^{m} = \tau$ in the output algorithmic context.[5]

Polymorphic promotion. Fig. 7 presents the novel polymorphic promotion judgment. The judgment $\Gamma \vdash \sigma \leadsto^{\pm}_{m} \tau \dashv \Delta$ reads: under the algorithmic context Γ and at level m, promoting type σ under polarity $\pm$ produces a monotype τ, updating the algorithmic context to Δ. Intuitively, the polarity $\pm$ indicates that the type being promoted is a subtype $(+)$ or supertype $(-)$ of a type variable of level m. Since m indicates the promoted level, it never changes in the rules. Recall that rule AS-SOLVEL uses promotion under $(-)$, while rule AS-SOLVER uses it under $(+)$.

Rule PR-INT and rule PR-TYCTOR return the type unchanged. Rule PR-SK promotes a type variable a^{m_1}. This rule requires that the variable's level m_1 be no greater than the promotion level m_2 ($m_1 \leqslant m_2$), as the promotion result will become part of the solution for a unification variable at

[5]There is overlapping between, e.g. rule AS-SOLVEL and rule AS-FORALLL. Such overlap is benign, as their application produces equivalent results. Deterministic behavior could be enforced with additional side conditions.

level m_2, which cannot refer to type variables of higher levels. In other words, promoting a type variable can (correctly) fail. Such failure corresponds to the case where a type variable would otherwise escape its scope through a unification variable.

Promoting unification variables involves two rules: rule PR-UVAR and rule PR-UVARPR. Intuitively, a unification variable at a wider scope can be promoted to a smaller scope for it to be part of a solution for a unification variable with a smaller scope. Specifically, if the unification variable's level is no greater than the promotion level, rule PR-UVAR returns the variable unchanged. Otherwise, rule PR-UVARPR adjusts the level by introducing a new unification variable $\hat{\alpha}_2^{m_2}$ at level m_2, and setting $\hat{\alpha}_1^{m_1} = \hat{\alpha}_2$.

Rule PR-FUNC promotes function types. Due to contravariant function typing, the rule first promotes the argument type under the flipped polarity (denoted as $\mp$), obtaining τ_1 and Θ. Then, it promotes the result type $[\Theta]\sigma_2$ to τ_2 under the original polarity. The final promoted type is $\tau_1 \rightarrow \tau_2$.

Promoting polymorphic types depends on polarity. Rule PR-FORALLPOS promotes a polymorphic type $\forall a.\ \sigma$ under $(+)$. This means that the polymorphic type $\forall a.\ \sigma$ needs to be a subtype of a monotype. Thus, we instantiate a with a fresh unification variable $\hat{\alpha}^m$ of the promotion level m. Conversely, rule PR-FORALLNEG promotes a polymorphic type under $(-)$, which requires $\forall a.\ \sigma$ to be a supertype of a monotype. The rule instantiates a with a fresh type variable at level $m+1$, while promotion stays at level m, effectively preventing a from appearing in σ.

Examples. To see how polarity works, consider the following derivations, with $\hat{\alpha}^0$, for $\forall b.\ b \rightarrow b <: \hat{\alpha}$ on the left, and $\hat{\alpha} <: \forall b.\ b \rightarrow b$ on the right, respectively:

$$\dfrac{\dfrac{\hat{\alpha}^0, \hat{\beta}^0 \vdash \hat{\beta} \rightarrow \hat{\beta} \rightsquigarrow_0^+ \hat{\beta} \rightarrow \hat{\beta} \dashv \hat{\alpha}^0, \hat{\beta}^0}{\hat{\alpha}^0 \vdash \forall b.\ b \rightarrow b \rightsquigarrow_0^+ \hat{\beta} \rightarrow \hat{\beta} \dashv \hat{\alpha}^0, \hat{\beta}^0}\ \textsc{pr-forallPos}}{\hat{\alpha}^0 \vdash^0 \forall b.\ b \rightarrow b <: \hat{\alpha} \dashv \hat{\alpha}^0 = \hat{\beta} \rightarrow \hat{\beta}, \hat{\beta}^0}\ \textsc{as-solveR} \qquad \dfrac{\dfrac{\hat{\alpha}^0, b^1 \vdash b \rightarrow b \rightsquigarrow_0^- ? \dashv ?}{\hat{\alpha}^0 \vdash \forall b.\ b \rightarrow b \rightsquigarrow_0^- ? \dashv ?}\ \textsc{pr-forallNeg}}{\hat{\alpha}^0 \vdash^0 \hat{\alpha} <: \forall b.\ b \rightarrow b \dashv ?}\ \textsc{as-solveL}$$

In the left case, we promote $\forall b.\ b \rightarrow b$ under $(+)$, allowing us to instantiate b with $\hat{\beta}^0$. This is valid as $\forall b.\ b \rightarrow b$ is indeed a subtype of any monotype $\tau \rightarrow \tau$. Conversely, in the right case, we instantiate b with b^1, and promoting b will fail, since rule PR-SK does not apply. This failure is expected, as indeed no monotype can be a subtype of $\forall b.\ b \rightarrow b$.

