1 Guidelines

Your solution must be submitted in a single zip file, named as *Project_XXX.zip* with *XXX* replaced by your UID. Please submit your solution to Moodle before the deadline. The deadline (*24 May 2022*) is set by the University, and we cannot change it. So please **make sure you arrange your time accordingly** to submit on time.

1.1 Code

The main advice for you is to get the basic functionalities correct (i.e. Parts A&B). This alone will probably guarantee a nice grade for the course. If time permits, you can try to add some extra features (i.e. Part C) to get a higher grade.

Though we have provided some test cases for basic features in `examples/`, you would better add more for your extra features on your own. You can test your code against all the examples using `stack test`.

1.2 Report

You must also submit a small report in the PDF format, which

- describes how you have implemented each basic feature,
- introduces the extra features that you have implemented, and
- indicates what examples you have used to test your interpreter.

Note that your report is important for us to evaluate your free extensions. You should elaborate on your proposed features, give some examples, and describe how these features are implemented. If you referred to any external resources, please cite them in your report.

2 Introduction

In this project, we will explore the design of a simple object-oriented language. The language is based on the ς-calculus (pronounced sigma calculus), which is a core object calculus proposed by Abadi and Cardelli [1]. We refer interested students to the original paper but this document is self-contained and explains the major concepts involved.

As the λ-calculus lays the foundation for functional programming, the ς-calculus is an attempt to lay the foundation for object-oriented programming. Instead of encoding objects as complex λ-terms, it introduces the concept of primitive objects in a minimal calculus. There are four basic constructs in the ς-calculus:

1. variables,
2. objects,
3. method invocations, and
4. method updates.

If you have used object-oriented languages before, you should be familiar with the first three constructs. The fourth construct is related to method overriding but is significantly more powerful.

Variables are standard and appear in all programming languages.

Objects are collections of methods. An important point of the ς-calculus is that classes are not needed to build objects. Instead, objects can be built directly. In the ς-calculus, there is no `new` construct, which is used by object-oriented languages like Java to create an object from a class. In other words, the ς-calculus is an *object-based* language rather than a class-based language. As a side note, JavaScript is also object-based, though classes have been added as syntactic sugar since 2015. We will also add classes to our source language later. Let us see an example of object literal first:

```
{ l1: self => self, l2: self => 0 }
```

Here, we create an object directly. The syntax for objects is minimalistic, and it is basically a collection of components enclosed by square brackets. Each component starts with a label and has a method definition.

Note that a method definition in the ς-calculus takes a special parameter, which is the *self-reference* to the object. For instance, the method definition for `l1` is `self => self`.

The left-hand side introduces a variable called `self`, which is the self-reference. You can find such a notion in Scala with a very similar syntax, and it plays a similar role to `this` in object-based languages like JavaScript. But the ς-calculus is different from those languages in that:

- Every method has to explicitly introduce a variable for the self-reference, in contrast to languages like JavaScript, where `this` is a keyword and is implicitly available in an object.
- The self-reference can be arbitrarily named. For instance, we could rewrite the previous object as:

  ```
  { l1: x => x, l2: x => 0 }
  ```

  In both examples, the first method (`l1`), returns the object itself. The second method `l2` returns `0`. The two versions of the program behave in exactly the same way. The only difference is syntactic: the self-reference is called `self` in one program, while it is called `x` in the other. You could even choose different names for different methods.

Method invocations allow calling a method on an object, and they are similar to method invocations in standard object-oriented languages. A simple example illustrating method invocations is:


```
{ l1: self => self, l2: self => 0 }.l1.l2
```

In this example, right after the creation of the object, we invoke methods `l1` and `l2`. Since `l1` just returns the object itself, the subsequent method invocation `l2` will return `0`.

A more interesting example is to invoke `self.l2` via the self-reference in the method `l3`:

```
{ l1: self => self, l2: self => 0, l3: self => self.l2 }
```

Method updates are the most unusual construct. A method update allows *replacing* an existing method implementation in an object with another one. Some dynamically-typed languages offer similar functionality, but most statically-typed object-oriented languages do not allow this. An example of updating a method is:

```
{ contents: x => 0, set: self => self.contents := y => 10 }
```

In the code above, the `set` method will replace the implementation of `contents` by `dummy => 10`. Thus, if you execute:

```
{ contents: x => 0, set: self => self.contents := y => 10 }.set.contents
```

You should get `10`. Note that in the example, the self-references `x` and `y` are unused. In our grammar, we can omit unused self-reference variables. Thus, you can rewrite the previous object as:

```
{ contents: 0, set: self => self.contents := 10 }
```

The provided parser can handle both forms of methods and will add a dummy self-reference for the simplified form.

