Azalea Type System: Hindley-Milner Type Inference

The Azalea Team

2025-08-02

Abstract

Azalea employs a variant of the Hindley-Milner type system, specifically *Algorithm W*; a well established type inference algorithm used in languages like Standard ML, Ocaml and Rust to provide parametric polymorphism.

This paper describes the type system in detail and how it is used and implemented in Azalea.

The main intention of this paper is to serve as a form of documentation for myself when developing the type system, as it is a fairly complex system with lots of moving parts. However, since Azalea borrows its type system from languages in the ML family (and partially Haskell), I have liberally used the syntax from those languages to aid understanding.

Type Variables

A type variable is a placeholder for a type to be instantiated later. In the type `List[`*a*`]`, *a* is a type variable. We say that `List` is polymorphic over *a*.

This really means `List` is a **generic type construction** that can work with any type *a*.

Type Constructors

A type constructor is a function that takes one or more types and returns a new type. `List` is a type constructor that takes a type *a*.

Type Inference Process

Type inference is performed by Azaleas type checker. The process is as follows.

1. **Assign type variables**: Assign type variables to each expression or subexpression.
   - Example: For a function `id :` $a \to a$ we assign a type variable *t0* to the argument and return type, so `id` has type $t0 \to t0$.
2. **Generate constraints**: Generate constraints that map type variables to types based on how expressions are used.
   - Example: For the expression `id(42)`, we generate a constraint that *t0* must be `Int`, resulting in the constraint *t0* = `Int`.
3. **Unification**: Solve the constraints by unifying types.
   - Example: Assume we have the constraints *t0* = `Int` and *t0* = `String`, we unify them to find a common type. This is not always possible, and if it fails, the type checker reports a type error.
4. **Generalization**: When a value is assigned to a variable, the type checker generalizes its type by quantifying type variables with `forall`. In order to do this, we need to find all the free type variables
   - Example: If `id` is inferred to have type $t0 \to t0$, it is generalized to id : $\forall a.\ a \longrightarrow a$, meaning it can work with any type *a*.
5. **Instantiation**: When a polymorphic function is used, the type checker instantiates it with a specific type, narrowing it down.
   - Example: If `id` is used with an `Int`, it is instantiated to `id` : `Int` → `Int` from $\forall a.\ a \longrightarrow a$.
6. **Recursion**: The process is recursive, meaning that type inference can handle nested expressions and complex types.

Unification Rules

For any two types *t1* and *t2*, the unification rules are as follows:
- **Equivalence**: If *t1* and *t2* are the same type, they unify.
- **Type variables**: If *t1* is a type variable and *t2* is not a type variable, *t1* is unified with *t2* by way of a substitution.
- **Type constructors**: If *t1* and *t2* are type constructors, first occurs check their type parameters and then perform unification on their type parameters.
- **Function types**: If *t1* is a function type (*fn*) and *t2* is a function type (*fn*), unify their argument and return types.
- **Array types**: If *t1* is an array type and *t2* is an array type, unify their element types.

- **Record types**: If *t1* and *t2* are record types, first check the lengths of both records, and then perform unification on their fields if the names match.