We now consider a larger example to see how things work together. Specifically, consider typing $(\lambda x.\ \mathbf{let}\ y = f\ x\ \mathbf{in}\ y)$ under $\Sigma = f : \sigma$, where $\sigma = \forall a.\ (a \rightarrow a) \rightarrow a$, with the following derivation:

$$\dfrac{\dfrac{\dfrac{\mathcal{D}}{\hat{\alpha}^0 \mid \Sigma, x : \hat{\alpha} \vdash^1 f\ x \Rightarrow \hat{\beta} \dashv \Delta_3} \qquad \mathrm{ftv}^1_{\Delta_3}([\Delta_3]\hat{\beta}) = \emptyset \qquad \Delta_3 \mid \Sigma, x : \hat{\alpha}, y : \hat{\beta}_1 \vdash^0 y \Rightarrow \hat{\beta}_1 \dashv \Delta_3}{\hat{\alpha}^0 \mid \Sigma, x : \hat{\alpha} \vdash^0 (\mathbf{let}\ y = f\ x\ \mathbf{in}\ y) \Rightarrow \hat{\beta}_1 \dashv \Delta_3}\ \textsc{at-let}}{\bullet \mid \Sigma \vdash^0 (\lambda x.\ \mathbf{let}\ y = f\ x\ \mathbf{in}\ y) \Rightarrow \hat{\alpha} \rightarrow \hat{\beta}_1 \dashv \Delta_3}\ \textsc{at-lam}$$

Here, rule AT-LAM creates a new unification variable $\hat{\alpha}^0$ as the type of $x : \hat{\alpha}$. Rule AT-LET type-checks $f\ x$, and generalizes the result as the type of y. The derivation $\mathcal{D}$ is given in Fig. 8.

There are a few notable things. First, at 1, we match f's type σ to $(\hat{\beta} \rightarrow \hat{\beta}) \rightarrow \hat{\beta}$, with $\hat{\beta}^1$. Then, rule AT-SUB checks if x's type $\hat{\alpha}$ is a subtype of the expected argument type $\hat{\beta} \rightarrow \hat{\beta}$. At 2, rule AS-SOLVEL applies, promoting $\hat{\beta} \rightarrow \hat{\beta}$ under 0, which is $\hat{\alpha}$'s level. Rule PR-UVARPR promotes $\hat{\beta}$ by creating a new unification variable $\hat{\beta}_1^0$, and sets $\hat{\beta}^1 = \hat{\beta}_1$, effectively lowering $\hat{\beta}$'s level to 0. Then, rule PR-UVAR promotes $[\Delta_2]\hat{\beta} = \hat{\beta}_1$, returning $\hat{\beta}_1$. Therefore, at 3, promotion succeeds, and we set $\hat{\alpha} = \hat{\beta}_1 \rightarrow \hat{\beta}_1$. As the final result, rule AT-APP returns $\hat{\beta}$ and the typing context Δ_3.

Returning to rule AT-LET, there are no level 1 variables within $[\Delta_3]\hat{\beta} = \hat{\beta}_1$. Thus y has type $\hat{\beta}_1$, and the final type is $\hat{\alpha} \rightarrow \hat{\beta}_1$.

$$\Delta_1 = \hat{\alpha}^0, \hat{\beta}^1$$
$$\Delta_2 = \hat{\alpha}^0, \boxed{\hat{\beta}^1 = \hat{\beta}_1}, \hat{\beta}_1^0$$
$$\Delta_3 = \boxed{\hat{\alpha}^0 = \hat{\beta}_1 \to \hat{\beta}_1}, \hat{\beta}^1 = \hat{\beta}_1, \hat{\beta}_1^0$$

