# Doing arithmetic with diagrams

# DOING ARITHMETIC WITH DIAGRAMS

by

Alan Bundy

Memo No 61

# A B S T R A C T

A theorem prover for a part of arithmetic is described which proves theorems by representing them in the form of a diagram or network. The nodes of this network are 'typical integers', i.e. objects which have all the properties that integers have without being any particular integer. The links in the network are relationships between 'typical integers'. The routines which draw this diagram make elementary deductions based on their built-in knowledge of the functions and predicates of arithmetic. This theorem prover is intended as a model of human problem solving behaviour.


---


Key Words and Phrases:

Artificial Intelligence, Semantic Network,
Automatic Theorem Proving, Heuristic Method,
Arithmetic, Integer, Logic, Typical Integer.

C O N T E N T S

## 0. MOTIVATION.

The objective of this work is to investigate human problem-solving behaviour by trying to simulate it on a machine. The main source of information about human behaviour is still self-observation and so the author has deliberately chosen a domain in which he has some experience, arithmetic (see [1]). Arithmetic had the additional advantage that as one of the oldest branches of mathematics it is rich in proof techniques and easily stated but difficult theorems (e.g. Fermat's Last Theorem). Also an efficient arithmetic theorem prover is likely to find applications in both Robot Problem Solving and Program Correctness Proofs.

## 1. INTRODUCTION.

This is a report of work in progress. It describes what is, to the best of the author's knowledge, a new kind of theorem prover. This theorem prover does not use axioms or rules of inference to prove theorems. Instead it represents the candidate theorem as a network (or diagram) in which the nodes are the property lists of arithmetic terms and the links describe relationships (e.g. $=$, $\neq$, $\leq$) between them. Statements are asserted by adding new links and proved by accessing the diagram. Knowledge about arithmetic is built into the routines that draw the diagram so that elementary deductions are made (and simple theorems proved) automatically.

The theorems proved are drawn from the Elementary Theory of Natural Numbers. (i.e. the non-negative integers.) At present the theorem prover can only handle universally quantified equations, inequations and inequalities, between terms composed of: numbers, variables, addition, modified difference, successor and predecessor. However this class of formulae is being constantly increased (see Section 6) and already fairly difficult theorems can be proved. (See Section 5.)

Modified difference, written $x \dot{-} y$, is a non-standard function. It is just subtraction, modified so that its result is always a non-negative integer.

$$x \dot{-} y = \begin{cases} x - y & \text{if } y < x \\ 0 & \text{if } x \leq y \end{cases}$$

where $x - y$ is subtraction between integers.

Also $\text{Pre}(x) = x \dot{-} 1$ (predecessor)

and $\text{Suc}(x) = x + 1$ (successor)

## 2. EXAMPLES.

Before describing the representation of the diagram and the routines which draw it, it is best to illustrate how the theorem prover works with some examples.

(i) The Associative Law of Addition

(a) $x + (y + z) = (x + y) + z$

The theorem prover always starts by constructing nodes for 0 and 1 and drawing in the appropriate links (e.g. $0 \neq 1$, $0 \leq 1$, etc.). New nodes are always appropriately related to 0 and 1. However, since these links play no part in the proof which follows they have been omitted for clarity.

The machine proceeds by creating nodes for each of the subterms involved in (a) and drawing in the appropriate links. e.g.

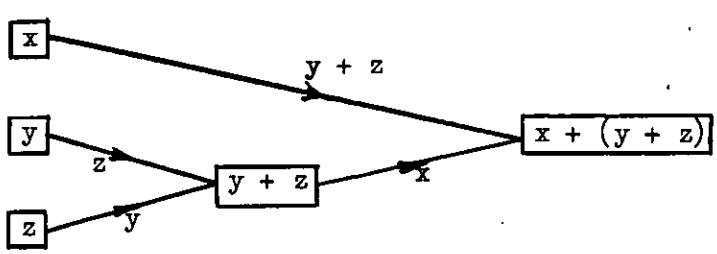



Fig. 1

The links in the above diagram are distance and direction links, e.g. x is a distance y + z less than x + (y + z). When a new link is added to a node it is compared with the old links to see if any conclusions can be drawn.

e.g.