2.1 Goal of the Project

As part of the project bundle, we already provide an implementation of an interpreter for the ς-calculus. The project consists of three main parts:

1. To extend the basic interpreter for the ς-calculus with more constructs.
2. To create a more handy object-oriented source language that "*compiles*" into our core ς-calculus.
3. To allow creative extensions where students are free to add their own features and make improvements.

The first two parts of the project will account for 75% of the marks, while the third part will account for the remaining 25%.

2.2 Project Bundle

Just like previous assignments, we provide a fully configured Stack project for you. There are several useful commands you might have already known:

- `stack build`: compile the whole project.
- `stack run`: run a REPL (read–eval–print loop).
- `stack run -- examples/xxx.obj`: load the file and run it in the REPL.
- `stack test`: test all the files in the directory `examples/`.
- `stack clean`: delete build artifacts for the project.

The project bundle is organized as follows:

```
-- app
  |-- Main.hs
-- examples
  |-- xxx.obj
  |-- ......
-- src
  |-- Tokens.hs
  |-- Parser.hs
  |-- Common.hs
  |-- Sigma.hs
  |-- Interp.hs
  |-- Source.hs
  |-- Translate.hs
-- test
  |-- Spec.hs
```

In this project, you will mainly write code under `src/`:

- `Tokens.hs` is the tokenizer, generated from `Tokens.x`.
- `Parser.hs` is the parser, generated from `Parser.y`.

- `Common.hs` includes the common definitions used by the ς-calculus and the source language.
- `Sigma.hs` defines the abstract syntax of the ς-calculus.

- `Interp.hs` contains an interpreter for the ς-calculus.
- `Source.hs` defines the abstract syntax of the source language.
- `Translate.hs` implements the translation from the source language to the ς-calculus.

The directory `examples/` provides some examples that help you check your implementation. Feel free to add more.

3 Part A: ς-Calculus

You can find an implementation of an interpreter for the ς-calculus in `Interp.hs`. The definitions of abstract syntax can be found in `Sigma.hs`.

We start with the abstract syntax:


```
data Term = Var Var                             -- x, y, ...
          | Object [(Label, Method)]            -- { l: self => 0 }
          | Invoke Term Label                   -- x.l
          | Update Term Label Method            -- x.l := self => 0
          | Lit Value                           -- 0, 1, ..., true, false
          | Unary UnaryOp Term                  -- -10, !false
          | Binary BinaryOp Term Term           -- 4+8, true||false
          | If Term Term Term                   -- if (x < 0) -1; else 1
          | Let Var Term Term                   -- var x = 1; x + 2
          deriving Eq

data Method = Method Var Term deriving Eq
```


The first four constructs are the basic building blocks of the ς-calculus, which are already presented in the introduction. In addition to those, we also have some other constructs that have been covered in the lectures and tutorials: literals, unary and binary operations, conditionals, and local variable declarations. For example, we support code such as:


```
var x = true; { l: if (x) 1; else 0 }.l
```

If we evaluate this code, the result should be `1`.

3.1 Evaluation

The interpreter uses a standard environment that keeps track of local variables and the values associated with them. Moreover, it also uses a memory model, like in Lecture 12. The provided interpreter is written in a direct style (i.e. it does not use monads). However, monads can be helpful for making the code cleaner and easier to understand and modify. While you do not have to write your code in a monadic style, you are encouraged to do so since this may be easier to work with. Moreover, this can give you some extra points for Part C.

The type `Mem` models virtual memory (again, you are encouraged to replace `List` with a more suitable data type for `Mem` in Part C):

```
type Obj = [(Label, MethodClosure)]
type Mem = [Obj]
```

It is used to store objects in memory. For this language, we have to have memory, just as in the interpreter with mutable state, because objects are mutable. That is, the operation of method updates can actually modify methods stored in objects. Therefore, the memory stores all the objects that are allocated in the program.

Note that the objects stored in memory are essentially collections of method closures. Each method has a label as its name and a method closure. The method closure, similar to function closures, stores the environment at the point of definition of the method.

We use a `replace` operation to update a method in an object stored in memory:

```
replace :: Int -> Label -> MethodClosure -> Mem -> Mem
```


We have three kinds of values, namely integers, booleans, and object references

```
data Value = IntV Int
           | BoolV Bool
           | ObjRef Int
           deriving Eq
```


Whenever evaluating an expression that computes an "object", the result is not the object itself but a reference to the location of the object in memory.

