CS4201 P1: λ-Calculus

Student ID: 200007413

October 13, 2023

Contents

1 Introduction **1**

1.1 Aims and Objectives 1

1.2 Summary 1

1.3 Report Structure 1

1.4 Instructions for Running Code/Tests 2

2 Design and Implementation **3**

2.1 Code Structure 3

2.2 Starter Code 3

2.3 Lambda Evaluation 3

2.4 Type Checking 5

2.5 Polymorphism 6

3 Testing **7**

4 Discussion and Analysis **10**

References **11**

1 Introduction

1.1 Aims and Objectives

Following the specification, my primary aim for this piece of coursework was to develop a practical understanding of λ-calculus and type theory. Secondarily, I aimed to improve my ability in coding clean, simple, and robust solutions to problems through more rigorous design and testing.

My primary objective for this practical was to create a well-tested, simply-typed λ-calculus - consisting of a λ-calculus evaluator and type-checker. Further objectives developed to extending this implementation to allow for more complex types. Specifically, parametric polymorphic types to create a system-f λ-calculus.

1.2 Summary

The above objectives were met, mostly in line with the practical aims. Through maintaining a clear distinction between type-checking and evaluation, the code is well organised and relatively intuitive. The use of unit-testing and property testing alongside implementation aided in creating a robust solution.

However, there are particular quality-of-life quirks with the system-f in regards to type-labelling. This could be improved given more time, but ultimately are not fatal bugs. Additionally, the polymorphic extension could be refactored to aid with readability.

1.3 Report Structure

In the design and implementation section of this report, I will give my process and reasoning for the design of my simply-typed λ-calculus submission. I will mention how I overcame design challenges, both those I anticipated, and those which came up in implementation. I will also discuss my research into further topics, in particular into polymorphism, leading to the extension of the code to a system-f implementation.

The following section will include an overview of the testing approach used to ensure reliability of the code, as well as the final test suite output. It will also discuss the unique challenges faced in testing this program.

The final section will be a discussion on the strengths and weaknesses of the overall design, implementation, and approach with regards to the aims and objectives. There will also a note areas where the design could be extended to a develop a more useful programming language.

1.4 Instructions for Running Code/Tests

```
cd lambdacalc
cabal run st

cd systemf
cabal run st
```

Listing 1: Instructions for running (with cabal)

```
cd lambdacalc
ghci -I src src/Lambda/Main.hs
$ *Lambda.Main> main

cd lambdacalc
ghci -I src src/Lambda/Main.hs
$ *Lambda.Main> main
```

Listing 2: Alternative instructions for running (with ghci)

```
cd lambdacalc
cabal test --test-show-details=direct

cd systemf
cabal test --test-show-details=direct
```

Listing 3: Instructions for running tests (with cabal)

2 Design and Implementation

2.1 Code Structure

The file structure for this submission is split into two parts. The first folder `lambdacalc` holds source code for the initial simply-typed lambda calculus required by the specification. The second folder `systemF` is a copy of the first, extended to allow parametric polymorphism. Despite all simply typed commands working as expected in system-f, the decision to separate the extension was made as the changes were wide-ranging, and calculus no longer considered ”simply” typed.

2.2 Starter Code

With the substantial starter code for this practical. It was important to understand the given functionality before starting. The code included a REPL with three primary functions: `assume` commands store types and variables into the typing context and environment; `let` commands run type checking, type inference and evaluation, then stores the results in the context and environment, as well as outputting the evaluation result (without type); finally, evaluation can be called through direct entry of expressions into the REPL. This runs the same processes as `let`, but outputs the results instead of storing them.

The important design decision the starter code had was having type-checking separate from evaluation. This meant that the `Eval.hs` functions solely focused on the untyped λ-calculus rules, and `Check.hs` functions solely on the type-checking and inference rules.

2.3 Lambda Evaluation

The evaluator recursively applies λ-calculus application rules, to reduce terms to a normal forms. The starter code split terms into two types: `iTerm` and `cTerm` for inferable terms and checkable terms. This is useful for type-checking, and so I will discuss this later in this section, but means that evaluation is split into `iEval` and `cEval`, but they

both just apply the relevant evaluation rules for the terms given.

These terms are the basic terms from the lambda calculus: applications (App), lambda abstractions (Lam), free variables (Free) and bound variables (Bound). For type-checking, there is additionally wrapper types Inf and Ann which allow the evaluator to pass type-checking to the correct function - they also allow the evaluators to pass terms to each other when necessary.