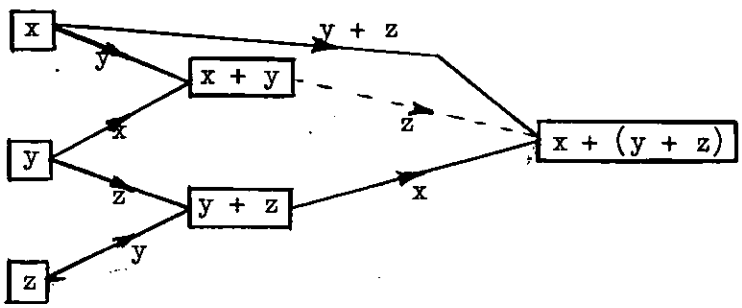



Fig. 2

For instance in Fig. 2 the link from x to x + y is compared with the link from x to x + (y + z). This causes the distance y to be compared to the distance y + z. y is found to be less than y + z by a distance z in the diagram; so the machine deduces that x + y must be less than x + (y + z) by a distance z, and draws in the appropriate link (dotted).

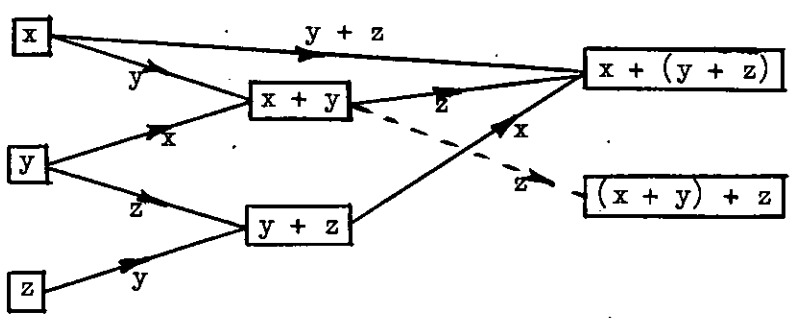



Fig. 3

Finally the node for (x + y) + z is created but as the first link is drawn (dotted in Fig. 3) it is compared with the link connecting x + y with x + (y + z) and found to be of equal length. The machine deduces that (x + y) + z = (x + y) + z and achieves this by merging the two nodes and deleting the new link (Fig. 4).

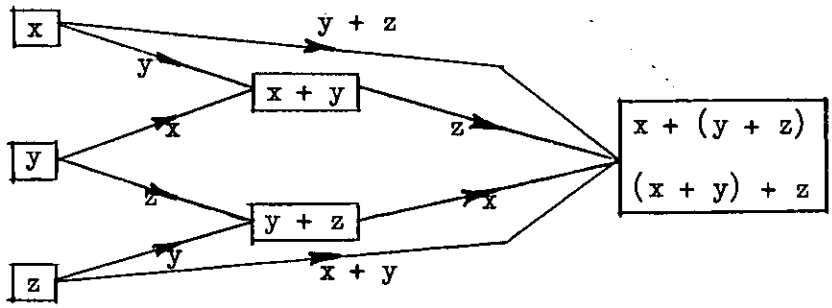



Fig. 4

The machine now proves the theorem by accessing the diagram and discovering that these two terms have the same node.

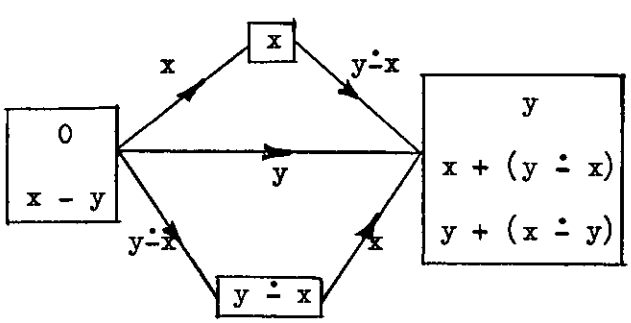
(ii) Commutativity of Maximum.

We can define $\max(x, y)$ as $x + (y \dot{-} x)$ and prove that max commutes by proving

(b) $x + (y \dot{-} x) = y + (x \dot{-} y)$

As before we draw nodes for each of the subterms involved in (b), except that this time we consider two cases. The machine always draws formulae of the form $A \dot{-} B$ by first interrogating the diagram to see if either of the cases $A \leq B$ or $B \leq A$ hold. Otherwise it duplicates the diagram, asserts $A \leq B$ in one and $B < A$ in the other and then tries to prove the theorem in both. So in this example, the final diagrams are:

case $x \leq y$



and

case $y < x$

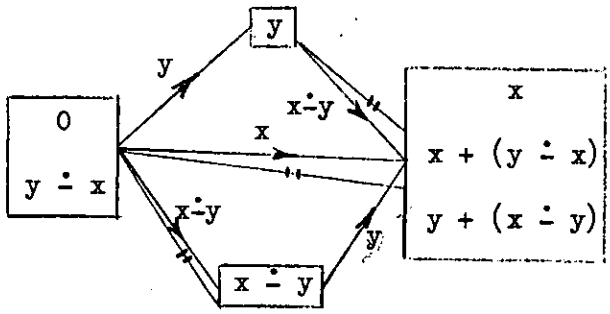



Fig. 5

A link of the form

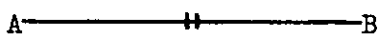



Fig. 6

means that $A \neq B$. These links are not needed in this example but are included for completeness.

