Автономная некоммерческая организация высшего образования
«Университет Иннополис»
(АНО ВО «Университет Иннополис»)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)
по направлению подготовки
09.04.01 – «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

GRADUATION THESIS
(MASTER GRADUATE THESIS)
Field of Study
09.04.01 – «COMPUTER SCIENCE»

Направленность (профиль) образовательной программы
«Программная инженерия»
Area of Specialization / Academic Program Title:
«Software Engineering»

Тема / Topic	**Руководство по реализации лямбда-исчисления с параметрическим предикативным полиморфизмом произвольного порядка / A Tutorial Implementation of a Lambda Calculus with Parametric Predicative Arbitrary-Rank Polymorphism**	
Работу выполнил / Thesis is executed by	**Данько Данила Константинович / Danko Danila Konstantinovich**	подпись / signature
Руководитель выпускной квалификационной работы / Graduation Thesis Supervisor	**Кудасов Николай Дмитриевич / Nikolai Kudasov**	подпись / signature
Консультанты / Consultants		подпись / signature

Иннополис, Innopolis, 2025

Contents

List of Tables

List of Figures

Abstract

Advanced type systems, such as arbitrary-rank polymorphism, are powerful but notoriously difficult to implement, creating a pedagogical gap between seminal theory, like that of Peyton Jones et al., and complex production compilers like the Glasgow Haskell Compiler (GHC). This thesis addresses this gap by presenting `Arralac`, a small, tutorial-focused compiler for a lambda calculus with arbitrary-rank polymorphism. Rather than replicating the eager unification algorithm of foundational papers, `Arralac` implements a modern, GHC-style, two-phase architecture that cleanly separates constraint generation from constraint solving, demonstrating a more modular and pedagogically clear approach. This design is supported by a Trees That Grow (TTG) Abstract Syntax Tree (AST), which enables pass-specific annotations crucial for tooling. The system is delivered as a complete, interactive toolchain, featuring a Language Server Protocol (LSP) implementation that makes the internal state of the typechecker transparent and explorable within a code editor. The implementation is validated by its ability to correctly handle higher-rank types, reject invalid programs through level-based skolem escape checks, and ultimately demonstrates that a constraint-based model serves as a clearer instructional tool than its eager counterpart. By synthesizing foundational theory with modern engineering patterns and interactive feedback, `Arralac` provides an accessible and effective resource that demystifies a cornerstone of advanced compiler design.

Chapter 1

Introduction

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke

1.1 Background and Motivation

Advanced type systems are a cornerstone of modern functional programming, enabling developers to write safer, more expressive, and more maintainable code. One such powerful feature is **arbitrary-rank polymorphism**, which allows functions to accept other polymorphic functions as arguments. This capability, while seemingly abstract, is fundamental to implementing advanced language features like GADTs, type-level programming, and sophisticated forms of generic programming.

The theoretical foundations for implementing this feature in a practical compiler were laid out in the seminal paper ”Practical Type Inference for Arbitrary-Rank Polymorphism" by Peyton Jones et al. [1]. The Glasgow Haskell Compiler (GHC) [2], the de facto standard compiler for the Haskell programming language, serves as the most prominent real-world implementation of these ideas. For many developers and language enthusiasts, including the author, the inner workings of GHC represent a pinnacle of compiler technology — a "sufficiently advanced technology" that can feel like magic.

1.2 The Problem: A Pedagogical Gap in Understanding Compiler Internals

Despite the existence of both foundational theory and a production-grade implementation, a significant pedagogical gap remains. Aspiring language developers and students face a steep learning curve when trying to understand how the elegant theory of arbitrary-rank polymorphism translates into practical code. This gap is not merely one of complexity, but of translation between different models of computation:

- The academic literature, including [1], presents a theoretically dense system built on the intricate dance of **subsumption**, **deep skolemization**, and **bidirectional checking**. While formally sound [3], the paper's description of an **eager unification** algorithm omits many of the practical engineering details required to build a robust system.

- The GHC source code, while an invaluable resource, is a large, highly-optimized production system. Its modern counterpart MicroHs [4] [5] is large too. Their current **constraint-based** type inference architecture is a significant evolution from the eager model described in the foundational papers, making it difficult for a newcomer to trace the connection between theory and implementation.

- Existing educational compilers for Haskell like Hugs [6] are outdated or do not incorporate these modern architectural patterns, leaving students without a clear bridge from first principles to the state of the art.

There is a clear need for a resource that bridges this gap — a ”middle ground” that is more concrete than a paper but more focused and accessible than a full production compiler, one that explicitly demonstrates the architectural evolution from eager to constraint-based inference.

1.3 Contribution: The `Arralac` Compiler

To address this pedagogical gap, this thesis presents the design and implementation of a small, typed functional language named `Arralac` (**Ar**bitrary-**ra**nk polymorphism + **la**mbda **c**alculus). The `Arralac` compiler (in the sense that it translates a source code into a target program [7]) serves as a well-documented, tutorial implementation that explicitly models the architectural principles of a modern typechecker for arbitrary-rank polymorphism.

The central thesis of this work is that a focused, modern, and interactive implementation that consciously diverges from older algorithmic models can serve as a more effective learning tool than studying foundational papers or production compilers in isolation.

The primary contributions of this thesis are:

1. **A Didactic Implementation of the Bidirectional Algorithm:** A clear, step-by-step implementation of the core type inference algorithms from [1], designed specifically to make the mechanism of the constraint-based checking using levels understandable.

2. **A Modern, Constraint-Based Architecture:** A deliberate architectural choice to implement a GHC-style, two-phase inference engine. This separates **constraint generation** from **constraint solving**, providing a modular and pedagogically superior alternative to the eager unification model.

3. **An Interactive Toolchain:** A complete language toolchain, featuring a parser, a typechecker, an evaluator, and a Language Server Protocol (LSP) implementation. The LSP makes the results of the type inference pipeline directly visible and interactive, transforming abstract rules into concrete feedback.

4. **A Public Repository:** A publicly available and permissively licensed codebase [8] to serve as a community resource for learning and experimentation.

1.4 Research Questions

The development of this thesis is guided by the following research questions:

1. How can the core algorithms of a bidirectional type system for arbitrary-rank polymorphism, including subsumption and deep skolemization, be implemented in a clear and modular manner for educational purposes?

2. How can a constraint-based inference model serve as a clearer pedagogical tool for explaining type inference than the eager unification model presented in the foundational literature?

3. How can the Language Server Protocol be leveraged to create an interactive development environment that makes the behavior and results of a language's type inference pipeline transparent and explorable?

1.5 Thesis Outline

This thesis is structured as follows:

- **Ch. 2** provides the theoretical foundations for this work by reviewing the evolution of polymorphic type systems. It traces the path from the Hindley-Milner compromise to the practical, bidirectional approach for arbitrary-rank types that forms the theoretical core of this thesis.