Figure 1: Reference image for my understanding of De Bruijn Indexing. [1]

One important design decision in regards to the way that terms are stored is with bound variables and lambda abstractions. As λ-calculus specifies, the value of bound variables are not set until the associated lambda abstraction is applied to a term. In the mathematical calculus, variable names are used for these bound variables which must be α-renamed when conflicting upon application. However, De Bruijn indexing stores bound variables as relative indexes which represent the number of other lambda abstractions between the variable and it's defining abstraction. This prevents indexes from conflicting as we can then keep a track of the number of abstractions we have parsed, and when applying to abstractions, store the value in context at that index, then retrieve them using the relative index by taking the current abstraction count and subtract the bound index.

This functionality can be maintained through application through the way we store the value of abstractions, and in the fact that evaluation is done from the outer-most term to the innermost term. In regards to the number of lambda expressions found, both terms in an application will have the same value. When we apply a term containing abstractions to an abstraction, the values might no longer have the same abstraction count, thus breaking the De Bruijn Indexing. However, as Haskell is a functional language, we can store the value of a function to apply with the

If we store the value of lambda abstractions as functions which take in the value to replace the bound value with, and outputs the evaluation of the inner term with the original context, then application of a term containing a lambda expression to another lambda expression will maintain the original bound variable values in it's context. This works because a bound variable cannot point towards a lambda expression of an adjecent application, and only points towards external lambda applications. As we evaluate outside-in, then we are guaranteed to have all necessary bound variable values in context.

The reason this automates α-abstraction is because when applying a term containing a lambda abstraction to a lambda abstraction, the bound values

The evaluator knows a term is a normal form, by representing evaluated terms as Value data types. There are two of these normal form value types: lambda abstractions (VLam), and Neutral types. This is because fully evaluating the inside of a lambda abstraction to a normal form can still be altered by applying the abstraction to another value. All other fully evaluated terms can only be applied to a lambda abstraction, but there do not change themselves until the lambda abstraction is then re-evaluated (hence neutral).

These neutral types are: free variables (NFree), which refer to values which must be in the environment); bound variables (NBound) (variables representing the yet-to-be-set values from a parent lambda abstraction, indexed using De Bruijn Indexing); and finally applications (NApp) representing applications of a non-lambda term to a term, thus not applicable (i.e. weak head or head normal forms).

However, the mathematical lambda calculus doesn't store normal forms as separate value types. Instead, evaluation of terms gives a more reduced term. To output the final normal form to the user, the Quote wrapper allows the REPL to convert values to their associated terms.

2.4 Type Checking

The type checking is very basic for the simply typed lambda calculus. The objective of the type checking is to ensure that the types of variables do not conflict in evaluation. It also seeks to infer the output types of evaluated expressions using the typing rules.

There is a limitation on the types that can be inferred however. For lambda abstractions, the output type is determined by the input that is given, and so must be annotated. This is why the distinction is made between `iTerm`s and `cTerm`s, as lambda abstractions must be annotated with a Function type which determine what can apply to it, and sets the output type. Type checking an application to an abstraction first makes sure that the abstraction has a function type, then checks that the input type matches, and then returns the annotation output type.

2.5 Polymorphism

After implementing the simply-typed evaluator, I researched more complex data types that may be interesting to implement. I found through using the original implementation that multiple versions of true and false were having to be defined for use in conditionals with different types (e.g. defining conditionals where the output had type int required defining true and false with type int -> int -> int. I wanted to allow true/false to be reused.

Research lead to parametric polymorphism, which is where the structure of a type remains the same, but can adapt to work with any type. Further research lead me to the theoretical extension of lambda calculus implementing parametric polymorphism.

This type of polymorphism adds and additional valid type with the form $\forall t.T$ where $t \in T$ and $T \in \{t, T \rightarrow T, *\}$. [4] This allows variables to take on different types at run-time, whilst keeping the same basic functionality, preventing the need for repeated code (i.e. the definition of separate conditionals and Booleans for each type). [2].

Before adding in the type-checking change, the parser required some design changes to allow for type-labelling. It was decided that the notation to distinguish universal types would be done by first putting an @ symbol, followed by an identifier for the universal variable, then wrapping the scope of the universal in parenthesis (i.e. `@ A ( )`). It is important to note that a peculiarity with the parser requires that there be spaces between the @ symbol and the identifier, and between the identifier and the first parenthesis.

