Project Report

Cppish: Enhancing Cish Compiler for Object Creation, Pointers, and Smart Pointers Support

Team Members:

- Manas Vegi (mv2478)
- Vashu Raghav (vr2326)
- Jayanth Reddy (nr2686)

Code link: https://github.com/jayanthreddy1997/CPPCompiler

---

1. Introduction

Our project aims to enrich the capabilities of the existing Cish compiler by incorporating advanced features like dynamic object creation, pointers, and smart pointers. The implementation of these smart pointers was an interesting endeavor since most smart pointer implementations are written in higher-level programming languages as a library whereas we are trying to handle them by generating instructions at the compile stage.

By integrating these features, we envisioned facilitating developers to craft more intricate programs in Cppish (a C++-like language), thereby fostering enhanced memory management and code structuring. Moreover, this compiler design and programming effort served as a valuable educational experience, deepening our understanding of compiler design principles, particularly on memory management support.

2. Related Work

Our project builds upon the foundation laid by our previous work on the Schimish to Cish conversion. Specifically, we extended the capabilities of the Cish compiler by adding support for object creation, pointers, and smart pointers. This project serves as a natural progression from our earlier assignments, demonstrating our continued learning and development in compiler design.

In designing our code for class compilation and object creation, we drew inspiration from the lecture slides on "Compiling Object-Oriented Languages." Specifically, the concepts discussed in these slides provided valuable insights into memory layout, which greatly influenced our approach. By understanding how class structures and object instances are represented in memory, we gained a clearer understanding of how to support classes and objects, and extend our support to smart pointers within the Cish compiler. One key takeaway was the realization that class member variables could be efficiently stored as offsets within the object's memory layout. However, for the compilation of class methods and handling of smart pointers, we opted for a different approach, which we will elaborate on in the subsequent sections.

3. Proposed Features Support

The features we provide in addition to Cish follow below:-

1. Class Definition: We supported the definition of Classes. The class definition is made up of the class name, attributes belonging to the class, and methods that an object can invoke including a parameterized constructor. In our implementation, we restrict that the first parameter of any class method must be ‘this’. This is similar to classes in Python where ‘self’ is a compulsory first parameter for methods of a class. The syntax goes as follows.

```
class Calculator {
  let x;
  let y;

  Calculator(this, a, b){
      this.x = a;
      this.y = b;
  }

  add(this){
      return this.x+this.y;
  }
}
```

2. Object creation: We supported the creation and usage of dynamic objects with the new keyword. The syntax for this is as follows:

```
let Calculator obj = new Calculator(10, 5);
```

This syntax strays away from real C++ a little due to the absence of the '*' operator. However, the semantics remain the same where obj is now a pointer type belonging to that class.

3. Class attribute and method invocation by object: We provide the feature for invocation of attributes or class methods from the object. We also provide the feature of accessing the attributes within class method definitions using the 'this' keyword. Examples of the syntax for these features in our Cppish are as follows:

```
// Outside class

let x = obj.add();
let y = obj.var1;

—-----------------------------

// Inside class
add(this){
      return this.x + this.y;
}
```

The next round of features involves pointers, shared pointers & unique pointers.

4. Pointer initialization and usage: We support int pointers (same as in Cish) and pointers to class objects. For non-smart int pointers, in our Cppish, they must be explicitly malloced instead of the '*' syntax. Dereferencing pointers is the usual C++ or Cish syntax.

```
main() {
  let x = 12; {
  let y = malloc(4);
  return (*y);
  }
}
```

5. Shared pointers:

```
let shared_ptr<ClassName> obj1 = new ClassName(arg1, arg2);{...}
let shared_ptr<ClassName> obj2 = obj1; {...}
```

Shared pointers are supported by our CPPish. These pointers keep track of how many variables are pointing to a memory location. When variables go out of their let scope, the reference counts are decremented. When the reference count of an object hits 0, we invoke the free operation in Cish (which is an extension we put in Cish to reflect the use of shared_ptrs).

6. Unique pointers:

```
let unique_ptr<ClassName> obj1 = new ClassName(arg1, arg2);{...}
```

Unique pointers make sure that only one variable points at that memory location and throws a compile error when it sees an assignment with a unique pointer variable on the RHS.

4. Implementation

The following provides an in-depth overview of the implementation details supporting our features.