- **Ch. 3** details the design and implementation of the `Arralac` language. It covers the full compilation pipeline, with a special focus on the constraint-based architecture of the typechecker and the Trees That Grow AST representation.

- **Ch. 5** evaluates the resulting system against its pedagogical goals, analyzing the trade-offs of its architectural decisions and discussing the insights gained during development.
- **Ch. 6** summarizes the contributions of this work, revisits the central thesis in light of the results, and outlines potential directions for future research and development.

Chapter 2

Literature Review

2.1 The Foundations of Typed Functional Languages

At the heart of modern, statically-typed functional languages like Haskell and OCaml lies a sophisticated type system [9], [10], a collection of formal rules assigning a property, or **type**, to every construct in a program [11]. A well-typed program is guaranteed to be free from a large class of runtime errors, such as applying an arithmetic operation to a string, a crucial property known as **type safety**. The theoretical basis for these languages is the **Lambda Calculus**, a formal system for expressing computation through function abstraction and application.

The most foundational typed variant is the **Simply-Typed Lambda Calculus (STLC)**. It ensures type safety by requiring every function parameter to be explicitly annotated with a type, and it verifies that functions are only ever applied to arguments of the matching type [12]. For example, the boolean `not` function, `\(x : Bool). if x then false else true`, is correctly assigned the type `Bool -> Bool`.

However, STLC's safety comes at the cost of expressivity. The system is fundamentally **monomorphic**: each function can only have one, specific type. An identity function, for instance, can be written for booleans (`\(x : Bool). x`) or for integers (`\(x : Int). x`), but a single, universal identity function that works for *all* types is impossible to express. This inherent rigidity makes pure STLC impractical for building reusable, large-scale software and creates a fundamental need for more expressive polymorphic systems.

2.2 System F: The Ideal of Polymorphism and its Practical Limits

To overcome the limitations of monomorphism, modern languages rely on **polymorphism**, which 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** [13]. System F extends the lambda calculus with type abstraction and type application, enabling the creation of truly generic functions through what is known as **parametric polymorphism**. In System F, a universal identity function can be written with the polymorphic type `forall a. a -> a`, where the `forall` quantifier introduces a **type variable** `a` that can be instantiated with any concrete type at the function's call site.

Within such powerful systems, a crucial distinction exists between predicative and impredicative polymorphism. In a **predicative** system, a quantified type variable (like `a`) can only be instantiated with a **monotype** — a simple type that does not itself contain any `forall` quantifiers. In contrast, a more powerful **impredicative** system allows a type variable to be instantiated with another **polytype**. While this ”first-class” polymorphism is highly expressive, full type inference for impredicative systems is undecidable [14], [15]. This fundamental trade-off between expressive power and decidability directly informs the scope of this thesis, which, following the pragmatic tradition of GHC [1], operates within the simpler and more predictable predicative fragment.

2.3 Practical Polymorphism: The Hindley-Milner Compromise and the Rank-N Gap

While System F provides the theoretical ideal for polymorphism, its power makes it impossible to create an algorithm that can always infer a program’s types without explicit annotations from the programmer. To make type inference both practical and automatic, languages like ML adopted a decidable and well-behaved subset of System F known as the **Damas-Hindley-Milner (HM)** type system [16].

The practicality of the HM system stems from a foundational compromise: it only supports **Rank-1** types. This restriction means that `forall` quantifiers are only permitted at the very outermost level of a type definition. For example, `forall a. a -> Int` is a valid Rank-1 type, but a type where the quantifier is nested, such as `(forall a. a -> a) -> Int`, is known as a **higher-rank type** (specifically, a Rank-2 type) and is forbidden in a standard HM system.

This Rank-1 restriction is the key to the system's tractability, ensuring that type inference is decidable and can be implemented efficiently by algorithms like Algorithm W [1]. However, this compromise simultaneously introduces a significant expressive limitation: the "Rank-N gap". It becomes impossible to pass a polymorphic function as an argument to another function, as doing so would require a higher-rank type. This "Rank-N gap" represents the central challenge that must be overcome to support many powerful functional programming idioms, motivating the development of more sophisticated, yet still practical, type systems.

2.4 A Practical Solution for Higher Ranks

The definitive solution to the Rank-N gap within a practical compiler context was presented in *Practical type inference for arbitrary-rank types* by Peyton Jones et al. [1]. This seminal paper provides the direct theoretical and architectural foundation for this thesis. Rather than attempting full, undecidable inference, the authors propose a system that extends the predictable Hindley-Milner framework by cleverly leveraging programmer-supplied type annotations to guide the typechecker.

The core contribution is a **bidirectional type inference algorithm**. The system operates in two modes: an **inference mode** that synthesizes the most general type for an expression, and a **checking mode** that verifies an expression against a known, expected type. This duality is the key to its practicality: when a higher-rank type is required (e.g., as a function argument), programmer annotations are used to switch the algorithm into the more constrained checking mode, thereby avoiding the need for full inference while still enabling the use of polymorphic arguments.

To correctly handle the ”more polymorphic than” relationship between types, the algorithm introduces a robust **subsumption** check. The mechanical heart of this check is a technique called **deep skolemization**, which correctly and efficiently handles quantifiers nested within function types. The detailed mechanics of this algorithm, including the type hierarchy (σ, ρ, τ), the monotype invariant for unification, and the contravariant handling of function arguments, will be presented in Ch. 3 as they form the direct basis for the implementation in this thesis.

2.5 The Modern Landscape of Bidirectional Typing

The bidirectional system from [1] provides the foundation for this thesis, but the field has continued to evolve. This section surveys key subsequent developments, contextualizing the chosen approach and further justifying its suitability for a pedagogical project. The continued interest in bidirectional typing is evidenced by numerous publicly available implementations, which serve as valuable resources for researchers and students alike [17], [18], [19].

2.5.1 The Rise of Formal Bidirectional Systems

A significant thread of modern research has focused on placing bidirectional typing on a more rigorous formal foundation. Dunfield and Krishnaswami [20] presented a declarative, bidirectional algorithm for higher-rank predicative polymorphism, proving its soundness and completeness with respect to System F [21]. This foundational work, extended by Dunfield and Krishnaswami in [22] and surveyed in [23], provides a clean type-theoretic basis using ordered contexts, in contrast to the more operational, constraint-based techniques of GHC.

2.5.2 Refining and Generalizing Bidirectional Typing

Subsequent research has focused on refining the bidirectional model. Xie [24] proposed refinements to the algorithm from [20], while more recently, Xue and Oliveira [25] generalized the approach to **contextual typing**, allowing for more fine-grained propagation of type information.

2.5.3 Exploring Alternatives: Impredicativity and Subtyping

In parallel, another line of research has explored supporting full, first-class impredicativity. Parreaux et al. [26] propose **SuperF**, a novel algorithm that supports impredicativity by replacing unification with **subtype inference**[1]. While extremely powerful, this approach represents a significant departure from the unification-based systems that are the focus of this thesis.

[1]The SuperF implementation can be found at `https://github.com/hkust-taco/superf`.