$$\dfrac{\begin{array}{c}① \\ \hat{\alpha}^0 \mid \Sigma, x : \hat{\alpha} \vdash^1 f \Rightarrow \sigma \dashv \hat{\alpha}^0 \\ \hat{\alpha}^0 \vdash^1 \sigma \triangleright (\hat{\beta} \to \hat{\beta}) \to \hat{\beta} \dashv \Delta_1\end{array} \quad \dfrac{\Delta_1 \mid \Sigma, x : \hat{\alpha} \vdash^1 x \Rightarrow \hat{\alpha} \dashv \Delta_1 \quad \dfrac{\dfrac{\dfrac{}{\Delta_1 \vdash \hat{\beta} \rightsquigarrow^+_0 \hat{\beta}_1 \dashv \Delta_2}\,\textsc{pr-uvarPr} \quad \dfrac{}{\Delta_2 \vdash [\Delta_2]\hat{\beta} \rightsquigarrow^-_0 \hat{\beta}_1 \dashv \Delta_2}\,\textsc{pr-uvar}}{② \; \Delta_1 \vdash \hat{\beta} \to \hat{\beta} \rightsquigarrow^-_0 \hat{\beta}_1 \to \hat{\beta}_1 \dashv \Delta_2}\,\textsc{as-func}}{\Delta_1 \vdash^1 \hat{\alpha} <: \hat{\beta} \to \hat{\beta} \dashv \Delta_3 \; ③}\,\textsc{as-solveL}}{\Delta_1 \mid \Sigma, x : \hat{\alpha} \vdash^1 x \Leftarrow \hat{\beta} \to \hat{\beta} \dashv \Delta_3}\,\textsc{at-sub}}{\hat{\alpha}^0 \mid \Sigma, x : \hat{\alpha} \vdash^1 f\,x \Rightarrow \hat{\beta} \dashv \Delta_3}\,\textsc{at-app}$$

Fig. 8. Example derivation

$\boxed{\Gamma \vdash^n \Sigma}$ *(Term Context Well-Formedness)*

$$\dfrac{}{\Gamma \vdash^n \bullet} \qquad \dfrac{\Gamma \vdash^n \sigma \quad \Gamma \vdash^n \Sigma \quad x \notin \mathrm{dom}(\Sigma)}{\Gamma \vdash^n \Sigma, x : \sigma} \qquad \dfrac{\Gamma \vdash^n \sigma \quad \Gamma \vdash^n \Sigma \quad D \notin \mathrm{dom}(\Sigma)}{\Gamma \vdash^n \Sigma, D : \sigma}$$

$\boxed{\Gamma \vdash^n \Delta}$ *(Algorithmic Context Well-Formedness)*

$$\dfrac{}{\Gamma \vdash^n \bullet} \qquad \dfrac{\Gamma \vdash^n \Delta \quad a \notin \Delta \quad m \leqslant n}{\Gamma \vdash^n \Delta, a^m} \qquad \dfrac{\Gamma \vdash^n \Delta \quad T \notin \Delta \quad m \leqslant n}{\Gamma \vdash^n \Delta, T^m}$$

$$\dfrac{\Gamma \vdash^n \Delta \quad \hat{\alpha} \notin \Delta \quad m \leqslant n}{\Gamma \vdash^n \Delta, \hat{\alpha}^m} \qquad \dfrac{\Gamma \vdash^n \Delta \quad \Gamma \vdash^m \tau \quad \hat{\alpha} \notin \mathrm{ftv}([\Delta]\tau) \quad \hat{\alpha} \notin \Delta \quad m \leqslant n}{\Gamma \vdash^n \Delta, \hat{\alpha}^m = \tau}$$

Fig. 9. Well-formedness of contexts

$\boxed{\Gamma \longrightarrow \Delta}$ *(Context Extension)*

$$\dfrac{}{\bullet \longrightarrow \bullet} \qquad \dfrac{\Gamma \longrightarrow \Delta}{\Gamma, a^n \longrightarrow \Delta, a^n} \qquad \dfrac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, a^n} \qquad \dfrac{\Gamma \longrightarrow \Delta}{\Gamma, T^n \longrightarrow \Delta, T^n} \qquad \dfrac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, T^n}$$

$$\dfrac{\Gamma \longrightarrow \Delta}{\Gamma, \hat{\alpha}^n \longrightarrow \Delta, \hat{\alpha}^n} \qquad \dfrac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \hat{\alpha}^n} \qquad \dfrac{\Gamma[\hat{\alpha} := \tau] \longrightarrow \Delta[\hat{\alpha} := \tau]}{\Gamma, \hat{\alpha}^n \longrightarrow \Delta, \hat{\alpha}^n = \tau}$$

$$\dfrac{\Gamma[\hat{\alpha} := \tau] \longrightarrow \Delta[\hat{\alpha} := \tau'] \quad [\Delta]\tau = [\Delta]\tau'}{\Gamma, \hat{\alpha}^n = \tau \longrightarrow \Delta, \hat{\alpha}^n = \tau'} \qquad \dfrac{\Gamma \longrightarrow \Delta[\hat{\alpha} := \tau]}{\Gamma \longrightarrow \Delta, \hat{\alpha}^n = \tau}$$

Fig. 10. Context extension

6.3 Soundness

We prove that the algorithm is sound and complete (§6.4) with respect to the declarative system. We start with definitions for reasoning about contexts.

