Lambda-X

Owen Meyer
Nick Stephens

June 8, 2013

Project Goal

The ultimate object of the Lambda-X project was to explore Alonzo Church's Lambda Calculus as a model for computation and language construction, as a follow up to the use of finite state machines and Turing machines that was presented in Computability taught at Evergreen by Neal Nelson and Sherri Shulman.

The Lambda Calculus and Dot-lam

In the first phase of the project, Pure Lambda Calculus was used to demonstrate how complex meta computation is possible in this system. Following the idea behind the Universal Turing Machine is the Universal Lambda Function: a Lambda expression that takes an encoded Lambda term as an argument and performs beta reduction on the encoded term. Lambda Calculus expresses all of the necessary logic and symbolic capacities necessary to represent functional completeness in it's simple, syntactic grammatical structures. When the rules of reduction are applied to well formed Lambda terms, any possible computation can be represented symbolically. In the pure Lambda Calculus, both the operators and operand of any computational expression are themselves Lambda terms . Additionally Lambda Calculus provides the control framework that is required of such a Turing Complete system in the form of choice, recursion, $< variable, expression >$ bindings, and scoping. It is interesting to note that in Pure Lambda Calculus, terms of Lambda Calculus, in turn manipulated by Lambda Calculus' control structure, can at one point be operands to symbolic expressions but at

another point be a control mechanism of the system. For example, $(\lambda a.\lambda b.a)$ symbolically represents the value "true" as an operand that can be used with logic operators and at another point, $(\lambda a.\lambda b.a)$ is the control mechanism of choice in a conditional branch selecting the first of two arguments. Thus, an encoding for a data type such as booleans is not merely syntactic but there is an underlying mechanism of logic at work as well. Combined with the interpretive key to some set of encodings of operators and operands (such as Church encodings), Lambda expressions are denotations of the semantic meaning behind expressions. For example, given the set of encodings (which is sufficient for functional completeness):
True := $(\lambda a.\lambda b.a)$
False := $(\lambda a.\lambda b.b)$
Or := $(\lambda p.\lambda q.p\ p\ q)$
Not := $(\lambda m.\lambda a.\lambda b.m\ b\ a)$
The expression $(\lambda m.\lambda a.\lambda b.m\ b\ a)\ ((\lambda p.\lambda q.p\ p\ q)(\lambda a.\lambda b.a)(\lambda a.\lambda b.b))$ denotes the expression $\neg(T\ \vee\ F)$. It is easy to see how other logic propositions can be constructed and given De Morgan's law, functional completeness can be attained with "True", "False", "Or", and "Not".

This phase of the project explores Pure Lambda Calculus by implementing a universal Lambda Function in the vein of the Universal Turing Machine. This function (Lambda Expression) requires that the inputs be encoded into some form that allows for equality testing between terms. This is a primary concern as there is no ability within the Lambda Calculus for checking universally whether two terms are equivalent, due to the purely syntactic nature of the Lambda Calculus. However with Church encoding for positive integers (Church numerals) there is an effective test for whether a term is Church zero upon which a general equality between any two Church numerals can be built. With this test, and an encoding of Lambda terms consisting of nested Church pairs of Church numerals, an equality function for testing whether any two encoded Lambda terms are equivalent can be built. Without this ability there is no way to know what the inputs are and respond accordingly. This response amounts to selective application of a Variable/Expression substitution function in the service of beta reducing the input term. To write the Universal Lambda term in a way that modeled the pure Lambda Calculus as close as possible we used a couple of auxiliary modules to construct a tool we called Dot-lam. Dot-lam is a minimalistic programming language which compiles to high-level Haskell code; it consists of the following parts:

- A lambda term parser which parses input text according to the grammar for pure Lambda Calculus.

- A file parser which applies the term parser to a *.lam file of Lambda Calculus expressions. This module outputs a *.hs file of Abstract Syntax for Lambda expressions.
- A Lambda Expression encoder and decoder. The encoder inputs parsed Lambda expressions and translates them into Church pairings of Church numerals using an environment of $< Variable, Churchnumeral >$ to assign numberings to the names of variables per scoping requirements. The decoder works by reversing the pairings in the environment from (a,b)'s to (b,a)'s. The encoder, evaluator and decoder are linked up with Haskell's sweet monad transformer libraries.
- A reduction module which can perform varying evaluation strategies on Lambda Expression represented by the Haskell abstract data types.
- Lastly, the LambdaCore. A file written entirely in Dot-lam that contains all of the Lambda terms making up the Universal Lambda Function. This core is the evaluator of the Lambda Calculus written in Lambda Calculus.

