WORKING NOTE AC50

Alan Creak
12 May 1986

ECCLES

WHAT ABOUT ECCLES ?

This note records some of the things I remember about my various tries at implementing an Environment for Computing with Complicated Logical Expressions Simply. Some of it is straight memory; some is supported by notes and/or records of experiments; some may be pure invention. But at least I'll get it written down, so it can't degrade any further.

HISTORY.

Eccles started when I was at Derby, in about 1970. I was looking for a way of handling the logical part of my project Peter (Programme for Evaluating Theoretical and Experimental Results), and something based on zero-order logic seemed to be the right thing. (Though at the time I didn't know it was zero-order logic – but I went to a really excellent introductory course on Boolean Algebra, given by a Professor Goodstein of Leicester University (1); there I found out about sentence logic, which obviously fitted my purposes very well.) As the computer at my disposal was – by present standards – very small (an IBM 1130 with 8K of 16-bit words of memory), I was very concerned to find a way of conducting arguments which would require as little as possible in the way of storage and time; and the possibility of packing assertions 16 to a word, and thereby getting 16-fold parallelism in the argument evaluation, seemed to kill two birds with one stone. (It still seems like a good idea; however big your computer, there are always going to be some problems which won't fit into it – so economical techniques will always have some value.)

When I came to Auckland, I started again. The original Eccles was in Fortran and IBM1130 assembly language, and didn't all work too well; I wanted to start again in Algol, and to take advantage of the even bigger 48-bit word of the Burroughs B6700. I'd also developed quite a few ideas about how the superstructure should be, though I hadn't been able to put those onto the IBM1130.

I managed to get quite a bit of Eccles going. It didn't all work; I suspect that was mainly because I couldn't resist the urge to "optimise" my logic routines by using clever tricks, so they became enormously complex and faulty. Then Paul Lyons came along as a Ph.D. student, and developed the system further in his own way, finding quite a few mistakes in the process. Paul did manage to get it going with a reasonable "front end", which could accept logical statements in a reasonably comprehensible form, parse them, produce equivalent Eccles statements, and evaluate the whole system. Then he went away, leaving no trace of his work behind.

I had always intended to carry on with Eccles, but didn't have time for some years. Then they got rid of the B6700, and all the work instantly became useless. And that's where we are now.

THE ESSENTIAL IDEAS.

Eccles is intended as a way of evaluating logical expressions (see the name). An expression is a formula , containing symbols representing atomic statements (`a, b, c, …`) connected by logical operators (`&, v, ~, =>, <=>`). Parentheses may be used to control the order of evaluation. Some examples of expressions are :

```
a & b
a & b v c => d <=> e
( ( a & b ) v c ) => ~ ( d <=> e )
```

and so on. The second expression demands that some convention regarding operator precedence be established; I assumed that `~` was executed as a unary operator at the highest precedence, and that `&` and `v` had highest, equal, precedence among the binary operators, followed by `=>`, followed by `<=>`.

Expressions in Eccles have NAMES: the data for Eccles consist, apart from possible housekeeping instructions, of a set of expressions with names. The names are kept in a name table, and each is linked to the representation of the corresponding statement body. The name table also contains

names of elementary statements, which, of course, have no corresponding statements body. For many purposes the names of elementary statements and of expressions can be used interchangeably – they both, after all, denote a statement of some sort. There are some complications with names if you want to leave them unevaluated in expressions, and I never really worked these out.

