March 23, 2023

Elara Language Specification

Alexander Wood

Contents

1 Introduction

Elara is a statically-typed multi-paradigm programming language targeting the JVM and 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 notable features include:

- Structural pattern matching with exhaustiveness checking
- Type classes for polymorphism
- Complete sound type inference with higher-kinded and higher-rank types

1.1 Code Examples

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

1.1.1 Hello World

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

1.1.2 Pattern Matching over Lists and Higher Order Functions

```
let map f list =
    match list with
        [] → []
        (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 a very flexible type system which allows for many different forms of data types to be defined.

```
type Name = String // Simple type alias

type Animal = Cat | Dog // Simple Discriminated Union

type Person = { // Record Type
    name : Name,
    age : Int,
}

type RequestState = // Complex Discriminated Union with different constructor arities
      Connected String
    | Pending
    | Failed Error

type Option a = // Polymorphic (generic) data types
      Some a
    | None

type Fix f = Fix (f (Fix f)) // Recursive, higher-kinded data types
```

```
// Combination of multiple type features
type JSONElement =
      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 with
    JSONNull → "null"
    JSONString str → str
    JSONNumber num → toString num
    JSONArray arr → "[" <> (String.join ", " (map jsonToString arr)) <> "]"
    JSONObject components →
        let componentToString { key, value } =
            "\"" <> key <> "\" : " <> jsonToString value
        in "{" <> String.join ", " (map componentToString components) <> "}"
```

1.1.5 Type Classes

```
type String = [Char]

type class ToString a where
    toString : a → String

instance ToString String where
    toString s = s

instance ToString Char where
    toString c = [c]

-- Impure function taking any s with an instance ToString s, returns Unit
def print : ToString s ⇒ s → IO ()
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 Monad Comprehension / Notation

```
def sequenceActions : Monad m ⇒ [m a] → m [a]
def sequenceActions list = match list with
    [] → 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 with
    [] → pure ()
    (x:xs) →
        do! x
        sequenceActions_ xs
```

1.2 Syntax

1.2.1 Literals

1.2.2 Identifiers

1.2.2.1 Variable Identifiers

1.2.3 Patterns

A large part of Functional Programming is *pattern matching*, the process of matching a value against a pattern and extracting information from it.

Elara supports a large number of patterns and pattern combinators to try and make this as painless as possible:

- **Var patterns**:
  `vn` where `vn` is any valid variable identifier (see Section 1.2.2.1) matches any value and binds it to the name `vn`.
- **Wildcard patterns**:
  `_` matches any value and discards it. This is useful when you only care about some of the arguments of a function or constructor.
- **Literal patterns**:
  Any of the supported literals (see Section 1.2.1) may be used as patterns. These match only if the value is equal to the literal.
- **Constructor patterns**:
  `(C p1 p2 ... pn)` matches a value `v` if `v` is a constructor `C` applied to `n` arguments, and if `p1`, `p2`, …, `pn` match the arguments of `v` respectively.
- **Tuple patterns**:
  `(p1, p2, ..., pn)` matches a value `v` if `v` is a tuple of `n` elements, and if `p1`, `p2`, …, `pn` match the elements of `v` respectively.
- **List Patterns**:
  `[]` matches the empty list.
  `[p1, p2, ..., pn]` matches a list of `n` elements, and if `p1`, `p2`, …, `pn` match the elements of the list respectively.
  `p1 :: p2` matches a list of `n` elements, and if `p1` matches the first element of the list and `p2` matches the rest of the list. (Note that this is technically a constructor pattern, just with special syntax)