All tests on running the Universal Lambda Function were successful, including taking the Lambda term, application of the Universal Lambda Function to some encoded Lambda term, and encoding that and evaluating the application of the Universal Lambda Function to that.

Thoughts on the Universal Lambda Function

The universal Lambda Function is a morphism of a potential programmable computer. While researching for and writing the Universal Lambda Function, some interesting findings came to light which suggested further exploration of these ideas. Particularly that the core "language" written in pure Lambda Calculus that was used to write the Universal Lambda Function very closely resembles the core functional language Lispkit which the original (soon to be mentioned) SECD machine has been used to evaluate.

The SECD, A Symbolic Architecture

While there are precise rules for the evaluation of Lambda terms in the form of beta reduction, they allow for much choice in the sequence of how the reductions can be carried out. In order to implement a reduction strategy for acquiring the evaluation expressions such as these, an operational semantic must be applied to Lambda Calculus. As there are numerous options for what order reductions can occur in, there are numerous reduction semantics that can be applied to Lambda Calculus. For example, Launchbury's natural semantics for lazy evaluation or those outlined by Sestoft for call by value, call by name, etc... The implementation of any one of these semantic interpretations of Lambda Calculus can be performed by translating functional expressions into imperative sequences of instructions. Regardless of which operational semantic is at work, scoping is determined in the static composition of the Lambda expressions themselves (more is said below about scoping). There are many stack based virtual architectures that implement these various reduction semantics of Lambda Calculus. These abstract machines where initially introduced by Peter Landin in his paper "The Mechanical Evaluation of Expressions"[1] where he outlines the mechanisms of the SECD machine. The SECD virtual machine employs a four stack architecture. The "S" stack is a stage, the "E" stack is the environment, the "C" stack is the control and the "D" stack is used for storing continuations.

Implementing a Version of the Lambda Calculus

Excepting purely theoretical purposes, it doesn't make too much sense to implement one of these abstract machines on Pure Lambda Calculus. Rather, simply typed or polymorphically typed Lambda Calculus is more suitable where these machines are concerned. The reasons being efficiency and useability. Pure Lambda Calculus isn't nearly as efficient and requires deciphering to understand it's meaning. The simply typed Lambda Calculus trades encoding of what would be the operands and operators of expressions for base data types ,such as `int` and `char`,and the operators which operate upon them. The polymorphically typed Lambda Calculus additionally offers algebraic data types to the mix. With these systems, the program control is still being provided by Lambda Calculus, including scoping, recursion, and bindings. An informal picture of a typed Lambda Calculus grammar:

Expression := $Lambda \mid Application \mid Variable \mid Operations \mid\ < optional : Let >$
Lambda := $\lambda < Variable >\ .\ Expression$
Application := $\lambda < Expression > < Expression >$
Operations := $\ a\ set\ of\ base\ values\ and\ operators$
Let := $Let\ \ < Variable > < Expression >\ \ In\ \ < Expression >$

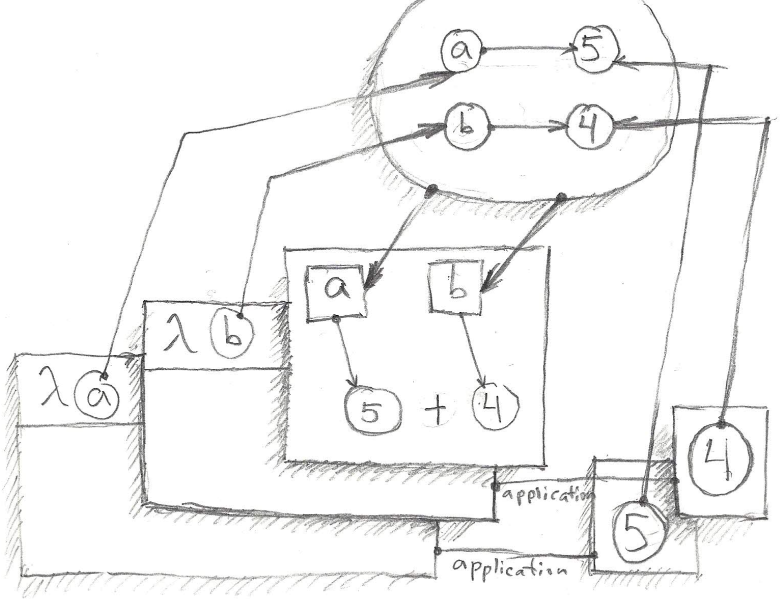
Lambda Calculus

3 kinds of expressions — examples

- $\boxed{\boxed{\text{Variable}}\ \boxed{\text{Expression}}}$ Lambda Abstraction — $(\lambda a.\lambda b.\, a)$

- $\boxed{\text{Variable}}$ Variable — (z)

- $\boxed{\boxed{\text{Expression}}\ \boxed{\text{Expression}}}$ Application — $(\lambda a.\lambda b.a)\ (\lambda s.\lambda z.\, s\ \ z)$



$((\lambda a.\lambda b.\, a + b)\ \ 5)\ \ 4$

Note: the $\oplus$ is not Pure L.C.

