Computer Science 3MI3 – 2020 homework 7

Pretty printing a representation of the untyped -calculus.

Mark Armstrong

November 1st, 2020

Contents

Introduction

For this homework, you will extend the implementations of the `UL` untyped -calculus in Scala and Ruby, which are provided in assignment 2, with a *pretty printing* method.

This language implementation makes use of unnamed variables through *de Bruijn* indices. This simplifies the implementation; specifically,

- we eliminate the need to replace names of variables with "fresh" names during substitution; instead,
  - (a relatively tedious problem; how do we keep track of fresh names?)
- we must only shift variable indexes according to the number of enclosing 's (variable binders.)
  - (a relatively trivial problem.)

The one major downside to a representation using unnamed variables is that the terms are far less human readable. Instead of terms such as

```
 x →   y →   z → x y z
```

we now have terms such as

```
      2 1 0
```

Hence, the task of this homework to implement a *pretty printer*, that chooses variable names for a term and "prints" the term (in fact, it should convert terms to strings, not print them directly.)

Updates

November 5th

- Testing and Docker setup has been added.
- The suggested output has been adjusted slightly to account for parentheses being added around terms being applied to each other.
- Sample code for how to interact with the `ULTerm` type was added to the assignment; you may want to try it out.
- The `ULTerm` code provided for the assignment has been updated,
  - first to correct a typo with a variable name, and
  - second to add `toString` methods which result in better output of the Scala `ULTerm` type.

  You may wish to apply those updates to your code as well.

Boilerplate

Submission procedures

Submission method

Homework should be submitted to your McMaster CAS Gitlab respository in the `cs3mi3-fall2020` project.

Ensure that you have **pushed** the commits to the remote repository in time for the deadline, and not just committed to your local copy.

Naming requirements

Place all files for the homework inside a folder titled `hn`, where `n` is the number of the homework. So, for homework 1, use the folder `h1`, for homework 2 the folder `h2`, etc. Ensure you do not capitalise the `h`.

Unless otherwise instructed in the homework questions, place all of your code for the homework in a single file in the `hn` folder named `hn.ext`, where `ext` is the appropriate extension for the language used according to this list:

- For Scala, `ext` is `sc`.
- For Prolog, `ext` is `pl`.

- For Ruby, `ext` is `rb`.
- For Clojure, `ext` is `clj`.

If multiple languages are used in the homework, submit a `hn.ext` file for each language.

If the language supports multiple different file extensions, you must still follow the extension conventions above.

Incorrect naming of files may result in up to a 10% deduction in your grade.

Do not submit testing or diagnostic code

Unless you are instructed to do so in the homework questions, **you should not submit testing code with your homework submission**.

This includes

- any `main` function,
- any `print` statements which output information **that is not directly requested as console output in the homework questions**.

If you do not wish to remove diagnostic print statements manually, you will have to find a way to ensure that they disabled in your final submission.

For instance, by using a wrapper on the print function or macros.

Due date and allowance for technical difficulties

Homework is due on the second Sunday following its release, by the end of the day (midnight). Submissions past 00:00 may not be considered.

If you experience technical difficulties leading up to the submission time, please contact Mark **ASAP** with the details of the problem and, if possible, attach the current state of your homework to the communication. This information will help ensure we are able to accept your submission once the technical difficulties are resolved.

Proper conduct for coursework

Individual work

Unless explicitely stated in the homework questions, all homework in this course is intended to be *individually completed.*

You are welcome to discuss the content of the homework in the public forum of the class Microsoft Teams team homework channel, though obviously solutions or partial solutions should not be posted or described.

Private discussions about the homework cannot reasonably be forbidden, but such discussions should follow the same guidelines as public discussions.

Inappopriate collaboration via private discussions which is later discovered by course staff may be considered academic dishonesty.

When in doubt, make the discussion private, or report its contents to the course staff by making a note of it in your homework.

To clarify what is considered appropriate discussions of homework content, here are some examples:

1. Discussing the language features introduced or needed for the homework.
    - Such as relevant builtin datatypes and datatype definition methods and their general use.
    - Code snippets that are not partial solutions to the homework are welcome and encouraged.
2. Questions of the form "What is meant by `x`?", "Does `x` really mean `y`?" or "Is there a mistake with `x`?"
    - Of course, questions of those form which would be answered by partial solutions are not considered appropriate.
3. Questions or advice about errors that may be encountered.
    - Such as "If you see a `scala.MatchError` you should probably add a catch-all `_` case to your `match` expressions."

Language library resources

Unless explicitely stated in the questions, it is not expected that you will use any language library resources in the homeworks.

Possible exceptions to this rule include implementations of datatypes we discuss in this course, such as lists or options/maybes, if they are included in a standard library instead of being builtin.

Basic operations on such types would also be allowed.

- For instance, `head`, `tail`, `append`, etc. on lists would not require explicit permission to be used.
- More complex operations such as sorting procedures would require permission before you used them.