2.5.4 Justification of the Chosen Approach

This survey confirms that while many innovative approaches exist, the practical, predicative, and unification-based system pioneered in [1] remains the most suitable foundation for this project. The formal systems of Dunfield et al. are elegant but are further removed from the concrete implementation techniques (like constraint solving and mutable metavariables) used in GHC. The subtyping-based approach of SuperF is a different paradigm entirely.

Therefore, for the pedagogical goals of this thesis — to create a tutorial implementation that demystifies a production-grade approach — basing the work on the established and practical foundation of Jones et al. is the most direct and effective strategy. It provides a clear path to understanding the core challenges of higher-rank types and the architectural patterns used to solve them in a real-world compiler.

Chapter 3

Design and Methodology

This chapter details the architectural design and core methodologies employed in the implementation of `Arralac`, a tutorial compiler for a lambda calculus with arbitrary-rank polymorphism. The design is heavily inspired by the bidirectional type inference system described in *Practical type inference for arbitrary-rank types* [1], but it incorporates several modern implementation techniques and diverges in key areas to prioritize clarity and extensibility. This chapter will first outline the overall system architecture, then justify the choice of Abstract Syntax Tree (AST) representation, and finally, delve into the specifics of the constraint-based type inference engine and the methodology used to validate its correctness.

3.1 System Architecture: The Compilation Pipeline

The process of transforming a source file from plain text into an evaluated term is managed by a multi-stage pipeline, depicted in Fig. 3.1. Each stage performs a distinct transformation on the program representation, passing its output to the next. This standard pipeline structure [27] was chosen to promote modularity and a clear separation of concerns, which are essential for creating an understandable and maintainable tutorial compiler.

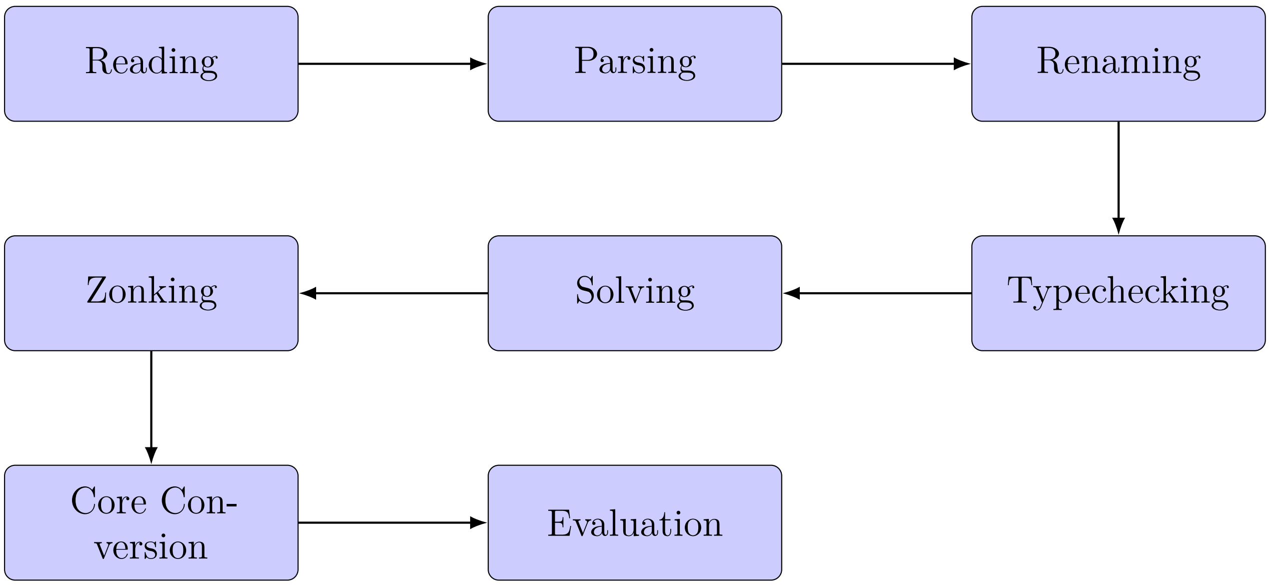


Fig. 3.1. The `Arralac` Compilation Pipeline.

The stages are as follows:

- **Reading & Parsing:** The source text is read and transformed by a BNFC-generated parser [28] into an initial AST, with each node annotated with its source location.

- **Renaming:** This stage performs α-conversion, assigning a unique identifier to every variable binding to resolve shadowing and prepare the program for typechecking.

- **Typechecking (Constraint Generation):** The renamed AST is traversed to generate a set of type constraints. This pass gathers equality and implication constraints but does not solve them.

- **Solving:** A separate iterative solver pass attempts to unify variables with types in the generated constraints, performing skolem escape and occurs checks.
- **Zonking:** After solving, a final pass substitutes all solved metavariables with their inferred types, producing a fully-typed AST.
- **Core Conversion:** The final, zonked AST is converted into a simpler Core language.
- **Evaluation:** The term in that Core language is evaluated to its weak head normal form (WHNF).

3.2 Abstract Syntax Tree Design

The choice of AST representation is a foundational design decision. My design was guided by several key technical constraints: extensibility for future language features, the ability to annotate nodes for tooling, and smooth integration with the BNFC parser generator.

3.2.1 Evaluating AST Representation Strategies

Several alternatives were considered to meet these constraints. While the default AST generated by BNFC is simple and integrates perfectly with the parser, it lacks the necessary structural flexibility for a multi-pass compiler because it is too close to the concrete language syntax.

Libraries like `hypertypes` [29] and `compdata` [30] offer powerful approaches to building extensible ASTs, serving as a solution to the Expression

```
-- In GHC (compiler/Language/Haskell/Syntax/Expr.hs)
data HsExpr p
  = HsVar (XVar p) (LIdP p)
   ↪   -- LIdP is a type synonym, not a type family
  ...

type LIdP p = XRec p (IdP p)
```

Fig. 3.2. GHC's TTG AST Structure (Simplified)

Problem [31]. However, they introduce a significant degree of a conceptual overhead that was deemed counterproductive for a tutorial project aimed at clarifying compiler internals.

Therefore, the **Trees That Grow (TTG)** pattern was selected [32]. It strikes an ideal balance, offering the extensibility needed for a phased compiler while remaining conceptually straightforward. It directly models the evolution of the AST through different stages, making it a powerful pedagogical tool in its own right and mirroring the architecture of GHC [33].

My implementation of TTG differs slightly from GHC's. Whereas GHC uses concrete types in some fields, my implementation uses type family applications for *all* fields, ensuring that every part of an AST node can be customized for a given compiler pass. Another difference is that I did not yet introduce an additional constructor for extending data types defined in the TTG representation because I did not expect usage of these types outside of the `Arralac` compiler.