By analogy, the relationship between Combinational Logic and Sequential Logic (where Sequential Logic is Combinational Logic with the addition of time and memory) suggests the relationship between Lambda Calculus and it's Operational Semantics. If a Lambda Expression is represented as a tree, then time correlates to the traversal of the tree, and memory correlates to the Set of $< Var, Expression >$ bindings, or the Scope. The nodes of the expression tree are Lambda Expressions. A Lambda node is a Scope (also known as an Environment) and a Body. The body of the Lambda is control. An Application node is the injection of an argument (Expression) into the Scope of the Lambda. A Variable node accesses a specific binding in the Scope. Recursion is possible with a Lambda existing as a bound element within it's own Scope (which is in itself a recursive structure: a Scope within a Scope within a Scope...). We introduce Operations to abstract away the details of base values and their operators. They can be thought of as black box input/output where the Lambda Calculus feeds in arguments and fields results.

A fundamental principle of computability lies in crafting structures with maximum reusability. The more generally applicable some structure is, the more ultimately useful it is. Lambda Calculus gives it's Expressions the ability to have a high degree of general applicability. As the Scope of some Lambda can potentially be injected with many different entities, and as the Lambda itself can exist in any number of Scopes of other Lambdas, it embodies genericity within and without. The current Scope is the set of all bindings that the control has access to right now. At any point in time, there is a current Scope and Body of control. At the beginning of time the Scope is empty and the body of control is the program (Expression) that awaits evaluation. As the Scope begins empty, the program cannot begin with an access (Variable). And since, under certain operational semantics, a Lambda is considered to be evaluated (weak normal form, and weak head normal form [2]), Control must begin with either an Application or a Let. The Let in this case provides a special functionality that allows for recursion and cyclic data structures (such as a graph where a node has a name and an adjacency list). Excepting the abilities of recursion and cyclic data, a let expression can be represented with plain Lambda Calculus. For example: Let one = $(\lambda f.\lambda x.fx)$ in $< ... >$ can be translated to: $(\lambda one. < ... >)$ $(\lambda f.\lambda x.fx)$. For this to work in the recursive case, would require something like a potentially unlimited number of self applications. For example:

```
fac = (\f.\n . if (n==1) then (1) else (n*(f (n-1)));
main = fac fac 1; -- returns 1
main = fac (fac fac) 2; -- returns 2
main = fac (fac (fac fac)) 3; -- returns 6  etc...
```

The idea is that since the scoping is done statically, a binding cannot be referenced before it has been made, thus,

```
fac a c = if (c==1) then (a) else (fac (a*c) (c-1));
```

is not possible with the above style of lettting because the definition of `verb`fac will not yet be bound to the variable `fac` at the time of the call to access `fac` from the environment in the body of `fac`. Thus an architecturally based Let is usefull in providing the ability to let definitions into the environment that reference themselves.

The Parts of PCONS

After implementing a version of the SECD from detailed specs found online, a more modern variant of the SECD was discovered, the SEC machince (the SEC machine using only 3 stacks instead of four and using a single list as the Environment where the SECD uses a list of lists) which was found to be more aesthetically pleasing. Unfortunately, the specs on the SEC did not implement conditionals, operators/base data types, environment indexing, or recursion. With the exception of environment indexing and recursion, these were straight forward to implement and both of these were easy after understanding the underlying mechanism though this understanding took great effort to attain. Thus PCONS was born as an attempt to compile a simple functional grammar to an SEC abstract machine. The modules of PCONS are as follows:

The Parser

A parser which takes advantage of the monadic library Parsec [3] to accept inputs of PCONS and output an initial Abstract Syntax. Creating a parser for a functional language requires a grammar of enormous flexibility. This flexibility can be seen in the production rule for *Expression*. Almost everything, besides environmental bindings (*Alias*) is considered to be an *Expression*. It is crucial that the Abstract Syntax Tree allows this to enable us to compose, curry, and pass functions. Functional programming relies on being able to treat functions as data. The PCONS parser compensates for this by treating all things as an

