COBRAS: Classes and Objects By Reapplying Array Structures

Zhemin Qu, Hongxiao Zheng

October 2022

The technical documentation for Garter — and the real COBRAS!

1 Concrete Syntax

The Class and Object function involves declaring and defining class with field and methods, constructing objects based on existing classes, and calling methods on the objects.

Also, it is natural for the programmers to modify the values in the fields of an object, so the fields are required to be mutable, and the SetField syntax is needed.

As a result, the following concrete syntax should be added to the language:

```
<expr>:
    ...

    |class identifier { ids } : decls in expr

    |new identifier

    |expr.identifier( exprs )

    |identifier := expr
```

which are used to define classes, construct objects, call methods, and modify fields correspondingly.

All object fields as set to be private intentionally, thus accessing to object fields without a method is not allowed.

2 Example Snippets

Following is an example of the usage of classes and objects. The program should simply return 483.

```
class Dummy { dummy }:
    def SetDummy(x):
        dummy := x
    and def GetDummy():
        dummy
in let dum = new Dummy
in dum.SetDummy(483)
in dum.GetDummy()
```

3 Abstract Syntax and Semantic Checking

Due to the new Concrete Syntax, the parsed Abstract Syntax Tree may be modified to include the following patterns correspondingly:

```
pub enum Exp<Ann> {
    ...

    Exp::Classdef {
        name: String,
        fields: Vec<String>,
        methods: Vec<FunDecl<Exp<Ann>, Ann>>,
        body: Exp,
        ann: Ann,
    }

    Exp::Object {
        name: String,
        ann: Ann,
    }

    Exp::CallMethod {
        object: Exp,
        fun: String,
        args: Vec<Exp>,
        ann: Ann,
    }

    Exp::SetField {
        field: String,
        newVal: Exp,
        ann: Ann,
    }
```

```
}
```

The following errors should be checked and return compiling error:

- Undefined Class, when trying to construct an object with an undefined class.
- Undefined Method, when trying to call an undefined method.
- Duplicate Field, when duplicate fields with the same identifier are declared in a single class.
- Duplicate Method, when duplicate methods with the same identifier are defined in a single class.
- Class Used as Variable, when trying to call a variable but found a class.

When method trying to manipulate an undefined field, it should still return UnboundVariable as before.

Type checking for objects when calling methods will be done during run time.

4 Uniquify

Class names and fields should be uniquified in the similar way as the function and variable names.

Due to the dynamic typing property, when calling methods, the type of objects remain unknown in the compiling process. Then when a method name is shared by multiple classes, all the uniquified names of the specified method should be recorded related to the corresponding classes. Technically, this information should be stored in a HashMap sturcture.

Thus, a new pattern for CallMethod Exp should be created when Uniquifying method names:

```
pub enum Exp<Ann> {
    ...

    Exp::CallUniqMethod {
        object: Exp,
        fun: String,
        uniqfun: HashMap<String, String>,
        args: Vec<Exp>,
        ann: Ann,
    }
}
```

5 Class Lift

Before conducting lambda lifting, class lifting is now newly added to the compiling process. Since we have already checked the scope of all classes, it won't affect the correctness of the program.

The process is similar to lambda lifting, with a new structure defined:

```
pub struct ClassInfo {
    id: usize,
    fieldsize: usize,
}
```

The prototype of class lifting should be as following:

```
fn class_lift<Ann>(p: &Exp<Ann>)
    -> (HashMap<String, ClassInfo>, Exp<()>)
```

The HashMap maps from the class name to its corresponding ClassInfo structure. Here, notice that besides lifting the classes, there are a few other jobs to be done:

1. Regard the fields as an array, and insert the array to the front of the argument vectors of the class methods declaration.
2. Replace all the fields applied in the methods as array element. In detail, changing field varibles to Prim2 Arrayget, and changing SetField to ArraySet.
3. Define a new pattern Fielddef in Exp as following:

   ```
   pub enum Exp<Ann> {
       ...

       Exp::Methoddef {
           classidx: usize
           decls: Vec<FunDecl<Exp<Ann>, Ann>>,
           body: Box<Exp<Ann>>,
           ann: Ann,
       }
   }
   ```

   Methoddef holds a classidx field that fundef doesn't have, which represent the unique usize id of the corresponding class (1-indexed).

