Project Summary

Intellectual Merit

Broader Impacts

Project Description

Motivating questions

What are the structural properties at the level of networks of neurons, modules of networks of neurons, and perhaps higher order forms of organization necessary to support the capacity for abstraction, which is fundamental to computation and may likewise be fundamental to cognitive architecture, development, and function [1]? Given that boolean logic devices are capable of being implemented in vitro using neurons [2], is it possible to biologically engineer analogous devices capable of performing computations that have natural embeddings within first or higher order logic? Moreover, can we define environmental conditions that lead to the development of an extensionally equivalent machine, but whose architecture may differ from the human-engineered biological implementation? Would multiple developmental implementations conserve identifiable structural features? And, how do the structural properties of networks with the capacity for zeroth, first, or higher order computations compare?

Physical implementations of abstractly characterized computation

Since the development of the electronic digital computer, which makes use of the digital abstraction [3] from analog electrical circuits there has been a close heuristic association between boolean logic and computation. However, developments in formal logic over the past century have been largely motivated by its applications to computation and the theory of programming languages that sometimes build upon and sometimes go beyond boolean logic to support additional forms of abstraction. In fact, the capacity to support abstraction mirroring various systems of formal logic has become a means to order and thereby assign value to programming languages [4]. Electrictronic devices that are capable of supporting such forms of abstraction provide a concrete physical instantiation of the ideas inherent to the logical systems they are designed to faithfully implement.

Neuronal logic devices (NLDs) represent an alternative physical modality to digital electronics for the purpose of performing computation. However, rather than attempting to directly parallel the history of the development of digital electronics via the digital abstraction, high-level descriptions of computation, such as the λ-calculus, serve as a specification of computation that is agnostic to the physical modality of implementation. If the specification that a neuronal computation device should be capable of implementing the λ-calculus, a natural first step toward this broad goal is to investigate simple neuronal systems that are capable of performing well-defined computations that require a capacity for first- or higher-order logic.

A higher-order function to be implemented as a higher-order NLD

The λ-calculus notation is helpful in order to define any higher order function. Some of the notation may be implicitly familiar to users of imperative programming languages under the heading "anonymous functions". We provide only an extremely informal set of examples necessary to explain our intended work; however, complete details can be found in [5]. We can define a standard binary boolean function like the "and" function as

$$\lambda x.\lambda y.(\wedge\ x\ y) \tag{1}$$

Such a lambda expression can apparently be applied to any inputs; however the inputs to such a function are not necessarily restricted to booleans unless we infer that the standard logical operator "$\wedge$" only accepts boolean arguments. Note that we have used the prefix or Polish notation for the $\wedge$ operator, which in the perhaps more common infix notation is written with its arguments flanking the operator as $x \wedge y$ to mean "x and y". We can provide explicit type annotations for the bound variables x and y, which indicate the types of the arguments

$$\lambda x : \mathit{bool}.\lambda y : \mathit{bool}.(\wedge\ x\ y) \tag{2}$$

We can now also provide a type annotation for this expression as a whole

$$[\lambda x : \mathit{bool}.\lambda y : \mathit{bool}.(\wedge\ x\ y)] : [\mathit{bool} \to \mathit{bool} \to \mathit{bool}] \tag{3}$$

Depending upon conventions with respect to currying, one could read the type annotation as "the function that takes two arguments each of type *bool* and returns a values of type *bool*" or "the function that takes an argument of type *bool* and returns a function that takes an argument of type *bool* and returns a value of type *bool*". The first formulation may be easier to read, but the second is standard as a result of the way in which functional programming languages implement such functions.

Now we can imagine that if we wish to abstract from the particular binary boolean function implied by the $\wedge$ operator, we need to introduce a functional variable, whose type will be explicitly denoted for concreteness despite the fact that it could be inferred, for which this operator can be substituted. Doing this results in the following second order function

$$\begin{aligned}&[\lambda f : (\mathit{bool} \to \mathit{bool} \to \mathit{bool}).\lambda x : \mathit{bool}.\lambda y : \mathit{bool}.(f\ x\ y)]\\ &\qquad\qquad : [(\mathit{bool} \to \mathit{bool} \to \mathit{bool}) \to \mathit{bool} \to \mathit{bool} \to \mathit{bool}]\end{aligned} \tag{4}$$

This function is intended to be read, in the less verbose uncurried form, as "the function that takes as its first argument (a function that takes two boolean values as arguments and returns a boolean value) and as its second and third arguments two boolean values and returns a boolean value". It is perhaps remarkable that this simple abstraction is now capable of implementing any of the 16 possible boolean functions, provided that the proper binary boolean operator is submitted as the first argument to this function. Furthermore, the statement of this function in terms of a simply typed lambda calculus notation is agnostic to any physical implementation capable of realizing extensionally equivalent behavior.

