CS4201 – Programming Language Design and Implementation

Practical 2: Denotational Semantics

Chris Brown
cmb21@st-andrews.ac.uk

2020

Weighting: 50% of coursework

Deadline: 17th November 2020 at 21:00
Deadlines on MMS are definitive

You are expected to have read and understood all the information in this specification at least a week before the deadline. You must contact the lecturer regarding any queries well in advance of the deadline.

Purpose

This practical will help develop your skills in:

- Lambda Calculus, including beta reduction, alpha conversion and eta-expansion
- Denotational Semantics;
- Environments;
- Reductions.

Overview and Background

The purpose of this practical is to implement a denotational semantics for an untyped Lambda Calculus for a small Haskell-lite language. The idea is to *denote* into the semantics, i.e. translate a Haskell-lite program into its equivalent untyped Lambda Calculus form and then evaluate it, using the reduction rules for Lambda Calculus as described in the lectures.

This means that there are essentially two parts to this practical. The first part is to take programs written in Haskell-lite and translate them into Lambda Calculus, using the methods similar to the ones in the lecture on Denotational Semantics. For this, you'll need to construct environments and value domains for the identifiers, for example. The second part consists of building a lambda-calculus interpreter. Once programs are translated into your representation of Lambda Calculus, you can evaluate them using the reduction rules to evaluate to a normal form (or not).

The denotational aspect is similar to that of a compiler back-end in some respects. Instead of generating assembly code, we will use the lambda calculus as our machine for representing computations.

The Haskell-lite language is a simple functional language. It has functions, arguments, where clauses, simple expressions, and lists. The grammar for Haskell-lite is given in the ANTLR grammar file, `Haskell.g`, together with Java files for walking the tree, similarly to Practical 1.

Here is an example of Haskell-lite, where we define factorial:

```
main = apply(fact,10)

fact n = if n == 0 then 1 else n * apply(fact, (n - 1))
```

Functions are applied to arguments using the apply construct. Functions with multiple arguments can be called by chaining the apply construct, such as in the Fibonacci example:

```
main = apply(fib, x) where { x = 42 }

fib n = if n == 0 then 0
      else if n == 1 then 1 else apply(fib, (n - 1)) + apply(fib, (n - 2))
```

Lists are also allowed in Haskell-lite, and there is an example in the practical directory for generating primes using the sieve of Eratosthenes.

The Lambda Calculus system should ideally be closed, i.e. all computations should be represented as lambda expressions. Recursion should be translated into a lambda calculus program using the fixed-point Y combinator (recall the lectures). Boolean values and numbers can be encoded using Church numerals and lambda expressions. Lists can be encoded using similar lambda expressions. A list of example Lambda expressions is given in the practical directory. For a core version you can use the given lambda constructs and use constants to represent numbers, e.g. 1, 2, … and normal arithmetic operators.

For example, Booleans can be defined:

```
TRUE  = lambda a. lambda b. a,
FALSE = lambda a. lambda b. b

AND   = lambda p. lambda q. p q FALSE,
OR    = lambda p. lambda q. p TRUE q,
NOT   = lambda p. lambda a. lambda b. p b a,
IF    = lambda p. lambda a. lambda b. p a b,
EQ    = lambda x. lambda y. IF (x == y) TRUE FALSE
```

Example

Suppose:

`AND TRUE TRUE` (we expect to get `TRUE` from this ...)

Then by application, we get:

```
= (AND TRUE) TRUE
```

If I expand AND ...

```
= ((lambda p. lambda q. p q FALSE) TRUE) TRUE
```

Beta reduction (replace all free occurrences of p with TRUE...)

```
= (lambda q . TRUE q FALSE) TRUE
```

BETA reduction on q ...

```
= TRUE TRUE FALSE
```

Expand TRUE ...

```
= (lambda a. lambda b. a) TRUE FALSE
```

Beta reduction on a ...

```
= (lambda b . TRUE) FALSE
```

Beta reduction on b ...

```
= TRUE
```

```
= lambda a. lambda b. a
```

Which is a normal form (it can't be reduced further) so we stop.

Tasks

The first task is to translate programs in Haskell-lite into Lambda calculus. The output of this task can be a Java Abstract Syntax representation of the lambda calculus representation of the Haskell-lite program. Here, the design of the Abstract Syntax Tree for Lambda Calculus is left up you as a design choice.

The second task is then to build a lambda calculus interpreter that would take as input a Lambda Calculus Abstract Syntax Tree and reduce it to a normal form and output the result. Recall that diverging computations in the calculus do not have normal forms, so may never terminate. The interpreter should have an option to print out a trace of the reduction steps in a readable form, using a syntax that is similar to lambda calculus. E.g.

```
   (lambda x. x) (lambda y. y) 42
⇨ (lambda y. y) 42
⇨ 42
```

(where 42 could be a Church numeral).

