A Haskell Interpreter for Object Calculus

Shobhit Maheshwari
Univ. of California, Santa Cruz
shmahesh@ucsc.edu

Aneesh Neelam
Univ. of California, Santa Cruz
aneelam@ucsc.edu

April 19, 2016

Abstract

We present an interpreter developed in Haskell for the Object Calculi introduced and detailed by M. Abadi and L. Cardelli. Object Calculus has been devised to represent objects directly as primitives, rather than to try to make use of Lambda Calculus that use function primitives, to objects. Since Haskell is not an object oriented language, we shall be deviating slightly from the syntax of the Object Calculi in our interpreter. But the Haskell interpreter will be based on the semantics, and typing rules of object-oriented languages specified by these Object Calculi.

1 Introduction

Lambda Calculus has been introduced by Church in the 1930s as a formal system in mathematical logic to express computation based on function abstraction and application using variable binding and substitution [9, 11]. Church numerals and encoding are used to represent data and operators in Lambda Calculus [10]. Functions are primitives in Lambda Calculus, and are directly implemented by functional programming languages.

For a given Lambda Calculus, one can implement a Haskell interpreter trivially. Various Lambda Calculi have been used as a foundation for procedural languages [2]. Interpreters can be built quickly for them.

However, Object-Oriented languages cannot be represented easily by Lambda Calculus. Object Oriented Programming constructs like Inheritance, Polymorphism cannot be represented easily by the Lambda Calculus and notation devised by Church. This is especially true for typed Object Oriented languages, when using typed Lambda Calculus [3]. It is not trivial even with Object Calculus to develop an interpreter in Haskell, a typed functional language, but it certainly is easier than directly adapting Lambda Calculus for object-oriented languages and then developing an interpreter for that [2].

Abadi and Cardelli devised Object Calculus to complement Lambda Calculus, and help represent object-oriented concepts in programming languages more easily [1–4]. There are a number of approaches that others have tried with various degrees of success when representing these languages and concepts.

In the next section, we shall see approaches that others have taken to model object-oriented languages, either adapting or directly using Lambda Calculus or in this case, devising a new formal mathematical system. In Section III, we shall have a quick overview of the Object Calculi devised by Abadi and Cardelli [2], and walk through the given syntax. In Section IV, we shall present our Haskell interpreter for untyped imperative Object Calculi, and in Section V, we conclude and discuss future work.

2 Related Work

These are the broad approaches taken to model object-oriented languages. More work has been done pertaining to each of the following approaches.

One of the earliest approaches to developing a formal system for modeling object-oriented languages was to use typed Lambda Calculus to encode objects, classes and methods in more primitive constructs like functions. Pierce specifies how lambdas and recursive records can be used to obtain a foundation for object oriented languages [8]. However, Abadi and Cardelli argue that Lambda Calculus is quite unsuitable for representing some of the concepts in object-oriented languages and hence devised Object Calculus [2, 3].

Castagna uses multimethod calculus to help represent object oriented theory and abstractions [6]. This approach is somewhat similar to Object Calculus, but it is more verbose and complicated. Object Calculus was devised to have a minimal syntax [2].

Bruce devised class-based calculi to model class-based programming. But the focus is on class-based languages,

and a limited set of object oriented concepts like classes and inheritance [5]. Abadi and Cardelli's Object Calculus manages to represent both Class-based and the broader Object-based languages [3].

3 Overview

We chose to implement the interpreter for the imperative untyped Object Calculus devised by Abadi and Cardelli in Haskell. The Object Calculus consists of objects, method invocation, method update, object cloning and local variable definitions [2, 3]. The syntax is as follows:

$a,b ::=$	terms
x	variable
$[l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$	object formation (l_i distinct)
$a.l$	field selection / method invocation
$a.l \Leftarrow \varsigma(x)b$	field update / method update

Figure 1: The syntax of untyped Object Calculus as defined by Abadi and Cardelli [3].

In Object Calculus, the Methods and Fields of the Objects are not defined individually. Methods and Fields are both defined as a mapping from label to method body in an object [2]. Labels are unique in Objects [2], otherwise it can break the mapping in the Object. A Method body without a function is a field [3], and treated as a field of the object. The function body can be construed as an adapted form of Lambda Calculus.

Objects are defined as a collection (or map from) of labels to methods. The labels are distinct within an object. Methods in an object can be looked up using the labels.

Method invocation evaluates the body of the method corresponding to the given label in an object, and assigns the result to the method's label and returns it [2].

Method update is used to add or update the methods of an object [3]. It replaces the method corresponding to the given label with the given method, or adds the given method and assigns the given label in the object.

Objects can be cloned easily in Object Calculus [2]. It is required to create multiple objects from an existing template. This can also be represented as a creating objects from classes in class-based object-oriented languages.

clone(o) produces a new object, with the same labels and the same corresponding methods as Object o [2].

Inheritance can be demonstrated by cloning an existing object. This is more akin to dynamic inheritance in object-based languages than the class-based inheritance in class-based languages [2]. The methods of the cloned object can be updated with different ones or new methods can be added to it. This represents method overriding and polymorphism in conventional object-oriented languages.

4 Implementation

Based on the rules and syntax specified in the previous section we can now implement an Haskell interpreter for untyped imperative Object Calculus. First, we define the data types and then the functions to evaluate method invocation, method update and object cloning.

