Writing a Program to Evaluate Lambda Expressions

Yola Jones

August 8, 2019

Chapter 1

Design and Implementation

1.1 Design

The overall program is split into a number of distinct elements. A grammar is used to define the syntax of a lambda term. Antlr is used to turn this grammar into an abstract syntax tree, which is used by an expression evaluator to traverse the tree and determine the result of the input lambda term.

A web interface is used to house this program, providing the user with a convenient and simple way to interact with the program.

Since Antlr is used to turn the grammar into an abstract syntax tree, this will not be documented in the design section, as this process has already been well-defined and will not be deviated from [14]. However, the grammar, expression evaluator and the web interface are key blocks which need to be implemented with key factors in mind, and so have been discussed below.

1.1.1 Grammar

The key aim in creating this grammar is creating a syntax which sticks as closely as possible to the rules of lambda calculus. Lambda Calculus grammar is already clearly defined, with a lambda term being either a variable, an abstraction or an application [9]. Applied lambda calculus adds functions and constants to this definition [19]. Therefore the grammar of a lambda calculus term becomes:

| Application (of form `[term] [term]`

| Abstraction (of form `[abstraction_term].[term]` where `[abstraction_term]` is of the form λ`[variable]`)

| Function (of form `[term] [operation] [term]`)

| Value (of form `[variable]` (the letters a-z) or `[number]` (constant))

With the addition of types, the grammar adds the option of a `:[type]` term to each variable, with each type being either a ground type (bool, int or none), or in the form `[type]->[type]`. This follows the standard syntax for typing used in the lecture material [9] [5], and will allow each student to input a lambda term directly from the lecture slides with minimal adjustment.

1.1.2 Expression Evaluator

The fundamental idea behind this evaluator is tree traversal, which navigates through the token nodes and performs different operations depending on the type of token encountered. For example, application tokens in the form MN will pass the right-hand term N to the left-hand term M. Abstractions will take the incoming value and substitute it into its own function. This will allow an evaluated expression to be built up, and a result evaluated.

Antlr provides two mechanisms for traversing an abstract syntax tree: listeners and visitors.

A listener is a passive way of evaluating a syntax tree, an antlr Walker object is declared alongside the desired listener class, the walker traverses the tree using a depth-first approach, triggering methods from the listener as it enters and exits each token [14]. These listener methods can't return values, so expressions and evaluations are to be handled using separate objects within the listener class. As the walker traverses the tree, the listener builds up a running evaluation of the term, returning the result when it exits the topmost node [22].

The key difference between a listener and a visitor is that a visitor controls its own traversal of the tree. By visiting the children of the current node explicitly, the path they take around the tree can be controlled [14], for example some children not being visited until their parents are evaluated, or a right-hand child being visited and evaluated before the left.

Visitors also allow custom return types, meaning rather than having to rely on separate objects for expression value return, nodes can return their resultant expressions directly to their parent node [22].

With beta reduction, different methods evaluate terms in different ways. In an application MN using a call-by-value approach, N is evaluated before M. In a call-by-need approach, N is passed into M before being evaluated.

This means that depending on the type of reduction selected, the evaluator will have to traverse the tree in a different order, suggesting visitor being more appropriate for this task than a listener. This is supported by the fact that when evaluating expressions, the evaluation will be happening as the tree is visited, and therefore there will be a great deal of return values needed. Having a separate object for storing these values could get complex, and so the visitor methods being able to return values directly to their parents will be more convenient for this task.

The visitor therefore will be the main code written in this project. A parser will be passed to a custom visitor interface, with different visitors being defined for each of the beta-reduction methods being implemented, since these methods will each traverse the tree in a different way.

The visitor should return tree things upon returning to the topmost node: the value of the expression, whether or not the term is typable, and what type the expression will be. It will also return details of any errors, for example syntax or "normal form does not exist" errors where applicable.

1.1.3 Web Interface

The web interface will allow the user to input a lambda term, along with the types of any terms. It will also allow the user to select which reduction strategy they would like to have the term evaluated by, with call-by-need (or normal order reduction) being selected as the default.

After the user has entered the data, a HTTP POST request will be used to send the data through the back-end code, which will process the term and return the result, type validity and the type of the final term from the visitor. This will then get displayed back to the user.

The layout should be simple and uncluttered, and should be suitable for those with a visual impairment.

1.2 Implementation

The majority of the code has been written in Python, specifically Python3, as there is plenty of support for Antlr being used with Python3. While Java is more extensively supported in Antlr, the MSc Software Development course which this dissertation is being written for teaches Java and not Python, so Python was chosen as a way of expanding the number of programming languages encountered throughout the course.