To make this more concrete, we can very simply implement the above function in any functional programming language, or any programming language supporting the passing of function handles to other functions. An implementation in the programming language OCaml appears as follows

```
let hobf = fun (bf : ('a 'a bool)) (i1 : bool) (i2 : bool)
          -> bf i1 i2
```

Listing 1: a higher order boolean function

What is required in order to evaluate this function are corresponding implementations of boolean functions to be substituted either for f in the lambda calculus notation or for bf in terms of the corresponding OCaml implementation. For example we can implement the XOR function using pattern matching to define a truth table.

```
let xor p1 p2 = match (p1, p2)
with (false, false) -> false
   | (false,  true) -> true
   | ( true, false) -> true
   | ( true,  true) -> false
```

Listing 2: implementation of an XOR boolean operator

Other binary boolean functions are implemented in an analogous fashion. In order to evaluate hobf we then simply provide the name of a binary boolean function and two boolean values as

```
>hobf xor 0 0 = 0
>hobf xor 0 1 = 1
>hobf xor 1 0 = 1
>hobf xor 1 1 = 0
```

Listing 3: Example output of hobf

Assessment of abstraction potential in NLDs

An important consideration is to state precisely some criterion for determining that a particular NLD has achieved potential for an explicit form of abstraction such as is indicated in the relationship between expressions 3 and 4. In abstracting the binary boolean operator $\wedge$ to the functional variable f, which is the fundamental transformation enabling the derivation of 4 from 3, we imply that any physical implementation must take at least three rather than two inputs and the first of these must specify a particular binary boolean operator to apply to the latter two boolean input values. This roughly means that any system that at least partially implements the lambda expression or function specified in equation 4 and listing 1 respectively , must be capable of interpreting the concept of "selection from a set" whose size is determined by the subset of the 16 binary boolean operators that is already implemented in a lower-level form. Indeed another representation of equation 4 as a partial or total set function could be written as $hobf : Hex \times Bool \times Bool \rightarrow Bool$ where we interpret $Bool$ and Hex as two and sixteen element sets respectively. This point of view makes clear that the capacity for selection from two element sets is already apparent in the binary boolean operator written as a set function $\wedge : Bool \rightarrow Bool$. The difference between these is that the system must implement the typing constraints necessary to distinguish a set that takes on sixteen possible values from one that takes on two. For example, in the application of the function $hobf$ the following evaluate as expected given the definition: $hobf \wedge\ 1\ 0 = 0$ or $hobf \wedge\ 1\ 1 = 1$. However, what is to be expected given inputs such as: $hobf 1\ \wedge\ 0$ or $hobf 1\ 1\ \wedge$? In these cases an output type intuitively associated to $error$ is required to indicate that the realization of a system implementing equation 4 has indeed correctly implemented the necessary typing constraints to claim that the function has been realized. In the case of neuronal logic devices, this alternative output should differ in a measurable way from those associated to 1 and 0.

Utilizing synaptic plasticity to generate higher order NLDs

In order to take advantage of the immense capacity for computation available in biological systems and neural networks, the biological complexity of each neuron and synaptic connection must be taken into consideration. Namely, the presence of paired pulse depression and facilitation, which has been previously demonstrated in cultured hippocampal neurons [6], could be exploited to create neural objects with similar design, but more complex behavior than existing NLDs. Specifically, if paired pulse coordination can be shown to exist, especially in the forward direction more significantly than the reverse, then perhaps chains of action potentials could be used as the logical readouts instead of single action potentials. If this is the case, then something approaching a hexadecimal computing system can be created using these logical elements, which would allow for significant information compression. In the long term, something approximating the design of a "neural VLSI" supporting a full logic system such as the simply typed lambda calculus could be implemented.

From NLDs to principles of cognition

From the point of view that identifies the capacity for abstraction with computation, the investigation of neuronal logic devices capable of such function provides a framework in which various hypotheses from cognitive science could begin to be evaluated at the level of well-defined neural circuits. By attempting to isolate minimal implementation criteria, this approach may serve to complement, enable simpler explanations of, or identify paradoxical results derived from studies that treat whole brains and their associated sensory apparatus as their object of study [7, 8].

Broader Impacts

Results From Prior NSF Support

References Cited

[1] J. B. Tenenbaum, C. Kemp, T. L. Griffiths, and N. D. Goodman. How to Grow a Mind: Statistics, Structure, and Abstraction. *Science*, 331(6022):1279–1285, March 2011.

[2] Ofer Feinerman, Assaf Rotem, and Elisha Moses. Reliable neuronal logic devices from patterned hippocampal cultures. *Nature Physics*, 4(12):967–973, October 2008.

[3] Stephen A. Ward and Robert H. Halstead. *Computation Structures*. The MIT Press, 1989.

[4] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs - 2nd Edition (MIT Electrical Engineering and Computer Science)*. The MIT Press, 1996.

[5] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North Holland, 1985.

[6] D D Cummings, K S Wilcox, and M a Dichter. Calcium-dependent paired-pulse facilitation of miniature EPSC frequency accompanies depression of EPSCs at hippocampal synapses in culture. *The Journal of neuroscience*, 16(17):5312–23, September 1996.

[7] James L McClelland, Matthew M Botvinick, David C Noelle, David C Plaut, Timothy T Rogers, Mark S Seidenberg, and Linda B Smith. Letting structure emerge: connectionist and dynamical systems approaches to cognition. *Trends in cognitive sciences*, 14(8):348–56, August 2010.

[8] Thomas L Griffiths, Nick Chater, Charles Kemp, Amy Perfors, and Joshua B Tenenbaum. Probabilistic models of cognition: exploring representations and inductive biases. *Trends in cognitive sciences*, 14(8):357–64, August 2010.

Biographical Sketch: Your Name