Context definitions. Fig. 9 defines well-formedness of contexts. The judgment $\Gamma \vdash^n \Sigma$ states that the term context Σ is well-formed under the algorithmic context Γ at level n. The judgment ensures that all types in Σ are well-formed under Γ at level n.

The judgment $\Gamma \vdash^n \Delta$ states that Δ is well-typed under Γ at level n, ensuring that all variables in Δ have levels no greater than n. The only interesting case is the last rule, which checks that τ is well-formed at level m, with $m \leqslant n$. Additionally, the rule checks that $\hat{\alpha}$ is not free in $[\Delta]\tau$. Lastly, it also requires Δ to be well-formed. Intuitively, we need Γ as Δ may still refer to $\hat{\alpha}$.

We write $\Gamma \vdash^n \Gamma$, or often just Γ^n, to denote that a context Γ is well-formed under itself at level n. When the level does not matter, we also write Γ^∞ to mean that Γ is well-formed at some level.

Fig. 10 defines *context extension*, where the judgment $\Gamma \longrightarrow \Delta$ states that Γ is extended by Δ. Intuitively, context extension expresses a form of information increase, where Δ may contain more variables or solutions for existing variables. The last three rules substitute the solution for $\hat{\alpha}$ in the rest of the contexts, since the contexts may still refer to $\hat{\alpha}$.

Soundness. We now establish soundness, starting from soundness of promotion. Notably, since Ω is a complete context with all unification variables resolved, $[\Omega]\sigma$ produces a declarative type for any well-formed type σ.

Lemma 6.1 (Soundness of promotion). *If Γ^∞ and $\Gamma \vdash^n \sigma$ and $\Delta \longrightarrow \Omega$ and Ω^∞, we have:*
(1) if $\Gamma \vdash \sigma \rightsquigarrow^+_m \tau \dashv \Delta$, then $\vdash^m [\Omega]\sigma <: [\Omega]\tau$.
(2) if $\Gamma \vdash \sigma \rightsquigarrow^-_m \tau \dashv \Delta$, then $\vdash^m [\Omega]\tau <: [\Omega]\sigma$;

The lemma captures the essence of promotion: promoting a polymorphic type σ under positive polarity produces a supertype of σ, while promoting it under negative polarity produces a subtype. With that, we prove the soundness of subtyping:

Theorem 6.2 (Soundness of subtyping). *If Γ^∞ and Δ^∞ and $\Gamma \vdash^n \sigma_1$ and $\Gamma \vdash^n \sigma_2$ and $\Gamma \vdash^n \sigma_1 <: \sigma_2 \dashv \Delta$ where $\Delta \longrightarrow \Omega$ and Ω^∞ then $\vdash^n [\Omega]\sigma_1 <: [\Omega]\sigma_2$.*

Lastly, we prove the soundness of typing, where we extend context application to term contexts, and thus $[\Omega]\Sigma$ produces a declarative context:

Theorem 6.3 (Soundness of typing). *Given $\Delta \longrightarrow \Omega$, where Γ^∞, Δ^∞, and Ω^∞,*
(Inference) If $\Gamma \vdash^n \Sigma$ and $\Gamma \mid \Sigma \vdash^n e \Rightarrow \sigma \dashv \Delta$ then $[\Omega]\Sigma \vdash^n [\Omega]e \Rightarrow [\Omega]\sigma$.
(Checking) If $\Gamma \vdash^n \Sigma$ and $\Gamma \vdash^n \sigma$ and $\Gamma \mid \Sigma \vdash^n e \Leftarrow \sigma \dashv \Delta$ then $[\Omega]\Sigma \vdash^n [\Omega]e \Leftarrow [\Omega]\sigma$.

6.4 Completeness

We now move to completeness. We start with completeness of promotion.