As a possible **going further** task, you may want to think about extending the core system by:

- splitting the translator and interpreter: this would mean that the translator could output a text file containing lambda calculus, and then an additional parser would parse this and generate the AST for Task 2. (Medium – Hard).
- Creating a completely closed system, so that numbers and arithmetic operators are also defined in lambda calculus. (Hard)
- Creating an additional typed Lambda Calculus equivalent and demonstrate how some computations in the simply typed calculus cannot exist (recall the lecture slides). (Very hard)

These are just some ideas and I encourage you to be creative with this practical!

Again, you will require antlr-4.7.2-complete.jar to be installed, available from

https://www.antlr.org/download/index.html

You can begin by modifying `HaskListener.java`, and overriding the methods similarly to Practical 1. (Please ensure you have correctly setup Java and the classpath).

```
javac *.java
java Translate "examples\fac.hs"
```

Assumptions/Clarifications

1. Functions in Haskell-Lite have a one-to-one correspondence with the Lambda calculus. E.g.

   ```
   f x = x + 1
   = \x . x + 1
   ```

   Function applications correspond to applications in the calculus and are reduced using beta reduction:

   ```
   g = apply(f, 42)
   =  f 42
   = (\x . x + 1) 42
   = 42 + 1
   = 43
   ```

2. You can assume that all the examples that you create or use are already correct, contain no type errors, scoping errors, etc. Do not write a semantic analyser for this practical, please assume this is already done and the examples are correct, well scoped, well typed, etc. You can also assume all the variables in a scope are unique.

Don't try to do scope checking. For example, `f a b = c where { c = 42 }` is allowed, but `f a b = b where { b = 42 }` is not (here we have `b` defined twice in the same scope).

3. Mutual recursion in Haskell-Lite is not allowed. E.g.
   ```
   f x = g x
   g x = g x
   ```
   You can assume that the examples you use contain no mutual recursion.
4. Normal recursion will need to be implemented using the Y combinator from the lectures.
5. If you find computing the results takes a long time due to reducing lambda expressions and their representations, that's fine. Optimisation is not the key here.
6. When you print the output, all I'm looking for is a simple trace of the reduction. You do not need to try to make the output readable or "understandable". A simple dump of it reducing would be enough.
7. There are no negative numbers in Haskell-lite. A version of the grammar was updated to remove the negative operator.
8. Please assume all equality, EQ, NEQ, LT, LTE, GT, GTE are over numbers, AND and OR are over Booleans. Do not write a type checker for this. Again assume all examples are correct in this regard. Arithmetic operations are over numbers only.
9. You do not need to produce an entire closed system for this practical. You can use the supplied lambda functions on Page 2 and use numbers and normal arithmetic operators, numbers, etc. for the rest. A pure closed system is a Going Further task. It is reasonable to get the core version working before attempting this. Remember, a really exceptional core solution (without going further elements) could still get a mark of 18 and beyond if it shows really exceptional clarity, precision and in-depth understanding. I encourage you to read this spec carefully.
10. Assume a leftmost-outermost reduction order.
11. Note the updated rule for equality where == here returns either a `TRUE` or `FALSE`
    ```
    EQ    = lambda x. lambda y. IF (x == y) TRUE FALSE
    ```
12. Declarations in the where clauses are just functions, which have optional arguments (in fact, all functions in Haskell-lite have optional arguments). Declarations in the where clause are surrounded in braces. E.g.

    ```
    f x = apply(a, x) + apply (b, x)
         where {
              a y = y
              b z = z
         }
    ```

Submission

Your implementation must be in Java. You should hand in the sources of your implementations, together with any recipes needed to build them. In your report, briefly discuss key design decisions, any problems or unexpected features you encountered, and the checking results of your implementations. Make a **zip archive** of all of the above and submit via MMS by the deadline.

Marking

I am looking for:

- Good design and understandable code, which is well documented, with major design decisions explained;
- A report giving a description of the denotational semantics of your translator. You should include in this description how you have constructed environments value domains.
- A report giving a critical analysis of the design and applicability of the Haskell-lite translator to a range of examples. You are encouraged to come up with Haskell-lite programs that show interesting behaviours of the system and document them in the report. Have fun with it!
- An in-depth understanding of lambda calculus and denotational semantics.

The standard mark descriptors in the School Student Handbook will apply:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

In particular, a very basic implementation of a lambda calculus translator for one example with very basic reductions and lambda terms fit the definition of a reasonable attempt achieving some of the required functionality and could get marks up to 10. More than that would need work on analysis and critical thinking of the algorithms and semantics.
As usual, marks of 19 or 20 would typically need an exceptional solution with demonstrated insight into the problem and demonstrations of going beyond the core system (see examples of Going Further). A really exceptional core solution (without going further elements) could still get a mark of 18 and beyond if it shows really exceptional clarity, precision and in-depth understanding.

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

The University policy on Good Academic Practice applies:
https://www.st-andrews.ac.uk/students/rules/academicpractice/