Iron

Official Language Specification

Table of Contents

Introduction

The purpose of this document is to provide an overview of the syntax and language features of the Iron programming language. It takes inspiration from many different, modern languages with the goal of making it easier to build secure, scalable systems.

Types

Iron has a set of fundamental base **Types**, and allows developers to define new types through *aliasing* or *composition*. These types are strictly enforced in operations and function calls to ensure program safety and improve performance.

Boolean

A boolean can be one of two values: *true* or *false*. Booleans implement the logical operators that you would expect, allowing you to compare them to each other to create more elaborate control flow structures.

```
let x: Boolean = true
let y: Boolean = false

let andResult = x and y
let orResult = x or y
let xorResult  = x xor y

assert(andResult equals y)
assert(orResult equals x)
assert(xorResult equals y)
```

String

The Iron String type is an alias for the `[U8]` list type, which we will discuss later. It provides an interface for UTF-8 strings, which consist of characters ranging from 1 to 4 bytes in size. If you want a more traditional string type, we also provide `ASCIIString`.

```
let hello: String = "Hello "
let world: String = "world!"

let greeting = hello + world
assert(greeting equals "Hello world!")

let nihongo: String = "日本語"
for index, char in nihongo {
    io.printLine("{char} starts at index {index}")
}
```

Numbers

There are a suite of number types that allow you to work with integer, floating point, and complex values of various sizes. Integer values can be signed or unsigned. The complex type, `C128`, is an alias for the `(F64, F64)` tuple type.

```
let a: I8 = 1
let b: I16 = -2
let c: I32 = 3
let d: I64 = -4
let e: I128 = 5

let f: U8 = 6
let g: Byte = 7 // Byte is an alias for U8
let h: U16 = 8
let i: U32 = 9
let j: U64 = 10
let k: U128 = 11

let l: F32 = -12.0
let m: F64 = 13.0

let n: C128 = (14.0, 15.0)
```

Collections

When you need to work with collections of a particular type (or group of types), you will want to use one of Iron's standard collection types.

List

A list is an array-like structure that contains elements of a single type. You indicate a list type by wrapping some `Type` in square brackets: `[Type]`.

```
let a: [U8] = [1, 2, 3]
let b: [U8] = [4, 5, 6]
let fruits: [String] = ["apple", "orange", "banana"]

assert(a.length() equals 3)
assert(b.length() equals 3)
assert(c.length() equals 3)
assert(a + b equals [1, 2, 3, 4, 5, 6])

try {
    assert((a + c).length() equals 6)
} catch TypeError {
    io.printLine("Type mismatch!")
}
```

Tuple

Lists are great, but have the draw-back that all elements have to be the same type. What if you want to work with a collection of related data that has different types? This is fairly common, so we make that possible with the `Tuple` type.

```
let a: (U8, String, Boolean) = (1, "hello", true)
let b: (U8, String, Boolean) = (2, "goodbye", false)

// You can also define lists of tuples!
let c: [(U8, String, Boolean)] = [a, b]

for element in a {
    match type(element) {
        U8 => io.printLine("It's a number!"),
        String => io.printLine("It's a string!"),
        Boolean => io.printLine("It's a boolean!"),
        _ => io.printLine("Big oopsie!"),
    }
}
```

Set

Sets are like tuples, but with the additional constraint that every value in them is unique. Their API exposes the typical algebraic functionality you would expect.

```
let a: Set<U8> = {1, 2, 3}
let b: Set<U8> = {2, 3, 4}

let c: Set<Boolean> = {true, false}
let d: Set<String> = {"red", "white"}

assert((a union b) equals {1, 2, 3, 4})
assert((a intersection b) equals {2, 3})
assert((a - b) equals {1})

// We use the * operator for the cartesian product of two
// sets. Notice that our assertion compares the result to a
// set of tuples, rather than a set of sets. Specifically,
// the result type here is `Set<(Boolean, String)>`.
//
// This allows us to satisfy the constraint that all elements
// of a set are the same type.
assert((c * d) equals {
    (true, "red"),
    (true, "white"),
    (false, "red"),
    (false, "white")
})
```

Map

A `Map` contains a set of key-value pairs. Keys can be any type that implements the `Hashable` protocol, and values can be of any type. Types must match for every key-value pair in the map.

```
let m: mutable Map<U8:String> = {}

m.set(1, "do")
m.set(2, "re")
m.set(3, "mi")

assert(m.get(2) equals "re")

try {
    assert(m.get(4) equals "fa")
} catch KeyError {
    io.printLine("Key is not present!")
}
```

Other Types

Self

This is a special type that can only be used within methods defined in a `Protocol`. Specifically, it refers to the type that is *implementing* the protocol. For more information, please see the Protocol section.