Lemma 6.4 (Completeness of promotion). *If $\Gamma \longrightarrow \Omega$ and Γ^∞ and Ω^∞ and $\Gamma \vdash^n \tau'$, then:*

- *if $\vdash^n [\Omega]\tau' <: [\Omega]\sigma$ then there exist Δ and Ω' such that $\Delta \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$ and $\Gamma \vdash \sigma \rightsquigarrow^-_m \tau \dashv \Delta$ where $[\Omega']\tau = [\Omega']\tau'$;*
- *if $\vdash^n [\Omega]\sigma <: [\Omega]\tau'$ then there exist Δ and Ω' such that $\Delta \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$ and $\Gamma \vdash \sigma \rightsquigarrow^+_m \tau \dashv \Delta$ where $[\Omega']\tau = [\Omega']\tau'$.*

Note that Ω and Δ may contain different but equivalent solutions for unification variables, such as $\Omega = (\hat{\alpha}^0 = \mathsf{Int} \rightarrow \mathsf{Int})$ and $\Delta = (\hat{\alpha}^0 = \hat{\beta} \rightarrow \hat{\beta}, \hat{\beta}^0 = \mathsf{Int})$. Therefore, we show that there is a context Ω' that extends both Δ and Ω. The lemma states that subtyping between a polymorphic type and a monotype can be resolved by promoting the polymorphic type to τ, with $[\Omega']\tau = [\Omega']\tau'$.

We proceed to completeness of subtyping and typing:

Theorem 6.5 (Completeness of subtyping). *If $\Gamma \longrightarrow \Omega$, $\Gamma \vdash^n \sigma_1$, $\Gamma \vdash^n \sigma_2$, and $\vdash^n [\Omega]\sigma_1 <: [\Omega]\sigma_2$ then there exist Δ and Ω' such that $\Delta \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$ and $\Gamma \vdash^n [\Gamma]\sigma_1 <: [\Gamma]\sigma_2 \dashv \Delta$.*

Theorem 6.6 (Completeness of typing). *Given $\Gamma \longrightarrow \Omega$ and Γ^∞ and Ω^∞ and $\Gamma \vdash^n \Sigma$:*
(Inference) If $[\Omega]\Sigma \vdash^n e \Rightarrow \sigma$ and $\vdash^n [\Omega]\Sigma' <: [\Omega]\Sigma$, there exist Δ, Ω', and σ' such that $\Delta \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$ and $\Gamma \mid \Sigma' \vdash^n e \Rightarrow \sigma' \dashv \Delta$ and $\vdash^n [\Omega']\sigma' <: \sigma$.
(Checking) If $[\Omega]\Sigma \vdash^n e \Leftarrow [\Omega]\sigma$ and $\Gamma \vdash^n \sigma$ then there exist Δ and Ω' such that $\Delta \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$ and $\Gamma \mid \Sigma \vdash^n e \Leftarrow [\Gamma]\sigma \dashv \Delta$.

Notably, completeness of inference allows algorithmic typing to produce a more general type than the declarative system, since the declarative system may not always generalize a let binding (§4.2). Thus, the theorem uses notion of *context subtyping*, denoted as $\vdash^n \Psi_1 <: \Psi_2$, where Ψ_1 assigns more general types to the same binding compared to Ψ_2, and the inferred type is also more general.

7 Implementation

We have implemented level-based type inference for the Koka language [Leijen 2013], and included the modified Koka compiler in the supplementary materials.

Compiler implementation. Koka supports both let generalization and higher-rank polymorphism. Following the traditional approach, the existing implementation traverses the entire typing context to collect free type variables for generalization, and includes additional checks for skolem escape.

We implemented level-based type inference in a Koka compiler. Following the formalism, we associate each unification and skolem variable with a level, and keep track of the current level throughout type inference. The typing levels are incremented upon entering a new polymorphism scope and decremented before generalization. This eliminates the need for context traversal during generalization. Similarly, skolemization happens at the incremented level when a lambda is checked against a propagated polymorphic type or when a type is checked to subsume another polymorphic type. The promotion process rejects the program if a skolem attempts to leak to a lower level.

We note that Koka has a polymorphic type-and-effect system with algebraic effect handlers [Plotkin and Power 2001; Plotkin and Pretnar 2009] and mutable reference cells. Our level-based generalization naturally supports effect polymorphism. In particular, generalization happens when typing named functions and top-level bindings that are total (akin to the *value restriction* [Wright 1995]).

