CS4201 P2 - Denotational Semantics

190006526

November 17, 2022

1 Instructions

To run the program on one of the test file (`<TestFile>`), run the following:

```
cd src/
javac *.java calculus/*.java
java Translate examples/<example_file>.hs [--no-output]
```

The no output flag prevents the program file from printing. This program is dependent on `antlr-4.7.2-complete.jar`

2 Abstract

For this practical, we were tasked with creating and implementing the denotational semantics required to translate a simple functional language, called Funl, to the untyped lambda calculus, extended with primitive integers and integer operations. We then had to evaluate this lambda calculus, printing out each reduction step as it goes. Funl supports some interesting features, such as Boolean values, recursive functions, lists, and local function definitions. I have completed these tasks to the best of my abilities, and additionally expanded Funl to include support for lambda expressions, and a modified grammar that supports a more ergonomic function application syntax.

3 Grammar Modifications

To begin, I will discuss the modifications made to the grammar. In the original grammar, functions are applied using the following syntax:
`apply(...apply(function, argument1), ..., argumentN)`. In a functional language, like Funl, this is quite cumbersome, because essentially everything is a function application, and you can only apply one argument at a time. To make this more ergonomic, and to make it so that the application of builtin functions like `hd` do not have a different syntax, the function application syntax has been changed to `function arg1 ... argN`. This is also more similar to

the way Haskell, of which Funl is essentially a trimmed down version, handles function application, leading to better comprehension.

In addition to the improved function application syntax, I also modified the grammar to support lambda expressions, using the syntax `\var -> expression`, because doing so allows all of the builtin functions, Boolean operations, Boolean values, and the Y combinator, all of which we will get to later, to be defined in the syntax of the language, instead of clunkily defined in Java using nested constructors. It also is a very easy feature to implement, as Funl corresponds directly to the lambda calculus anyways.

One important note is that semicolons are used to signify the end of a function declaration (including those in where statements), as I was having trouble figuring out how to get newlines to delimit the end of function declarations, as they do in Haskell.

4 Prelude

As mentioned previously, instead of defining builtin functions in Java, and then substituting them when called, I instead define them almost entirely in the language. This is with the exception of integer arithmetic and integer comparison operations, which are primitive to the lambda calculus we are using. Theoretically, they could also be defined using just the untyped lambda calculus using church numerals, but this would be significantly more involved and tricky to implement. To define these builtin functions, I use a prelude file, called `prelude.hs` that is loaded before each file is run, that adds all of the builtin functions to the environment in which the user defined functions end up getting evaluated in. All of the prelude functions use the lambda expressions I added to the grammar in their definitions, which was the main motivation behind adding support for the lambda syntax. This means that nothing is primitive to the language except integers. The definitions I used for the builtin functions were taken straight from the practical specification, with the exception of the Y combinator, which I took from Wikipedia[1].

5 Translation

Once both the input file and prelude have been loaded in, they are passed to the parser generated from my modified grammar by ANTLR. I chose to use a visitor, as this gave me more flexibility in implementation. The visitor's visit method returns a `CalcExpr`, which is the class used to represent expressions in the lambda calculus. This class is extended by the various different types of expressions that are valid in the calculus, like lambda abstractions, variables, integer constants, arithmetic operations, and function application.

To construct the final expression, the parser starts at the top of the program, evaluating all function definitions, then the main expression. When a function

[1] https://en.wikipedia.org/wiki/Fixed-point_combinator

definition is being parsed, it has an environment, which describes the ways in which various identifiers should be substituted. On the first pass, identifier substitution expressions (`IdentSub`) are left in the expressions, and then those substitution expressions are later replaced with the appropriate expression once the environment has been constructed, and has all of the necessary information. All of the substitution expressions must be replaced before an expression is reduced. The identifiers cannot be substituted on the first pass, as the identifier that needs substituting may not have been seen in the parse tree traversal yet.

Environments themselves are simple hashmaps, with the keys as strings, and the values as `CalcExpr`s. This allows for the efficient lookup of identifiers within the environment while the lambda expression is being built.

This substitution works fine in most cases, but has a few caveats. The most obvious is recursion. If a function is defined in terms of itself, when the identifier is substituted to get a resulting expression, it will expand forever. To solve this problem, the Y combinator is used. The Y combinator is of the form $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. While this function is a bit arcane, in essence it allows an anonymous function to call itself, that is, it creates a fixed point of f. To use the Y combinator, one must abstract over the function, substituting all recursive calls with some function f, which is the variable of the lambda abstraction. Then, this abstracted function turns is applied to the Y combinator, creating a fixed point. Take the function $\text{rec}(n) = \text{if } n = 0 \text{ then } 0 \text{ else } n + \text{rec}(n-1)$, which computes the sum of numbers between 0 and n. To remove the recursive call, we first abstract over the function as follows: $\lambda f.\lambda n.$ if $n = 0$ then 0 else $n + f(n-1)$. Call this function G. We then we apply this function to the Y combinator, called Y here: YG. We now have our fixed point function, which can be evaluated using the standard rules of the lambda calculus.

For implementation

Another important part of substituting expressions for identifiers is naming variables of functions. This is historically a difficult problem in the lambda calculus. In my implementation, however, it is sidestepped almost completely by using DeBruijn indices. Instead of variable names, indices are used that specify unambiguously which lambda abstraction the variable refers to, removing the need for the α-renaming step traditionally used in lambda calculus evaluation. The β-reduction of these indices will be discussed in the reduction section.

The translation of constants and builtin functions/operators is fairly trivial, they are just translated to the appropriate type of `CalcExpr`. In the case of infix operators which need translating into a function applications, they are translated to the names of builtin functions. For example, `x :: y` is translated into `cons x y`.