- **Named Record Patterns**
  `{f1: v1, f2: v2, ..., fn: vn}` matches a value `v` if `v` is a record with fields `f1`, `f2`, …, `fn` and if `v1`, `v2`, …, `vn` match the values of the fields respectively. As records are first-class types, this pattern
- **As patterns**:
  `p as vn` matches a value `v` if `p` matches `v` and binds the name `vn` to `v`. This is useful in more complex patterns. For example `([1, _, _] as l, _)` matches a 2-tuple whose first element is a list of exactly 3 elements, starting with `1`, and binds the name `l` to the *whole list*
- **If patterns (guards)**:
  `p if e` matches a value `v` if `p` matches `v` and `e` evaluates to `True`. The expression `e` is evaluated in the same scope as the pattern, so it can refer to any variables bound by the pattern.
  A practical example of this is `n if isEven n` which only matches even numbers (assuming `isEven` works as the name suggests)
  Note that guards often break exhaustiveness checking. For example, this will not compile:

  ```
  def f : Int → Int
  let f n = match n with
      n if isEven n → n + 1
      n if isOdd n → n - 1
  ```

  This is because the compiler cannot prove that `isEven` and `isOdd` are mutually exclusive so cannot prove that the match covers all cases.

1.2.4 Multi-line Environments

Some syntactic structures in Elara can create multi-line environments. Formally, this means that rather than a single expression, a semicolon-separated list of *statement*s, surrounded by braces, can be used where a multi-line environment is permitted.
Practically, this allows the imperative idea of "blocks" of code to be used, rather than having a binding be a single long expression. Note that this feature is merely syntax-sugar and does not change the purely-functional semantics of Elara. When in a multi-line environment, the syntax is extended to allow imperative statements:

- Standalone let bindings:
  `let x = 1;`
- Monadic let expressions:
  `let! x = action in x + 1;`
- Monadic let bindings:
  `let! x = action;`
- Monadic do statements:
  `do! action;`
- Monadic (applicative) return statements:
  `return! 1;`

1.2.5 Lightweight Syntax

Elara allows lightweight syntax, a feature heavily inspired by F#. This makes newlines and indentation significant, allowing the omission of many braces and other tokens. Its use is recommended in almost every case.

Note that the tokens that can be ignored when using lightweight syntax may still be written manually, making the use of lightweight syntax effectively optional. The only difference is that

the lexer should insert them implicitly *if and only if they are missing* when using lightweight syntax.

1.2.5.1 Lightweight Syntax Rules by Example

The following describes the lightweight syntax rules in an informal, example-based manner.

1.2.5.1.1 Optional Semicolons

In normal syntax, semicolons are required to separate statements and must appear at the end of every declaration or statement. In lightweight mode, semicolons are optional and are inferred by the presence of a newline (`\n`) character.

Normal Syntax

```
let x = 1;
let y = 2;
let main = println (x + y);
```

Lightweight Syntax

```
let x = 1
let y = 2
let main = println (x + y)
```

1.2.5.1.2 Optional Braces

In normal syntax, braces are required when beginning a multi-line environment (see Section 1.2.4) or in a few other cases (such as the body of a match expression)

In lightweight syntax, braces are optional and can be inferred by newlines and indentation.

Normal Syntax

```
let x = {
    1;
}
```

```
let y = \x -> {
    1;
}
```

```
let main = match x with {
    1 -> {
        println "it's 1";
    };
    _ -> {
        println "it's not 1";
    };
}
```

```
let test x = {
    if x then {
        1;
    } else {
        2;
    }
}
```

Lightweight Syntax

```
let x =
    1
```

```
let y = \x ->
    1
```

```
let main = match x with
    1 ->
        println "it's 1"
    _ ->
        println "it's not 1"
```

```
let test x =
    if x then
        1
    else
        2
```

1.2.5.2 Offside Rule

When using lightweight syntax, the indentation is flexible but not arbitrary. The "offside rule" is used to determine which columns code should be indented to. The offside rule marks specific columns in the source code as "offside" for a given syntactic construct. Code on proceeding lines must be indented to the same column to be considered part of the same syntactic construct. If the code is indented further and a new offside rule cannot be triggered, an error is raised. If

the code is indented less, the offside rule is exited and the code is considered to be part of the parent syntactic construct

The following tokens trigger the offside rule at their respective columns:

- The first non-whitespace token after the `=` token of a `let` construct
- The first non-whitespace token after the → token of a lambda
- The first non-whitespace token after the `then` token of an `if` expression
- The first non-whitespace token after the `else` token of an `if` expression
- The first non-whitespace token after the `with` token of an `match` expression
- The first non-whitespace token after the → token of an `match` case
- The first non-whitespace token after a `{` token
- The start of a `let`, `if` or `module` token

