Contents

List of Tables

List of Figures

Chapter 1

Introduction

> Any sufficiently advanced technology is indistinguishable from magic.
>
> *Arthur C. Clarke*

For me, the Glasgow Haskell Compiler (GHC) [1] represented that "sufficiently advanced technology." Despite programming in Haskell for several years, I had not yet explored the internal workings of its primary compiler.

As one of the most advanced and widely-used compilers for a functional programming language, the Glasgow Haskell Compiler (GHC) serves as a benchmark for research and development in type systems. However, a key challenge in understanding systems like GHC is the pedagogical gap between academic theory and production code. While GHC's design is documented in seminal papers like "Practical Type Inference for Arbitrary-Rank Polymorphism" [2], these resources can be difficult for learners to approach. The academic literature is often theoretically dense, while GHC's source code is a large, highly-optimized production system, which can hide the core

algorithms.

This thesis aims to bridge this gap. It presents the design and implementation of a small, typed functional language called **Arralac** (**Ar**bitrary-**ra**nk polymorphism + **la**mbda **c**alculus). This project serves as a tutorial implementation of arbitrary-rank polymorphism. By using modern architectural patterns from GHC - such as the "Trees that Grow" AST representation and level-based scoping for unification - in a focused and simplified context, this work provides a clear path for understanding these advanced concepts.

The central thesis of this work is that a focused, well-documented implementation can serve as a more accessible learning tool for advanced type systems than studying the original papers or production compilers directly. To support this, the project includes a complete toolchain, featuring a parser, a constraint-based typechecker, and a Language Server Protocol (LSP) implementation for an interactive development experience.

To guide this work, the following research questions are addressed:

1. How can the core algorithms for type inference with arbitrary-rank polymorphism be implemented in a clear and step-by-step manner?

2. How can compiler implementation patterns from GHC, like "Trees that Grow" and level-based unification, be simplified for an educational setting while retaining their key benefits?

3. How can the Language Server Protocol be used to create an interactive development experience that helps in understanding a language's type system?

The resulting implementation is publicly available on GitHub [3] under

the MIT license to serve as a resource for the community, aiming to fill the observed gap in well-documented, GHC-like educational compilers.

1.1 Overview

This thesis is structured as follows:

Ch. 2 reviews existing approaches to building parts of the desired system. Subsequently, Ch. 3 explains my technical decisions and details the theoretical foundations. Ch. 4 evaluates the results obtained and discusses the development process and experience. Following this, Ch. 5 outlines potential directions for future work.

Chapter 2

Literature Review

2.1 Type Systems and the Challenge of Polymorphism

At the heart of many modern, statically-typed functional programming languages like Haskell and OCaml lies a sophisticated type system that is based on numerous scientific publications [4], [5]. A type system is a set of formal rules that assigns a property, known as a *type*, to various constructs of a computer program [6]. The primary purpose of a type system is to ensure program correctness by ruling out certain classes of errors at compile time, a property often referred to as *type safety*. If a program is *well-typed*, it is guaranteed to not suffer from runtime errors caused by misapplication of operations, such as adding an integer to a string.

2.1.1 Simply-Typed Lambda Calculus

The theoretical foundation for most functional programming languages is the **Lambda Calculus**, a formal system for expressing computation based on function abstraction and application. The most basic typed version, the **Simply-Typed Lambda Calculus** (**STLC**), introduces a set of rules for assigning types to terms, preventing runtime errors by ensuring functions are only applied to arguments of the correct type. Using the notation based on the one specifid in [7], the syntax of terms (`t`) and types (`T`) in a minimal STLC (using only booleans as a base type) can be defined as follows:

⟨*T*⟩ ::= '`Bool`'
    | ⟨*T*⟩ '`->`' ⟨*T*⟩

⟨*t*⟩ ::= ⟨*x*⟩
    | '`\`' '`(`' ⟨*x*⟩ '`:`' ⟨*T*⟩ '`)`' '`.`' ⟨*t*⟩
    | ⟨*t*⟩ ⟨*t*⟩
    | '`true`'
    | '`false`'
    | '`if`' ⟨*t*⟩ '`then`' ⟨*t*⟩ '`else`' ⟨*t*⟩

Fig. 2.1. Backus-Naur form for STLC

Here, `\(x : T). t` represents a function abstraction, where `x` is the parameter, `T` is the type of that parameter, and `t` is the function's body. Correspondingly, `t t` represents function application.