This uniform use of type families provides a type-safe guarantee that pass-specific information, such as inferred types, is only available in the AST after that pass has successfully completed.

```
-- In Arralac (Language/Arralac/Syntax/TTG/SynTerm.hs)
data SynTerm x
  = SynTerm'Var (XSynTerm'Var' x) (XSynTerm'Var x)
   ↪  -- All fields use type families
  ...

-- Type families for each field, defined separately
type family XSynTerm'Var' x
type family XSynTerm'Var x
```

Fig. 3.3. Arralac's TTG AST Structure

3.3 Type Inference and Constraint Solving

The design of the `Arralac` typechecker required a concrete implementation of several core concepts from the bidirectional system of [1]. This section outlines these foundational mechanisms that the methodology must realize and then describes the significant architectural divergence taken in `Arralac`.

3.3.1 Foundations from *Practical Type Inference*

The design of `Arralac`'s typechecker, while architecturally distinct, is a direct evolution of the bidirectional system presented in *Practical type inference for arbitrary-rank types* [1]. To fully appreciate the design choices made in `Arralac`, it is essential to first understand the foundational mechanisms of this influential system, which elegantly balances expressive power with practical, decidable inference.

Type Hierarchy and Context.

At its core, the system stratifies types into a precise hierarchy to manage polymorphism. The type context, denoted by Γ, maps program variables to their types.

- **Monotypes (τ-types):** These are simple types with no polymorphism whatsoever, such as `Int` or `Bool -> Int`. They form the bedrock of the system.
- **Polytypes (σ-types):** These are types that begin with a `forall` quantifier, such as `forall a. a -> a`. They introduce polymorphism.
- **Rho-types (ρ-types):** These are an intermediate category, representing types that can appear on the right-hand side of a function arrow or under a `forall`. In the full higher-rank system, a ρ-type can be either a τ-type or a function type $\sigma_1 \to \sigma_2$.

This stratification is crucial for controlling where and how polymorphism can appear and be inferred.

Metavariables and the Monotype Invariant.

During inference, the typechecker encounters expressions whose types are not yet known. It handles this by creating placeholder types called **metavariables** (e.g., α, β). These are temporary, flexible variables that are gradually refined as the algorithm gathers more information. The process of solving for these metavariables is called **unification**.

A cornerstone of the system's practicality is the **monotype invariant**: a metavariable can only ever be unified with a monotype (τ-type). This restriction is what guarantees that type inference is decidable and, crucially, **order-independent**. If a metavariable could be unified with a polytype, the result of inference could depend on the order in which expressions were traversed, leading to unpredictable and non-deterministic behavior. By enforcing this invariant, the algorithm ensures that the principal type it infers is unique and consistent.

The Bidirectional Algorithm.

The master strategy for handling higher-rank types is the **bidirectional type checking** algorithm. Instead of a single, monolithic inference function, the system operates in two distinct modes, preventing the need for full, undecidable inference:

- **Inference Mode ($\uparrow$):** This mode synthesizes or ”infers” the most general type for an expression from the bottom up. In the paper's implementation, this corresponds to functions like `inferRho` and `inferSigma`. It is used for expressions where the type is not already known.
- **Checking Mode ($\downarrow$):** This mode verifies or ”checks” an expression against a known, expected type that is pushed down from the surrounding context. This is implemented by functions like `checkRho` and `checkSigma`.

This duality is key. For most of the program, the system can infer types. However, when it encounters a programmer-supplied annotation for a higher-rank type (e.g., on a function argument), it switches to the more constrained checking mode, using the annotation to guide the process and avoid intractable search.

Subsumption: The "More Polymorphic Than" Check.

A central challenge is determining if a function can be used where a more-specific polymorphic type is expected. This is handled by a **subsumption** check, which formalizes the notion of one type being "more polymorphic than" another. In the paper's implementation, this is the role of the `subsCheck` function. The mechanics of subsumption rely on two dual concepts: instantiation and skolemization.

- **Instantiation:** When *using* a polymorphic value (e.g., calling a function bound to a σ-type), its quantified type variables are replaced with fresh metavariables. This process, handled by `instantiate`, makes the type usable in a specific context.

- **Deep Skolemization:** The more complex case is checking against a polymorphic type `forall a. T`. To do this, the algorithm replaces the quantified variable `a` with a new, rigid constant called a **skolem**. A skolem constant is unique and cannot be unified with any other type. If a skolem constant "escapes" its scope (**skolem escape**) during subsequent unification (e.g., by being unified with a metavariable from an outer scope), a type error is raised.

  For arbitrary-rank types like `T1 -> (forall a. T2)`, the `forall` is nested. **Deep skolemization** is the paper's crucial innovation to handle this. It first transforms the type into a **weak prenex form**

$$pr(\forall\, c.(\forall\, a.a \to c) \to (\forall\, b.b \to c)) =$$
$$= \forall\, bc. \to (\forall\, a.a \to c) \to b \to c$$

Fig. 3.4. Transforming a polytype into a weak-prenex form

using an auxiliary function, $pr(\sigma)$, which pulls nested quantifiers from the return types of functions to the outermost level. For example,

This makes all quantifiers accessible for skolemization, enabling a consistent and robust subsumption check.

Finally, when checking function types for subsumption, such as $(\sigma_1 \to \sigma_2) \leq (\sigma_3 \to \sigma_4)$, the algorithm respects **contravariance**. The argument types are checked in the opposite direction ($\sigma_3 \leq \sigma_1$), while the result types are checked in the same direction ($\sigma_2 \leq \sigma_4$).

These mechanisms — a strict type hierarchy, the monotype invariant, bidirectional checking, and subsumption check via deep skolemization — work in concert to create a system that is powerful enough to support higher-rank types yet constrained enough to remain practical and decidable. This is the foundational system that `Arralac` reinterprets through a modern, constraint-based lens.

3.3.2 Architectural Divergence: A Constraint-Based Model

While the theoretical underpinnings are rooted in [1], the implementation of the inference mechanism diverges significantly. Instead of the **eager unification** described in the paper, where constraints are solved as they are discovered, `Arralac` adopts a **constraint-based** approach inspired by GHC. This architectural choice was made to enhance modularity and to lay a more robust foundation for high-quality error diagnostics, reflecting modern compiler practice. Table 3.1 summarizes the key differences.

TABLE 3.1
Comparison of Type Inference Architectures

Feature	**[1] (Eager Unification)**	**`Arralac` (Constraint-Based)**
Unification	Solves constraints immediately as they are generated.	Gathers all constraints first; solves them in a separate, dedicated pass.
Modularity	Inference logic is tightly coupled with unification logic.	The Typechecker (generation) and Solver (unification) are fully decoupled modules.
Error Reporting	Errors are reported at the first point of unification failure, which can be obscure.	Already reports the location of a sub-term where a constraint originated. Has the potential for holistic error analysis by examining all conflicting constraints at once.
Let-Generalization	Requires a global type context traversal at the point of the `let`-binding.	Not implemented, but the architecture permits solving of scoped constraints during type checking.

This separation is realized through two primary constraint types:

1. **Canonical Equality Constraints (`EqCt`):** Represent a required equality between a metavariable and a type.