1.2.1 Grammar

The lambda grammar is contained in a .g4 file which defines the parser and lexer rules for lambda calculus, as required by Antlr. A section of the grammar is shown in 1.1, and indicates the parser rules for the term, application and abstraction token nodes, alongside the lexer rules in 1.2.

1.2.2 Abstract Syntax Tree

Having defined a grammar, antlr can then be used to create an abstract syntax tree. This is very simple to do, and involves passing the input term through an Input Stream class which converts this to a string of characters, then feeding this into a lexer (which outputs a set of tokens), a class which converts the tokens into an indexable list, and finally passing this through a parser which turns the tokens into an abstract syntax tree which can be explored [24]. This process is well documented and therefore will not be deviated from.

1.2.3 Expression Evaluator

This is the largest block of code in the program, and defines how the incoming lambda term should be evaluated depending on the input given by the user. This code can be split into three distinct sections as discussed in the Background chapter of this report: alpha conversion, beta reduction and typing rules.

Alpha Conversion

The alpha conversion is based on the rules for explicit alpha conversion with substitution, as defined by [1], and as discussed in detail previously. These rules are as follows:

$$[t/x]y = \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \tag{1.1}$$

$$[t/x](t_1 t_2) = [t/x]t_1[t/x]t_2 \tag{1.2}$$

$$[t'/x](\lambda y.t) = \begin{cases} \lambda y.t & \text{if } x = y \\ \lambda z.[t'/x][z/y]t & \text{if } x \neq y \wedge z \notin FV(t) \cup FV(t') \end{cases} \tag{1.3}$$

```
/* Parser Rules */
term
    : abstraction
    | function
    | value
    | application
    ;

application
    : application term
    | abstraction term
    | value term
    | function term
    | LBRACKET application RBRACKET
    ;

abstraction
    : abstraction_term '.' term
    | LBRACKET abstraction RBRACKET
    ;

abstraction_term
    : '%' variable
    ;
```

Figure 1.1: Parser Rules

```
/* Lexer rules */
NUMBER : [0-9]+ ;
BOOL : 'TRUE'|'true'|'True'|'FALSE'|'false'|'False' ;
VARIABLE : [a-zA-Z] ;
ADD : '+' ;
SUBTRACT : '-' ;
MULTIPLY : '*' ;
DIVIDE : '/' ;
POWER : '^' ;
LBRACKET : '(' ;
RBRACKET : ')' ;
AND : '&' ;
OR : '|' ;
GT : '>' ;
LT : '<' ;
EQ : '==' ;

WS : [ \t\r\n]+ -> skip ;
```

Figure 1.2: Lexer Rules

The final rule is the rule to be focused on, and contains the process of finding a variable z that does not appear in the free variables of the incoming term or the existing term, replacing all bound variables y with this new term z, and then substituting in t' as normal. The alpha conversion code follows this process. First, the set of free variables in the term are determined, by taking the set of all alphabetic characters in the term and eliminating the bound variables. These variables are then replaced with letters that are not in the list of free variables, in the situation where a clash in free variables between the two expressions are found. This produces an alpha-converted term, and regular substitution happens using a string.replace() python method, as abiding by the Barendregt Convention.

Beta Reduction

Two beta reduction strategies are taught on the Theory of Computation course, call-by-value and call-by-name. The differences between these two strategies have been discussed, but the key difference is in when terms are evaluated in an application MN, whether N is evaluated before being passed into M, or whether it is substituted first and then evaluated inside M.

Because of these differences in evaluation strategy, two separate visitors are used. However, since they share a lot of the same common functionality (when alpha conversion happens, typing rules, what happens inside functions), a BaseVisitor was defined which contains all common code between these two strategies. The two call-by-value and call-by-need visitors are subclassed from this base visitor, and define their unique behaviour for the application and abstraction terms.

The abstraction term differs between the two methods due to nothing other than typing, since the type of a term happens during the evaluation of that term as will be discussed in more detail below. In call-by-value, the type of N is known before substitution, so can be carried throughout the function. In call-by-need, the whole term needs to be type checked after substitution has happened to determine the type of M with N incorporated.

Typing

Lambda Calculus has clearly defined typing rules which are taught in the Theory of Computation course. These rules have been discussed in more detail previously, and are as follows:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} TVar \tag{1.4}$$

$$\frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : T \to U} TAbs \tag{1.5}$$

$$\frac{\Gamma \vdash M : T \to U \qquad \Gamma \vdash N : T}{\Gamma \vdash MN : U} TApp \tag{1.6}$$

There are two minimal types (`int` and `bool`) and one supertype (`none`) as defined in the Theory of Computation lecture slides [5]. These will be the only types allowed by the program.