4.1 Object Calculus

The Object Calculus is defined as a simple data structures in Haskell:

```
-- Object Calculus Term
data Term =
   Var String
  | Obj Object
  | MetInv String
  | MetUpd String Method
  deriving (Show, Eq)
```

We shall break the syntax down and specify what each of them correspond to in the Object Calculus specified by Abadi and Cardelli [2]:

```
Var String
```

This refers to the local variable definitions in Object Calculus. Since Haskell is a typed language, we specified it as a String. However, in untyped imperative Object Calculus, it can be of any type.

```
type Object = (Map String Method)
Obj Object
```

This refers to an object in Object Calculus. In Haskell, we defined an Object as a Map (Haskell's Data.Map) between labels(of type String) and Methods. Methods are defined as a separate data type in Haskell.

```
MetInv String
```

This corresponds to Method Invocation of Object Calculus. When evaluated, the Method corresponding to the given label (String) is evaluated. If the Method body is a field and not a function body then the evaluation simply returns the same field. If the Method body is a function, then it is evaluated in a manner similar to Lambda Calculus and the result is assigned to the same label in the object.

```
MetUpd String Method
```

This corresponds to Method Update of Object Calculus. When evaluated, the Method corresponding to the given label (String) is replaced by the given method. If there is nothing corresponding to the given label, then the given Method is added to the Object as a new Method. As mentioned before, Methods can be Fields or Function Bodies [3].

4.2 Object Method and Body

The Method is another data type defined to be either a field, or a Function Body.

```
data Method =
   VarM Int
  | Fun Body
  | FunM (Maybe Body)
  deriving (Show, Eq)
```

The Method can either be a Field or a Function Body. A Field in this case is defined to be an Integer. However, it can be of any other type as well. The Function Body is defined similar to Lambda Calculus.

4.3 Method Invocation

Method Invocation is defined as a function that evaluates invokes a Method in an Object. It is used by when evaluating the Object Calculus. The label is looked up in the given Object, and the method is evaluated. The function body is evaluated in a manner similar to Lambda Calculus.

```
-- Method Invocation
methodInvocation :: Object -> String ->
    Object
methodInvocation obj label = resultantObj
    where
  m = (lookup label obj)
  m' = evalMethod m
  f m2 = if (m2 == m') then Just m' else
      Nothing
  resultantObj = Map.update f label obj

-- Evaluate a Method
evalMethod :: Method -> Method
evalMethod (VarM number) = VarM number
evalMethod (Fun body) = FunM (evalBody
    body)
```

4.4 Method Update

Method Update is simple, and is defined as a function that updates the method corresponding to the given label in an Object. Like Method Invocation, this is also used to evaluate the Object Calculus. The label is looked up in the given Object, and the method is replaced and the modified Object is returned.

```
-- Update Method with given Method
methodUpdate :: Object -> String ->
    Method -> Object
methodUpdate obj label m = resultantObj
    where
  f m' = if (m' == m) then Just m else
      Nothing
  resultantObj = Map.update f label obj
```

4.5 Object Cloning

This is the simplest one of the functions. It simply returns the same object, as a copy. Method update can be applied on the copy to obtain a derived object, this represents object-based inheritance in many object-based languages.

```
-- Clone an object
objClone :: Object -> Object
objClone obj = obj
```

5 Conclusion

The Haskell interpreter for Object Calculus can be used to gain a better understanding of how Object-oriented languages are represented using Object Calculi. As we mentioned in a previous section, there are other approaches to modeling object-oriented languages. However, we chose to build upon Abadi and Cardelli's Object Calculus because we feel it is a more generic representation of object-oriented languages and hence can be used to model any type of object oriented language [3].

Mapping Object Calculus devised for representing Object-Oriented Languages in a Functional Language like Haskell has been quite difficult. Untyped Imperative Object Calculus is the more generic and minimal form of Object Calculus. But as Haskell is a typed functional language with lazy evaluation, there is often a mismatch between the implementation, and the given typing rules, semantics and the Object Calculus syntax. Unfortunately, the current implementation of the interpreter does not fully parse the more advanced expressions in Object Cal-

culus. But it is able to parse and evaluate the untyped imperative Object Calculi of the previous sections.

In the future, this interpreter can be extended to support Typed Object Calculus as well [7]. A Type system for the Object Calculus has already been devised by Abadi and Cardelli [2]. However, whether it is feasible to do so in Haskell remains to be seen.

References

[1] M. Abadi and L. Cardelli. A semantics of object types, 1994.

[2] M. Abadi and L. Cardelli. An imperative object calculus: Basic typing and soundness, 1995.

[3] M. Abadi and L. Cardelli. A theory of objects, 1996.

[4] M. Abadi and L. Cardelli. A logic of object-oriented programs, 2003.

[5] K. B. Bruce. Foundations of object-oriented languages, 2002.

[6] G. Castagna. Object-oriented programming, 1997.

[7] M. Odersky, V. Cremet, C. Rockl, and M. Zenger. A nominal theory of objects with dependent types, 2003.

[8] B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming., 1994.

[9] R. Rojas. A tutorial introduction to the lambda calculus, 2015.

[10] Wikipedia. Article on church encoding.

[11] Wikipedia. Article on lambda calculus.