A Tutorial Implementation of a Lambda Calculus with Parametric Predicative Arbitrary-Rank Polymorphism

Danila Danko, MS-SE [1]

Supervisor: Nikolai Kudasov [1]

[1] Innopolis University

July 17, 2025

Outline

The Magic of Modern Type Systems

- Modern functional languages like Haskell have powerful type systems.
- One feature is **arbitrary-rank polymorphism**, allowing polymorphic functions as arguments.
- This enables highly generic and safe code, but its implementation can feel like magic.
- *How do compilers like GHC handle this?*

```
-- Pass a polymorphic function
-- 'myShow' as an argument
let
  applyMyShow =
    (\x. \y. x y) ::
      forall a.
        (forall b. b -> String)
          -> a -> String
in
let
  myShow = \x. "Hello"
in
  applyMyShow myShow
```

The Problem: A Pedagogical Gap

There is a steep learning curve for understanding how arbitrary-rank polymorphism is implemented.

Foundational Papers

- Seminal work like *Peyton Jones et al. (2007)* is theoretically dense.
- Describes an **eager unification** algorithm, which is conceptually different from modern practice.

Production Compilers (GHC)

- Huge, highly-optimized codebase.
- Uses a modern **constraint-based** architecture, a significant evolution from the papers.
- Difficult for a newcomer to trace the connection between theory and implementation.

The Gap

We need a "middle ground": a resource more concrete than a paper, but more focused and accessible than a production compiler.

The Solution: A Tutorial Compiler

- This thesis presents **`Arralac`**, a small, typed functional language and compiler.

The Solution: A Tutorial Compiler

- This thesis presents `Arralac`, a small, typed functional language and compiler.
- **Goal:** To bridge the pedagogical gap by serving as a well-documented, tutorial implementation of a modern typechecker.

The Solution: A Tutorial Compiler

- This thesis presents `Arralac`, a small, typed functional language and compiler.
- **Goal:** To bridge the pedagogical gap by serving as a well-documented, tutorial implementation of a modern typechecker.
- **Central Thesis:** A focused, modern implementation that consciously diverges from older models can be a more effective learning tool than studying theory or production code in isolation.

Key Architectural Contributions

`Arralac` was built on three core design choices to maximize clarity and modernity.

1. **A Modern, Constraint-Based Architecture**
   - Deliberately implements a GHC-style, two-phase engine:
     1. Constraint Generation
     2. Constraint Solving
   - This is a pedagogically superior alternative to the eager unification model.

Key Architectural Contributions

`Arralac` was built on three core design choices to maximize clarity and modernity.

1. **A Modern, Constraint-Based Architecture**
   - Deliberately implements a GHC-style, two-phase engine:
     1. Constraint Generation
     2. Constraint Solving
   - This is a pedagogically superior alternative to the eager unification model.
2. **An Extensible "Trees That Grow" AST**
   - The AST representation evolves with each compiler pass, enabling type-safe annotations required for tooling.

Key Architectural Contributions

`Arralac` was built on three core design choices to maximize clarity and modernity.

1. **A Modern, Constraint-Based Architecture**
   - Deliberately implements a GHC-style, two-phase engine:
     1. Constraint Generation
     2. Constraint Solving
   - This is a pedagogically superior alternative to the eager unification model.
2. **An Extensible "Trees That Grow" AST**
   - The AST representation evolves with each compiler pass, enabling type-safe annotations required for tooling.
3. **An Interactive Toolchain (LSP)**
   - A Language Server makes the typechecker's results visible and interactive, turning abstract rules into concrete feedback.

The Compilation Pipeline

A modular, multi-stage pipeline transforms source text into an evaluated result.

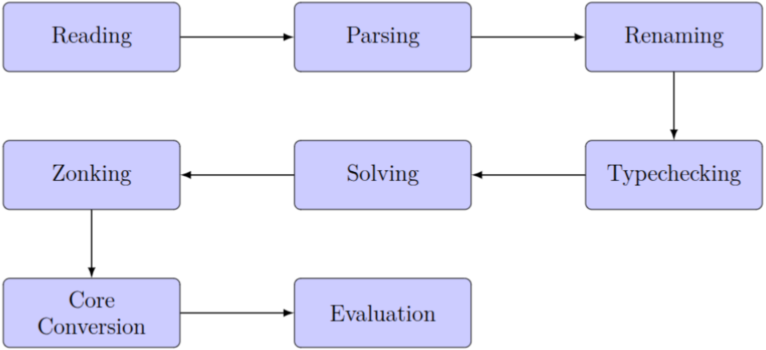

Figure 1: The `Arralac` Compilation Pipeline. Note the separation of Typechecking (Generation) and Solving.