Since the visitNode methods already evaluate their children nodes, these typing rules can be integrated directly into these nodes, with each visitNode method returning a value and a type. Type checking happens throughout the code, in each method which could contain conflicting types. This is implemented as follows:

Abstraction

Users enter types in the form `x:type`, which can be applied to any variable included in the term. This means that the term λ `x:int.x:bool` is valid, but type invalid (since the type of x is different for each x term). This is checked inside the visitAbstraction method.

The abstraction method takes the input type T (as determined either from input by the user or through type inference from a function, discussed below), and joins it with the output type (again, either from user input or determined from the function) with `->`, to become of type `input->output`.

Application

Application typing is far simpler than abstraction, for an application MN the code simply takes the type of M (in the form `T` or `T->U`) and iteratively removes the first type from both M and N until the type of N is None, at which point it returns type M. If at any point the first type in M does not match the first types in N, or the number of types in N is larger than the type of M, the type checking is declared invalid. This is as per rule 1.6.

Variable

The variable terms return the type given to them by the user, as defined by Rule 1.4. Any number is given type `int`, and any boolean values TRUE or FALSE are given type `bool`.

In the visitFunction node, the input and output types of terms can be inferred by examining the operation term. The following is stated in code:

- The operations {&,|} take two boolean values and return a boolean. Therefore the type of the incoming term can be inferred to be a bool, likewise with the output term
- The operations {+,-,*} take two integer values and return an integer. As above, the input term can be inferred to be an integer, matching the output term
- The operations {==,>,<} take two integer values and return a boolean, so it can be inferred that the input term is an integer, and the output is a boolean.

This is used to determine the output type of a function which is used to determine the type output by its parent term. In an abstraction, this type inference can be used to determine the type of its input term, for example the lambda term $\lambda x.x + 1$ can be inferred to have type `int->int` despite no input type given by the user.

There is a limit to this type inference, the function will infer the input type when there is only one operation term. For example, the term $\lambda x.(x == 1)\&b$ contains two operations, $==$ and &. In this case, working out the input type is more complicated, since the typing has to be broken down in to sub-functions, which need to determine what they think the input type should be, which then needs to be examined collectively as a complete term. The type of this output is returned by the program therefore as `None->bool`, which is correct, it's just not as refined as `int->bool`.

While this is definitely possible, it is not the key goal in the project, and is a small edge-case when it comes to improving students understanding. The result is correct, just not completely minimal. Because of this, and due to the finite nature of this project, this has been deemed an acceptable limitation of the code, with other tasks which more greatly contribute to students understanding of lambda calculus being prioritised.

Overall Lambda Code

Each visit method in the visitor returns an evaluated result and a type, which is used to build the evaluated term and determine the final result, type and type validity of the input expression.

However, during this process, there are a number of occurrences which could stop the program from being able to return a final result. These are broken down into two key issues: syntax errors and occurrences where a term doesn't have a normal form and therefore cannot be evaluated to termination. These have been discussed below.

Syntax Errors
Lambda terms often include nested parentheses. It is easy for a user to mismatch brackets, for example for a term to include three open brackets but only two close brackets. Because of this, before the term is passed to antlr, the code checks to ensure brackets are matched correctly, and if not, returns an error to the user informing them of the issue and asking them to re-enter.

Apart from this, general syntax errors are likely to occur, for example a user forgetting to put a . in an abstraction term. To handle this, antlr's ErrorListener class is overridden, instead throwing a custom SyntaxTokenError exception, which can be caught by the program and passed back to the user.

Recursion Errors
Recursion errors are thrown in Python when a maximum stack depth is reached in order to protect Python from crashing [7]. This is the exception which is thrown when a lambda term has no normal form, as the code keeps trying to evaluate it before eventually throwing a RecursionError.

To handle this, the recursion limit for the code is set to 200, smaller than the python default to prevent excessive time waiting for the program to fail, but large enough to ensure that any reasonable lambda term a student could want to input can be processed. A recursion limit for the code equates to a lambda term with approximately 50 nested abstractions (since each abstraction could potentially be comprised of the following nodes: its parent term, the abstraction contained within parentheses, the actual abstraction and its inner terms). While this is a limitation, it's is deemed to be a reasonable one.

If a recursion error is thrown, it is caught by the code in a `try/except` block, and an error message informing the user that a normal form cannot be found for this term is given to the user.

Web Interface

Flask is a web framework designed for use with Python, and allows python scripts to be controlled and run from a web interface which is served by Flask[13]. The front-end of the web interface is written in HTML, and is connected to the python file using Flask's render_template() function.

Clicking the *Check Expression* button on the web interface sends a HTTP post request to the Flask server, which takes the value held in the user input box along with the selection of reduction strategy, and sends this to the main antlr Lambda program. The result outputted is then given to the HTML code, and the webpage is re-rendered to display the results, be that an error message indicating a syntax or normal form not found error, or the result, type and type validity of the lambda term input by the user.