1.2.5.2.1 The Offside Rule in Action

The following examples demonstrate the offside rule in action, noting when it is used incorrectly and errors should be raised.

```
module Main          | module keyword triggers the offside rule
let x = 1            | marks offside column
  let y = 2          | considered part of x, error!
let z = 3            | correctly indented
```

As the `module` keyword triggers the offside rule, all top-level declarations must be indented to the same column. This can produce some interesting, but correct results. All top-level declarations can be indented as long as it's consistent:

```
module Main                 | module keyword triggers the offside rule
  let x = 1                 | marks offside column
  let y = 2                 | same column, fine
let z = 3                   | bad! not considered part of the module, error!

let x =                     | = triggers the offside rule
    let y = 1               | correctly indented
      let z = 2             | bad! indented too much
    y + z                   | correctly indented

let main =                  | = triggers the offside rule
  match [1, 2, 3] with      | with triggers the offside rule
    x :: [] →  "one"        | x's column marked as offside
   x :: xs → "many"         | bad! not indented enough
     [] → "empty list"      | bad! too far indented
```

1.3 Code Structure

1.3.1 Packages

The basic unit of compilation in Elara is a *package*. A package is a collection of modules, which are typically compiled and distributed together. Packages are defined in a `elara.json` file located in the root directory of the package. This file provides metadata about the package such as name, author, version, etc.

1.3.1.1 `elara.json` structure

The `elara.json` file is a standard JSON file that must contain the following attributes at the top level:

- `name: string` - The package's name. This may only contain alphanumeric characters, and the `-` symbol. Conventionally, package names are written in `lower-snake-case`

- `version: string` - The package's version. This must be a valid semantic version string. Typically `1.0.0` is used for initial releases and should usually be the default value when generating `elara.json` files.

1.3.2 Modules

Inside packages, Elara code is organised into hierarchial *modules*. Modules are single files containing a (possibly empty) list of declarations which define namespaces for these declarations. Modules are named using the `module` keyword which must appear at the start of the file, and must be named as at least 1 `UpperCamelCase` section, separated by `.` characters. For example, the module names `Foo`, `Foo.Bar`, and `Foo.Bar.Baz` are all valid. Module names must be unique within a package, and must reflect the file structure of the package. For example, a module named `Foo.Bar` must be located at `src/Foo/Bar.elara`.

Importantly, modules are hierarchial with respect to importing. Given modules `Foo` and `Foo.Bar`, the module `Foo.Bar` is a "child" of `Foo`. This has 2 important implications:

- The module `Foo.Bar` can reference declarations in `Foo` without explicitly importing it
- Modules importing `Foo` will also import `Foo.Bar`, and any other child modules of `Foo`

The following code across multiple files demonstrates the above points:

src/Foo.elara

```
module Foo

def x : Int
let x = 1
```

src/Foo/Bar.elara

```
module Foo.Bar

def y : Int
let y = Foo.x + 1
```

src/Main.elara

```
module Main
import Foo

def main : IO ()
let main = print Foo.Bar.y
```

1.3.2.1 Imports

Importing is the action of bringing a module's declarations into scope. This is done using the `import` keyword, which must appear at the top of the file under the `module` declaration.

By default, imports are **qualified** and expose **everything**. This means that when referencing a member imported from another module, the module name must be prefixed, e.g. `Foo.Bar.y` rather than `y`

1.3.2.1.1 Qualification

Qualified imports can be made unqualified by using the `unqualified` keyword after the module name, e.g. `import Foo.Bar unqualified`.

Unqualified imports should be used sparingly as they can lead to name clashes and scope pollution. Note that even with an unqualified import, explicit qualification is still permitted.

1.3.2.1.2 Expositions

By default, all declarations in a module are exposed (brought into scope) when imported. If this is not desired, a subset of the declarations can be imported using the `exposing` keyword after the module name, e.g. `import Foo.Bar exposing (x, y)`.