Additionally, Koka supports *impredicativity* [Leijen 2008], where type variables can be instantiated with polymorphic types. As a result, the promotion implementation for Koka can produce a polymorphic type and does not require the polarity as shown by the rule on the right. For example, promoting $\forall a.\ a \to a$ produces the type itself. In Koka, this is implemented by treating a as a bound variable without a level, rather than a skolem variable.

$$\frac{\Gamma, a^0 \vdash \sigma \rightsquigarrow_n \sigma'}{\Gamma \vdash \forall a.\, \sigma \rightsquigarrow_n \forall a.\, \sigma'}$$

Validation. To validate the implementation, we have run the modified compiler on the entire Koka test suite which includes 308 positive and negative tests. Our implementation produced results identical to the original compiler for 275 tests.

For the remaining tests, the modified compiler produced equivalent results after alpha-renaming of bound variables for 19 tests, where alpha-equivalence is needed because the constraint solver in the modified compiler generates different numbers of variable identifiers, which then appear in the generated core programs. Both compilers correctly rejected 7 negative tests (involving issues like skolem escapes). The modified implementation produced different error messages, as promotion detected skolem escapes earlier than the traditional context traversal approach. The remaining 7 tests involve an analysis to remove tail effect variables. The analysis is known to be fragile [Ikemori et al. 2022, §4.5], where the typability of a program is sensitive to small program transformations (specifically, lifting a term to a let binding influences typability). We leave developing a more robust analysis to future work.

Evaluation of generalization. It is clear that level-based generalization is computationally more efficient, as it does not involve traversing the entire typing context. Rémy [1992] introduced level-based generalization as "a simple and efficient presentation of ML type system". Nevertheless, Rémy [1992], and subsequent work such as Kuan and MacQueen [2007], did not provide an evaluation.

We evaluated our level-based implementation in the Koka compiler against the original one, focusing on performance gains of level-based generalization in a relatively modern type-checker. To this end, we generated programs that stress the generalization process. Specifically, these generated programs have 200 simple functions nested within a top-level function with varying numbers of parameters. This structure models scenarios where a function relies on numerous local functions. Each nested function simply takes three parameters and returns one parameter from the top-level function. As a result, a program runs generalization 200 times, in a typing context whose size is the sum of the number of parameters in the top-level function and the number of functions already type-checked. The evaluation was performed on a MacBook Pro 2023 with 8-Core 64-bit Apple M3 CPU and 24GB unified memory.

We present the evaluation results in Fig. 11, comparing Level Koka, the level-based implementation, with Koka the original compiler, where let generalization and skolem escape detection traverse the typing context for free type variables. We disabled tail effect removal for both compilers, ensuring identical typing results, so that the performance difference is due to the generalization strategies and the promotion overhead. We report the average type-checking time in milliseconds (ms) over 10 runs for each program. The results show that generalization in Level Koka is 2.9-3.7x faster than Koka on the programs. Moreover, Koka gets slower as the number of parameters in the top-level function grows, leading to a larger typing context.

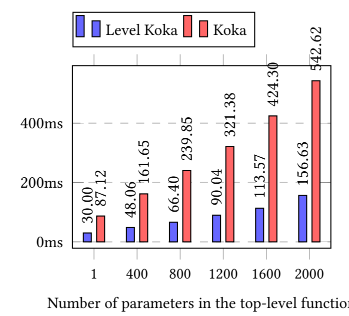

Fig. 11. Evaluation

It is important to note that these benchmark programs are specifically designed to stress the generalization process, and the observed performance is specific to the data structures used within the Koka type checker. In practice, the typing context may not reach the size of 2000, and the overall running time of a compiler is impacted by various other phases beyond type checking. We interpret the evaluation results as preliminary empirical evidence supporting the folklore that level-based generalization is more efficient. In the future we are interested in studying the performance impact on larger-scale Koka applications.

8 Language Extensions

We explore related language extensions and discuss how modern type checkers, specifically the Glasgow Haskell Compiler (GHC) and the OCaml type checker, use levels in their implementations.

Kind polymorphism. While type variables in this paper all have the same kind (i.e. the kind $\star$), modern type checkers often employ higher kinds or *kind polymorphism* [Yorgey et al. 2012]. With kind polymorphism, the kind of a type variable can include a kind variable. Extending levels to support kind variables is relatively straightforward: each kind variable is associated with a level, and promoting a type variable also promotes its kind. Xie et al. [2019] provide a detailed formalism of kind inference in the setting of ordered contexts [Dunfield and Krishnaswami 2013].

GADTs. Similar to the formalism presented in this paper, GHC associates each type variable with a level and uses levels for generalization and skolem escape check. Additionally, GHC uses levels when type-checking programs with *generalized algebraic datatypes* (GADTs).