Expression.
PCONS parses function definitions solely in what is known to the Haskell community as "Expression Style" syntax.
Here is the concrete Syntax used to describe PCONS:
$Program \;\rightarrow\; \{Alias\}$
$Alias \;\rightarrow AliasCenter\ ;\ |\ Letrec\ AliasCenter\ ;$
$AliasCenter \;\rightarrow Name\ \{Pattern\}\ =\ Expression$
$Pattern \;\rightarrow Variable\ |\ (\ Variable\ :\ Variable\ )\ |\ (Variable\ ,\ Variable\ )$

$Expression \;\rightarrow \backslash Variable\ .\ Expression\ |\ Expression\ Expression\ |\ Variable\ |$
$(Expression)\ |\ Expression\ Operator\ Expression\ |\ Case\ |\ Conditional\ |\ Let\ |\ Value$
$Case \;\rightarrow case\ (\ Expression\ )\ of\ \{(\ Pattern\ '\rightarrow'\ (\ Expression\ )\ )\}$
$Conditional \;\rightarrow\; if\ (\ Expression\ )\ then\ (\ Expression\ )\ else\ (\ Expression\ )$
$Let \;\rightarrow let\ Alias\ in\ Expression$
$Variable \;\rightarrow Letter\ \{Digit\ |\ Letter\}$
$Operator \;\rightarrow +\ |\ -\ |\ /\ |\ *\ |\ \tilde{}\ |\ \hat{}\ |\ :\ |\ <>$
$Value \;\rightarrow Int\ |\ Float\ |\ Char\ |\ List\ |\ Pair$
$List \;\rightarrow [\ \{\ Value\ ,\ \}\ Value\ ]\ |\ []$
$Pair \;\rightarrow (\ Value\ ,\ Value\ )$

The Desugarer

A desugarer which analyzes the intial abstract syntax for patterns used either during function definition, or functional case statements. It reduces the patterns into a hidden let statement depending on the pattern at hand. A (x:xs) pattern will reduce to the syntactic equivalent of "let x = car lst in let xs = cdr lst in ..." where "lst" is the list expect to be passed to either the case expression or function and "..." is the remainder of the expression being desugared.

The Translator

A translator that converts desugared Abstract Syntax into a secondary Abstract Syntax that is more readily flattened into SEC instruction sequences.

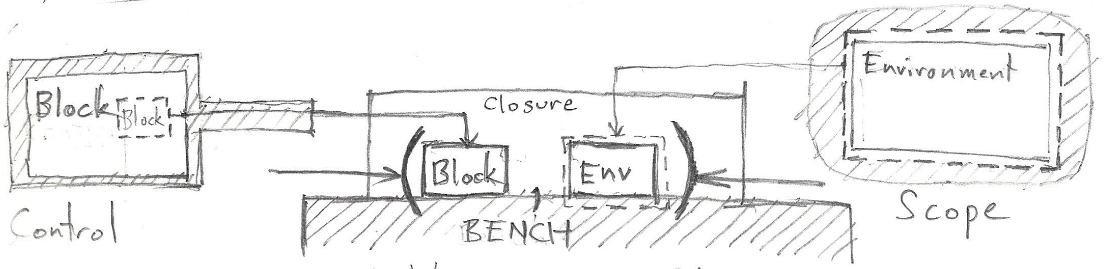
Code Generation

A code generator that produces the executable instruction sequences from the secondary Abstract Syntax.