The primary contribution of the type system is to assign a type to every valid term. For example:

1. The term `\(x : Bool). if x then false else true` (the boolean `not` function) is assigned the type `Bool -> Bool`.

2. A higher-order function, such as `\(f : Bool -> Bool). f (f true)`, which takes a function and applies it twice, is assigned the type `(Bool -> Bool) -> Bool`

While STLC guarantees type safety through these rules, it is fundamentally **monomorphic**. A function can only have one specific type. For instance, an identity function must be written for a single, concrete type, such as `\(x : Bool). x`, which has the type `Bool -> Bool`. It is impossible to write a single, universal identity function that works for all types. This lack of generality makes STLC, in its pure form, impractical for writing large-scale, reusable software.

2.1.2 Polymorphism and System F

To address the limitations of monomorphic systems like STLC, the concept of **polymorphism** was introduced. Polymorphism allows a single piece of code to operate on values of multiple types. The foundational system that formally introduced this capability is Jean-Yves Girard's **System F** [8]. System F extends the lambda calculus with type abstraction and type application, allowing the creation of truly generic functions.

In System F, the identity function can be given the polymorphic type `forall a. a -> a`. The `forall` quantifier introduces a **type variable** `a`, which can later be instantiated with a concrete type as needed. This form of universal quantification is known as **parametric polymorphism**.

A crucial distinction in polymorphic systems is between **predicative** and **impredicative** polymorphism. In a **predicative system**, a quantified type variable (like `a` in `forall a. ...`) can only be instantiated with

a **monotype** — a type that does not itself contain any quantifiers. For example, in a predicative system, one can instantiate `a` with `Int` or `Bool`, but not with another polymorphic type like `forall b. b -> b`.

In contrast, an **impredicative system** allows a type variable to be instantiated with a **polytype**. This makes polymorphism 'first-class,' allowing polymorphic types to appear anywhere any other type can, such as inside a list type `[forall a. a -> a]`. While highly expressive, full type inference for impredicative systems is undecidable, and designing practical, predictable inference algorithms for them has been a long-standing research challenge [9], .

The work in this thesis, along with the foundational system for higher-rank types in GHC [2], operates within the simpler and more predictable predicative fragment of polymorphism.

2.1.3 The Practicality Problem: Hindley-Milner and Rank-1 Polymorphism

While System F is powerfully expressive, its impredicative nature makes full type inference undecidable [10]. This means it is impossible to create an algorithm that can always determine the types of a program without explicit annotations from the programmer. To make type inference both practical and automatic, languages like Haskell and ML adopted [2] a variant of a decidable system known as **Damas-Hindley-Milner** (HM) [11].

The HM system imposes two key restrictions on the polymorphism of System F:

1. It is predicative. As discussed in the previous section, type variables

can only be instantiated with monotypes.

2. It only supports **Rank-1** polymorphism. This means that `forall` quantifiers may only appear at the very outermost level of a type. A type such as `forall a. [a] -> Int` is a valid Rank-1 type, as the quantifier is on the outside.

A type where a quantifier is nested, particularly to the left of a function arrow, is known as a **higher-rank type**. For example, the type `(forall a. a -> a) -> Int` is a Rank-2 type and is forbidden in a standard HM system. This restriction ensures that type inference can be performed efficiently by algorithms like Algorithm W [2].

However, the Rank-1 restriction re-introduces a practical limitation. It becomes impossible to pass a polymorphic function as an argument to another function without it losing its polymorphic character. For a wide range of programs this is acceptable, but for expressing certain powerful programming patterns — such as operating on monadic structures or implementing generic traversals — it proves to be a significant constraint.