Specifically, consider the example on the right taken from Vytiniotis et al. [2011]. We can type *test* with either of the following two types that are not a subtype of each other: (1) $\forall a.\, T\ a \to Bool \to Bool$; (2) $\forall a.\, T\ a \to a \to a$. This example demonstrates the known issue that type inference for GADTs does not always have principal types [Cheney and Hinze 2003;

```
data T :: ⋆ → ⋆ where
  T1 :: Int → T Bool
  T2 :: T a

test (T1 n) _ = n > 0
test T2 r = r
```

Vytiniotis et al. 2011]. GHC rejects *test* using the concept of *untouchable* variables. Specifically, if the return type of *test* is a unification variable $\hat{\alpha}$, this variable is considered untouchable (i.e. cannot be solved) under the local assumption $a \sim Bool$, as unifying $\hat{\alpha}$ with $Bool$ has two incompatible solutions: $\hat{\alpha} = a$, or $\hat{\alpha} = Bool$.

GHC implements untouchability using levels. Specifically, GHC's type inference is based on constraint generation and solving. In the above example, the GADT match introduces an *implication constraint* $a \sim Bool \Rightarrow \hat{\alpha} \sim Bool$. Importantly, GHC increments the level when checking a GADT match, and associates the implication constraint with the incremented level (say 2). Since $\hat{\alpha}$ has a lower level (say 1) and is under a local assumption, it is considered untouchable, as unifying $\hat{\alpha}$ may not produce principal types. This mechanism prevents solving $\hat{\alpha}$ with $Bool$.

We remark that using levels for untouchable variables shares similarities with skolem escape checks. In both cases, levels indicate the valid scope of a unification variable: it prevents unification with a skolem variable of a higher level, or within an implication constraint that has a higher level.

Type families. GHC also supports *type families* [Eisenberg et al. 2014; Stolarek et al. 2015]. Recall that when unifying $\hat{\alpha}^1$ with $Maybe\ \hat{\beta}^2$, we promote $\hat{\beta}^2$ to level 1. Interestingly, given a type family F, unifying $\hat{\alpha}^1$ with $F\ \hat{\beta}^2$ should not promote $\hat{\beta}$, as $F\ \hat{\beta}^2$ can potentially reduce to, say, Int, which is well-formed at level 1. As a result, GHC does not promote variables under a type-family application.

The OCaml type checker. Levels are also used in the OCaml type checker, as detailed in a blog post by Kiselyov [2022]. While promotion in our system and GHC involves creating new unification variables, levels in OCaml use mutable references and can thus be updated in-place.

OCaml prevents local definitions from leaking. For example, the program with local modules on the right does not type-check. OCaml achieves that by incrementing the typing level when type-checking a local module, and later checking that the level of the result type has the original level.

```
let y =
  let module M =
    struct
      type t = Foo
      let x = Foo
    end
  in M.x
```

Interestingly, in OCaml, every type is associated with a level, maintained during unification. The design enables efficient level access through a constant-time lookup. OCaml thus employ several techniques to optimizing level-related operations. For example, during generalizing, if a type's level is not greater than the current typing level, the type checker doesn't need to traverse that type's structure. OCaml also adjusts levels to relax the value restriction [Garrigue 2004], by lowering the level of type variables appearing in contravariant positions, preventing their generalization.

OCaml also supports GADTs using the concept of *ambivalent types* [Garrigue and Rémy 2013]. More concretely, types in OCaml carry an additional *scope*. Ensuring that an ambivalent type does not escape its scope is equivalent to checking if its scope is no greater than its level.

Another interesting use of levels is that OCaml associates bound variables with a very large level (10^8). Thus, instantiating can skip types without such a level as they have no bound variables.

9 Related Work and Conclusion

We have discussed most related work on levels throughout the paper [Kiselyov 2022; Kuan and MacQueen 2007; Rémy 1992]. *Ordered contexts* [Dunfield and Krishnaswami 2013; Gundry et al. 2010] is an approach adopted in several subsequent works [Dunfield and Krishnaswami 2019; Xie et al. 2019; Zhao et al. 2019]. While ordered contexts offer an elegant framework for reasoning about type inference, they focus more on theoretical foundations than practical implementations.

This work seems to be the first comprehensive formalism of level-based type inference beyond let generalization. While we explored a range of language features implemented using levels, our investigation is not exhaustive. We are interested in extending the formalism with more features, such as GADTs. Furthermore, mechanization of type inference algorithms requires significant effort [Garrigue 2015; Zhao et al. 2018, 2019]. We would like to mechanize the proofs for our algorithmic system in the future.