2. **Implication Constraints (`Implication`):** Capture the scoping of polymorphism. An implication bundles a set of skolem variables created at the ambient level with **wanted** [27] constraints generated from checking an expression after incrementing the ambient level.

Furthermore, `Arralac` manages polymorphism scoping using **levels**, a technique also used in modern compilers [34]. Every variable is assigned an integer `TcLevel` at creation. The solver then enforces the skolem escape check by a simple rule: a metavariable at level n cannot be unified with a type containing any free variable (skolem or a metavariable) from a level $m > n$ [1].

3.4 Validation Methodology

To validate the correctness of the implementation and ensure it meets its pedagogical goals, a methodology of targeted, feature-driven testing was employed. Rather than relying on a large, undifferentiated test suite, specific test cases were developed to exercise the core mechanisms of the arbitrary-rank type system and its implementation.

1. **Correct Handling of Higher-Rank Polymorphism:** The primary validation case involved a program that requires passing a polymorphic

[1]See Note [Unification preconditions]

function as an argument, similar to `Program1.arralac` (Sec. 4.3). The successful typechecking and evaluation of this program served as the baseline validation that the core bidirectional algorithm, including subsumption and deep skolemization, was implemented correctly.

2. **Skolem Escape and Level Checking:** A specific negative test case was created to ensure that a skolem variable could not escape its scope during unification. The test involved attempting to unify a metavariable with a type containing a skolem from a deeper scope. The expected outcome was a type error from the solver, which validated that the level-based checking mechanism was correctly preventing unsound unifications.

3. **Language Server Functionality:** The interactive tooling was validated manually within Visual Studio Code. This involved checking that (a) hovering over identifiers displayed the correct, fully-zonked types as inferred by the pipeline, and (b) introducing syntactic or semantic errors (e.g., unbound variables) triggered appropriate and timely error diagnostics from the language server.

While this methodology does not constitute a formal proof of correctness, it provides enough evidence that the key features of the system are implemented correctly and robustly, satisfying the primary objectives of the thesis.

3.5 Summary of Design Choices and Limitations

The design of `Arralac` makes several deliberate trade-offs to prioritize its tutorial nature and extensibility over feature completeness. The key design choices were:

- An extensible **Trees That Grow** AST to support future language features and tooling annotations.
- A GHC-style, two-phase **constraint-based type inference** engine, which separates constraint generation from solving.
- **Level-based scoping** for skolem variables, providing an efficient mechanism for escape checking.

This design, however, comes with several limitations compared to a production compiler. The accompanying Technical Appendix to the paper [3] provides proofs of soundness and completeness for the theoretical system, but this implementation does not attempt to formally prove its own correctness. The most notable limitations are:

- **No `let`-generalization:** Local `let`-bindings are not generalized, a simplification suggested in [35].
- **No recursive `let`-bindings.** just like in [1].
- **No floating-out of constraints:** The solver does not attempt to move equality constraints out of implications to enable further solving.

- **Untyped Core Language:** Unlike GHC and [1], `Arralac` translates `SynTerms` to a simple, untyped Core language, forgoing the powerful consistency checks that a typed intermediate language provides. This simplification was a deliberate trade-off to keep the focus of the thesis squarely on the front-end type inference algorithm.

- **Simplified Constraint Solving:** The solver does not rewrite Wanteds with Wanteds [2] and only reports the first constraint that it could not solve, not all residual constraints.

[2]See Note [Wanteds rewrite Wanteds]

Chapter 4

Implementation and Results

The previous chapter detailed the architectural design and core methodologies for the `Arralac` compiler. This chapter transitions from design to practice, describing the concrete Haskell implementation that realizes this architecture.

First, I will examine the implementation of the core data structures, particularly the Trees That Grow (TTG) Abstract Syntax Tree (AST). Next, I will detail the type inference pipeline, focusing on the separation between constraint generation and solving. Finally, I will present concrete results, including the output for both successful and unsuccessful typechecking scenarios, and a demonstration of the functional Language Server Protocol (LSP) features.

```
data SynTerm x
  = -- | \ (x :: a). x
    SynTerm'ALam (XSynTerm'ALam' x) (XSynTerm'ALam'Var x)
                 (XSynTerm'ALam'Type x)
                     (XSynTerm'ALam'Body x)
  | -- | (f x) :: Int
    SynTerm'Ann (XSynTerm'Ann' x) (XSynTerm'Ann'Term x)
                (XSynTerm'Ann'Type x)
  -- ... other constructors
```

Fig. 4.1. The Generic `SynTerm` Data Type in `Language.Arralac.Syntax.TTG.SynTerm`

4.1 AST Implementation with Trees That Grow

As outlined in the design (Sec. 3.2), the AST is built using the Trees That Grow (TTG) pattern to ensure extensibility and to facilitate annotations. The core data types, `SynTerm` for expressions, `SynType` for syntactic types, and `Type` for non-syntactic types are parameterized by a type variable `x` that represents the current compiler pass (e.g., `CompRn` for Renamed, `CompTc` for Typechecked, `CompZn` for Zonked).

Unlike a simpler AST where fields have concrete types, every component of an `Arralac` AST node is defined by a type family. This allows the structure of a node to be radically different across passes. The generic definition for `SynTerm` is shown below.

The power of TTG is realized through the instantiation of these type families for each specific compiler pass. For the typechecking pass (`CompTc`), the extension point fields (e.g., `XSynTerm'Ann'`) are instantiated with a

```haskell
-- In Language.Arralac.Syntax.Local.SynTerm.Tc
type instance XSynTerm'Ann'Term CompTc = SynTerm CompTc
type instance XSynTerm'Ann'Type CompTc = SynType CompTc

-- In Language.Arralac.Syntax.Local.Extension.Tc
-- The AST node is annotated with its type
-- after typechecking.
type instance XSynTerm'Ann' CompTc = TcAnno
data TcAnno = TcAnno { annoSrcLoc :: SrcSpan, annoType ::
 ↪   Expected TcType }
```

Fig. 4.2. Type Family Instantiation for the Typechecked Pass

`TcAnno` record. This record is crucial for tooling, as it contains the inferred type for that node, which is later used by the language server.

This approach provides a compile-time, type-safe guarantee that pass-specific information is only available in the AST after that pass has successfully completed.

4.2 The Type Inference and Solving Pipeline

The implementation of the type inference engine closely follows the constraint-based design laid out in Sec. 3.3. The process is divided into distinct, sequential stages, each managed by its own set of modules.

4.2.1 Constraint Generation (`Typechecker`)

The first phase, implemented primarily in the `Language.Arralac.Typechecker.TcTerm` module, traverses the renamed AST to produce a set of wanted constraints. The `unify` function merely emits canonical equality constraints with a metavariable on the left-hand side and a monotype on the right-hand side.