Eccles combines logical statements to produce new statements. For example, given the initial statements `X` and `Y`, denoting, respectively, `a & b` and `c & d`, Eccles will evaluate `( X => Y )` as a new expression in disjunctive form :

```
( ~a ) v ( ~b ) v ( c & d )
```

Statements in Eccles are represented bitwise within machine words; each bit position in a word corresponds to a single atomic statement. A pair of words (strictly, bit arrays, which may be multiple words, depending on how many atomic statements are to be represented) is used to represent a conjunction of atomic statements : so, assuming that the bits from left to right of the word are used to represent `a, b, c`, …, the simple conjunctive expression `a & ~b & d` would be represented by the pair of words

```
@..@ …
.@.. …
```

Notice that the logical value associated with an atomic statement is defined by the combination of corresponding bits in the two words. We may call a simple expression of this sort a *clause*; then any logical statement can be represented as the disjoined assertion of a number of clauses. We can read such an expression as " clause 1 *or* clause 2 *or* clause 3 ……", which underlines its useful interpretation as a list of possible configurations of the logical universe defined by the initial statements.

Two examples of applications of Eccles within the Peter project follow. Both of these really do work, and were running (before 1973 !) on the IBM1130 version of Eccles.

Chemical reaction mechanisms.

One of the aims in the field of gas phase chemistry is to elucidate the "mechanisms" of chemical reactions which occur in gas mixtures. Typically, one knows the sorts of reaction which can occur in the system under investigation, and the task is to select a plausible set of actual reactions (the "mechanism") which will account for the observed reactants and products. Eccles is useful in this context because the arguments to be carried out can be simply expressed in logical form. Thus, if we have the chemical reaction :

R1 : A + B –> C

then we can formulate these logical statements:

- If reaction R1 happens, then all the chemicals involved must be present in the system :

  ```
  R1 —> A & B & C
  ```

- If compounds A or B are known to be consumed, then reaction R1 (or some other reactions which consume these substances) must occur :

  ```
  Aconsumed —> R1 v ……
  ```

- If C is found among the products, but was not initially present, then R1, or some other reaction which produces C, must have occurred :

  ```
  C —> Cinitial v R1 v ……
  ```

And so on. By asserting these logical statements, and providing the known facts about the substances which were observed to be produced and consumed, we can use Eccles to derive a final expression in which each clause describes a possible mechanism for the reaction.

Double and single bonds in molecules.

A molecule in which the geometry leaves atoms with formally unsatisfied bonds is often best described in terms of structures involving multiple bonds; carbon atoms take part in many such bonds. If a carbon atom forms fewer than four bonds to other atoms, it is called *unsaturated*. If two such unsaturated carbon atoms are adjacent, their "spare" bonds may combine together, producing a double bond between the two atoms; if several such atoms are close together in a molecule, then there may be several possible ways of drawing the double and single bonds, leading to the idea of resonance and resonance stabilisation. While it is possible to treat such phenomena by quantum mechanical methods (albeit very approximately), practising chemists manage very well by simply drawing the bonds; and they have managed in this way since long before the quantum mechanical interpretations became available. This empirical approach is therefore of great interest in a study of chemical reasoning, and it offers a satisfactory way of determining how multiple bonds are configured given only the topological properties of a molecule, such as may be derived from certain line-formula representations. Most such notations designed for use by people provide ways of representing the order of each bond; but, using the method about to be described, such an explicit representation is not necessary, and is in any case not necessarily reliable. The same is true, with even more force, for formulæ produced by computer calculation, and this approach was developed in order to discover the properties of molecules represented by such formulæ.

For each unsaturated atom in the molecule, one can construct a statement which makes certain assertions about the bonds made by the atom. For example, if the atom forms three bonds, say b1, b2, and b3, with other unsaturated atoms then we can construct the statement :

```
( b1 & ~b2 & ~b3 ) v ( ~b1 & b2 & ~b3 ) v ( ~b1 & ~b2 & b3 )
```

where we write `b1` for the statement "b1 is a double bond", and so on. Combining all such assertions, we arrive at a final expression which is a disjunction of the possible configurations of double and single bonds in the molecule : there may be none of these, which suggests some oddity about the molecule (it may be a free radical); or there may be exactly one, in which case there is a single possible configuration; or there may be several, in which case the molecule is resonance-stabilised, and the results can be used to make guesses about the orders of the various bonds involved.

REFERENCE.

(1) R.L. Goodstein : *Boolean Algebra* (Pergamon, 1963).