Programming in Three Dimensions

William Troiani and Daniel Murfet
University of Melbourne, AMSI

Objectives

This project was broken up into 2 parts. First, was getting up to date with the current research. Then we had to use this understanding to design and build a computer program.

- To understand what "computability" means.
- To understand the relationship between Comptuer Science, Mathematics, and Logic.
- Specifically, to understand the differences between classical logic, and linear logic.
- To write a passer for the language of linear λ-calculus
- To write an interpreter for the language of linear λ-calculus

Introduction

What can and can't a computer do? This question is central to the area of Mathematics now referred to as *computability theory*. In the early 1900's, there was interest in not only finding solutions to specific questions in mathematics, but also in finding algorithms which can solve these problems for us. The first step in achieving this is to first answer the question "what is an algorithm"? Godel, Church, and Turing all came up with different looking, but logically equivalent definitions of an 'algorithm' as a mathematical object. With this definition under our belts, we can now start exploring what kinds of problems can be solved by following an algorithm, and what kinds of problems can't. Perhaps surprisingly, there are some general problems for which there exists no algorithmic solution. For example, it has been proven that there exists no algorithm which takes as input a mathematical statement, and returns "true", if the statement is true, and returns "false" if the statement is false. Note: The phrase "Mathematical statement" here means a statement of 1st order logic [1].

For this project, we focused on Church's approach, which is that of the λ-calculus. Essentially, λ-calculus is a formal language used to describe functions. According to Church, an algorithm is a list of instructions to follow, where the tasks in the list can be described and executed in this language [2]

An example

The following process shows how λ-calculus adds two numbers together.

Note, at each step we are performing an instance of "β reduction", which is simply substitution, ie, $(\lambda x.x)y \rightarrow_\beta y$ (all we have done is substituted y in for x).

Define:

$1 := \lambda f.\lambda x.fx$

$2 := \lambda f.\lambda x.f(fx)$

$3 := \lambda f.\lambda x.f(f(fx))$

$\text{PLUS} := \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$

Then,

$$\begin{aligned}&\text{PLUS } 1\ 2 = \\ &(\lambda m.\lambda n.\lambda f.\lambda x.mf(nfx))(\lambda f.\lambda x.fx)(\lambda f.\lambda x.f(fx)) \\ &\rightarrow_\beta (\lambda f.\lambda x.(\lambda f.\lambda x.fx)f((\lambda f.\lambda x.f(fx))fx)) \\ &\rightarrow_\beta \lambda f.\lambda x.f((\lambda f.\lambda x.f(fx))fx)) \\ &\rightarrow_\beta \lambda f.\lambda x.f(f(fx)) = 3\end{aligned}$$

What about other logical systems?

Let us recall our previous claim that there exists no algorithm which takes as input a statement of first order logic, and returns "yes" if the statement is true, and "no" if the statement is false. This result was established by Church using his definition of an algorithm which was based off his language of "λ calculus". Perhaps this would motivate us to explore a different, but similar language, which can express statements in other logical systems, for example, Linear Logic.

Linear Logic is a logic system which can be interpreted as one which cares about "resources". For instance, it would make sense that if I have two different reasons as to why some statement A is true, and I also know that another statement B follows from A, then I don't need both of those reasons as to why A is true, to insist on the truth of B. However, if we change our thinking a bit, and rather than thinking of A as a statement, but rather as a resource, and change "two different reasons", to "two different ways of obtaining 1 of that resource", then it becomes clear that I will absolutely sometimes needs both those "reasons" to infer B.

As a concrete example. If A was the statement "It will rain today". And if B is the statement "I will wear my coat". Then it is true that B follows from A, and so any non-zero amount of reasons as to why it will rain today will be enough to convince me that I will wear my coat. On the other hand, if A was the resource "$1", and B was the resource "1 bottle of $2 water", then I need two ways of obtaining 1 resource of A, in order to "infer" B.

These two examples highlight the difference between classical, and linear logic.

If we think about logic in this way, then in doesn't require a big stretch of the imagination to realise that computer scientists (who spend all their time thinking about how to optimally shuffle the memory states of their computer around) would be interested in a formal mathematical system where they can rigorously talk about how to acquire and distribute computational resources.

The code

The main objective of our project was to write a program which takes as input a sentence from the language linear λ-calculus (the extension of Church's λ-calculus which interprets sentences from linear logic), and returns a passing tree (presented by nested lists) for that sentence.

The figure below displays an example of the input and output of the program.

```
In [12]: term = '(lq.(lp.(copy q as r, s in (lz.
(derelict(p) (derelict(s) (derelict(r) z)))))))'

In [13]: syntax_tree(term)
Out[13]:
['Abs',
 'var q',
 ['Abs',
  'var p',
  ['Contraction',
   ['var q'],
   ['var r'],
   ['var s'],
   ['Abs',
    'var z',
    ['App',
     ['Dereliction', ['var p']],
     ['App',
      ['Dereliction', ['var s']],
      ['App', ['Dereliction', ['var r']], ['var
z']]]]]]]]]
```

Future research

The next step with this program is to write an extension to it which not only returns a passing tree for a given linear λ-calculus sentence, but then goes ahead and performs all possible β-reduction steps (reducing it to a normal form). This would turn our passer into a full interpreter, which would have many uses for people studying this area of computability theory.

Additional Information

So where does the programming in three dimensions part come in?

As it turns out, we can also use categories to represent these logical systems. This is all part of a correspondence referred to as the "curry howard isomorphism", which relates select areas of λ-calculus, logic, and category theory. The important realisation to have, is that there is a natural notion of three dimensionality in the category theory interpretation, and since we have this correspondence, there must also be one in the λ calculus, which is a model of computation. So the bottom line is "three dimensional" programs exist, but we don't know what they are yet.

References

[1] Alonzo Church, "A note on the Entscheidungsproblem", Journal of Symbolic Logic, 1 (1936), pg 40,41.

[2] Sorensen and Urzycyn, "Lectures on the curry howard isomorphism".

Contact Information

- Email: wtroiani@student.unimelb.edu.au
- Email: d.murfet@unimelb.edu.au

AMSI AUSTRALIAN MATHEMATICAL SCIENCES INSTITUTE

THE UNIVERSITY OF MELBOURNE