Language semantics

Outline - 3/11/2019

Semantics

The language (*name tbd*) is going to be a single threaded, asynchronously executed language. It will contains classeswith classic Java style single inheritance. It will be dynamically typed, with relatively strong type coercion rules. The language will have first order functions. It will be garbage collected. There will be a keyword to execute any function asynchronously.

Decision reasons

There are many language semantics details to be decided on, including

- Type system (strong vs weak, static vs dynamic, duck typing)
- Paradigm (Functional, Object-Oriented, Declarative).
- First-class types (integers, float, combined number type, bool, strings, lists, functions, structs, colours).
- Memory management (manual, garbage collection, reference counting).
- Concurrency primitives (threads, co-routines, none?).
- Meta-programming constructs (macros, code-as-data, none?).

Type System

Static vs Dynamic

As the domain of the language is education, I find myself leaning towards a dynamic type system. What this means is that variables and functions are not strict about what types they accept, they will at runtime attempt to behave according to the language spec and raise a runtime error if this is not possible.

This will mean trading away catching type errors at compile time for the freedom of allowing the user to be more free and fanciful with the types of objects they are passing around.

Personally I dislike using dynamically typed languages, but I do appreciate how it allows the user to quickly write short and effective code, at the expense of throwing away the compilers guidance. If this language was intended for use in real systems I would not use dynamic typing, but I think it is the best fit for the domain.

Weak vs Strong

The weak vs strong decision is more of a choice of where on the spectrum I want to lie, On one end of the spectrum we have weak typing, where the runtime will try it's very best to cast one type to the correct type needed by the operation. The opposite end of the spectrum is strong typing, where the runtime will refuse to compute any function or store any value who's type is at-all different from the one it expects.

For this language I am leaning towards a slightly strongly typed language. In practice what this will mean is I want the runtime to take the initiative of converting ints to floats if you multiply by a float (*If I have ints/floats*) or converting ints to bools for checks. However it will not do things that languages like JavaScript will do like allowing adding strings and integers, or equality comparisons of lists and structs. I think this relatively middle ground will be forgiving enough for users to not have to waste characters casting types to do every calculation, but not so forgiving to create unpredictable behaviour.

For an example of unpredictable behaviour of a weak type system type `["0","1","10"].map(parseInt);` into any JavaScript console.

Paradigm

The choice of paradigm for the language is a big one, but I think I will fall in line with the status quo for educational languages and use an OOP-esque approach. The language won't be an OOP zealot's dream of perfect encapsulation and abstraction, but the language will rely on inherently mutable objects, which possess attributes, and can in some way inherit attributes or behaviour from some ancestral objects.

I've come to this decision largely due to the influence of the graphics display stage, this stage is an inherently mutable environment with agents or actors which can move and interact with the stage and each other. There will be much less of a cognitive load on users if they feel free to directly mutate the objects.

Inheritance is not a necessary consequence of the above decision, but it is a prevalent feature of many (*one might say most*) common industry languages. There is also a cultural Irish connection with inheritance, Irish names historically and traditionally are inherited from the parents first names. For example the name *O'Carroll* means "From Carroll", and *MacDiarmuid* means "Son of Diarmuid".

Inheritance

There are many approaches open to me to implement inheritance.

- **Single Inheritance**:
  This is the approach used by a lot of mainstream languages, notably Java and C#. A new structured type can inherit attributes and methods from a parent type. The child types here can be seen as the parent type with more bolt's attached. I think that using this approach in the language would be beneficial as it is a common foundation to learn the ideas of inheritance from.
- **Multiple Inheritance**:
  This is basically the same as the above approach, but instead of inheriting properties of parent classes, it inherits from multiple classes, this does slightly break the idea of the child class being a direct subtype of the parent, but it does find prominent use in languages like Python. I think that the language doesn't necessarily need to support multiple inheritance, it's absence will likely not be felt or desired by beginners.
- **Prototypal Inheritance**:
  This approach is used in JavaScript and Lua, instead a new class inheriting attributes and behaviours from some parent object, it inherits properties from a parent **object**. A given object has a link to a parent object embedded inside it, this means that if many objects all inherit from the same object and that object is mutated during the runtime, then all those children's inherited properties will also mutate.
  This approach is also relatively beginner friendly, but the potential mutation of inherited properties during the execution of the program might prove problematic.
- **Struct Embedding**:
  This approach is similar to the prototype approach, but it can be argued it isn't *really* inheritance. Instead of a link to some parent object being included with every object, a parent object is directly into the child object. The child object inherits the properties from this parent object, but unlike the prototype approach the parent object doesn't really exist outside of the context of providing behaviour to it's child object. This approach is used by Go.

In conclusion I think that it's probably simplest and best to just use classic single inheritance.

First-class types

I would like to have as many objects be first class as possible. This includes the obvious primitive types, e.g. bools, numbers, characters, but also functions and structs. I think that

higher order functions and first class functions is a valuable concept for someone to learn, and I personally find them very useful, So I would like them to be included.

Memory Management

This decision is relatively simple, demanding manual memory management on a general education language seems like lunacy to me, so I will probably use GC, which will either be provided by the JavaScript engine, or the Go compiled WASM runtime.

Concurrency primitives

I think the cost-benefit balance of including some sort of concurrency primitives in the language is tilted in favour of not doing it. It adds a layer of semantic and cognitive complication that I don't think is at all necessary in a beginners language, as well as requiring massive amounts of work to implement behind the scenes. So I think I will pass on this.

There is also a dichotomy between the Go runtime's use of *goroutines* and the single threaded world of javascript that I think would not be fun to try and wrangle.

Update 31/10/2019

Prototyping semantics in JavaScript is showing that some sort of concurrency will be needed. It need not be multi-threaded however, so we will not need things like mutices.

Meta-programming constructs

My thoughts on meta-programming are almost exactly the same as the ones for concurrency primitives. I think that, while interesting, they are out of place in a beginner friendly language.