References

Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reason.* 49, 3 (2012), 363–408. https://doi.org/10.1007/S10817-011-9225-2

James Cheney and Ralf Hinze. 2003. *First-class phantom types.* Technical Report. Cornell University.

Coq Team. 2024. *The Coq Proof Assistant.* https://coq.inria.fr/

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.* 207–212.

Jana 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.* 429–442.

Jana Dunfield and Neelakantan R Krishnaswami. 2019. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–28.

Richard A Eisenberg, Guillaume Duboc, Stephanie Weirich, and Daniel Lee. 2021. An existential crisis resolved: Type inference for first-class existential types. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–29.

Richard A Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed type families with overlapping equations. *ACM SIGPLAN Notices* 49, 1 (2014), 671–683.

Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. 2020. Freezeml: Complete and easy type inference for first-class polymorphism. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 423–437.

Jacques Garrigue. 2004. Relaxing the value restriction. In *International Symposium on Functional and Logic Programming.* Springer, 196–213.

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

Jacques Garrigue and Didier Rémy. 2013. Ambivalent types for principal type inference with GADTs. In *Programming Languages and Systems: 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings 11.* Springer, 257–272.

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* (Baltimore, Maryland, USA) *(MSFP '10)*. Association for Computing Machinery, New York, NY, USA, 43–54. https://doi.org/10.1145/1863597.1863608

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

Kazuki Ikemori, Youyou Cong, Hidehiko Masuhara, and Daan Leijen. 2022. Sound and Complete Type Inference for Closed Effect Rows. In *Trends in Functional Programming*, Wouter Swierstra and Nicolas Wu (Eds.). Springer International Publishing, Cham, 144–168.

Oleg Kiselyov. 2022. How OCaml type checker works – or what polymorphism and garbage collection have in common. (2022). https://okmij.org/ftp/ML/generalization.html

George Kuan and David MacQueen. 2007. Efficient type inference using ranked type variables. In *Proceedings of the 2007 workshop on Workshop on ML.* 3–14.

Konstantin Läufer and Martin Odersky. 1992. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications.* 78–91.

Daan Leijen. 2008. HMF: Simple type inference for first-class polymorphism. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming.* 283–294.

Daan Leijen. 2013. *Koka: Programming with Row-Polymorphic Effect Types.* Technical Report MSR-TR-2013-79. Microsoft. https://www.microsoft.com/en-us/research/publication/koka-programming-with-row-polymorphic-effect-types/

David MacQueen, Robert Harper, and John Reppy. 2020. The history of Standard ML. *Proc. ACM Program. Lang.* 4, HOPL, Article 86 (June 2020), 100 pages. https://doi.org/10.1145/3386336

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* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 54–67. https://doi.org/10.1145/237721.237729

Lionel Parreaux, Aleksander Boruch-Gruszecki, Andong Fan, and Chun Yin Chau. 2024. When Subtyping Constraints Liberate: A Novel Type Inference Approach for First-Class Polymorphism. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 1418–1450.

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.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. *ACM SIGPLAN Notices* 41, 9 (2006), 50–61.

Gordon Plotkin and John Power. 2001. Adequacy for algebraic effects. In *Foundations of Software Science and Computation Structures: 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001 Proceedings 4*. Springer, 1–24.

Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 18*. Springer, 80–94.

Didier Rémy. 1992. *Extension of ML type system with a sorted equation theory on types*. Ph. D. Dissertation. INRIA.

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

Jan Stolarek, Simon Peyton Jones, and Richard A Eisenberg. 2015. Injective type families for Haskell. *ACM SIGPLAN Notices* 50, 12 (2015), 118–128.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn (X) Modular type inference with local assumptions. *Journal of functional programming* 21, 4-5 (2011), 333–412.

Andrew K Wright. 1995. Simple imperative polymorphism. *Lisp and symbolic computation* 8, 4 (1995), 343–355.

Ningning Xie, Richard A Eisenberg, and Bruno C d S Oliveira. 2019. Kind inference for datatypes. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28.

Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. 53–66.

Jinxu Zhao, Bruno CDS Oliveira, and Tom Schrijvers. 2018. Formalization of a Polymorphic Subtyping Algorithm. *INTERACTIVE THEOREM PROVING, ITP 2018* 10895 (2018), 604–622.

Jinxu Zhao, Bruno C d S Oliveira, and Tom Schrijvers. 2019. A mechanical formalization of higher-ranked polymorphic type inference. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–29.