The type signature of the evaluator for the ς-calculus is:

```
evaluate :: Term -> Env -> Mem -> Maybe (Value, Mem)
```

Basically, we have three inputs and two outputs. The types of the inputs are:

1. `Term`: the expression to be evaluated;
2. `Env`: the current environment;
3. `Mem`: the current memory.

The outputs are:

1. `Value`: the value that the expression is evaluated to;

2. `Mem`: the updated memory (in case method updates have been performed).

The main point is that when objects are created, memory is allocated to store the object information in memory. To access the objects in memory, we use object references.

Question 1. (10 pts.) Your first task is to complete the definitions of `evaluate (If _ _ _)` and `evaluate (Let _ _ _)` in `Interp.hs`.

3.2 Clones

The semantics of `clone(o)` is that it returns a *new* object with the same methods as the object `o` has. Any changes to the cloned value should not affect the original `o`. During the evaluation, you should look up the value of `o` in memory and allocate a fresh object with the same methods as `o` has. For example:

```
var o1 = { l: 0 }; var o2 = clone(o1).l := 10; o1.l + o2.l
```

We create an object `o1`, and then create a clone of the object, but with an updated implementation of `l`. The update of `l` should not affect `o1`. Thus, the final result should be `10` instead of `20`.

Question 2. (10 pts.) Complete the definition of `evaluate (Clone _)` in `Interp.hs`.

3.3 Strings

Besides integers and booleans, there are a variety of data types that are useful in our everyday programming. For example, strings are missing in the previous interpreters that are presented in tutorials, but they are ubiquitous in all programming languages. Therefore, we want to add strings as a primitive data type and support some operations on strings.

We have added `StringV` to `Value` and extended the parser to handle string literals. There are some new operators defined in `UnaryOp` and `BinaryOp` in `Common.hs`:

- `Length` (`#s`): return the length of the given string `s`.
- `Null` (`?s`): return `true` if the given string `s` is empty; otherwise `false`.
- `Index` (`s !! i`): return another string that only contains the `i`-th character of `s`.
- `Append` (`s1 ++ s2`): append two strings.
- `Remove` (`s1 \\ s2`): remove all occurrences of `s2` if `s2` is a substring of `s1`.

Question 3. (10 pts.) Implement the aforementioned operations in `Common.hs`.

4 Part B: Source Language

Realistic programming languages are often built upon a small core like the ς-calculus or the λ-calculus, but providing many convenient source-level features that can be encoded in terms of the small core calculus. We will take a similar approach in the project. In addition to the ς-calculus, which is our core language, we will have a richer language that translates to the ς-calculus.

The abstract syntax of our new source language is:

```
data Exp = Var Var
         | Object [(Label, Method)]
         | Invoke Exp Label
         | Update Exp Label Method
         | Clone Exp
         | Lit Value
         | Unary UnaryOp Exp
         | Binary BinaryOp Exp Exp
         | If Exp Exp Exp
         | Let Var Exp Exp
         deriving Eq

data Method = Method Var Exp deriving Eq
```


It is quite boring at the moment because it just duplicates the abstract syntax of the previous ς-calculus. We will add some non-trivial constructs to make the source language more interesting soon.

The most important function for our source language is:

```
translate :: Source.Exp -> Sigma.Term
```

It converts expressions in the source language to terms in the ς-calculus. The translation is straightforward since all constructs are mirrored so far.

Question 4. (10 pts.) Complete the definition of `translate` in `Translate.hs`.

4.1 Functions

The first thing you might notice is that we still do not have (first-class) functions. Actually, we can encode functions using the existing constructs in the ς-calculus. We will use JavaScript-like syntax for functions as before.

```
data Exp = ...
         | Fun Var Exp              -- new!
         | Call Exp Exp             -- new!
```

For instance, we can parametrize the `set` method in the previous example:

```
{ contents: 0
, set: self => function(n) { self.contents := n }
}.set(10).contents
```

This program will evaluate to `10`.

Question 5. (10 pts.) Support first-class functions as an encoding in terms of objects in the ς-calculus. In other words, you should implement `translate (Fun _ _)` and `translate (Call _ _)` in `Translate.hs`.

