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

March 11, 2016

Abstract

We present an interpreter developed in Haskell for the Object Calculi introduced and detailed by M. Abadi and L. Cardelli in A Theory of Objects. Object Calculus has been devised by Abadi et al. to represent objects directly as primitives, rather than to try to match Lambda Calculus that use function primitives, to objects. 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 [3,7]. Church numerals and encoding are used to represent data and operators in Lambda Calculus [4]. 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, and 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 [1]. It is not trivial even with Object Calculus to develop an interpreter in Haskell, a functional language, but it certainly is easier than directly adapting Lambda Calculus and then developing an interpreter for that.

Abadi et al. devised Object Calculus to complement Lambda Calculus, and help represent object-oriented concepts in programming languages more easily [1,2] 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 et al. and walk through the semantics and rules that we made use of to build our interpreter in Haskell. In Section IV, we shall present our interpreter for 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, et al. specifies how lambdas and recursive records can be used to obtain a foundation for object oriented languages [8].

Castagna uses multimethod calculus to help represent object oriented theory and abstractions [9].

Bruce devised class-based calculi to model class-based programming. But they focus on class-based languages, and a limited set of object oriented concepts like classes and inheritance [10]. Abadi, et al.?s Object Calculus manages to represent both Class-based and the broader Object-based languages [1].

3 Overview

We first implemented an untyped calculus of objects called Zeta Calculus. To start off, we specify the encoding of object primitives in lambda calculi as described in Abadi, et al [1].. An object is a map of attribute label li - method zeta(xi)bi pair for i elem 1,...n. Here xi is self parameter of object and method body is bi. We have two operations - method invocation and method update.

¡Placeholder for semantics table¿

Abadi, et al. doesn?t treat object fields separate from methods, if a method self parameter is not used, then method invocation is equivalent to field selection and method update is equivalent to field update

The reduction steps of zeta calculus are as follows [1]:

¡Placeholder for primitive semantics table¿

The following are rules for Free variables and substitution for Zeta method body [1]:

¡Placeholder for both tables¿

The following are rules for translation from lambda terms to objects [1]:

¡Placeholder for table¿

With the above rules, we implement a calculator based on the description by Abadi, at el [1]:

¡Define Calculator¿

The enter method will set the arg of self object to input value. The add method will change the equals method of the object, but before doing so, it will store the equals method of the object to the acc field. Thus when the equals method is invoked for the new object, the sum of arg and acc is returned. Similarly sub method is also defined. The equals method by default will return arg value.

The trait is defined as ¡placeholder¿ for an object ¡placeholder¿ [1]. Traits are a collection of pre-methods. We can reconstruct an object from a trait. Thus a new class is defined with a new method which will return an object as:

¡Define c and o = c.new¿

Inheritance in this case simply means copying the class and adding more pre-methods for the derived class [1].

¡Define c?¿

We plan to implement the above calculator example and an example interpreter demonstrating inheritance in Haskell.

4 Implementation

¡Placeholder dummy text¿

5 Conclusion

¡Placeholder dummy text¿

6 References

1. M. Abadi and L. Cardelli. A Theory of Objects. Springer-Verlag, 1996.

2. M. Abadi and L. Cardelli. A Logic of Object-Oriented Programs, 2003.

3. Wikipedia Article on Lambda Calculus.

4. Wikipedia Article on Church Encoding.

5. Odersky, et al. A Nominal Theory of Objects with Dependent Types, 2003.

6. M. Abadi and L. Cardelli. A Semantics of Object Types, 1994.

7. Raul Rojas. A Tutorial Introduction to the Lambda Calculus, 2015.

8. Benjamin C. Pierce and David N. Turner. Simple Type-Theoretic Foundations for Object-Oriented Programming. Journal of Functional Programming, 4(2):207-247, April 1994.

9. Castagna G. Object-oriented programming. Boston: Birkhauser; 1997.

10. Bruce K B. Foundations of Object-oriented Languages. Cambridge Mass.: MIT Press; 2002.

11. ¡and more if neccessary¿