Towers of Hanoi in Lambda Calculus

Martijn Vermaat*

29th February 2004

Abstract

In this document we will see how a solution to the Towers of Hanoi problem can be constructed in pure, untyped Lambda Calculus.

Contents

1 The problem of the Towers of Hanoi

The problem definition below is taken from Wikipedia[1].

> The Tower of Hanoi (also called Towers of Hanoi) is a mathematical game or puzzle. It consists of three pegs, and a number of discs of different sizes which can slot onto any peg. The puzzle starts with the discs neatly stacked in order of size on one peg, smallest at the top, thus making a conical shape.
>
> The object of the game is to move the entire stack to another peg, obeying the following rules:
>
> - only one disc may be moved at a time
> - a disc can only be placed onto a larger disc (it doesn't have to be the adjacent size, though: the smallest disc may sit directly on the largest disc)

*E-mail: mvermaat@cs.vu.nl, homepage: http://www.cs.vu.nl/~mvermaat/

[1] Tower of Hanoi, http://en.wikipedia.org/wiki/Tower_of_Hanoi

This problem is often used in programming courses to illustrate the use of recursion. Solutions have been programmed in allmost every language known to human kind. Having taken a introductory course on Lambda Calculus, I searched for a solution for the Towers of Hanoi problem in Lambda Calculus. Much to my surprise, I wasn't able to find one.

2 A solution in untyped Lambda Calculus

2.1 The simple solution in Haskell

The solution is basically a translation to Lambda Calculus of the following obvious solution in Haskell:

```
hanoi (0, _, _, _)          = []
hanoi (n, from, to, using) = hanoi(n-1, from, using, to) ++
                                (from, to) :
                                hanoi(n-1, using, to, from)
```

That is, to move `n` discs from `from` to `to` (using `using`), all we have to do is move `n-1` discs to `using`, then move the last disc to `to`, and finaly move the `n-1` discs from `using` to `to`. The trivial case here is moving 0 discs–just don't do anything (return the empty list of pairs). Calling `hanoi` as follows will return a list of pairs, where each pair denotes a move:

```
> hanoi (3, 1, 3, 2)
[(1,3),(1,2),(3,2),(1,3),(2,1),(2,3),(1,3)]
```

In Lambda Calculus, we can try to do the same thing, using notations for pairs, lists, and natural numbers. Because we are talking about untyped Lambda Calculus, the result will be complicated to read (but nevertheless it can well be a correct solution).

2.2 Translating to Lambda Calculus

To denote numbers, we will use the common notation known as Church Numerals. The most important part of the translation is what we can do in Haskell with simple pattern matching: to distinguish between the number 0 and all numbers above.

By realising what happend if we would apply two arguments on a Church Numeral, we find the following body for our Hanoi function:

$$\lambda n.n \quad (\lambda x.A) \quad B$$

where A is some Lambda term not containing x, and B an arbitrary Lambda term. This function expects one argument, a Church numeral, and will reduce to A on applications on numbers greater than zero, and B on application on zero.

Now, moving on, we can fill in the B part. The resulting term for zero has to be the empty list. We do have to make sure though, to discard the three

arguments $from$, to, and $using$. So, using $empty$ as the empty list constructor, we have our new Hanoi function:

$$\lambda n.n \quad (\lambda x.A) \quad (\lambda f\ t\ u.empty)$$

The A part is a bit more complicated, but can nevertheless be translated from the Haskell solution fairly straightforward by making use of some predefined functions:

$$\begin{array}{lll} \lambda f\ t\ u\ . & append & (hanoi\ (pre\ n)\ f\ u\ t) \\ & & (cons\ (pair\ f\ t)\ (hanoi\ (pre\ n)\ u\ t\ f)) \end{array}$$

where:

- $append$ is the append operator for lists
- pre is the predecessor function
- $cons$ is the list constructor function
- $pair$ is the pairing constructor function
- $hanoi$ is a recursive call to the Hanoi function itself

Combining these results, we then have a complete Hanoi function:

$$\begin{array}{lllll} hanoi \rightarrow & \lambda n.n & (\lambda f\ t\ u\ .\ append & (hanoi\ (pre\ n)\ f\ u\ t) & \\ & & & (cons\ (pair\ f\ t)\ (hanoi\ (pre\ n)\ u\ t\ f))) & \\ & & (\lambda f\ t\ u.empty) & & \end{array}$$

2.2.1 Recursion in Lambda Calculus

The problem is that we now have a definition in terms of it self. We say we know how $hanoi$ goes, but only if we know $hanoi$. We can transform this recursive definition to a non-recursive one using a little trick and the fixed-point combinator Y[2].

We can rewrite the term we just had to the following (taking the $hanoi$ references outside the term):

$$\begin{array}{lllll} hanoi \rightarrow & (\lambda h\ n.n & (\lambda f\ t\ u\ .\ append & (h\ (pre\ n)\ f\ u\ t) & \\ & & & (cons\ (pair\ f\ t)\ (h\ (pre\ n)\ u\ t\ f))) & \\ & & (\lambda f\ t\ u.empty)) & & \\ & hanoi & & & \end{array}$$

The resulting definition is still recursive, but if we look closely, we can see that $hanoi$ seems to be a fixed-point of the function consisting of the complex inner term (everything between the two $hanoi$'s).

That observation makes it possible to write $hanoi$ using the fixed-point combinator Y, as follows:

[2]For a short explanation of the fixed-point combinator, see for example http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?fixed+point+combinator on FOLDOC

$$\begin{array}{lllll} hanoi \rightarrow & Y & \lambda h\ n.n & (\lambda f\ t\ u\ .\ append & (h\ (pre\ n)\ f\ u\ t) \\ & & & & (cons\ (pair\ f\ t)\ (h\ (pre\ n)\ u\ t\ f))) \\ & & & (\lambda f\ t\ u.empty) & \end{array}$$

And there we have our final definition of a solution for the Towers of Hanoi problem in untyped Lambda Calculus. Please don't try to prove its correctness by reducing

$$hanoi\ 4\ left\ right\ middle$$

by hand. Let's instead just assume it's correct enough.

A Function definitions

Below are the definitions of the functions used in the construction of our solution. These definitions are widely used and are certainly not 'invented' by me. Because there we have smart people for.

General combinators

$$\begin{array}{rcl} I & \rightarrow & \lambda x.x \\ Y & \rightarrow & \lambda f.\ (\lambda x.f(x\ x))\ (\lambda x.f(x\ x)) \end{array}$$

Church numerals

$$\begin{array}{rcl} zero & \rightarrow & \lambda s\ z.z \\ suc & \rightarrow & \lambda x\ s\ z.s\ (x\ s\ z) \\ pre & \rightarrow & \lambda c.c\ (\lambda z.\ (z\ I(suc\ z)))\ (\lambda a.\ (\lambda x.zero)) \end{array}$$

Pairing constructor

$$pair \rightarrow \lambda l\ r\ z.z\ l\ r$$

Lists

$$\begin{array}{rcl} empty & \rightarrow & \lambda x\ y.y \\ cons & \rightarrow & \lambda h\ t\ z.z\ h\ t \\ append & \rightarrow & Y\quad (\lambda a\ l\ r.l\ (\lambda h\ t\ z.cons\ h\ (a\ t\ r))\ r) \end{array}$$