This can be quite useful when combined with unqualified imports. For example, suppose a library provides a `HashMap` module whose members' names clash with the Prelude. We could write something like

```
import HashMap unqualified exposing (HashMap)
import HashMap
```

```
def testMap : HashMap String Int
let testMap = HashMap.singleton "foo" 1
```

to allow the use of the `HashMap` type name without qualification, but everything else must be qualified.

Modules may also control their exposed members in the module declaration with a very similar syntax: `module Foo exposing (x, y)`. This means that at most, the listed members can be imported. Any members not in the exposition list can be considered "private" to the module.

1.3.2.1.3 Aliasing

Finally, it can be convenient to rename a module when importing it. This can be done using the `as` keyword, e.g. `import Foo.Bar as Bar`. This allows us to refer to the module as `Bar` rather than `Foo.Bar` in the current file.

Going back to the previous `HashMap` example, we might rename the unqualified `HashMap` module to `Map` to avoid confusion:

```
import HashMap unqualified exposing (HashMap)
import HashMap as Map

def testMap : HashMap String Int
let testMap = Map.singleton "foo" 1
```

1.4 Types

honestly man I'm not gonna pretend to know what all the theory actually works, so here's just a list of what Elara's type system can do (theoretically)

1.4.1 Basic Types

As Elara is based on the Hindley-Milner system, types are either **monotypes** or **polytypes**.

Monotypes are simple, non-polymorphic types that can be either:

- A *concrete type* such as `Int` or `String`
- An *application* of a *type function* such as `Int → String`, which is the application of the type function `→` to the types `Int` and `String`, representing a function from `Int` to `String`

Polytypes (often called *generic types* in imperative languages) contain type variables which are bound by zero or more universal quantifiers.
The simplest example of a polytype is $\forall a.a \to a$, written in Elara as `forall a. a → a` (the `forall a.` is optional and may be omitted, all type variables are universally quantified). This type represents a function that takes a value of any type and returns a value of the same type.

Polytypes must be *instantiated* before they can be used practically. This is the process of substituting the type variables with concrete types. This is generally done automatically based on context.
For example, the following code is valid:

```
def id : forall a. a → a
let id x = x

def num : Int
let num = id 1
```

The type of `id` is `forall a. a → a`, which is a polytype. In `num`, `id 1` is called. Since `1` is a monotype (an `Int`), the `a` is substituted with `Int` to give the specialised type `Int → Int` for `id`. This implies that the type of `id 1` is `Int` which matches the declared type of `num`.

However, polytypes are not *required* to be instantiated. Consider a higher-order function `map : forall a b. (a → b) → [a] → [b]`. Calling `map id` is valid despite `id`'s type being a polytype. In this example, the types `forall a b. (a → b)` and `forall a. a → a` are *unified* to give the type `forall a. a → a` (since `b` must "equal" `a` for the 2 types to be equivalent). Therefore the type of the expression `map id` is `forall a. [a] → [a]`. Note that while `b` is instantiated to `a`, `a` is not instantiated.

Note that the *type functions* mentioned earlier are typically polytypes. `Int → String` is a monotype, but → is a polytype with 2 type variables.

1.4.2 Record Types

Elara supports first-class record types. A record type `{k1: t1, k2: t2, ...}` is a monotype that represents a record with keys `k1`, `k2`, etc. and types `t1`, `t2`, etc. respectively.

A record type is defined as a set of *fields*, each with an explicit name and type, written as `name: type`, separated by commas (`,`) and enclosed in curly braces (`{}`).

For example, the following defines a record type with 2 fields, `x` and `y`, both of which are `Int`s:

```
{ x: Int, y: Int }
```

Record types are constructed using a similar syntax:

```
def origin : { x: Int, y: Int }
let origin = { x = 0, y = 0 }
```

TODO: document row polymorphism

1.4.3 Custom Data Types

Elara supports user-defined data types, which are defined using the `type` keyword followed by a name and an optional set of type variables, separated by spaces.

User-defined data types may be either *algebraic* or *alias* types, or a combination of the two.

1.4.3.1 Algebraic Data Types

An algebraic data type (commonly called a *discriminated union*) is defined as a set of named *constructors* that each take a nullable set of arguments, separated by the `|` character.

