Language Design:

OOP or not OOP or better OOP

www.huawei.com

Author’s name: Aleksei Nedoria
Date: Dec, 2019

HUAWEI

Huawei New Programming Language

- **Goal**: to improve the application development **ecosystem** for various Huawei devices
- **Ecosystem**: should include a unified, developer-friendly development environment that provides multi-platform application development and a high level of reuse.
- **Language**: support of component-oriented programming (COP): assembling (an essential part) of the program from ready-made components.
- One of the steps in the direction of COP is the right choice of OOP features.
- In this talk we do not consider COP directly, focusing on the OO paradigm implementation.

Confusing state of OOP Paradigm

The situation with the OO paradigm is quite confusing. In fact, there is no consensus in the IT community on what OOP is.

OOP in Go, Rust, Elegant Objects (EO) and Lua is fundamentally different from OOP in C++ and Java.

Feature	C++	Java	Go	Rust	EO	Lua
Class	+	+	-	-	-	-
Class (static) methods and properties	+	+	-	-	-	-
Interface	-	+	+	+ (trait)	+	-
Interface inheritance (abstraction)	+	+	-	+	+	-
Implementation inheritance	+	+	-	-	-	-
Immutability	-	-	-	+	+	-
Dynamic creation of a class	-	-	-	-	-	+ (аналогов классов)

Criticism of class-based OOP

Languages with object orientation based on classes and implementation inheritance (CLOP languages, where CLOP – Class-Oriented Programming) are criticized for the lack of flexibility and for the problems of developing reusable components.

Joe Armstrong, author of Erlang:

> *I think the lack of reusability comes in object-oriented languages, not in functional languages. Because the problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.*

We need better OOP.

Our steps to better OOP or CLOP is a Bug

1. Not CLOP
2. No implementation inheritance
3. Better support for component reuse

Better reuse

- the ability to reuse components (**compile once, use everywhere**) for different devices significantly increases the efficiency of COP.
- Hence, the requirement to extend objects (add methods) without having to make changes to the source code and to minimize recompilation.

An example of the object extension - 1

Let's define List data structure with Append and Remove methods.

```
type List = struct {
    … fields …
    fn (l: List) Append(e: Element) …
    fn (l: List) Remove(e: Element) …
}
```

Suppose some applications need an operation to add a list to the list.

Method 1*: implement as external function in a separate compilation unit*

```
fn AppendList(to, from: List) …
```

Disadvantages:

1. One must import the AppendList function additionally (and be aware of Its existence);
2. Lack of uniformity: calling the AppendList function is different from calling Append;
3. The performance of this function is likely not best.

An example of the object extension - 2

Method 2*: add AppendList to original List*

```
List = struct {
     fn (l: List) Append(e: Element) …
     fn (l: List) Remove(e: Element) …

     fn (l: List) AppendList(from: List) …
}
```

This eliminates the drawbacks of method 1, but leads to a change in the source code and the need to recompile all software parts that use List.

Both of these methods do not suit us, we want to provide an extension without compromising performance, and without having to recompile those parts that do not use AppendList.

An example of the object extension - 3

Method 3: *separating to unite*

Let's separate atomic (separately compiled) entities:

- List definition as a hidden type
- List:impl structure with specific implementation details
- Methods Append, Remove that use List:impl

And then assemble the “object” from its parts using a “usebox” unit:

```
usebox std.containers.list
import …
export List:impl as List + Append + Remove
```

In order to add AppendList we implement it separately and then define another usebox:

```
usebox std.containers.list2
export List:impl as List + Append + Remove + AppendList
```

Method 3 Advantages

- All program parts that don’t need AppendList still use first usebox (no need to change or recompile)
- All program parts that do need AppendList will use second usebox (need to change import and recompile)
- There is no code duplication, since usebox is a compile-time unit. Both std.containers.list и std.containers.list2 can be used in one program, but the code of Append, Remove methods will be not duplicated.

Encapsulation

The proposed mechanism contradicts the conventional encapsulation, and, at first glance, may reduce the reliability of programs due to the possibility of access to the List implementation details.

In fact, we are used to conventional encapsulation and do not notice its oddities. Instead of mechanical protection from "all", it is necessary to protect the implementation details from those who can spoil it. Or from those who are not authorized.

Access levels:

- Access to the method code: for method developer
- Access to the structure implementation for use (not for change): for method developers
- Access to the structure implementation for change: for code owner

Conclusion

Our experimental language will contain several OOP-like features to improve extensibility and reuse.

The proposed feature is used for so-called horizontal extension, other features will help with vertical extension (like inheritance), adding dynamics (trait objects and duck typing) and generics.

We are working on a language, compiler and run-time system prototype to check the usefulness of the proposed approach.

Thank You