## 3. THE REPRESENTATION OF THE DIAGRAM.

The diagram is represented in a straightforward way in order to simplify the code. At the top level there is a list of pairs called the diagram list. Each pair is made from a diagram and a label, which says which case it represents. Case labels are just the formulae which define these cases (e.g. $x \leq y$, $y < x$). Each diagram is either the null list, for a contradictory diagram, or a list of pairs made from terms and their associated property lists. Each property list is a POP-2 record containing five slots in which information relating its terms to the other terms is kept. This information consists of:

(a) The name of the syntactically simplest term with this property list.

(b) A list of terms proved to be unequal to the present one.

(c) A list of pairs, each of which is a term greater than or equal to the present one, followed by the distance between them.

(d) A similar list for terms less than or equal to the present one.

(e) A slot used for markers, when measurements are being made in the diagram.

The normal list representation is used for formulae and terms. e.g. $x \leq x + y$ is represented internally as [LESS X [ADD X Y]].

## 4. ROUTINES THAT 'KNOW'.

In order to distinguish functions in the theorem proving program from functions in arithmetic, we will call the former 'routines'.

There are three classes of top level routines arranged roughly in a hierarchy. They are:-

(a) Drawing Routines: which extract information from the original formulae and use this information to draw the diagram.

(b) Asserting Routines: which make relationships between terms hold in the diagram by adding links.

(c) Interrogating Routines: which discover whether relationships between terms hold by accessing the diagram.

### (i) The Drawing Routines.

There is a drawing routine for each arithmetic function symbol, and this routine deals with any term beginning with this function symbol. It assumes that the term's arguments have already been successfully drawn in the diagram and that the term's property list has been created. If necessary it can use the interrogation routines to discover the state of the diagram, before using its knowledge of arithmetic to assert relationships with the asserting routines.

As an example we describe the drawing routine for terms of the form $A \dot{-} B$ :

If $A \leq B$ (in the diagram) then make $A \dot{-} B$ equal to 0 (in the diagram)

else if $B \leq A$ then make $A \dot{-} B$ less than or equal to A by an amount B.

else

Make another copy of the present diagram.

If 'case' is the label of the old diagram label the old one '$A \leq B$ & case' and the new one '$B < A$ & case'.

In the old diagram

Create a node for $B \dot{-} A$ if necessary.

Make A less than B by amount $B \dot{-} A$.

Make $A \dot{-} B$ equal to 0

In the new diagram

Make $A \dot{-} B$ unequal to 0

Make $A \dot{-} B$ less than A by amount B.

end.

These routines are sufficiently close to the original function definitions to make one feel optimistic about having the machine 'learn' a new function by constructing its drawing routine from its definition.

Apart from the nodes for 0 and 1 and the node for $B \dot{-} A$ mentioned above, nodes are only constructed for terms explicitly mentioned in the candidate theorem. This kind of conservatism is essential to prevent the diagram exploding. The strictly-controlled creation of new nodes, in order to prove more difficult theorems, will eventually be allowed (see Section 6).

(ii) The Asserting Routines.

There is an asserting routine for each predicate symbol and for each negation of a predicate symbol. These routines assert new facts into the property lists of the terms, thus creating new links in the diagram. Before and after these links are created they are compared with existing links using the interrogation routines, to see if any more information can be deduced. In particular a contradiction may be detected in some case and then registered by making the diagram of that case the null list.

It would be misleading to emphasize the precise definition of one of these asserting routines, because they are constantly evolving as new knowledge is added to the theorem prover. We describe only the present state of the routine which makes A less than B by an amount C. The routine simultaneously makes C less than B by an amount A. This involves adding the new pairs (A . C) and (C . A) to the 'less' slot of B and the pairs (B . C) and (B . A) to the 'greater' slots of A and C respectively. The routine that puts the pair (B . C) in the 'greater' slot of A first checks to see if $B \leq A$ or $C = 0$ and in either case makes $A = B$ etc. It then compares (B . C) with the other pairs in the 'less' slot. Let (B' . C') be a typical pair.

If $B = B'$ then make $C = C'$

else if $C = C'$ make $B = B'$

and in either case exit the routine without adding the new pair.