The primary goal of this thesis is to explore a practical extension beyond the standard HM system to support these more expressive, **arbitrary-rank types** within a predicative framework. This work follows the path laid out by Peyton Jones et al. in [2], a work that bridged the gap between the limitations of Rank-1 systems and the full, undecidable power of System F.

2.2 GHC's Type Inference

The GHC type inference engine, comprising approximately 50000 lines of code [12], has evolved incrementally over many years to support numerous extensions to the Haskell type system. Many of these extensions were accompanied by scientific publications [4].

2.2.1 Arbitrary-Rank Polymorphism

One foundational extension is `RankNTypes`. This extension enables parametric predicative arbitrary-rank polymorphism using a version of the bidirectional typing algorithm described in [2].

2.2.2 Bidirectional Typing

Bidirectional typing systems typically operate in two modes: an inference mode, that determines the type of a program construct and helps reduce the need for explicit type annotations, and a checking mode, which verifies top-down whether a program construct conforms to an expected type [13]. This second mode allows typing constructs (like lambda abstractions without annotated arguments in certain contexts) for which types cannot be uniquely inferred.

2.2.3 The French approach

Type checking in GHC is implemented using the **French approach**, as Peyton Jones calls it in his talk *Type Inference Using Constraints* [14]. This approach was described by French researchers Pottier and Rémy in [15].

When applied to Haskell, this approach implies the following main stages:

1. Parse the Haskell source program into an **Abstract Syntax Tree** (**AST**).
2. Traverse the AST and generate constraints using a bidirectional algorithm.
3. Identify "holes" - parts of typeable expressions with incomplete type information.
4. Solve constraints to obtain substitutions.
5. Substitute solutions (also known as **zonking**).
6. Elaborate the AST to include all available type information.
7. Report errors about residual constraints (constraints that could not be solved).

The main benefits of using this approach are as follows [14]:

1. Constraint generation is relatively easy, despite large Haskell syntax.
2. Although constraint solving is tricky, constraint language is extremely small and therefore easy to reason about.
3. The AST traversal order usually does not affect the results of solving constraints.
4. Elaborating the initial program is easy because constraint solver fills almost all holes.

5. Type error messages are generated after solving all solvable constraints and have helpful messages because most types are known.

2.2.4 Constraints

GHC's solver works with the following types of constraints [14]:

$$
\begin{array}{lll}
W & ::= \quad \epsilon & \text{(Empty constraint)} \\
& \mid \quad W_1, W_2 & \text{(Conjunction)} \\
& \mid \quad d : C\,\tau_1..\tau_n & \text{(Class constraint)} \\
& \mid \quad g : \tau_1 \sim \tau_2 & \text{(Equality constraint)} \\
& \mid \quad \forall\, a_1..a_n.\, W_1 \Rightarrow W_2 & \text{(Implication constraint)}
\end{array}
$$

Fig. 2.2. Simplified GHC constraints language

Here, d is an **evidence** of a class constraint (e.g., derived from an existing instance of a class) and g is an evidence of an equality constraint (e.g., derived from a `(b ~ c)` constraint in a function signature).

2.2.5 Implication Constraints

An implication constraint (Fig. 2.2) of the form $\forall\, a_1..a_n.\, W_1 \Rightarrow W_2$ states that the **wanted** constraint W_2 holds in the environment containing type variables $a_1..a_n$ when the **given** constraints W_1 hold in that environment.

Implication constraints were described in [15], where authors called these constraints *assumptions*. Later, Vytiniotis et al. [16] proposed OutsideIn(X) - a constraint-based approach for modular local type inference

that could work with implication constraints. Such approach enables type-checker to accept only programs where each function has a principal (the most general) type. A function type is inferred using the information about the function at its definition rather than call sites, thus making type inference faster. The authors implemented in GHC 7 a constraint solver for a combination of the following features: type classes, Generalized Algebraic Data Types (GADTs), and type families.

2.2.6 `let`-generalization

Consider the following Haskell program.

```
data B = B'
data C = C'
f x = let g y = (x, y) in (g B', g C')
```

Here:

- `B'` is a constructor of type `B`.
- `C'` is a constructor of type `C`.
- `f` is a top-level function with an argument `x`.
- `g` is a function introduced in a local `let`-binding inside the body of `f`.
- `(x, y)` is a pair of elements.
- `g 3` and `g False` are applications of `g` to values of type `B` and type `C`.

