RE-ENTERING THOUGHTS

William Bricken
February 1991

Re algebraic networks

(a (b)) is an expression. LoF implicitly treats it algebraically,

(a (b)) = ( )

Dataflow evaluation lets us determine the truth of the equation only by providing bindings to the variables. But the equation

(a (b)) = (a (b (a (b))))

can be evaluated without recourse to variable bindings. A dataflow approach must explore the space of bindings to see if all possibilities result in the same truth value for the equation. An algebraic approach is stronger.

Basically the algebraic technique is substitution and replacement of patterns which belong to the same equivalence class. EXTRACT establishes the equivalence class which permits the example equation to be reduced without evaluation of variables.

Re inversion

Computation should be bidirectional. But the central issue is that networks provide parallelism, so it local computation in general, of which bottom-up (dataflow) and top-down (goal seeking) are just two types of locality. There's also middle out, like in the above example.

I'm not sure what is meant by "Data values should flow both ways", values are not known from the "top" of a network. There are constraints that can propagate down. I like to think of variables as set objects. In a logical example, the variable "a" is a set object, {t, f}. Evaluation is constrained by the domain of the variable. Constraints then merely eliminate possible values (conditionally).

Bidirectionally is intimately part of imaginaries. We never got to that in the network implementation, but we did demonstrate the use of imaginaries in parens models.

Re recursion

I'm thinking of ways to remove recursion/iteration altogether. This is more of a wish than a solid idea. We do know that the repetition can be switched from function to data, as in

```
Fac(n) = n*Fac(n-1)     vs     Fac(n) = (apply times (1,2,...,n))
```

In boundary numbers, Fac(n) is achieved by parallel switching of n pointers, so there's no linearity in the times operation. We still have to list out the numbers 1..n.

So here are three models.

1. Get a number, operate and accumulate, calculate next number, repeat.

2. Generate (1..n), move the operator over it while accumulating.

3. Generate (1..n), switch 2n pointers.

Re network equality

Determining equivalence of networks is NP. The simplest hard case is

```
((a b)(a c)) = a ((b)(c))
```

Different networks, hard to prove equality without transformation. Network equality with imaginaries is harder still, cause of weak permitted transformations.

Re algebra

In playing with Boolean minimization, I found that transformation axioms must necessarily incorporate non-intuitive directions of expansion (the source of NP). It's possible, for instance, to construct complex expressions for which the wrong guess about which variable to evaluate will lead to huge inefficiency. So my proposition is that algebraic transformation can be more efficient than evaluation, particularly since it occurs on the structure of the code/database. The non-evaluated problem space is transformed, permanently changing the control structure of a network, independent of the variables.

So although theorems are not fundamental for expressability, I think that a well selected group is fundamental for efficiency of computation. In very large spaces, they make the difference between termination (in our lifetime) or not.

The situation is analogous to using search heuristics rather than brute force, to using a well-tailored representation rather than a random one.

Re bidirectionality

I agree that computation is propagation of change. Perhaps an issue is change where/what? As above, there is change in state and change in structure.

I've falling into the habit of thinking about variables as constrained domains. First, note that in numerical algebra, variables are used in different manners:

$x = 3$	as a label
$x = y+3$	as a constraint
$x+y = y+x$	as a pattern

Now, A+B=C can be read that A, B, C are unknown instances from a domain, or they can be read as sets on that domain. For eg, in a binary domain:

$$\{0,1\} + \{0,1\} = \{0,1\}$$

Here the equation excludes both A=1 and B=1.

Dialog

> If A+B=C as a constraint, then if A changes and B is constant, C must
> change. Similarly, if C changes, A and/or B must change. Since A+B and C
> are the same quantity, there is only one of it, and it can have only one
> value.

I'm proposing a different interpretation, where C changing impacts the choices of A and B, not their values. From this possibility calculus approach, structure is different, although value is maintained. The equality symbol establishes an equivalence class of values, but not of form. In purely algebraic approaches, theorems provide mechanisms for argument about structure independent of value.

From this angle, imaginary values come from constraint sets that eliminate all unique choices.

> I'm led to the view that specifying a computation is nothing more or less
> than saying what depends on what. The dependencies are everything.
> Computing is then just the propagation of change in an orderly fashion.

I agree. In a Losp network, dependencies are represented by the connectivity of the network. Change is change in network structure, not in values of variables. Orderly fashion is asynchronous and local. So propagation is broader that linear pulses.

In Losp, algebraic simplification occurs without variable binding, just as in matrix techniques, you can determine overspecifications without solving the equations.

So I think there is at least one other element (type of propagated change): dataflow (binding propagation), lazy (possibility propagation), and structural (algebraic propagation).