Core Mechanism: Bidirectional Typechecking

The system avoids undecidable inference by operating in two modes.

Inference Mode (↑)

- Synthesizes or "infers" the most general type for an expression.
- Used when the type is not known in advance.
- For

  ```
  let x = \x.  x
  ```

  infers

  ```
  x ::  a -> a
  ```

Checking Mode (↓)

- Verifies an expression against a known, expected type.
- Triggered by programmer annotations.
- This is the key to handling higher-rank types.
- ```
  (\x.  x) ::
      String -> String
  ```
  ```

Positive Case: Higher-Rank Polymorphism

The system correctly typechecks a program that passes a polymorphic function as an argument.

Input Code (`Program1.arralac`)

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

Inferred & Zonked Output (simplified)

```
let
  applyMyShow_0 =
    (\(x_1 :: forall b_4. b_4 -> String).
      (\(y_2 :: a_8).
        (x_1 :: a_8 -> String)
        (y_2 :: a_8)
      )
    ) :: (forall b_4. b_4 -> String)
      -> a_Unsolved_10 -> String
in
  (applyMyShow_0
    (\(z_8 :: b_11). "Hello")
  ) :: a_Unsolved_10 -> String
```

Result: Success. The higher-rank annotation guides the typechecker correctly.

Negative Case: Skolem Escape Detection

The solver correctly rejects invalid programs where a type variable would escape its scope.

Invalid Code (Program2.arralac)

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

This code tries to unify an outer-scope variable with an inner-scope ‘a‘.

Compiler Error

```
Skolem escape!
The variable:
  a_7[ID 7, L 0, Meta, ...]
has the TcLevel:
  L 0
but the type variable:
  a_11[ID 11, L 1, Meta, ...]
has a larger TcLevel:
  L 1
```

Result: Correctly rejected. The level-based check works as designed.

Interactive Tooling: The Language Server

The LSP brings the type system to life in the editor.

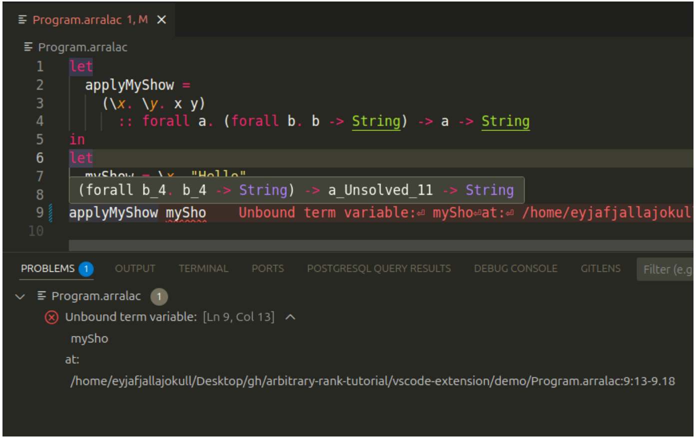

Figure 2: LSP features in VS Code: (1) Type on hover, showing the inferred polymorphic type, and

Conclusion

- `Arralac` successfully bridges the pedagogical gap between theory and practice for arbitrary-rank polymorphism.
- **Key Insight:** Architectural choices (constraint-based model, TTG, LSP) have a profound impact on a compiler's value as a learning tool.
- It demystifies the "magic" by providing a structured, modern, and understandable implementation.
- The project provides a clear, interactive bridge for students and aspiring language developers, delivered as a public, open-source repository.

Future Work

The tutorial foundation of `Arralac` enables several clear avenues for future research and development.

- **Implement `let`-generalization:** The most significant missing feature. The constraint-based architecture is an ideal foundation for adding scoped constraint solving.
- **Introduce a Typed Core Language:** Evolve the untyped Core language into a typed intermediate representation (like GHC's System FC) to provide end-to-end type safety.
- **Enhance the Constraint Solver:** Improve error reporting to show all residual constraints, not just the first failure. Implement advanced solving strategies like floating constraints.
- **Richer Language Features:** Support user-defined algebraic data types and type classes.

Thank You

Questions?

Code available at: `https://github.com/deemp/arbitrary-rank-tutorial`