The type environment of `g` contains all variables with their types; these variables are introduced in enclosing `let`-bindings and top-level definitions. In this case, `f` and `x` are in the type environment of `g`.

If I disable the `MonoLocalBinds` extension, GHC will **generalize** `g`, i.e., infer the polytype `forall {b}. b -> (p, b)` for `g`. It decided to `quantify` over `b` because that type variable was not in the type environment. However, `{b}` (in braces) means that I can't instantiate `b` with a particular type using the `TypeApplications` extension.

On the other hand, if I disable the `MonoLocalBinds` extension, GHC will not generalize any `let`-binding that cannot be made a top-level function. Since the body of `g` contains `x` introduced in the definition of `f`, `g` cannot be a top-level function like `f`. Therefore, GHC will not generalize `g`. Instead, it will do roughly the following:

1. Introduce two fresh **unification variables**, `a` and `b`.
2. Record in the type environment that `y` has type `a` and `g` has type `a -> b`.
3. Discover that `g` is applied to a value of type `B`.
4. Generate a constraint that `a` $\sim$ `B`.
5. Discover that `g` is applied to a value of type `C`.
6. Generate a constraint that `a` $\sim$ `C`.
7. Try to solve the constraints.
8. Substitute `B` for `a`.

9. Find out that the constraint `B` $\sim$ `C` cannot be solved.

10. Report a type error that the argument of `g` has an unexpected type.

    - `Couldn't match expected type ‘B’ with actual type ‘C’`
    - `In the first argument of ‘g’, namely ‘C’`
      `In the expression: g C`
      `In the expression: (g B, g C)`

2.2.7 Levels

The **levels** technique originally introduced by Rémy [17] is a crucial, practical implementation detail used to manage the scope of type variables during type inference [18], [14], [19]. A level is essentially an integer assigned to each unification variable and skolem variable that tracks its depth at "birth". This mechanism is used to solve two major problems efficiently:

1. **`let`-generalization**. In a `let` binding (`let x = ... in ...`), the type of `x` can be generalized (made polymorphic). However, it can only be generalized over type variables that do not appear in the surrounding environment. The traditional way to check this is to traverse the entire type environment, which is inefficient. GHC uses levels to optimize this: a new, deeper (one more than the current one) level is entered for the body of the `let`, and only type variables at this new, deeper level are candidates for generalization. This avoids a full type environment scan. To get the variables to quantify over, GHC gathers and solve local constraints with the ambient level one

more than the level of `let`-binding (`Note [Inferring the type of a let-bound variable]`).

2. **Skolem escape checking**. When checking higher-rank types (e.g., checking if type τ_1 is a **subtype** of `forall a.` τ_2), the `forall`-bound variable `a` is turned into a "skolem" constant. This skolem must not "escape" its scope by being unified with a variable from an outer scope. GHC assigns the skolem a deeper level than the outer unification variables. The typechecker then enforces a simple rule: a variable at level `n` cannot be unified with a variable at a deeper level `m` (where `m` > `n`) or a type containing such a variable. This elegantly and efficiently prevents skolem escape.

2.3 A Survey of Type Inference Algorithms Beyond GHC

While studying GHC and [2], I reviewed several more recent approaches to supporting arbitrary-rank polymorphism. I primarily focused on the bidirectional algorithms as they were said to improve error locality and produce better error messages [13].

Several of these and other algorithms have publicly available Haskell implementations [20], [21], [22].

The following subsections highlight some key developments.

2.3.1 Bidirectional Typing

Dunfield and Krishnaswami [23] presented a relatively simple bidirectional type inference algorithm for systems with higher-rank predicative polymorphism. Their algorithm exhibits properties similar [23, Fig. 15] to those described in [2, Sec. 6], including adherence to the η-law [24, Ch. 4] and soundness and completeness with respect to System F [24, Ch. 8]. The authors argue that their formulation, using ordered contexts, existential variables, and precise scoping rules, offers a better type-theoretic foundation compared to the "bag of constraints," unification variables, and skolemization techniques employed in earlier work like [2].