Additionally, the standard *higher-order* operations including `map`, `reduce`, `flatten`, and `filter` are permitted generally, unless the task is to implement such a higher-order operator.

Part 1: The "pretty printer" `prettify` method [30 marks]

Implement a method `prettify` on the `UTTerm` type provided in assignment. The implementation of the `UTTerm` type can be found

- here in Scala, and
- here in Ruby.

Your method should take a `UTTerm` are return a string. As an example, given the `UTTerm` representing the -term (with unnamed variables)

```
  2 (1 0)
```

should produce a string such as

```
lambda x . lambda y . lambda z . (x) ((y) (z))
```

The exact choices of variable names do not matter; however, you must not change the meaning of the term. For instance, considering the above term, output

```
lambda x . lambda y . lambda x . (x) ((y) (x))
```

would be *incorrect.*

(You may want to use a helper method which has an argument to keep track of the variable names already in use. But as usual, the implementation details are within your control.)

Testing

Unit tests for the requested methods/functions are available

- here h7t.sc and

- here h7t.rb.

The contents of the unit test file are also repeated below.

Unlike previous homeworks, passing these tests does not guarantee your solution behaves as expected.
The tests will print an example string and the string your method returns. They do not have to match exactly; instead, the terms they show must be equivalent except for naming of bound variables. (This is to allow for variations in how variable names are chosen.)

Automated testing via Docker

The Docker setup and usage scripts are available at the following links. Their contents are also repeated below.

- Dockerfile
- docker-compose.yml
- setup.sh
- run.sh

Place them into your `h7` directory where your `h7.sc` and `h7.rb` files and the `h7t.sc` and `h7t.rb` (linked to above) files exist, then run `setup.sh` and `run.sh`.

You may also refer to the README for this testing setup and those files on the course GitHub repo.

Note that the use of the `setup.sh` and `run.sh` scripts assumes that you are in a `bash` like shell; if you are on Windows, and not using WSL or WSL2, you may have to run the commands contained in those scripts manually.

The tests

The contents of the testing script are repeated here.

h7t.sc

```
import $file.h7, h7._

val x = ULVar(0)
val y = ULVar(1)
val z = ULVar(2)
val u = ULVar(3)

val appvars = ULApp(ULApp(x,y), ULApp(z,u))
val lappvars = ULAbs(appvars)
val lllappvars = ULAbs(ULAbs(lappvars))

println("a
 ↪  Sample")
println(prettify(x))
println("b
 ↪  Sample")
println(prettify(y))
println("((a) (b)) ((c) (d))
 ↪  Sample")
println(prettify(appvars))
println("lambda a . ((a) (b)) ((c) (d))
 ↪  Sample")
println(prettify(lappvars))
println("lambda a . lambda b . lambda c . ((c) (b)) ((a) (d))
 ↪  Sample")
println(prettify(lllappvars))
```

h7t.rb

```
require_relative "h7"

x = ULVar.new(0)
y = ULVar.new(1)
z = ULVar.new(2)
u = ULVar.new(3)

appvars = ULApp.new(ULApp.new(x,y), ULApp.new(z,u))
lappvars = ULAbs.new(appvars)
lllappvars = ULAbs.new(ULAbs.new(lappvars))

puts "a
 ↪  Sample"
puts x.prettify
puts "b
 ↪  Sample"
puts y.prettify
puts "((a) (b)) ((c) (d))
↪  Sample"
puts appvars.prettify
puts "lambda a . ((a) (b)) ((c) (d))
↪  Sample"
puts lappvars.prettify
puts "lambda a . lambda b . lambda c . ((c) (b)) ((a) (d))
↪  Sample"
puts lllappvars.prettify
```

The Docker setup

The contents of the Docker setup files and scripts are repeated here.
Dockerfile

```
# Define the argument for openjdk version
ARG OPENJDK_TAG=8u232

FROM ruby:2.7.2-buster

# Setup to install Scala
RUN apt-get update && \
    apt-get install scala -y && \
    apt-get install -y curl && \
    sh -c '(echo "#!/usr/bin/env sh" && \
    curl -L
↪   https://github.com/lihaoyi/Ammonite/releases/download/2.1.1/2.12-
↪   2.1.1) > /usr/local/bin/amm &&
↪   \
    chmod +x /usr/local/bin/amm'
RUN (rm -rf /root/.cache)

# Set the name of the maintainers
MAINTAINER Habib Ghaffari Hadigheh, Mark Armstrong
↪  <ghaffh1@mcmaster.ca, armstmp@mcmaster.ca>

# Set the working directory
WORKDIR /opt/h7
```

docker-compose.yml

```
version: '2'
services:
  service:
    build: .
    image: 3mi3_h7_docker_image
    volumes:
      - .:/opt/h7
    container_name: 3mi3_h7_container
    command: bash -c
      "echo 'Scala testing' &&
       echo
       ↪  '----------------------------------------------------------------------'
       ↪  &&
       amm h7t.sc &&
       printf '\\n\\n\\n' &&
       echo 'Ruby testing' &&
       echo
       ↪  '----------------------------------------------------------------------'
       ↪  &&
       ruby h7t.rb &&
       echo
       ↪  '----------------------------------------------------------------------'"
```

setup.sh

```
docker-compose build --force-rm
```

run.sh

```
# Run the container
docker-compose up --force-recreate
# Stop the container after finishing the test run
docker-compose stop -t 1
```