Otherwise add the new pair and compare it with the old pairs again

If $C \leq C'$ then make $B \leq B'$

or if $C' \leq C$ then make $B' \leq B$

or if $C \neq C'$ then make $B' \neq B$

Similarly if $B \leq B'$, $B' \leq B$ or $B \neq B'$.

Now suppose $(B' \cdot C')$ is a typical member of the 'less' slot of A.

If there is a term C'' in the diagram corresponding to the distance C + C' we make B' less than B by an amount C''.

In general we build as many links as possible in the diagram providing this does not involve creating nodes for terms not mentioned in the original formula. This redundancy is convenient and not explosive as only a finite number of links is possible.

The above routine is repeated for the other new pairs. Lastly if $A \neq 0$ then everything greater than B is made unequal to everything less than C. Similarly if $C \neq 0$.

(iii) The Interrogating Routines.

There is an interrogating routine for each predicate symbol and for each negation of a predicate symbol. They are used to settle all questions about the relationships between terms and they do this by accessing the diagram. Because they are used so frequently the present routines do very little searching. For instance the routine which asks whether A is equal to B returns true if A and B have the same property list or, like all these routines, if the diagram is contradictory. The routine which asks whether A is less than B, starts at A and climbs up through the 'greater' slots, marking its passage, until either it comes to B or it exhausts all possibilities. It would be possible to design routines which tried a lot harder than this, but their use would have to be selective.

## 5. RESULTS.

All of the theorems which have been attempted were drawn from [1], which contains about 700 suitable formal theorems. Only 20 of these are composed solely of $+$, $\dot{-}$, Pre, Suc, $=$, $\neq$, $\leq$ and $<$. The theorem prover

can now prove all 20, some of which are quite difficult. For instance, it is most unlikely that number 13 could be proved with a normalization algorithm. Considerable improvements have been wrought to its deductive power by examining its initial failures.

A list of the 20 theorems, which have been proved, follows:-

1. $x + y = y + x$
2. $x + (y + z) = (x + y) + z$
3. $Pre(x \dot{-} y) = Pre(x) \dot{-} y$
4. $Suc(x) \dot{-} Suc(y) = x \dot{-} y$
5. $(x + z) \dot{-} (y + z) = x \dot{-} y$
6. $x \dot{-} (x + y) = 0$
7. $1 \dot{-} Suc(x) = 0$
8. $1 \dot{-} (1 \dot{-} Suc(x)) = 1$
9. $x + (y \dot{-} x) = y + (x \dot{-} y)$
10. $x \dot{-} (x \dot{-} y) = y \dot{-} (y \dot{-} x)$
11. $x \dot{-} (y + z) = (x \dot{-} y) \dot{-} z$
12. $(x \dot{-} y) \dot{-} z = (x \dot{-} z) \dot{-} y$
13. $(x + y) \dot{-} z = (x \dot{-} z) + (y \dot{-} (z \dot{-} x))$
14. $x \dot{-} y \leq x$
15. $x \dot{-} (x \dot{-} y) \leq x$
16. $x \dot{-} (x \dot{-} y) \leq y$
17. $x \leq x + (y \dot{-} x)$
18. $y \leq x + (y \dot{-} x)$
19. $0 < Suc(x)$
20. $x < Suc(x)$

## 6. GOALS.

In order to prove the other 680 theorems in [1], the theorem prover requires the ability to deal with arithmetic and logical symbols outside its present scope. The plans for adding most of these abilities are well

advanced and they only await implementation. A description of these plans, and of the eventual goals, follows.

(i) Logical Connectives.

The ability to handle the truth functional connectives can be added relatively painlessly, provided we continue to tackle only formulae which would contain no existential quantifiers if written in Prenex Normal Form. The theorem prover must be able to assert a hypothesis (using the asserting routines) in a diagram, possibly dividing into several cases, and then prove the conclusion in each of these cases. For instance to prove a formula of the form A → B, assert A and prove B. If A is of the form C ∨ D then asserting A involves making 2 diagrams, in the first we assert C and in the second we assert D. Depending on the formulae C and D we may also assert ~D in the first or ~C in the second, but not both. We then prove B in both diagrams. Negations are passed down to the atomic formulae and the negated predicates are treated as predicates in their own right.

(ii) Further Arithmetic.

The next task is to add the multiplicative functions and relations to the present additive sub-system. There is a close correspondence between the multiplicative and additive sub-systems, and this will be reflected in the implementation. New drawing routines will be written for multiplication, quotient and remainder, and new asserting and interrogating routines for the relation 'x exactly divides y'. The property lists will be given new slots for divisors and their corresponding quotients etc. Theorems involving multiplicative functions alone (e.g. associative law of multiplication) will be proved in an analogous way to the corresponding additive theorems. The interaction between multiplication and

addition is more complicated, but the distributive law, for instance, has now been proved by hand, which bodes well for the future.