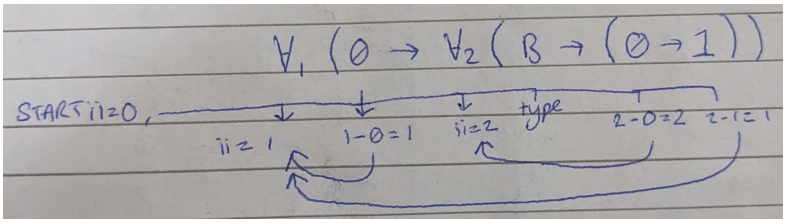

Figure 2: Sketch of indexing for polymorphic types

As the identifier is not relevant to the type-checking, it's main purpose is for the parser to distinguish each universal type from each other, and other standard types. As these types could be recursive (i.e. `@ t1 (@ t2 (t1 -> t2))` meaning the the type accepts any function of two types which can be either the same or different), I thought that De Bruijn indexing could be used to store the values for these types, similarly to the way bound

variables are represented Figure 2. Therefore, I added a type `TUni` representing the for-all statement scope, and added the `Uni` which would be wrapped in a `TFree` type (as with bound variable types) to represent the actual type variables.

To allow the `cKind` function to accept these new type variables, the `cKind` function had to be extended to accept the `Uni` both when they are labelled and when they are not. To do this, I added a parameter to the function which kept track of the number of universal types that have been encountered, and then if the De Bruijn index of the Uni type is less than or greater than

I then considered how to determine how to set the type for each t. My first idea was to keep a store of the values for each t and backtrack to make sure that assignations of each were consistent, thus inferring the type. However, I found out through further research that this was undecidable, and a decidable approach was to use annotations [3], much in the same was as λ-function types.

This change required an additional change to the parser for labelling. For this, typing a variable followed by `!` followed by a type variable tries to apply the type to the outermost `TUni`.

3 Testing

```
Test suite spec: RUNNING...

Check
  Unit test
    cKind type verification
      passes for variable types set in context [✓]
      fails for variable types not set in context [✓]
      passes for function types with valid input and output variable types [✓]
      passes for recursive functions with valid variable node types [✓]
      fails for function tpyes with invalid input type [✓]
      fails for function type with invalid output type [✓]
      fails for recursive functions with one invalid node [✓]
    iEval type inferrence
      returns free variable type in context [✓]
      throws error if free variable type not set in context [✓]
      of annotated lambda expression returns output type if input type matches [✓]
      of annotated lambda expression throws exception if annotation type isn'
```

```
    t a function [✓]
      of annotated lambda expression throws exception if input type doesn't
    match [✓]
  Property test
    cKind type checking verification
      fails when context is empty [✓]
        +++ OK, passed 100 tests.
Eval
  Test evaluation
    returns correct bound value set in environment [✓]
    throws an error when bound variable not wrapped in lambda expression [✓]
    returns correct free value set in environment [✓]
    for free vars not set in  [✓]
    Test Inferrable terms pass evaluation to iEval with same environment [✓]
    Test Annotations passes evaluation onto cEval with same environment [✓]
    Test unappliable application evaluates to a neutral [✓]
    Test quote lambda function value replaces with correct [✓]
    Test embedded lamda function values replace correctly [✓]
    Test application of lambda function to value replaces bound instance [✓]
    Test application of multiple lambda functions to values applies correctly
     [✓]
  Property test
    infer does not fail for valid expressions [✓]
      +++ OK, passed 100 tests.
Quote
  Unit Test
    Neutral Quote
      converts neutral free variable  [✓]
      converts neutral bound variable to index with one bound var [✓]
      converts neutral bound variable to index with multiple bound vars [✓]
      converts neutral application [✓]
    Quote
      converts neutral value by passing it to neutralQuote [✓]

Finished in 0.0042 seconds
30 examples, 0 failures
Test suite spec: PASS
```