Building on this, Dunfield and Krishnaswami [25] extended the approach to a significantly richer language featuring existential quantification, sums, products, and pattern matching. Their formal development utilizes a desugared core language [25, Fig. 11] derived from a more user-friendly surface language [25, Fig. 1].

Furthermore, Dunfield and Krishnaswami [13] provide an extensive survey of bidirectional typing techniques and offer practical guidance for designing new, programmer-friendly bidirectional type systems.

2.3.2 Modifications and Generalizations of Bidirectional Typing

Xie [26] review the algorithm from [23] (Sec. 2.3) and propose refinements, including adding an application mode alongside inference and checking (Sec. 3). They also present a novel algorithm for kind inference (Sec. 7) and compare their system to GHC's implementation (Sec. 8.6), identifying

potential areas for GHC improvement (Appendix Sec. C).

Xue and Oliveira [27] address limitations of traditional bidirectional typing (Sec. 2.5) by generalizing it to contextual typing. Instead of propagating only type information, this approach propagates arbitrary contextual information relevant to type checking. It replaces the binary inference/checking modes with counters that track the flow of contextual information. This allows for more fine-grained specification of precisely where programmer annotations are required.

2.3.3 Impredicativity

Parreaux et al. [28] propose SuperF, a novel, non-bidirectional type inference algorithm designed to support first-class (impredicative) higher-rank polymorphism via subtype inference. As mentioned in Sec. 2.1.2, impredicative polymorphism permits the instantiation of type variables with polytypes, whereas predicative polymorphism restricts instantiation to monotypes [2, Sec. 3.4]. SuperF infers a type for each subterm and then checks these against user-provided annotations written in System F syntax. The authors argue that the subtype inference employed by SuperF is better suited for implementing first-class polymorphism than approaches relying on first-order unification. As evidence of its expressive power, the authors demonstrate that SuperF can successfully type a wide variety of terms, often even without explicit type annotations (Sec. 5.4, Sec. 5.5). The algorithm implementation in Scala is available in the repository.

Chapter 3

Design and Implementation

The Hindley-Damas-Milner (HM) type system, foundational to languages like Haskell and ML, offers the remarkable property of complete type inference for rank-1 polymorphism. However, certain powerful programming patterns, such as passing polymorphic functions as arguments, require a more expressive system known as higher-rank polymorphism. The paper "Practical Type Inference for Arbitrary-Rank Types" by Peyton Jones et al. presents an elegant extension to the HM system that supports arbitrary-rank types in a practical manner. This chapter first outlines the key theoretical concepts from this paper and then describes our implementation, which is based on a constraint-generation and solving approach.

3.1 Theoretical Foundations

While full type inference for arbitrary-rank types is undecidable, the key insight of the paper is that programmers are willing to add type annotations to guide the inference engine. The goal is to create a system that requires minimal annotations while remaining sound, predictable, and not overly complex.

3.1.1 Type System Hierarchy: τ, ρ, and σ Types

To manage complexity, the type system is stratified into a three-level hierarchy. This stratification is crucial for defining the rules of the type system and its operations.

- **Tau types (τ)**: These are monomorphic types. They include type variables (like a), concrete types (like `Int`), and function types whose components are also τ-types. A τ-type contains no `forall` quantifiers.

  $$\tau ::= a \mid \texttt{Int} \mid \tau_1 \to \tau_2$$

- **Sigma types (σ)**: These are polytypes, which may have a top-level `forall` quantifier. They are the most general form of types in the system and are what we assign to top-level definitions.

  $$\sigma ::= \forall\, \bar{a}.\rho$$

- **Rho types (ρ)**: These are an intermediate form that lies between τ and σ. A ρ-type has no outermost `forall` quantifier but may con-

tain nested polytypes (σ-types). This structure allows for higher-rank types.

$$\rho ::= \tau \mid \sigma_1 \to \sigma_2$$

For instance, $\forall\, a.a \to a$ is a rank-1 σ-type. The type $(\forall\, a.a \to a) \to \texttt{Int}$ is a rank-2 ρ-type, which can be generalized to the σ-type $\forall\, b.((\forall\, a.a \to a) \to b) \to b$. The stratification ensures that operations like unification are restricted to the simpler τ-types, preserving decidability.