- **Bidirectional Logic:** The core function, `tcRho`, implements the bidirectional algorithm. When called in inference mode (via the `inferRho` wrapper), it is passed a new mutable reference (`Expected (IORef TcType)`) to hold the resulting type. When called in checking mode (via `checkRho`), it consumes the expected type passed to it.
- **Implication Constraints:** At each point where skolemization is required — specifically, within the `checkSigma` function when checking an expression against a polymorphic type — the typechecker enters a deeper scope. This is implemented by the helper function `pushLevelAndCaptureConstraints`. This function increments the current `TcLevel`, captures all new constraints generated within its scope, and packages them into an `Implic` constraint, which bundles the new skolem variables created at the current level with the constraints at a deeper level in which these skolems may occur. This directly models the scoped nature of polymorphism.

4.2.2 Constraint Solving (`Solver`)

The set of `WantedConstraints` generated by the typechecker is passed to the solver, implemented in `Language.Arralac.Solver.Solve`. The solver iteratively processes a worklist of simple equality constraints (`wc_simple`).

- **Occurs and Level Checks:** Before attempting to unify a metavariable with a type, the solver performs two critical checks implemented in

`Language.Arralac.Solver.Check`. First, an **occurs check** ensures the metavariable does not appear within the right-hand side of the type, preventing infinite types like $a\,a \to a$. Second, a **level check** verifies that the type on the right-hand side does not contain any skolem variables with a level deeper than the metavariable. This check is the concrete implementation of the skolem escape rule discussed in the design chapter.

- **Unification:** If the checks pass, the solver unifies the variable by writing to the metavariable's mutable `IORef`. Constraints within implications are solved recursively within their own, deeper-level scope. Any constraints that cannot be solved (e.g., due to a type mismatch or a failed check) trigger an exception. When a metavariable contains a type, the `unify` function (defined in `Solver.Unify`) tries to generate new constraints by equating the contained type and the type on the right-hand side of the constraint. Therefore, the solver may need several iterations to solve constraints.

4.2.3 Finalization (`Zonker`)

After the solver has completed, the `Zonker`, implemented in `Language.Arralac.Zonker.Zn.Zonk`, traverses the now-typed AST. It recursively resolves the `IORef` of each metavariable, replacing it with its final, unified fully zonked type. Metavariables that were not solved are explicitly renamed to indicate their status (e.g., `a_Unsolved_11`), producing a final AST that is free of mutable references and ready for code generation or evaluation.

```
let
  applyMyShow =
    (\x. \y. x y)
      :: forall a. (forall b. b -> String) -> a -> String
in
let
  myShow = \x. "Hello"
in
applyMyShow myShow
```

Fig. 4.3. `Program1.arralac`

4.3 Results and System in Action

This section demonstrates the functionality of the implemented system using both positive and negative test cases, validating the core mechanisms of the type system.

4.3.1 A Positive Case: Higher-Rank Polymorphism

To test the correct handling of higher-rank types, I use a program that requires passing a polymorphic function as an argument. The following program, `Program1.arralac`, defines a function `applyMyShow` that expects a polymorphic function of type `forall b. b -> String`.

Running the `Arralac` CLI to typecheck the program produces a fully-annotated AST. The output in Fig. 4.4 shows that the system correctly inferred and propagated the types. Crucially, the lambda-bound variable `x_1` is correctly assigned its type from the annotation containing a higher-rank type, and the final type correctly reflects that the type variable `a` was unconstrained and thus remains unsolved.

```
$ nix run .#arralac -- typecheck arralac/test/data/Program1.arralac

(let
  -- Correct higher-rank type for applyMyShow
  applyMyShow_0 =
    (\(x_1 :: forall b_4. b_4 -> String).
       (\(y_2 :: a_9). (x_1 :: a_9 -> String) (y_2 :: a_9) :: String
       ) :: a_9 -> String
    ) :: (forall b_4. b_4 -> String) -> a_Unsolved_11 -> String
in
  (let myShow_7 = (\(x_8 :: b_13). "Hello") :: b_13 -> String
   in (applyMyShow_0 (myShow_7)) :: a_Unsolved_11 -> String
  ) :: a_Unsolved_11 -> String
) :: a_Unsolved_11 -> String
```

Fig. 4.4. Typechecking output for `Program1.arralac` (compacted).

The evaluator correctly reduces the program to its weak head normal form (WHNF), as shown in Fig. 4.5.

```
$ nix run .#arralac -- evaluate whnf
 ↪   arralac/test/data/Program1.arralac

\y_2. (\x_8. "Hello") (y_2)
```

Fig. 4.5. Evaluation output for `Program1.arralac`.

4.3.2 A Negative Case: Skolem Escape Detection

To validate the robustness of the solver, a negative test case was created to trigger a skolem escape. The following program attempts to unify an outer-scope metavariable with an inner-scope skolem, which should be rejected.

```
let
  myRun = \(x :: forall a b. a -> b).
    (let myBad = \y. y x in myBad)
in
  myRun
```

Fig. 4.6. `Program2.arralac`

When run through the typechecker and solver, the solver correctly identifies the illegal unification and reports a level check failure, as shown in Fig. 4.7. This directly validates that the level-based checking mechanism is working as designed.

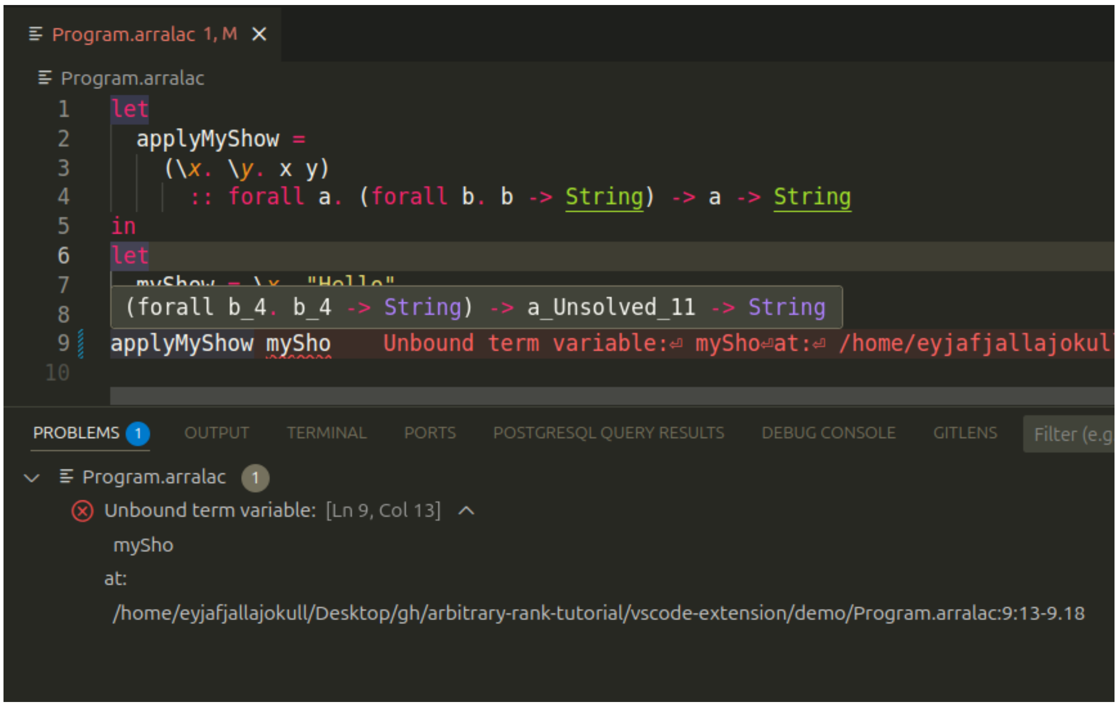
```
$ nix run .#arralac -- typecheck arralac/test/data/Program2.arralac
CallStack (from HasCallStack):
  SolverErrorWithCallStack, called at
       src/Language/Arralac/Solver/Error.hs:25:24
  dieSolver, called at src/Language/Arralac/Solver/Check.hs:51:5
Skolem escape!
The variable:
  a_7[ID 7, L 0, Meta, Unknown span: No location]
has the TcLevel:
  L 0
but the type variable:
  a_11[ID 11, L 1, Meta, Unknown span: No location]
has a larger TcLevel:
  L 1
in the constraint:
...
```