As for function definitions, the translation proceeds as follows (*pseudocode* is used here):


```
function(x) { body } -->

{ arg: self => self.arg
, val: self => |body|[ x ~> self.arg ] }
```


The idea is to encode a function using an object with two methods. The first method (`arg`) stores the argument. In the second method (`val`), `body` is first recursively translated (denoted by `|body|`), and then all the occurrences of `x` are substituted by `self.arg` in the translated `body`. A simple example is:

```
function(x) { x + 1 }  -->

{ arg: self => self.arg
, val: self => self.arg + 1 }
```

As for function calls, the translation proceeds as follows:

```
fun(exp)  -->

var f = clone(|fun|); var e = |exp|; (f.arg := e).val
```

The translation is also tricky: we first obtain a clone of the translated function, then perform a method update using the translated argument, and finally invoke `val` on the updated object. A simple example is:

```
(function(x) { x + 1 })(5)  -->

var f = clone({ arg: self => self.arg, val: self => self.arg + 1 });
var e = 5; (f.arg := e).val
```

4.2 Classes

In mainstream object-oriented languages, classes are a basic feature. To support classes in our source language, we need to add two new constructs:

```
data Exp = ...
         | Class [(Label, Method)]   -- new!
         | New Exp                   -- new!
```

For example, `Class [("l", Method "self" (Var "self"))]` is denoted as:


```
class { l: self => self }
```

Here is another example of a class definition:


```
class {
  contents: 0,
  set: self => function(n) { self.contents := n }
}
```


Classes look like objects. But instead of defining methods, they define *pre-methods*. Unlike objects in the ς-calculus, we cannot invoke a method on a class immediately. We must first create an instance of the class using the `new` construct. This is the same as class-based languages like Java. An example of `new` is:

```
var cell = class {
  contents: 0,
  set: self => function(n) { self.contents := n }
};
(new cell).set(10).contents
```

This program evaluates to `10`.

Question 6. (10 pts.) Support classes as an encoding in terms of objects in the ς-calculus.

The translation procedure of classes is basically to generate an object in two steps:

1. change all the pre-methods into methods that have a self-reference as a normal function parameter;
2. add a `new` method to create an object instance.

In other words, assume that `l` represents all methods in a class definition:

```
class { l: x => body }  -->

{ new: z => { l: x => z.l(x) }
, l: function(x) => body }
```

For example, if you have a class as follows:

```
class {
  contents: x => 0 ,
  set: x => function(n) { x.contents := n }
}
```


Then it should be translated into:


```
{ new: z => { contents: x => z.contents(x), set = x => z.set(x) }
, contents = function(x) { 0 }
, set = function(x) { function(n) { x.contents := n } } }
```


Object instances are created with the `new` construct. The expression `new klass` actually does two things:

1) translate `klass` into an object in the ς-calculus;
2) invoke the method `new` on the object.

4.3 Recursion

As you might expect, the current function definitions do not support recursion. But the good news is that we can also encode recursion in terms of objects. We introduce a new construct for this:

```
data Exp = ...
         | Letrec Var Exp Exp        -- new!
```

`Letrec` is different from `Let` in that the declared variable itself can be used in the declaration. For example, you can write a factorial function like this:

```
var rec fact = function(n) {
  if (n == 0) 1; else fact(n-1) * n
};
fact(10)
```

Question 7. (10 pts.) Support recursive variable declarations as syntactic sugar.

There are several encodings for recursion in terms of objects. Here we introduce one approach proposed by Mitchell et al. [3] This approach regards recursion as syntactic sugar and leverages the so-called *fixpoint* operator. First of all, we would like to desugar `Letrec` to `Let` in the source language:

```
var rec x = e; ...  ~~>

var x = fix( function(x) { e } ); ...
```

The recursive definition `e` is wrapped in a function taking `x` as its parameter, to which the fixpoint operator is later applied. The definition of `fix` is as follows:


```
fix = function(f) {
  { rec: self => f(self.rec) }.rec
}
```


Note that `fix` is defined in the source language instead of the ς-calculus since we want to use first-class functions. In essence, the encoding of recursion is different from the previous encodings because it is desugared to another expression in the source language, rather than translated to terms in the ς-calculus.

4.4 Arrays

If you are familiar with JavaScript, you will probably agree that an array is a special kind of object with its labels to be natural numbers. For example, the two-element array `[true, false]` can be represented as:

```
{ 0: true, 1: false }
```

We have added `Array` to `Exp` and extended the parser to handle array literals. The only thing you need to do is to implement `translate (Array _)`.

Question 8. (5 pts.) Translate arrays to objects in the ς-calculus. This is the last compulsory question.