Option

In some situations, you might want to handle scenarios based on whether or not some value exists, such as working with data retrieved from a SQL query. `Option` enables you to do this for any type.

```
type Person = {
    firstName: String,
    middleName: String?,
    lastName: String
}

// For this example, we define a function that will always
// return a Person type. However, a function like this should
// return a `Result`; see the next section for details.
let me: Person = getPerson(byUUID: MY_UUID)

if let middle = me.middleName {
    io.printLine("My middle name is {middle}!")
} else {
    io.printLine("I don't have a middle name!")
}
```

Result

Any operation that can cause an error, such as a function call, must use the `Result` type. It is essentially a set with two elements; the first element is the type of the value you are expecting, and the second element is some type that implements the `Error` protocol.

```
function getPerson (
    byUUID uuid: String
): Result<Person, Error> {
    try {
        let person: Person = queryPersonTable(byId: uuid)
        return person
    } catch Error {
        return Error
    }
}

function main {

    // Now that we have modified that function to return
    // a result, we can gracefully handle the case where no
    // user with said UUID exists in the database.
    match getPerson(MY_UUID) {
        Some(me) => io.printLine("Hi, I'm {me.firstName}!"),
        Error => io.printLine("I don't exist!")
    }
}
```

Custom Types

Aliasing

Type aliasing is a simple way to give another name to an existing type, in order to provide clarity with respect to the purpose of elements of that type. You can reference a specific type, or even provide specific, valid values for instances of the new type.

```
type Byte = U8
type Fruit = "apple" or "orange" or "banana" /* and so on */
```

Composition

If you want to provide a higher-level abstraction, such as a type to represent a Person, then you will want to create a *type composition*. It allows you to define a structure with a set of properties, with each property having some type.

```
type Color = {
    r: U8,
    g: U8,
    b: U8
}

type Person = {
    age: U8,
    name: String,
    favoriteColor: Color
}

let blue = Color (r: 0, g: 122, b: 255)
let me = Person (
    age: 25,
    name: "Sam",
    favoriteColor: blue
)
```

Type Parameters

Type parameters allow you to add additional constraints to its instances. For example, the standard library `List` type has a parameter indicating the type of each item in the list. When you try to add an item to the list, we verify that its type matches the type of the list.

To demonstrate, we show a rudimentary implementation of the Iron list structure below.

```
type MyList (Type) = {
    itemType: Type,
    items: [Type]
}

implement Collection for MyList {
    function append(to list: Self, _ item: Type) {
        if type(item) equals list.itemType {
            items.append(item)
        } else {
            throw TypeError
        }
    }

    /* Other protocol methods */
}

function main {
    let fruits: List<String> = []
    let nums: List<U8> = []

    fruits.append("apple")
    fruits.append("banana")
    fruits.append("orange")

    nums.append(1)
    nums.append(2)
    nums.append(3)

    try {
        stringList.append(true)
    } catch TypeError {
        io.printLine("Type mismatch!")
    }
}
```

Ownership

Ownership is an important aspect of the Rust programming language that ensures memory safety without relying on a garbage collector. Iron implements a very similar system to provide the same guarantees.

Concurrency

Being able to execute multiple tasks at the same time is a critical aspect of modern software systems. Iron draws inspiration from languages like JavaScript and Go.

The `Async` and `Await` Keywords

Here we will talk about how any function can be run asynchronously, by using the `async` keyword before the function call. When you do this, the expression returns a `Promise` type, optionally resolving to some value.

```
function futureString: String => "Hello from the future!"

function main {
    let helloPromise: Promise<String> = async futureString()

    /* Do some other stuff here... */

    let hello: String = await helloPromise
}
```

The `Channel` Type

Channels allow you to send data between asynchronous functions.

```
import { Channel } from "iron/concurrency"

function channelDemo (
    _ in: &Channel,
    _ out: &Channel,
    _ message: String
) {
    while true {
        let recv: String = await in.receive()
        io.printLine("Received message: {recv}")
        out.send(message)
    }
}

function main {
    // You have to use type hints here to specify what kind
    // of data will be sent and received by the channels.
    let pingChannel: Channel<String> = Channel()
    let pongChannel: Channel<String> = Channel()

    // With our channels created, we now spawn two routines
    // that use them to communicate with each other.
    let pinger = async channelDemo(
        &pongChannel, &pingChannel, "Ping!")
    let ponger = async channelDemo(
        &pingChannel, &pongChannel, "Pong!")

    // We need to send an initial message to get things
    // started. Otherwise, `pinger` and `ponger` would
    // wait forever for their first message.
    pongChannel.send("Let's get started!")

    // Just allows the process to go on indefinitely.
    // Demonstrates how we can use `await` on multiple items.
    await pinger, ponger
}
```