Fig. 4.7. Error message demonstrating successful skolem escape detection.

4.3.3 Interactive Tooling: The Language Server

The implementation includes a functional language server that leverages the annotated AST to provide on-the-fly diagnostics and type information.

Fig. 4.8 demonstrates two key features in Visual Studio Code: (1) hovering over an identifier (`applyMyShow`) to see its inferred polymorphic type, and (2) an error diagnostic for an unbound variable (`myShowww`). This closes the loop between the compiler's internal representations and a useful, interactive user experience.



Program.arralac 1, M

Program.arralac

```
let
  applyMyShow =
    (\x. \y. x y)
      :: forall a. (forall b. b -> String) -> a -> String
in
let
  myShow = \x. "Hello"
  (forall b_4. b_4 -> String) -> a_Unsolved_11 -> String
applyMyShow mySho    Unbound term variable:⏎ mySho⏎at:⏎ /home/eyjafjallajokul
```

```
Program.arralac 1
  Unbound term variable: [Ln 9, Col 13]
    mySho
  at:
    /home/eyjafjallajokull/Desktop/gh/arbitrary-rank-tutorial/vscode-extension/demo/Program.arralac:9:13-9.18
```

Fig. 4.8. LSP features: type on hover and error diagnostics.

4.3.4 Codebase Characteristics

The system was evaluated against several quality characteristics from the ISO 25010 standard.

- **Modularity:** The codebase is highly modular, comprising 86 distinct Haskell modules, as shown by the `cloc` [1] analysis in Table 4.1. This

[1]The tool is available at `https://github.com/AlDanial/cloc`

separation of concerns was critical for managing the complexity of the type inference engine.

- **Analysability:** The error-handling mechanism is designed for clear diagnostics. Each pipeline stage (e.g., Renamer, Typechecker, Solver) throws its own distinct error type, which captures a full call stack. This ensures that failures are easy to trace back to their source.
- **Installability:** The entire project is packaged with Nix, enabling a reproducible, single-line installation via the command `nix profile install`.

TABLE 4.1
Code Metrics for the `Arralac` Implementation Generated by `cloc`

Language	**Files**	**Blank Lines**	**Comment Lines**	**Code Lines**
Haskell	86	705	1085	3907
SUM	**86**	**705**	**1085**	**3907**

This chapter has demonstrated that the design laid out previously has been successfully realized in a functional, well-structured, and non-trivial implementation. The system correctly handles higher-rank types, robustly rejects invalid programs, and provides modern tooling, fulfilling the primary objectives of this thesis.

Chapter 5

Analysis and Discussion

This chapter moves beyond description to provide a critical analysis of the `Arralac` project. The analysis will proceed in three parts. First, I will evaluate how the chosen design patterns successfully achieved the project's primary objectives and what trade-offs they entailed. Second, I will offer a qualitative evaluation of the system's non-functional characteristics. Finally, I will critically examine the limitations of the current implementation and propose concrete directions for future research.

5.1 Analyzing the Architectural Contributions

The primary goal of this thesis was to create a modern, tutorial-focused implementation of arbitrary-rank polymorphism by synthesizing foundational theory with modern compiler engineering practices.

5.1.1 From Eager Unification to Constraint-Based Inference

The most significant architectural contribution of this work is its adoption of a two-phase, constraint-based type inference model, a departure from the eager unification algorithm presented in [1]. In their paper, the authors suggest that a ”more principled alternative is to... get the inference algorithm to return a set of constraints, and solve them all together” [1, Sec. 9.6].

`Arralac` serves as a direct, working implementation of this suggestion. By separating constraint generation (the `Typechecker`) from constraint solving (the `Solver`), this thesis demonstrates that the benefits of this architecture are fundamental to building a modular and understandable system.

Architectural Benefits and Trade-offs This separation yielded two primary benefits:

1. **Improved Modularity:** The logic of the typechecker and solver are cleanly decoupled. The `Typechecker`'s sole responsibility is to traverse the AST and emit a declarative set of `WantedConstraints`. The `Solver` operates on this abstract set of constraints, free from the complexities of AST traversal.

2. **A Foundation for Better Error Reporting:** A constraint-based model enables a holistic view of type errors. By gathering all constraints before solving, a future version of the solver could analyze the full set of conflicts to produce a much richer diagnostic than an eager unifier that fails on the first error.

The primary trade-off is the added complexity of the `WantedConstraints` data structure, which becomes the sole interface between the two largest components of the inference engine. This required careful engineering to ensure all necessary context, such as source locations and levels for scope checking, was correctly propagated.

5.1.2 The Trees That Grow AST: A Necessity for Modern Tooling

The second key architectural choice was the adoption of the **Trees That Grow** (TTG) pattern for the AST. The LSP requires annotating the AST with inferred types, and TTG provides a type-safe and elegant solution. As demonstrated in Sec. 4.1, the AST is parameterized by the compiler pass, and type families are used to "grow" the tree with annotations like `TcAnno` only after the typechecking pass. This provides a compile-time guarantee that type information is not accessed before it has been computed.

5.2 A Qualitative Evaluation of System Characteristics

Evaluating `Arralac` against ISO 25010 [36] sub-characteristics reveals the practical impact of its design.

- **Modularity:** The division of the codebase into 86 modules (Table 4.1) is a direct outcome of the pipeline architecture. The strict separation between stages creates a highly modular system. This modularity makes the codebase highly **analysable**; a student can study the `Solver` in isolation.