1. Class Compilation

A class gets compiled down to functions (since Cish doesn't support classes). Each compiled function takes a pointer to the object as the first argument. We use "classname_methodname" as a convention to name the generated functions.

An AST program is now a list of classes and functions as opposed to a function list in Cish as can be seen in the code below

```
type funcsig = { name : var; args : var list; body : stmt; pos : pos }
type func = Fn of funcsig

type classsig = {
  cname: class_name;
  cvars: var list;
  cmethods: func list
}

type class_member = {
  cvars: var list;
  cmethods: func list
}

type klass = Klass of classsig

type func_klass =
  | Fn2 of func
  | Klass of klass

type program = func_klass list
```

Objects of classes are represented by blocks of data allocated on the heap (using malloc). Class variable references are converted to load and store operations from the heap memory. Variables are laid out in the block in their order of definition in the class. We maintain a map in Ocaml from class to a list of variable names to identify the offset of each variable. We also maintain a map (class_method_map) from the class to its methods, this is to ensure valid functions are called for each class. When a method is invoked on an object (say "obj.method1(arg1, arg2)") we first look at the class it belongs to (maintained in object_class_map), then check if the called method is a valid class method (check from class_method_map), followed by a Call the function (named "classname_method1") with its first argument as obj pointer and followed by function's arguments.

2. Pointers

We support all pointer operations that are supported in Cish like malloc, load, and store. Currently, Cppish malloc redirects to Cish malloc, which is handled by the Cish evaluator that allocates a memory location in an Array. Since our language

restricts int pointers to be malloced, we can directly generate the corresponding malloc statement in Cish and assign it to the pointer variable.

3. Shared pointers

For each class object, we store a reference count variable on the heap to store the number of pointers that point to an object. The first 4 bytes of the object on the heap are the reference count followed by the rest of the class variables. Once the reference count hits zero, we trigger a call to “free” that would deallocate the memory from the heap. This is done by adding instructions to the compiled code that check the value of the reference code and call free if it is 0 in a Cish if-statement. Shared pointers also support assignments to other shared pointers, in which case the reference counts change accordingly.

4. Unique pointers

We have added checks to identify assignments of unique pointers to other pointers, in which case we throw compile errors. For this, we needed to keep track of which variables point to unique pointers, we maintained this using a hash table. Similar to shared pointers, unique pointers also support automatic deallocation once they are out of scope.

Language and Design Decisions

a. Every class method needs to have “this” as the first argument
   Every class method gets compiled down to a regular function with the first argument as a pointer to the class. This is similar to how Python handles class methods.
b. Objects of a class have a default attribute called ref_count
   In the space allocated for an object of a class, the first 4 bytes go towards the integer that will represent the ref_count of the object. This allows for easy updates to the ref_count when using shared_ptr or unique_ptr.
c. Each class needs one constructor. The compiler can be extended to support a default no-arg constructor and multiple constructors (function overloading)

5. Evaluation

To ensure the correctness of our compiler implementation, we employed a comprehensive testing approach, complete with manual validation and automated testing procedures.

Manual Validation:

1. **Parser Functionality**: We evaluated the parser to ensure the correct handling of our supported features. We made test programs for each proposed feature to verify parsing accuracy.
2. **Comparison of ASTs**: We validated the correctness of our CPPish compiler by verifying the corresponding Cish AST it generates for each test case. The Cish AST was tested by running the cish_eval file. This step ensured that our CPP compiler was producing the intended Cish code accurately.
3. **Evaluation of Generated Code**: Following AST comparison, we evaluated the generated Cish code to ensure its correctness and adherence to the intended semantics.

Automated Testing: In addition to manual validation, we developed an automated testing script to execute test cases. This script facilitated the evaluation of our compiler's performance across various scenarios, including both existing Cish test cases (approximately 50) and new test cases tailored to our added features. The script systematically ran each test case and gave the compiler output, providing a way to compare our results. The tests can be found here.

6. Conclusion

In conclusion, we successfully created a Cppish extending the Cish compiler to support dynamic object creation, pointers, and smart pointers, thereby enriching the language with advanced memory management capabilities. We ensured the correctness of our Cppish compiler through a comprehensive test suite. Thus, we have offered one possible implementation of these memory management features for a compiler.

7. Future works

1. Compile time polymorphism like function overloading and operator overloading
2. Type checking for types other than classes. The current compiler checks for the class types and throws errors if a wrong attribute or method is invoked.
3. Inheritance support for classes.