Types in Passerine

Weekly meeting 2021-04-17

Isaac Clayton

Quick facts

Y'all know, but: a type system allows a compiler to determine the specific structure of data at each point in the program:

- Strong vs Weak: Casting and comparing data
- Static vs Dynamic: When types are checked
- Nominal vs Structural: How types are described

Algebraic Data Types

A composite type, i.e. types made up of other types:

- Base types:
  - Integers, strings, floating point numbers, booleans, bytes, etc.
- Product types:
  - Structs or records or tuples
  - Made up of a number of *fields*
- Sum types:
  - Tagged unions
  - Made up of a number of *variants*

Mapping ADTs to Passerine

Product types:

- Tuples (indexed by number)
- Records (indexed by name)

Sum types:

- Enums (closed unions)
- Traits (open unions)

Base types:

- Boolean
- Integers
- Reals
- String
- Functions
- Fibers
- Alg. Effs.?
- (More; this for now)

Part I: Product Types

Tuples

Definition:

- (Label_0, …, Label_N)

Construction:

- (expression_0, …, expression_N)

Destruction:

- (pattern_0, …, pattern_N)

Update:

- (function_0, …, function_N)

Records

Definition:

- { field_0: Label_0, … field_N: Label_N }

Construction:

- { field_0: expression_0, … field_N: expression_N }

Destruction:

- { field_0: pattern_0, … field_N: pattern_N }

Update:

- { field_0: function_0, … field_N: function_N }

Aside:
Row Polymorphism

If the record A is a subset of record B, record A is a B.

- type Foo { a: Int }
- type Bar { a: Int, b: String }

Therefore, Foo is a Bar

Part II: Sum Types

Enums

Definition:

- { Label_0 type_0, Label_N type_N }

Construction:

- Label expression

Destruction:

- Label pattern

Update:

- Label function

Traits

I'm not sure at all… some options

1. Associated namespaces
2. Dynamic dispatch
3. Multiple dispatch (Julia)
4. Rust traits / Haskell typeclasses
5. ???

Part III: Traits

Associated Namespaces

Basic Idea:

- Each type has a record, i.e. an associated namespace
- We can index into that namespace:
    - Label::field
- We can define interfaces as records:
    - type Add T { add: T T -> T }
- Because of row polymorphism, if Label has associated value Label::add, Label is Add.

Associated Namespaces

Problems:

- How do we define associated namespaces?
  - impl
  - direct
- Diamond problem:
  - Two traits, Couch and Chair both have ::sit
  - We make an Armchair, both Couch and Chair apply
  - Which ::sit method is used for Armchair?

Dynamic / Multiple Dispatch

Think of this like open match statements:

- We define a ‘function’ X:
  - Has a number of methods.
  - Each method maps from A -> B
- If we call function(D):
  - Looks up method that takes type D
  - Runs that method against it
- No diamond problem, always call function.

Dynamic / Multiple Dispatch

Issues:

- Only granular on the basis of functions:
  - Can’t define a ‘Number’ that implements add, sub, mul, div, etc. methods.
- Not clear what method you are calling, or which methods take priority

Traits / Typeclasses

- You define a ‘trait’ X:
  - Has a number of ‘members’
  - Each member is a type, function, etc.
- You implement trait X for type Y:
  - All members must be implemented
- Dispatch based on type:
  - If X is implemented for Y and X::member exists:
  - Y.member() will dispatch appropriately.
- Diamond Problem is solved:
  - Use explicit function call syntax:
  - X::member(Y) and Z::member(Y)

Traits / Typeclasses

Issues:

- With languages like Rust (low-level), annoying to manage &dyn
    - Not an issue for high-level langs like Haskell or Passerine
- Trait objects are a tad confusing
- Have to explicitly implement it for each type
- Introduces additional syntax

Honestly probably the best solution?

Part V: Algebraic Effects

What is an Algebraic Effect?

- Separates state management from call site:
  - Computations can cause side effects:
    - I/O, FFI, Errors, etc.
  - Algebraic effects define ways to manage for these side effects
- Prior work:
  - Koka
  - Unison (abilities)
  - Eff
- Built on top of continuations:
  - An effect is a sum type that dispatches on continuation.

Basic example

We define an effect with a number of operations:

- **effect raise → has operation raise**

We can use the operation like a function:

- **raise("hello")**

We implement a handler to implement the behaviour of the operation:

- **handler raise(continuation) → handle and decide how to continue**

Why are they a good for for Passerine?

When you think about it, an FFI is an effect handler implemented in another language.

Passerine is an embedded language, so allows fine-grained control of side effects.

Natural fit for fibers, and Passerine's concurrency model.

Would simplify runtime interface upon implementation.

Discuss!

1. **Traits in Pn?**
2. **Alg. Effs. in Pn?**

Moving forward

Easiest ways to contribute:

- Get acquainted with project layout; I'm always open for questions
- Help-wanted labels on GitHub
- Coming to this Weekly Meeting!
- Discussing ideas on Passerine Discord.

I'm a bit burnt out, which is why I haven't been actively hacking on Passerine's codebase.

Also at a bit of a crossroads: Passerine is very simple, and there are a lot of directions to take, we have to make a choice and set a course. (w.r.t. Alg. Effs. and Traits)

Fin!

Thanks for tuning in!

Next Up:
Shaw will share his work on an alternative impl. for Passerine!