3.1.2 The Role of Metavariables

During type inference, the compiler often encounters expressions whose types are not yet fully known. To handle this, it introduces **metavariables** (also known as unification variables) as placeholders. These are distinct from the programmer-written type variables (like a, b). A metavariable, often written as α, β, stands for an unknown *monotype* (τ-type). As inference proceeds, constraints on these metavariables are collected and solved, typically through unification, to determine the actual type they represent. For example, in inferring the type of `\x -> x`, the type checker might assign x the type α and deduce the function's type to be $\alpha \to \alpha$.

3.1.3 Key Operations in Type Inference

The process of type inference relies on several core operations that manipulate these types.

Instantiation

Instantiation is the process of specializing a polymorphic type (σ-type) by replacing its quantified variables with fresh metavariables. It is the mechanism by which a polymorphic function is prepared for a specific use case.

$$\text{instantiate}(\forall\, a, b.a \to b) \Rightarrow \alpha \to \beta$$

Here, the quantified variables a and b are replaced with fresh metavariables α and β.

Skolemisation

Skolemisation is a more constrained form of specialization used when checking if a term conforms to a given polytype. Instead of replacing quantified variables with flexible metavariables, it replaces them with rigid **skolem variables** or constants. A skolem variable is a placeholder for a specific but unknown type that cannot be unified with any other type except itself.

$$\text{skolemise}(\forall\, a.a \to a) \Rightarrow s_a \to s_a$$

This is essential for the **skolem escape check**. A skolem variable must not ”escape” its scope, meaning it cannot appear in the type of a variable outside the expression being checked. This prevents unsound generalizations, such as inferring $\forall\, s.s \to \texttt{Int}$ from `id :: ` $\forall\, s.s \to s$.

Subsumption Checking

Subsumption is the core concept that allows higher-rank types to work. A type σ_1 is said to be subsumed by σ_2 (written $\sigma_1 \leqslant \sigma_2$) if σ_1 is more polymorphic than or equal to σ_2. This means any term of type σ_1 can be safely used where a term of type σ_2 is expected. For example:

$$(\forall\, a, b.a \to b) \leqslant (\forall\, c.c \to c) \leqslant (\texttt{Int} \to \texttt{Int})$$

The paper introduces an algorithm for checking subsumption, `subsCheck`, which operates recursively. Its key rules are:

- **DEEP-SKOL**: To check if $\sigma_1 \leqslant \sigma_2$, first skolemise the quantifiers in σ_2 to produce a ρ-type, ρ_2. This may involve floating quantifiers out from nested positions (e.g., from the right of an arrow).
- **SPEC**: Then, instantiate the quantifiers in σ_1 to match the structure of ρ_2.
- **FUN**: For function types $(\sigma_a \to \rho_b) \leqslant (\sigma_c \to \rho_d)$, the check proceeds contravariantly on the argument ($\sigma_c \leqslant \sigma_a$) and covariantly on the result ($\rho_b \leqslant \rho_d$).

3.1.4 Bidirectional Type Checking

To minimize the annotation burden, the paper employs a bidirectional type system with two modes of operation:

- **Inference Mode ($\uparrow$)**: Synthesizes or *infers* the most general type for an expression from the types of its sub-expressions. Denoted as

$\Gamma \vdash e \uparrow \sigma$.

- **Checking Mode ($\downarrow$)**: *Checks* if an expression conforms to an expected type. Denoted as $\Gamma \vdash e \downarrow \sigma$.