A value of an algebraic data type may be any one of the type's constructors with the appropriate arguments.

For example, the following defines an algebraic type `Option` with 2 constructors, `Some` and `None`:

```
type Option a
    = Some a
    | None
```

with this definition, we can pattern match over the constructors:

```
def getOrDefault : forall a. Option a → a → a
let getOrDefault opt default = match opt with
    Some x → x
    None → default
```

and we can construct values of the type using either of the constructors:

```
def some : Option Int
let some = Some 1

def none : Option Int
let none = None
```

Algebraic data types may be recursive:

```
data ConsList a
    = Cons a (ConsList a)
    | Nil
```

1.4.3.2 Alias Types

An alias type defines a name for an already existing type. For example, the following defines a type `Tuple2` which represents tuples as records:

```
type Tuple2 a b = { _1: a, _2: b }
```

Importantly, alias types may be recursive:

```
type Person = { name: String, children: [Person] }

def child : Person
let child = { name = "Bob", children = [] }

def parent : Person
let parent = { name = "Alice", children = [child] }
```

This means that an alias is not simply a synonym for an existing type, as this type would be impossible to express without an alias.

1.4.3.3 Combinations of Algebraic and Alias Types

Algebraic and alias types may be combined to create more complex types. For example, we can define a JSON AST as follows:

```
type JSON
    = Number Float
    | String String
    | List [JSON]
    | Object [{ key: String, value: JSON }]
    | Null
```

The JSON text

```
{
    "foo": 1,
    "bar": [1, 2, 3],
    "baz": {
        "qux": "hello"
    }
}
```

would be represented as the following value:

```
def jsonObj : JSON
let jsonObj =
    Object
        [ { key = "foo", value: Number 1.0}
        , { key = "bar", value: List [Number 1.0, Number 2.0, Number 3.0]}
        , { key = "baz", value: Object [ { key = "qux", value: String "hello" } ] }
        ]
```

1.4.4 Formal Syntax

The following BNF grammar defines the formal syntax of Elara types, where the term `<typeDecl>` is a user-defined type declaration.

```
<recordField> ::= <fieldName> <spaces> ":" <spaces> <type>

<recordType> ::= "{" <spaces> (<recordField> <spacesOrNewlines>) ("," <spaces>
<recordField>)* <spaces> "}"

<constructor> ::= <typeName> <spaces> (<type> <spaces>)*

<algebraicDataType> ::= <constructor> <spaces> ("|" <constructor> <spaces>)*

<type>
::= <typeName>
      | <recordType>

<topLevelType>
::= <type>
      | <algebraicDataType>

<typeDecl> ::= "type" " "+ <typeName> <fieldName>* <spaces> "=" <topLevelType>

<fieldName> ::= [a-z] ([a-z] | [A-Z] | [0-9])*
<typeName> ::= [A-Z] ([a-z] | [A-Z] | [0-9])*
<spaces> ::= " "*
<spacesOrNewlines> ::= (" " | "\n")*
```

1.4.5 Expanding User Defined Types

Some user defined types can be simplified by expanding them to their underlying type. This can make compilation easier and potentially reduce the number of allocations needed.

For example, when writing a simple alias such as `type Age = Int`, we would not expect usage of `Age` to behave any differently than `Int` or require any extra memory allocations, as they are isomorphic. Therefore, the compiler can simply substitute all uses of `Age` with `Int`.

Similarly, algebraic data types with a single constructor can be expanded to their underlying type. For example, `type Box a = Box a` should be expanded to `type Box a = a`. This also requires eliminating usages of the `Box` constructor in expressions and patterns:

```
type Box a = Box a

def box : Box Int
let box = Box 1

let main =
    match box with
        Box x → print x
```

```
type Box a = a

def box : Int
let box = 1

let main =
    match box with
        x → print x
```

Single constructor ADTs are often very useful for making non-standard instances of a type class. Their use should not incur any performance or memory penalty at runtime.

Compilers are expected to simplify non-recursive type aliases and single-constructor ADTs. While further simplification may be posible, it is not required.