Once all of the substitutions of identifiers are performed, including substitutions of functions defined in `where` clauses, which can use the variables of their parent functions, according to the rules laid out above, a lambda calculus expression is returned by the visitor, which is free from any identifiers, and ready for reduction.

6 Reduction

Once the program has been translated from Funl to the lambda calculus, as represented by the various implementations of `CalcExpr`, it can be reduced. Reduction is fairly straightforward, as the lambda calculus only has a few rules. Listed here are the procedures for reducing each kind of expression:

- **Binary Integer Operations** — To reduce a binary integer operation expression, if both sides are constant expressions, that is, integers, the operation can be performed, and the expression is reduced to either the constant result of an arithmetic expression, or the Boolean result of a comparison operation. If either of the sides are not reduced, one of the sides is reduced, with the left hand side taking precedence to ensure leftmost-outermost reduction.
- **Applications** — To reduce a function application expression, if the function expression of the application is a lambda expression, the lambda expression is β-reduced using the argument expression (β-reduction will be covered later). If the function expression is not a lambda, the program will attempt to reduce it, and if it is unable to, it will then reduce the argument expression. This ensures that if there is a normal form, it will be found, while still evaluating in leftmost-outermost order.
- **Lambda Abstractions** — On their own, lambda abstractions cannot be reduced without the application of an argument to perform a β-reduction. So, the interior expression of the lambda abstraction will be reduced.
- **Variables and Constants** — Variables, which in the program are represented as DeBruijn indices, cannot be reduced on their own. Constants, being just numbers, can also not be reduced further.

6.1 β-Reduction

Probably the trickiest step in reducing the lambda calculus is the β-reduction. β-reduction occurs when an argument is applied to a lambda abstraction. Because DeBruijn indices are used to represent variables, every index that corresponds to the lambda abstraction being reduced must be replaced with the argument. Additionally, all indices that are free in the expression being reduced must be decremented by 1, because otherwise they will no longer refer to the correct lambda expression. For example, take the following reduction:

$$\lambda\lambda(\lambda\lambda 4)1 \tag{1}$$

$$\lambda\lambda\lambda 3 \tag{2}$$

Although the variable represented by index 4 in (1) is not being β-reduced, it must be decremented so that it still refers to the same lambda abstraction in (2).

Another complication that can occur is when substituting a DeBruijn index into a lambda expression, its index may change if it is under more lambdas after the substitution. Take the following reduction:

$$\lambda\lambda(\lambda\lambda 3)1 \tag{3}$$
$$\lambda\lambda\lambda 3 \tag{4}$$

When reducing (3), we cannot simply replace the index 3 with 1, as the index 1 would refer to a different lambda expression than it did previously. So, we must increment it to the number of lambdas it is being substituted under, in this case 2.

7 Printing

The practical specification asks us to print out the reduction steps in the program. To do this nicely, I store the name of the variables in the lambda abstractions and indices, although they are not used during parsing. For example, if we have a DeBruijn index 7, which was called `x` in the Funl program, it will be printed as `x@7`. This makes for easier reading and debugging of the lambda calculus expressions.

Because some programs (particularly recursive ones) get extremely large, and require huge amounts of reduction steps, I have included the `--no-output` option, which skips the printing step of the reduction.

8 Testing

Since the practical specification states that we can expect only valid programs to be translated and evaluated by the program, only valid inputs need to be checked. For the tests we were given, I have adjusted them slightly to fit my updated syntax, as well as In addition to running the provided test files, I have added quite a few tests of my own, to check various interesting behaviour of the pattern. The output of these functions can be seen by running them with the instructions provided at the top of this report. I detail these tests below:

- `arith.hs` — This test runs some basic arithmetic operations. It demonstrates one important note about the grammar, which is that no operator precedence is specified, and as a result the standard order of operations are not used. This means that one must be careful performing arithmetic operations, using parenthesis where necessary.
- `bool.hs` & `bool_full.hs` — This test runs some basic Boolean operations. It demonstrates that nested Boolean expressions work as expected, and can be evaluated to a normal form. `bool_full.hs` enumerates and tests all the different combinations of truth values for the boolean operators, and integer operators that return booleans.

- `curry.hs` — One feature that Funl gets more or less for free because of its correspondence with the lambda calculus is currying. This is demonstrated in this test.
- `rec.hs` — This file shows a very basic recursive function, which computes the sum of the numbers between 0 and n. It is useful because the inner workings of the Y combinator can be examined, without having to sift through too much program output as with some of the more complex recursive functions.
- `where.hs` — This file demonstrates local function definitions with `where`, showing some more complex situations than are given in the provided tests.
- `map.hs`, `foldr.hs` & `filter` — These tests implement the `map`, `foldr`, and `filter` functions, respectively. `map` takes a function and a list, and applies the function to each argument, returning the resultant list. `foldr` takes a function that accepts two arguments, some base value, and a list, and combines the values of the list using the function and the base value as the final node. For example, `foldr f z (1 :: 2 :: 3 :: nil)` will evaluate to `f 1 (f 2 (f 3 z))`. Filter takes a predicate function and a list, and returns a list with all the elements for which the predicate function returns true. These are all invaluable operations in functional programming languages, and being able to implement them shows that Funl is capable of more useful operations.
- `env.hs` — This file tests that functions have the other functions in their scope, regardless of the ordering in the file.
- `list.hs` — This file test the basic list construction functions, `cons` (or `::`), `hd`, `tl`. Lists are used in more interesting ways throughout the tests as well.

9 Conclusion

Overall, this was a very interesting practical, and being able to write my own functional programming language has been very rewarding, even if it is just a toy language. I've greatly enjoyed implementing the extension features, and hope to do more work like this in the future. In addition to being satisfying, it has also given me significant insight into the theory of programming language design.