This duality is powerful. For instance, consider the expression `(\f -> f True) :: ( (a -> a) -> Bool)`. In a purely inferential system, `\f -> f True` would be inferred to have type $(\texttt{Bool} \to \alpha) \to \alpha$. The annotation would then fail to match. With bidirectional checking, the type checker enters checking mode. The annotation $(\forall\, a.a \to a) \to \texttt{Bool}$ provides the expected type for the lambda. The checker can then push the argument type, $\forall\, a.a \to a$, down as the expected type for `f`, successfully typing the expression without needing an annotation on `f` itself.

3.2 Implementation in Arralac

Chapter 4

Evaluation and Discussion

This chapter analyzes the research results.

Sec. 4.1 presents the main findings that are connected with the research purpose. Sec. 4.2 interprets how the research results support these findings. Sec. 4.3 contrasts my findings with the results of the past researches. Sec. 4.4 describes the limitations of my research. Sec. 4.5 suggests the possible applications of my research findings.

4.1 Key findings

...

4.2 Results and findings

...

4.3 Previous research

...

4.3.1 Subsection

...

4.4 Limitations

...

4.5 Potential applications

...

Chapter 5

Conclusion

...

Bibliography cited

[1] "The glasgow haskell compiler," Accessed: Apr. 27, 2025. [Online]. Available: https://www.haskell.org/ghc/.

[2] S. P. Jones, D. Vytiniotis, S. Weirich, and M. Shields, "Practical type inference for arbitrary-rank types," *J. Funct. Prog.*, vol. 17, no. 1, pp. 1–82, Jan. 2007, ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796806006034. Accessed: Apr. 3, 2025. [Online]. Available: https://www.cambridge.org/core/product/identifier/S0956796806006034/type/journal_article.

[3] D. Danko, *Deemp/arbitrary-rank-tutorial*, original-date: 2024-11-11T16:24:27Z, Apr. 27, 2025. Accessed: Apr. 27, 2025. [Online]. Available: https://github.com/deemp/arbitrary-rank-tutorial.

[4] "Research papers - type systems," Accessed: Jul. 8, 2025. [Online]. Available: https://wiki.haskell.org/index.php?title=Research_papers/Type_systems.

[5] "Ocaml papers," Accessed: Jul. 8, 2025. [Online]. Available: https://ocaml.org/papers.

[6] B. C. Pierce, *Types and Programming Languages*, 1st. The MIT Press, Jan. 2002, 645 pp., ISBN: 978-0-262-16209-8.

[7] B. C. Pierce et al. "Programming language foundations. "[Online]. Available: https://softwarefoundations.cis.upenn.edu/plf-current/index.html.

[8] J.-Y. Girard, "The system f of variable types, fifteen years later," *Theoretical Computer Science*, vol. 45, pp. 159–192, 1986, ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(86)90044-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397586900447.

[9] A. Serrano, J. Hage, S. Peyton Jones, and D. Vytiniotis, "A quick look at impredicativity," *Proc. ACM Program. Lang.*, vol. 4, 89:1–89:29, ICFP Aug. 3, 2020. DOI: 10.1145/3408971. Accessed: Apr. 9, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3408971.

[10] J. B. Wells, "Typability and type checking in system f are equivalent and undecidable," *Annals of Pure and Applied Logic*, vol. 98, no. 1, pp. 111–156, Jun. 30, 1999, ISSN: 0168-0072. DOI: 10.1016/S0168-0072(98)00047-5. Accessed: Jul. 8, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168007298000475.

[11] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '82, Albuquerque, New Mexico: Association for Computing Machinery, 1982, pp. 207–212, ISBN: 0897910656. DOI: 10.1145/582153.582176. [Online]. Available: https://doi.org/10.1145/582153.582176.

[12] S. P. Jones. "Secrets of the GHC typechecker in 100 type declarations," Simon Peyton Jones, Accessed: May 6, 2025. [Online]. Available: https://simon.peytonjones.org/secrets-of-typechecker/.

[13] J. Dunfield and N. Krishnaswami, *Bidirectional typing*, Nov. 14, 2020. DOI: 10.48550/arXiv.1908.05839. arXiv: 1908.05839. Accessed: Nov. 12, 2024. [Online]. Available: http://arxiv.org/abs/1908.05839.

