Elara Language Specification

Alexander Wood

December 2022

1 Introduction

Elara is a statically-typed multi-paradigm programming language based on the Hindley-Milner type system. It supports a succinct, Haskell-like syntax while preserving readability and ease of use.

Elara focuses on the purely functional paradigm, but also supports Object Oriented programming and imperative programming.

Elara's features include:

- Structural pattern matching
- A first-class effects system
- Type classes for polymorphism
- Complete sound type inference

Elara primarily targets the JVM but may also target other platforms in the future.

1.1 Code Examples

While all examples are syntactically correct, they may assume the existence of functions not provided in the examples.

1.1.1 Hello World

```
let main = println "Hello World!"
```

1.1.2 Pattern Matching on Lists

```
let map f list =
    match list
        [] -> []
        (x:xs) -> (f x) : (map xs f)
let main =
    let list = [1, 2, 3, 4]
    let doubleNum i = i * 2
    println (map doubleNum list)
```

1.1.3 Custom Data Types

Elara has an extremely flexible type system allowing many different types of custom data types

```
type Name = String # Type alias
type Animal = Cat | Dog # Simple Discriminated Union
type Person = { # Record Type
    name : Name,
    age : Int,
}
type NetworkState = # Complex Discriminated Union with
      Connected
    | Pending
    | Failed Error

type Option a = # Generic data types
      Some a
    | None

type JSONElement = # Combination of multiple type features
      JSONString String
    | JSONNumber Int
    | JSONNull
    | JSONArray [JSONElement]
    | JSONObject [{ # record syntax can be used anonymously
        key : String,
        value : JSONElement
    }]
```

1.1.4 Pattern Matching on Data Types

```
type JSONElement =
      JSONString String
    | JSONNumber Int
    | JSONNull
    | JSONArray [JSONElement]
    | JSONObject [{
        key : String,
        value : JSONElement
    }]

let jsonToString elem = match elem
    JSONNull -> "null"
    JSONString str -> str
    JSONNumber num -> toString num
    JSONArray arr -> "[" ++ (join ", " (map jsonToString arr)) ++ "]"
    JSONObject components ->
        let componentToString { key, value } =
            "\"" ++ key ++ "\" : " ++ jsonToString value
        in "{" ++ join ", " (map componentToString components) ++ "}"
```

1.1.5 Type Classes

```
type class ToString a where
    toString : a -> String
instance ToString String where
    toString s = s
instance ToString Char where
    toString c = [c]

def print : ToString s :> s -> () \ IO
# Impure function taking any s | ToString s, returns Unit
let print s = println (toString s)
let main =
    print "Hello"
    print 'c'
    print 123 # Doesn't compile, no ToString instance for Int
```

1.1.6 Polymorphic Effects

```
def map : (a -> b \ ef) -> [a] -> [b] \ ef
let map f list = match list
    [] -> []
    (x:xs) -> let! y = f x in y : (map f xs)
```

1.1.7 Monad Comprehension

```
def sequenceActions : Monad m => [m a] -> m [a]
def sequenceActions list = match list
    [] -> pure []
    (x:xs) ->
        let! x' = x
        let! xs' = sequenceActions xs
        pure (x' : xs')

def sequenceActions_ : Monad m => [m a] -> m ()
def sequenceActions_ list = match list
    [] -> pure ()
```

```
(x:xs) ->
    do! x
    sequenceActions_ xs
```