Hello! [8] [23] [1] [2] [3] [4] [5] [6] [9] [10] [11] [12] [14] [15] [16] [17] [18] [19] [20] [21] [24] [22]

Bibliography

[1] Umut Acar and Amal Ahmed. *Lecture 3: Lambda Calculus (Syntax, Substitution, Beta Reduction)*. Jan. 17, 2018. URL: `https://ttic.uchicago.edu/~pl/classes/CMSC336-Winter08/lectures/lec3.pdf` (visited on 08/01/2019).

[2] Henk Barendregt and Erik Barendsen. *Introduction to Lambda Calculus*. Mar. 2000. URL: `http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf` (visited on 07/27/2019).

[3] Steven Bird, Ewan Klein, and Edward Loper. *Natural Languge Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly, 2009, pp. 291–322.

[4] Alonzo Church and J. B. Rosser. "Some Properties of Conversion". In: *Transactions of the American Mathematical Society* 38 (3 1936), pp. 472–482.

[5] Ornela Dardha and Simon Gay. *The Theory of Computation*. 2019.

[6] Stacy Everett. *Arithmetic Grammar*. July 11, 2019. URL: `https://github.com/antlr/grammars-v4/tree/master/arithmetic` (visited on 07/28/2019).

[7] Python Software Foundation. *Built-in Exceptions*. Aug. 8, 2019. URL: `https://docs.python.org/3/library/exceptions.html` (visited on 08/08/2019).

[8] University of Glasgow. *Theory of Computation (H) Course Specification*. 2019. URL: `https://www.gla.ac.uk/coursecatalogue/document?type=sp&courseCode=COMPSCI4072` (visited on 08/02/2019).

[9] Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. 2nd ed. King's College Publication, 2004.

[10] Susan B. Horowitz. *Lambda Calculus (Part 1)*. 2013. URL: `http://pages.cs.wisc.edu/~horwitz/CS704-NOTES/1.LAMBDA-CALCULUS.html#normal` (visited on 08/01/2019).

[11] Fairouz Kamareddine. *Reviewing the Classical and the de Bruijn Notation for λ-calculus and Pure Type Systems*. Oct. 4, 2000. URL: https://pdfs.semanticscholar.org/2b65/df6c8c0cc14db2afcacbca0f0423588ec6f0.pdf? (visited on 08/02/2019).

[12] Ralph Loader. *Notes on Simply Typed Lambda Calculus*. Feb. 1998. URL: http://www.lfcs.inf.ed.ac.uk/reports/98/ECS-LFCS-98-381/ECS-LFCS-98-381.pdf (visited on 07/28/2019).

[13] Matt Makai. *Flask*. Aug. 8, 2019. URL: https://www.fullstackpython.com/flask.html (visited on 2019).

[14] Terrance Parr. *The Definitive Antlr4 Reference*. 1st ed. The Pragmatic Programmers, 2012.

[15] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[16] Raúl Rojas. *A Tutorial Introduction to the Lambda Calculus*. 1998. URL: https://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf (visited on 07/28/2019).

[17] Michael L. Scott. *Programming Language Pragmatics*. 4th ed. Morgan Kaufmann Publishers, 2016.

[18] Peter Selinger. *Lecture Notes on the Lambda Calculus*. 2013. URL: https://www.irif.fr/~mellies/mpri/mpri-ens/biblio/Selinger-Lambda-Calculus-Notes.pdf (visited on 07/27/2019).

[19] Ken Slonneger and Barry Kurtz. *Chapter 5: The Lambda Calculus*. Pearson, 1995, pp. 139–166.

[20] Kevin Sookocheff. *Eta Reduction*. Sept. 27, 2018. URL: https://sookocheff.com/post/fp/eta-conversion/ (visited on 08/02/2019).

[21] Kevin Sookocheff. *Normal, Applicative and Lazy Evaluation*. Sept. 18, 2018. URL: https://sookocheff.com/post/fp/evaluating-lambda-expressions/ (visited on 08/01/2019).

[22] Saumitra Srivastav. *Antlr4 - Visitor vs Listener Pattern*. Oct. 19, 2017. URL: https://saumitra.me/blog/antlr4-visitor-vs-listener-pattern/ (visited on 08/06/2019).

[23] Paul Tarau. *Church Encoding*. Feb. 17, 2017. URL: http://www.cse.unt.edu/~tarau/teaching/PL/docs/Church%20encoding.pdf (visited on 08/02/2019).

[24] Gabriele Tomassetti. *The ANTLR Mega Tutorial*. Mar. 8, 2007. URL: https://tomassetti.me/antlr-mega-tutorial/ (visited on 08/06/2019).