README.md

```
# Instructions for automated testing using Docker

We have already created a `Dockerfile` here which specifies
all the necessary packages, etc., for compiling and running
↪  your code.
You only need to follow the instructions below to see
```

~~~
the results of unit tests designed to check your
↪  implementation.

## Setup
We use `docker-compose` and its configuration file to build
↪  the image.
Assuming you have `docker` and `docker-compose` installed,
simply execute
```shell script
./setup.sh
~~~
to generate the image.

Prepare your code for the running the tests
You only need to place the `h7t.sc` and `h7t.rb` unit test
↪ files and
the `run.sh` file in the same directory as your `h7.sc` and
↪ `h7.rb` source files.

Running the tests
As with the build process, we have already put
the configuration needed for running the test inside
↪ `docker-compose.yml`.
Simply execute
```shell script
./run.sh
```
to run and see the results of the tests.

Sample solution

These sample solutions make use of `variableName` methods which calculate a name from an integer, `index`. Index `0` is assigned to `a`, `1` to `b`, etc., until `25` is assigned to `z`. After that, the pattern is repeated, but with a digit (or, if there are enough variables, digits) following the letter.

In Scala:

```scala
def prettify(t: ULTerm): String = {
  def variableName(index: Int): String = {
    val letter = (index % 26 + 97).toChar
```

```
    val number = index / 26
    if (number == 0)
      letter.toString
    else
      letter.toString + number.toString
  }

  def prettifyHelper(t: ULTerm, currentBinders: Int): String =
  ↪  t match {
    case ULVar(i) if i < currentBinders =>
      // The indexing goes from the innermost binder to the
      ↪  outermost.
      // So if i is 0, it refers to the (currentBinders -
      ↪  1)'th bound variable.
      //    If i is 1, it refers to the (currentBinders -
      ↪  2)'th bound variable.
      variableName(currentBinders - i - 1)
    case ULVar(i) =>
      // Indexing can go in increasing order for free
      ↪  variables.
      variableName(i)
    case ULAbs(t1) =>
      val name = variableName(currentBinders)
      val body = prettifyHelper(t1,currentBinders+1)
      "lambda " + name +  " . " + body
    case ULApp(t1,t2) =>
      val t1_pretty = prettifyHelper(t1,currentBinders)
      val t2_pretty = prettifyHelper(t2,currentBinders)
      "(" + t1_pretty + ") (" + t2_pretty + ")"
  }

  prettifyHelper(t,0)
}
```

And in Ruby (including only the relevant methods for each class):

```
class ULTerm
  def variableName(index)
    # Choose the index'th character past lowercase a.
    # If the index is more than 26, we'll also
    # append a number, starting from 1 and counting up as
    ↪  needed.
    letter = (index % 26 + 97).chr
    number = index / 26  # for some reason, the slash breaks
    ↪  my fontification
                         # until a matching one, so here: /
                         # If my students are reading this,
                         ↪  don't worry about it;
                         # it's some sort of bug with my
                         ↪  editor.
    if number == 0
      letter
    else
      letter + number.to_c
    end
  end

  def prettify
    prettify_helper(0)
  end
end

class ULVar < ULTerm
  def prettify_helper(current_binders)
    if @index < current_binders
      variableName(current_binders - @index - 1)
    else
      variableName(@index)
    end

    # This alternate implementation names all bound variables
    ↪  xn,
    # where n is an integer
    # and all free variables zn where n is an integer.
    #if @index < current_binders
    #  # This is a bound variable.
    #  'x' + @index.to_s
    #else
    #  # This is a free variable.
    #  'z' + (@index - current_binders).to_s
    #end
  end
end

class ULAbs < ULTerm
  def prettify_helper(current_binders)
    "lambda " + variableName(current_binders) + " . " +
    ↪  @t.prettify_helper(current_binders+1)
    # This alternate code matches that in the ULVar method
    # which gave all bound variables the name "x"
    #"lambda x" + current_binders.to_s + " . " +
    ↪  @t.prettify_helper(current_binders+1)
  end
end

class ULApp < ULTerm
  def prettify_helper(current_binders)
    pretty1 = @t1.prettify_helper(current_binders)
    pretty2 = @t2.prettify_helper(current_binders)
    "(" + pretty1 + ") (" + pretty2 + ")"
  end
end
```