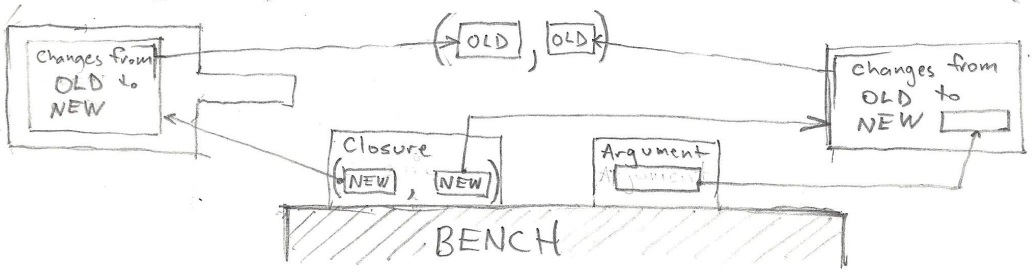
The SEC

The SEC abstract machine which has a three stack architecture that performs call by value style reductions. The SEC has an environment which is accessed using DeBruijn indecies, a control stack where the current control code is pushed, and a call stack where closures and arguments are staged as well as where return frames are stored. The DeBruijn indexing is a remarkable system with it's ostensible simplicity and clean handling of the most intricate, complicated scoping configurations. The DeBruijn indecies also do away with the superfluous naming of a Lambda Abstraction's argument. The control stack gets blocks of instruction sequences pushed to it and one by one they are popped off and executed telling the SEC how to transform the current state (of the three stacks) to the next state. The SEC implements a call by value reduction semantic. The basic intuition, following the above intuition of Lambda Calculus is as follows. A program moves incrementally through time. The state of now becomes the state of next. Now is always in a closure. A closure consists of an environment that is in scope, and a sequence of actions (the control stack) that are fed one at a time into the now dictating what happens next. The actions that can take place in the now are: form a new closure, enter a closure, access the environment, or return to the previous closure. To form a new closure, a block of code is popped from the control stack and pushed to the call stack. Then it is paired with a duplicate of the current environment. To enter a closure it must be on the control stack. First, the existing environment and control stack are stowed for return, then an argument is injected into the staged closure's environment, and the closure's block of code and environment are pushed to the environment stack and control stack. The environment is accessed by way of the DeBruijn indexing mentioned above that is determined statically at compile time. The environment can contain values or closures. When returning to a previous closure, the current environment and control are replaced with the stowed environment and control on the call stack. The result of the closure that is being returned from will be either a value or another closure. This result is left on the call stack for use by the continuation. The now closure is gone forever but the value of what took place there is remembered.

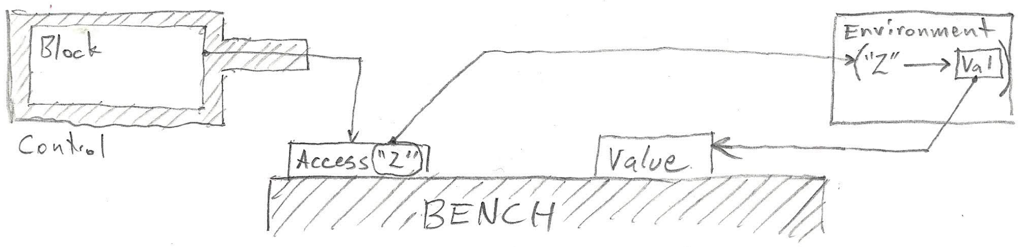
Actions from within a Closure



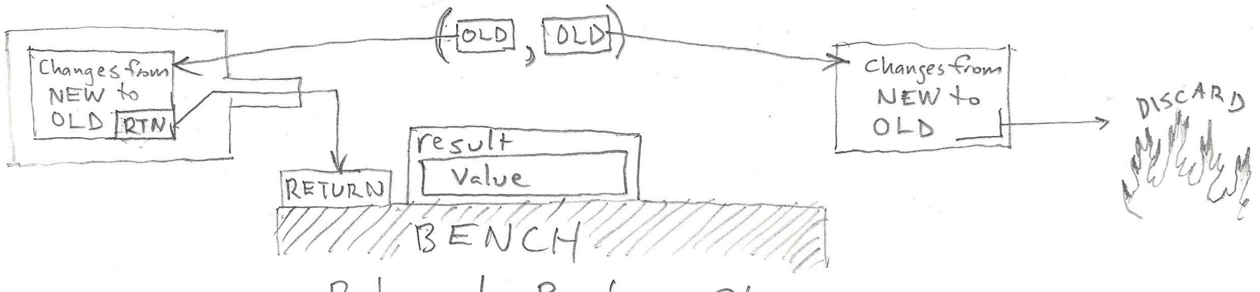
Making a new Closure



Entering a Closure



Access Environment



Return to Previous Closure

Further Work

Further exploration would involve implementing a lazy reduction abstract machine which would be reducing to weak head normal form with graph reduction. Also it would be nice to implement algebraic data types with user definable types as well. Another thing that was not accomplished with PCONS was to implement type checking. With algebraic types, the Hindley Milner type algorithm could be added into the pipeline.

References

[1] P. J. Landin, *The mechanical evaluation of expressions*. The Computer Journal 1964. http://comjnl.oxfordjournals.org/content/6/4/308.full.pdf

[2] Peter Sestoft, *Demonstrating Lambda Calculus Reduction*. Royal Veterinary and Agricultural University, Denmark 2002. http://www.itu.dk/people/sestoft/papers/sestoft-lamreduce.pdf

[3] Daan Leijen, *Parsec, a fast combinator parser*. University of Utrecht, The Netherlands 2001. http://legacy.cs.uu.nl/daan/download/parsec/parsec.pdf