The sub-systems of exponentiation and prime numbers create no special difficulties and can be added in a similar way.

(iii) Existential Quantification.

At present the theorem prover can only handle universal formulae (what we used to call 'identities' at school). To handle formulae containing existential quantifiers (e.g. to solve equations) is more difficult. The following is a conjecture at the solution. The theorem prover must recognise two kinds of variables: dummy variables, corresponding to the existentially quantified variables and skolem functions corresponding to the universally quantified variables. The arguments of the skolem functions serve only as markers, to preserve the original order of the quantifiers and to prevent illegal substitutions. We proceed in the ordinary way, except that a successful conclusion requires us to have identified each dummy variable, say x, with some term, which contains no dummy variables except in skolem functions, and no skolem functions containing x. This identification may not happen automatically, so we may have to cast about for suitable terms, or even construct suitable new terms, using construction routines.

(iv) Strategies.

With the additions listed above the theorem prover should be able to prove a wide class of straightforward theorems. However, it always terminates with or without a solution and performs no search. To prove more sophisticated theorems the theorem prover needs to have, and to make use of, a store of proof strategies.

For instance, it may decide to try mathematical induction, to construct some new terms, to use some previously proved theorem, or to set up some intermediate sub-goals. All these abilities are relatively straightforward to apply once a decision has been made on how to use them, but deciding what to use and how is difficult. The intention, therefore, is to implement this stage in two parts. The first task will be to make the theorem prover interactive. Procedures will be written to correspond to the mathematical use of such instructions as: Consider the cases.....; Try induction on.....; First prove.....; Prove by contradiction.....; Assume.....; Similarly prove.....; etc. Then the theorem prover will be led through the proofs of moderately difficult theorems in number theory. Using this experience it is hoped that it will be possible to evolve a language for classifying candidate theorems and selecting and applying suitable proof strategies.

## 7. CONCLUSION.

This theorem prover uses an analogical representation of arithmetic and lets this representation do all the work. In the traditional representations of mathematical theories, for instance a set of axioms in a resolution theorem prover, it is quite easy to assert a contradictory set of formulae without this situation being readily spotted. In the present representation any new assertion causes all sorts of conclusions to be drawn and asserted, and contradictions are usually spotted quickly.

In [2] Gelernter achieved a similar effect by using two representations: a syntactic one for proving theorems with rules and axioms and a diagram to guide this proof. His 'syntax computer' discovered groups of sub-goals which would imply the present goal, and the 'diagram computer' vetted these sub-goals, rejecting those that were false in the diagram. The diagram

was also used to prove certain very basic sub-goals (p.142, No. 7), but this is a risky process because although precautions were taken to prevent spurious coincidences entering into the diagram, there were still statements, true in the diagram, but not provable from the hypothesis of the candidate theorem. The analogue of Gelernter's diagrams in Arithmetic would be to substitute particular numbers for the variables in some sub-goal, and then compute the truth or falsity of the resulting formula.

The present theorem prover arose from an attempt to represent in the machine a 'typical integer', i.e. an object with all the properties of an integer (e.g. being equal to, less than, not equal to, or a divisor of some other integer) but which is not any particular integer (e.g. 3, 16, 13). In a 'diagram' composed of 'typical integers' no spurious coincidences arise so that anything true in the diagram is provable from the hypothesis of the theorem and anything false is not provable. So this diagram can be legimately used for rejecting and proving sub-goals. (Of course something may be neither true nor false in the diagram.)

If it is the author's hope that this method of theorem proving will prove applicable not just to arithmetic, but to all mathematical theories, especially classical systems with a single standard model, like analysis, geometry and set theory. Also he hopes that this representation may make some contribution to the study of human problem solving. In particular, maybe it sheds some light on the curious blackboard diagrams which mathematicians use to help them 'understand' problems.

ACKNOWLEDGEMENTS.

My debt to the M.I.T. Progress Report [3] will be obvious. Not quite as obvious, but equally important, are the conversations with my colleagues: Aaron Sloman, Bob Boyer, J Moore, Mike Liardet and numerous others.

REFERENCES.

[1] Bundy, A. The Metatheory of the Elementary Equation Calculus. Ph.D. Thesis, Leicester University, England, 1968.

[2] Gelernter, H. A Geometry Theorem Proving Machine. Computersand Thought, p. 134-52, McGraw-Hill, 1963.

[3] Minsky, M. and Papert, S. Project M.A.C. Progress Report p. 129-244, M.I.T., 1971.

---