   Replace the method definitions originally inside the class declarations as Fielddef, at exactly the same place that the classes are originally defined.

6 Lambda Lift

Similar as the original version, except for a tiny modification on the output fundecls: First define ClassFunDecl structure as following:

```
pub struct ClassFunDecl<E, Ann> {
    pub name: String,
    pub classidx: usize,
    pub parameters: Vec<String>,
    pub body: E,
    pub ann: Ann,
}
```

Then the function prototype may be modified to:

```
fn lambda_lift<Ann>(p: &Exp<Ann>) ->
    (Vec<ClassFunDecl<Exp<()>, ()>>, Exp<()>)
```

And the MakeClosure pattern should be modified accordingly:

```
MakeClosure {
    classidx: usize,
    arity: usize,
    label: String,
    env: Box<Exp<Ann>>,
    ann: Ann,
}
```

The classidx field is the unique identifier of classes as stated in class lifting process,

The classidx field in the structures should be set as 0 if the fundecl is a normal function instead of a class method.

Still, make closure at exactly the original place that the function is declared.

7 Sequentialize

Sequentialize should work similarly as the original version.

Similar as in lambda lifting, makeclosure pattern is required to be modified. Object and CallMethod patterns should be added.

```
pub enum SeqExp<Ann> {
    ...

    Object {
        name: String,
        ann: Ann,
    }

    CallMethod {
        object: ImmExp,
        method: String,
        uniqfun: HashMap<usize, String>,
```

```
            args: Vec<ImmExp>,
            ann: Ann,
        },
    }
```

Notice that the uniqfun field in CallMethod is changed from HashMap<String, String> to HashMap<usize, String>. The usize is the class identifier of the class name, which can be achieved from the class Hashmap returned in the Class Lift process.

8 Compile to Instructions

8.1 Arrays and Objects

Objects share many similarities with arrays, especially when our arrays are actually tuples that is not limited to one type. The only difference is that objects hold some specific type, which is the corresponding class. Thus, we can extend the array structure to implement the objects:

Objects and arrays should share exactly the same type tag on the stack, and the same structures on the heap.

The type tag follows the original tag of the arrays.

The structure of arrays on the heap can be partitoned into 3 parts: **class id**, **array length**, and **array elements**. The structure should hold the length of $n+2$ in total, where n is the length of the array.

Since we defined the class ids starting from 1, it would be natural to give the normal arrays the class id 0.

Then the type checking on ArrayGet and ArraySet methods should be operated according to the classidx of the functions:

- For main functions and functions with classidx 0, only arrays with classidx 0 on the heap should be allowed to apply arrayget and arrayset.
- For functions with classidx x, where $x \neq 0$, only arrays with classidx 0 and x on the heap should be allowed to apply arrayget and arrayset.

For simplicity, we set the array length as snake values (tagged with a tailing 0).

8.2 Methods and Closures

Just as the relationship between objects and arrays, methods are basically a special version of functions, then calling methods are basically calling functions with a class type restriction.

Thus, we extend the closure structure on the heap:

The structure can be partitioned into 4 parts: **class id**, **function pointer**, **arity**, **environment array pointer**.

Notice that when calling a method, the object itself should be inserted as the first argument (but after environment array).

The assembly code generation for calling methods would be a bit complicated than calling a normal function. Since as stated in uniquifying process, it is possible for two classes to hold methods with the same name, and when calling the method, it still requires to be checked which class the object is in, so as to determine which closure to be called.

When closures are called, it is required to check whether the closure is designed for the corresponding type, which can be shown in the classidx.

In technical detail:

- When calling normal functions, check whether the class identifier in the closure structure on the heap is 0. If not, a Misusing Methods as Functions error should be raised.
- When calling methods, iterate through the uniqfun HashMap field, check whether the class identifier matches with the object. If matches, break the loop and call the corresponding closure.

  Note that at most one item in the HashMap may match with the object in class identifier, since there is not allowed to be two methods sharing the same name defined in a single class.

  If none of the elements in the HashMap matches with the object in class identifier, a Misusing Methods from other Classes error should be raised.