Listing 4: Test results for the simply typed lambda calculus implementation

```
Test suite spec: RUNNING...

Check
  Test checking type Kind
    passes for variable types set in context [✓]
    fails for variable types not set in context [✓]
    passes for function types with valid input and output variable types [✓]
    passes for recursive functions with valid variable node types [✓]
    fails for function tpyes with invalid input type [✓]
    fails for function type with invalid output type [✓]
```

```
    fails for recursive functions with one invalid node [✓]
  Test inferring type
    returns free variable type in context [✓]
    throws error if free variable type not set in context [✓]
    of annotated lambda expression returns output type if input type matches
    [✓]
    of annotated lambda expression throws exception if annotation type isn't
    a function [✓]
    of annotated lambda expression throws exception if input type doesn't
    match [✓]
  Property test
    cKind type checking verification
      fails when context is empty [✓]
        +++ OK, passed 100 tests.
Eval
  Test evaluation
    returns correct bound value set in environment [✓]
    throws an error when bound variable not wrapped in lambda expression [✓]
    returns correct free value set in environment [✓]
    for free vars not set in  [✓]
    Test Inferrable terms pass evaluation to iEval with same environment [✓]
    Test Annotations passes evaluation onto cEval with same environment [✓]
    Test unappliable application evaluates to a neutral [✓]
    Test quote lambda function value replaces with correct [✓]
    Test embedded lamda function values replace correctly [✓]
    Test application of lambda function to value replaces bound instance [✓]
    Test application of multiple lambda functions to values applies correctly
     [✓]
  Property test
    infer does not fail for valid expressions [✓]
      +++ OK, passed 100 tests.
Polymorphic
  Test checking polymorphic type
    passes when single Uni type with empty context [✓]
    fails when single Uni type references out of bounds type [✓]
    passes with recursive TUni types [✓]
    fails with multiple recursive TUni types but Uni out of bounds [✓]
  Test inferring polymorphic type from label
    Replaces single Uni variable type correctly [✓]
    Replaces multiple Uni variables types correctly [✓]
Quote
  Unit Test
    Neutral Quote
      converts neutral free variable  [✓]
      converts neutral bound variable to index with one bound var [✓]
      converts neutral bound variable to index with multiple bound vars [✓]
      converts neutral application [✓]
    Quote
      converts neutral value by passing it to neutralQuote [✓]
```

```
56
57 Finished in 0.0182 seconds
58 36 examples, 0 failures
59 Test suite spec: PASS
```

Listing 5: Test results for the system-f lambda calculus implementation

The primary testing method for this practical was unit testing using the Hspec testing framework. This allowed easy behaviour-based testing to be conducted, which was useful as the functional nature of Haskell makes it more difficult to test the inner workings of a function than with object orientated languages.

The approach to testing was to test the basic functionality of features based on the design before implementation. This aided in bug-fixing code, and kept the intention of an addition to the code base clear.

I attempted to learn property testing, as this would have improved coverage given the fact that lambda calculus is Turing-complete, and so has an infinite number of terms that can be evaluated. However, there was not enough time to fully create full property tests which checked output of generated examples, but there is some basic kind checking, and evaluation property tests which check that valid terms do not throw exceptions.

As can be seen in Listing 5, most of the tests for this are repeated from the simply-typed lambda calculus. This shows how the functionality of the simply typed lambda calculus is retained with the extension.

4 Discussion and Analysis

Overall this submission meets all of the main objectives for creating a simply-typed lambda calculus evaluator, as well as extends to meet the objective of adding more complex parametric polymorphism. The code is overall quite readable given some basic understanding of the starter code, and has the core functionality well tested.

I feel that the system-f implementation could be refactored, given more time, to a simpler and more elegant system. Whilst the functionality is there, it is not possible to have more than one label in one evaluation/let command. I believe this limitation derives from the use of the De Bruijn indexing for the universal variables - as having multiple labelling

in one command causes them to conflict. One way to potentially fix this is to keep the universal values in a separate environment, where the indexing just takes the next available value, and is removed when the value is labelled. However, as can be found in the `prelude.st` file, this can be mitigated already by splitting labelling into different `let` commands.

Additionally, whilst testing does cover much of the vital functionality, more coverage could have been gained from more use of the property tests. I had trouble with creating the necessary arbitrary functions which would generate valid expressions. Especially with regards to making sure that bound variables do not reference out of range.

References

[1] *De Bruijn index*. URL: `https://en.wikipedia.org/wiki/De_Bruijn_index` (visited on 2023-10-12).

[2] Jean-Yves Girard. "The system F of variable types, fifteen years later". In: *Theoretical Computer Science* 45 (1986), pp. 159–192. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/0304-3975(86)90044-7`. URL: `https://www.sciencedirect.com/science/article/pii/0304397586900447`.

[3] *Lambda Calculus - System F. Type Spam*. URL: `https://crypto.stanford.edu/~blynn/lambda/systemf.html#_type_spam` (visited on 2023-10-09).

[4] The University of Birmingham. *Handout 6: Polymorphic Type Systems*. URL: `https://www.cs.bham.ac.uk/~udr/popl/06-18-polymorphic.pdf` (visited on 2023-10-09).