Arrays alone are not very useful, so you are encouraged to continue to support some operations on arrays. In Haskell, there is a fancy feature called *list comprehension*. You can try to implement it in terms of arrays. Using array comprehension, we can perform complex transformations and do filtering:

```
var arr = [1, -1, 2, -2, 3, -3];
[ n * n | n <- arr if n > 0 ]
```

The code above selects non-negative numbers out of the array and returns another array that contains the square of every selected number. The final result is `[1, 4, 9]`. Moreover, there can be more than one array being iterated:

```
var a = [1, 2, 3]; var b = [3, 2, 1, 0];
[ x + y | x <- a, y <- b ]
```

Both `a` and `b` are iterated above, and a new array containing the sum of every pair of elements is returned. Note that the length of the final result is the same as the minimal length of all the iterated arrays. Therefore, the result of the program is `[4, 4, 4]`.

Having answered quite a few questions, you should be proficient in adding new features. Therefore, we do not provide concrete instructions from now on. It is your freedom to decide how to model, parse and implement array comprehension, and feel free to add auxiliary constructs to the ς-calculus if you think it necessary. Array comprehension is just one of the ideas worth trying. Please see the next page for more ideas.

5 Part C: Free Extensions

Question 9. (25 pts.)

The final part of the project gives you the freedom to improve the interpreter and extend the language in various ways, including making the interpreter more convenient to use or maintain as well as adding new language features. We hope you can unleash your creativity in this part. The more interesting your extensions are, the better grades you will get. For example, you may want to consider the following ideas:

- *Going monadic*

  Refactor the code of your interpreter to adopt a monadic style. You can encapsulate memory management using a stateful monad like in Lecture 12.

- *Improving error handling*

  Besides memory management, you may want to improve error handling using a checked monad like in Lecture 11. Our suggestion for you is to create a single monad that combines both monads. This requires you to create a suitable data type and corresponding monad instances.

- *Deprecating lists as memory*

  We have been modeling memory as a list so far. However, such usage is actually not idiomatic nor efficient. Please find a more suitable data type by yourself and refactor related code.

- *Adding more language features*

  While the current interpreter already has a few interesting language features, there are many more that can be added, including array comprehension, multiple function parameters, (single or multiple) inheritance, method overriding, etc. Generally speaking, novel features can earn you more marks than the boring ones that have been shown in previous lectures or tutorials.

- *Comparing native support and encodings*

  Although it is interesting to encode functions and lists using objects in the ς-calculus, they are probably less efficient than the native support shown in previous lectures and tutorials. You can implement functions and lists as primitive types in the ς-calculus and do some benchmarking to compare their performance.

- *Type checking*

  The current language is untyped, but it is possible to support static typing as well. You could try to add a simple type system to the language, perhaps without supporting all the features of the language. The original paper [1] contains inference rules for a simple type system. For simplicity, you can avoid the use of subtyping by using only equality. Moreover, note that supporting methods that return `this` is tricky, as this requires recursive types. So you may want to consider those methods as ill-typed (i.e. they would not type-check) since recursive types require some advanced concepts.

Some Pointers

Here are a few pointers that can help you with your extensions:

- *A Theory of Objects* [2]

  The topics mentioned in previous sections are covered by Part I of the book. You may read the book to deepen your understanding of the object calculi and try to implement more advanced parts.

- *Types and Programming Languages* [4]

  You may find more interesting ideas for language extensions in the book. By the way, you may want to read the chapter on *Imperative Objects* to see how to encode objects in $\lambda_{<:}$ instead of using a calculus with primitive objects like the ς-calculus.

- *Monads for Functional Programming* [5]

  The paper discusses more about the uses of monads. It can be helpful for you to improve the code of your interpreter.

- Alex and Happy

  You may need to modify the concrete syntax when you extend the language. The provided implementation uses Alex to generate a tokenizer and uses Happy to generate a parser. You can find their documentation by clicking the links above.

References

[1] Martín Abadi and Luca Cardelli. 1996. A theory of primitive objects: Untyped and first-order systems. *Information and Computation* 125, 2 (1996). Retrieved from http://lucacardelli.name/Papers/PrimObj1stOrder.A4.pdf

[2] Martín Abadi and Luca Cardelli. 1996. *A theory of objects*. Springer. Retrieved from http://lucacardelli.name/Talks/1997-08-04..15 A Theory of Objects (Sydney Minicourse).pdf

[3] John C. Mitchell, Furio Honsell, and Kathleen Fisher. 1993. A lambda calculus of objects and method specialization. In *IEEE symposium on logic in computer science*. Retrieved from http://crypto.stanford.edu/~jcm/papers/objects-njc.ps

[4] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press. Retrieved from http://ropas.snu.ac.kr/~kwang/520/pierce_book.pdf

[5] Philip Wadler. 1995. Monads for functional programming. In *International school on advanced functional programming*. Retrieved from https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf