Algebra and the lambda calculus*

Aubrey Jaffer

Abstract

An algebraic system which includes Church's lambda calculus and currying is presented. Closures, applications, and currying are implemented using variable elimination.

The usual connection made between the lambda calculus and algebra is to construct the integers using the lambda calculus and then construct algebraic (and other formulas) by Godelizing them.

The system described here reverses this connection by implementing the lambda calculus in an algebraic system. It is used in the JACAL symbolic mathematics system and gives JACAL the ability to represent functions as members of the algebraic system. All of the system's operations (including simplification) can then be applied to functions as easily as to expressions.

1 Algebraic representation

Given an underlying representation for multivariate polynomials we can represent an equation as a multivariate polynomial with the understanding that the polynomial is equal to zero. We can convert typical equations to this form by multiplying both sides by the denominators and then subtracting the left side from both sides. For instance:

$$f/(c+d) = (a-b)/g;$$

Yields:

$$0 = (a-b)\,c + (a-b)\,d - f\,g$$

How to represent expressions? The usual approach is to use a polynomial or a ratio of polynomials. Instead, we will introduce a special variable "@", calling it the value variable. The value of a polynomial involving @ is that polynomial "solved" for @. For instance, the expression:

$$\frac{-1+x}{1+x}$$

[0]Original: `https://people.csail.mit.edu/jaffer/lambda.txt`.

is represented internally as:

$$0 = -1 + x + (-1 - x)\,@$$

This allows us to represent irrational expressions as well:

$$0 = -1 - y^5 - @ - @^5$$

Is the root of a fifth degree polynomial.

For the rest of this paper the examples will not show @ if the values can be represented in usual mathematical notation without it (as JACAL does).

2 Substitution

At this level of algebraic abstraction we do all operations using variable elimination. Variable elimination is the process of combining n polynomial equations so that m variables do not appear in the $(n - m)$ resulting equations (where $n > m$). Common techniques used include resultants [1][2][4][5] and Groebner Bases [3][4][5].

$$\text{eliminate}\big([a + c^2 = b, b + c^2 = 2], [c]\big);$$

Yields:

$$0 = 2 + a - 2\,b$$

Common symbolic transformations can be done by constructing auxiliary polynomial equations and eliminating variables between them and the original polynomial equations. The operation we are interested in for the next section is substitution. We can substitute an expression for a variable by constructing an auxiliary equation of the variable and then eliminating that variable. Suppose we want to substitute $(a\,x + b)^2$ for g in $g + 1/g$. We construct the equation $g = (a\,x + b)^2$ and then:

$$\text{eliminate}\big([g = (a\,x + b)^2, g + 1/g], [g]\big);$$

$$\frac{1 + b^4 + 4\,a\,b^3 x + 6\,a^2\,b^2\,x^2 + 4\,a^3\,b\,x^3 + a^4\,x^4}{b^2 + 2\,a\,b\,x + a^2\,x^2}$$

Eliminate deals only with polynomial equations so remember that $g + 1/g$ internally is:

$$0 = 1 + g^2 - g\,@$$

and similarly for the result.

3 Functions

A similar approach to the use of @ can be used for arguments as well. We will name these arguments $@_1$, $@_2$, and so on.

Functions do not need to use all of their arguments. However, in this system, a function must use at least one of its arguments. With this constraint, the only difference between a function and an expression is the presence of $@_n$ variables. Our functions can return either equations or expressions; those containing @ are expressions and those without, equations.

A function which ignores its first two arguments is $\lambda([x, y, z], 1/z - z)$, which would be represented as

$$\frac{1 - @_3^2}{@_3}$$

These functions can freely mix bound and free variables. The following expression, with free c and bound x and y,

$$f : \lambda\big([x, y], c\,(y + x)/(y - x)\big);$$

is simply:

$$\frac{c\,@_1 + c\,@_2}{-@_1 + @_2}$$

We can now apply this function. We don't have to always apply it to two arguments, we can apply it to just one also. This is called "currying" an argument. The application

$$g : f(x);$$

substitutes x for $@_1$ from the polynomial equation for f. It also "bumps" $@_2$ down to $@_1$ (also by substitution). This results in a new function of one argument:

$$\frac{c\,x + c\,@_1}{-x + @_1}$$

It this function is applied to one argument (which is a non-function) the result will be a non-function. For example $g(a + b)$ yields:

$$\frac{(-a - b)\,c - c\,x}{-a - b + x}$$

This result is exactly the same as the result of applying $f(x, a + b)$.

4 Alpha-conversion

The above method is sufficient when the arguments to functions are not themselves functions. But when applying functions to functions differences in the order of elimination

produce different results. Consider applying $@_1 - @_2$ to the arguments $(@_2, @_1)$. The result should be $@_2 - @_1$. But if we curry an argument we get $@_2 - @_2 \rightarrow 0$ applied to $@_1$.

This problem is similar to the inadvertent capture of free identifiers by macros in languages like C and Lisp. The solution to this problem is called alpha-conversion in the lambda calculus and Hygienic Macro Expansion in a paper of that title by E. E. Kohlbecker, D. P. Friedman, M. Fellinson, and B. Duba [6].

Since the names of bound identifiers are unimportant we will substitute new names for those lambda variables for which we will later substitute arguments. This eliminates possible conflicts between the variables bound in the current function and variables in its arguments (which are free relative to this function).

In the above example substitute $:@_1$ for $@_1$ and $:@_2$ for $@_2$ in $@_1 - @_2$ yielding $:@_1 - :@_2$. Now substitute $@_2$ for $:@_1$ and $@_1$ for $:@_2$. This then yields the desired result $@_2 - @_1$.

Currying an argument would substitute $@_2$ for $:@_1$ and $@_1$ for $:@_2$ in $:@_1 - :@_2$ to produce $@_2 - @_1$. When this function is applied to the remaining argument, $@_1$, the result is $@_2 - @_1$ as before.

5 Lambda

A symmetrical situation to currying of arguments is binding a variable over a function. $\lambda([y], \lambda([x], x - y))$ should yield the same function as $\lambda([y, x], x - y)$. The trick here is to "bump up" any lambda variables in an expression when binding additional variables. To execute $\lambda([y], @_1 - y)$ we substitute $@_2$ for $@_1$ and then $@_1$ for y.

6 Vectors and matrices

Vector and matrix valued functions can be represented by vectors and matrices some of whose entries are lambda expressions. Clearly a vector or matrix function applied to scalar arguments should return a vector or matrix with the same shape as the function.

The case of a scalar function applied vector arguments can work if the multiplication used by the function is of a type compatible with the arguments. Inner product is commutative while matrix multiplication is not. Another possibility here is to have a mechanism for allowing lambda expressions to reference elements vector and matrix arguments.

The case of vector or matrix functions applied to vector or matrix arguments gets stickier. Allowing only elements of the arguments to be operated on is one solution; Another is to incorporate the structure of the arguments inside the structure of the function or vice versa.

7 Differential algebra

These techniques can be extended to differential algebra as well. In differential algebra the derivative of a variable, written v', can act as an variable in polynomials. Derivatives of derivatives are also allowed and can be written v'' and so on.

When applying a function, each distinct derivative of a lambda variable with a corresponding argument requires that an equation be generated and that derivative variable be eliminated. We equate the nth derivative variable with the nth total derivative of its corresponding argument. For instance to apply the function $@_1'/@_2'$ to (x^3, x) we

$$\text{eliminate}\big([@1'/@2', @1' = (x^3)', @2' = (x)'], [@1', @2']\big);$$

$$3\,x^2$$

As illustrated by this example, differential operators are now as easily expressed as functions. This worked well for the univariate case; what about multiple variables? $(@_1'/@_2')\big((x+y)^2, x\big)$ yields

$$\frac{2xx' + 2x'y + (2x+2y)y'}{x'}$$

We need to set y' to 0 to get the expected answer $2x + 2y$. But when we curry we need to have the differentials remain until all the lambda variables are consumed. $(@_1'/@_2')((x+y)^2)$ gives

$$\frac{2xx' + 2x'y + (2x+2y)y'}{@_1'}$$

We don't want to set x' and y' to 0 in the numerator because the result would be 0. We don't know which differential until $@_1'$ is substituted for. I think the solution here is to set to zero differentials occurring only in the numerator and only for those expressions containing no lambda variables or their derivatives.

8 Conclusion

I have shown how to implement functional abstraction (lambda), application and partial application (currying) in a system built on variable elimination from polynomials.

References

[1] Bareiss, E.H.: Sylvester's identity and multistep integer-preserving Gaussian elimination. Mathematics of Computation 22, 565–578, 1968.

[2] Uspensky, J.V.: Theory of Equations McGraw-Hill Book Co., Inc., 1948

[3] Thomas W. Dube: "The Structure of Polynomial Ideals and Groebner Bases", SIAM JOURNAL ON COMPUTING, (19,4) (August 1990) pp. 750–773.

[4] Hoffmann, C. M.: Geometric and Solid Modeling: An Introduction. Morgan Kaufmann Publishers, Inc. San Mateo, California, 1989

[5] Geddes, K.O., Czapor, S.R., Labahn, G.: Algorithms for Computer Algebra. Kluwer Academic Publishers

[6] Hygienic Macro Expansion E.E.Kohlbecker, D.P.Friedman, M.Fellinson, and B.Duba 1986 ACM Conference on Lisp and Functional Programming Pages 151–159