- **Analysability (Error Handling):** The system's analysability is enhanced by its error handling strategy. By defining distinct error types for each pipeline stage, each capturing a full call stack, the system provides transparent diagnostics.
- **Installability and Portability:** The use of Nix for dependency management and installation is critical to the project's goal of being a reproducible tutorial artifact. It guarantees that any (Linux or macOS) user can build and run the software with a single command.

5.3 Limitations and Avenues for Future Work

A critical analysis requires acknowledging the project's limitations. These simplifications, however, suggest clear directions for future research.

1. **Lack of `let`-generalization:** The most significant functional limitation is the lack of ML-style generalization for local `let`-bindings. Future work could add a scoped-solving phase at each `let`-binding, quantifying over any metavariables whose level is local to the binding's scope, as discussed in [27].
2. **Untyped Core Language:** `Arralac` desugars its typed AST into an untyped lambda calculus. A major extension would be to design a typed Core language and extend the desugaring process to generate

the necessary type abstractions and applications, making the entire pipeline type-safe.

3. **Simplified Constraint Solver:** The current solver is basic. It halts on the first unsolvable constraint and does not implement advanced strategies, like floating constraints out of implications and promotion [1]. Enhancing the solver to handle these cases and to report a complete set of residual constraints would be a valuable research project.

In conclusion, the analysis confirms that `Arralac` is not merely a reimplementation, but a modernization of the ideas in its foundational literature. It successfully serves its pedagogical purpose while providing a robust foundation for future exploration.

[1]See Note [Unification preconditions]

Chapter 6

Conclusion

This thesis was motivated by a significant pedagogical gap between the foundational theory of advanced type systems and the complex reality of their production-grade implementations. I began with the central claim that this gap could be bridged not by yet another theoretical paper, but by a modern, tutorial-focused compiler that makes the core engineering principles of a system like GHC tangible and interactive. The design and implementation of `Arralac`, a lambda calculus with arbitrary-rank polymorphism, has served to validate this thesis.

To achieve this, `Arralac` was built upon a synthesis of carefully chosen architectural patterns. By integrating a modular, constraint-based type inference pipeline with an extensible Trees That Grow AST, the project established a foundation that is both robust and clear. This architecture was then brought to life through a complete toolchain, most notably a functional Language Server Protocol (LSP) implementation, which transforms the abstract algorithms of type inference into an explorable, interactive experience.

The development of `Arralac` yielded a key insight: the architectural choices made in a compiler's design have a profound and direct impact on its pedagogical value. The separation of constraint generation from solving does not merely improve modularity; it creates an explicit representation of the typechecker's ”reasoning” that can be inspected and understood in an easier way that an eager algorithm. Similarly, the use of the LSP confirms that modern tooling is a transformative element in computer science education. Providing on-the-fly type information and diagnostics directly within an editor makes the behavior of the type system immediately visible, turning abstract rules into concrete feedback.

While `Arralac` successfully demonstrates its core thesis, its journey as a practical tool is far from complete. The logical next step is to tackle the crucial feature of `let`-generalization, a challenge for which the existing constraint-based architecture is an ideal foundation. Subsequently, evolving the evaluator to use a typed Core language would introduce the end-to-end type safety characteristic of production compilers. Extending the language server to support richer features, such as ”go to definition,” would further enhance its utility as a development and learning environment.

This thesis began with the goal of demystifying the ”magic” of a production-grade type system. Through the design and implementation of `Arralac`, it has shown that the core principles of arbitrary-rank polymorphism can be implemented in a structured, modern, and understandable way. By combining foundational theory with practical architectural patterns, this work provides a clear and interactive bridge for students, researchers, and aspiring language developers. Ultimately, this project stands as a testament to the idea that even the most powerful compiler technologies can be made accessible, fostering a deeper understanding and appreciation for the art of programming language implementation.

Bibliography cited

[1] 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/S09567968060060` Accessed: Apr. 3, 2025. [Online]. Available: `https://www.cambridge.org/core/product/identifier/S0956796806006034/type/journal_article`.

[2] “The glasgow haskell compiler,” Accessed: Apr. 27, 2025. [Online]. Available: `https://www.haskell.org/ghc/`.

[3] D. Vytiniotis, S. Weirich, and S. Peyton-Jones, “Practical type inference for arbitrary-rank types - technical appendix,” [Online]. Available: `https://repository.upenn.edu/server/api/core/bitstreams/7c1dc678-93c6-4516-98fd-f82b384eb75d/content`.

[4] L. Augustsson, “MicroHs: A small compiler for haskell,” in *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium*, ser. Haskell 2024, New York, NY, USA: Association for Computing Machinery, Aug. 28, 2024, pp. 120–124, ISBN: 979-8-4007-1102-2. DOI: `10.1145/3677999.3678280`. Accessed: Jul. 15, 2025. [Online]. Available: `https://dl.acm.org/doi/10.1145/3677999.3678280`.

[5] “Augustss/MicroHs: Haskell implemented with combinators,” Accessed: May 20, 2025. [Online]. Available: https://github.com/augustss/MicroHs.

[6] “Hugs 98,” Accessed: Jul. 16, 2025. [Online]. Available: https://www.haskell.org/hugs/.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN: 0321486811.

[8] 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.

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

[10] “Ocaml papers,” Accessed: Jul. 8, 2025. [Online]. Available: https://ocaml.org/papers.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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/BidirectionalSystem.

[18] 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.

[19] 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`.

[20] 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`.

[21] 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`.

[22] 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`.

[23] 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`.

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

[25] 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.

[26] 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.

[27] ACM SIGPLAN. “[WITS’24] solving constraints during type inference,” Accessed: Jul. 8, 2025. [Online]. Available: https://www.youtube.com/watch?v=0ISat1b2-4k.

[28] “Welcome to BNFC’s documentation! — BNFC 2.9.6 documentation,” Accessed: Apr. 26, 2025. [Online]. Available: https://bnfc.digitalgrammars.com/.

[29] “Hypertypes,” Hackage, Accessed: Apr. 27, 2025. [Online]. Available: https://hackage.haskell.org/package/hypertypes.

[30] “Compdata,” Hackage, Accessed: Apr. 27, 2025. [Online]. Available: https://hackage.haskell.org/package/compdata.

[31] P. Wadler. “The expression problem,” Accessed: Jul. 16, 2025. [Online]. Available: https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

[32] S. Najd and S. Jones, "Trees that grow," *JUCS - Journal of Universal Computer Science*, vol. 23, no. 1, pp. 42–62, Jan. 28, 2017, Number: 1 Publisher: Journal of Universal Computer Science, ISSN: 0948-6968. DOI: `10.3217/jucs-023-01-0042`. Accessed: Apr. 28, 2025. [Online]. Available: `https://lib.jucs.org/article/22912/`.

[33] *Glasgow haskell compiler*, GitLab, Apr. 24, 2025. Accessed: Apr. 27, 2025. [Online]. Available: `https://gitlab.haskell.org/ghc/ghc`.

[34] 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`.

[35] 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`.

[36] "ISO 25010," Accessed: Jul. 15, 2025. [Online]. Available: `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010`.