[14] ACM SIGPLAN. "[WITS'24] solving constraints during type inference," Accessed: Jul. 8, 2025. [Online]. Available: https://www.youtube.com/watch?v=OISat1b2-4k.

[15] F. Pottier and D. Rémy, "10 the essence of ML type inference,"

[16] D. Vytiniotis, S. P. Jones, T. Schrijvers, and M. Sulzmann, "OutsideIn(x) modular type inference with local assumptions," *Journal of Functional Programming*, vol. 21, no. 4, pp. 333–412, Sep. 2011, ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796811000098. Accessed: Jun. 4, 2025. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/outsideinx-modular-type-inference-with-local-assumptions/65110D74CF75563F91F9C68010604329.

[17] D. Rémy, "Extension of ML type system with a sorted equation theory on types," Accessed: Jul. 8, 2025. [Online]. Available: https://inria.hal.science/inria-00077006/document.

[18] A. Fan, H. Xu, and N. Xie, "Practical type inference with levels," *Proc. ACM Program. Lang.*, vol. 9, pp. 2180–2203, PLDI Jun. 10, 2025, ISSN:

2475-1421. DOI: 10.1145/3729338. Accessed: Jun. 25, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3729338.

[19] ACM SIGPLAN. "[WITS'25] invited talk: Type inference in OCaml and GHC using levels," Accessed: May 20, 2025. [Online]. Available: https://www.youtube.com/watch?v=iFUrhTQi0-U.

[20] A. Goldenberg, *Artem-goldenberg/BidirectionalSystem*, original-date: 2024-11-23T14:06:35Z, Jan. 23, 2025. Accessed: Apr. 28, 2025. [Online]. Available: https://github.com/Artem-Goldenberg/BidirectionalSyst

[21] K. Choi, *Kwanghoon/bidi*, original-date: 2020-08-16T11:54:57Z, Jan. 3, 2025. Accessed: Apr. 28, 2025. [Online]. Available: https://github.com/kwanghoon/bidi.

[22] C. Chen, *Cu1ch3n/type-inference-zoo*, original-date: 2024-12-27T09:05:39Z, Apr. 26, 2025. Accessed: Apr. 28, 2025. [Online]. Available: https://github.com/cu1ch3n/type-inference-zoo.

[23] J. Dunfield and N. R. Krishnaswami, *Complete and easy bidirectional typechecking for higher-rank polymorphism*, Aug. 22, 2020. DOI: 10.48550/arXiv.1306.6032. arXiv: 1306.6032[cs]. Accessed: Apr. 2, 2025. [Online]. Available: http://arxiv.org/abs/1306.6032.

[24] P. Selinger, *Lecture notes on the lambda calculus*, Dec. 26, 2013. DOI: 10.48550/arXiv.0804.3434. arXiv: 0804.3434[cs]. Accessed: Apr. 28, 2025. [Online]. Available: http://arxiv.org/abs/0804.3434.

[25] J. Dunfield and N. R. Krishnaswami, "Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials

and indexed types," *Proc. ACM Program. Lang.*, vol. 3, 9:1–9:28, POPL Jan. 2, 2019. DOI: 10.1145/3290322. Accessed: Apr. 3, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3290322.

[26] N. Xie, "Higher-rank polymorphism: Type inference and extensions," [Online]. Available: https://i.cs.hku.hk/~bruno/thesis/NingningXie.pdf.

[27] X. Xue and B. C. D. S. Oliveira, "Contextual typing," *Proc. ACM Program. Lang.*, vol. 8, pp. 880–908, ICFP Aug. 15, 2024, ISSN: 2475-1421. DOI: 10.1145/3674655. Accessed: Apr. 4, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3674655.

[28] L. Parreaux, A. Boruch-Gruszecki, A. Fan, and C. Y. Chau, "When subtyping constraints liberate: A novel type inference approach for first-class polymorphism," *Proc. ACM Program. Lang.*, vol. 8, pp. 1418–1450, POPL Jan. 5, 2024, ISSN: 2475-1421. DOI: 10.1145/3632890. Accessed: Apr. 24, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3632890.

Appendix A

Extra code snippets

...