AN APPROACH TO DETERMINACY PROOFS

by

Robert M. Keller

University of Utah
Salt Lake City, UT 84112

UUCS-78-102

March 1978

This work was supported by NSF Grant MCS 77-09369.

AN APPROACH TO DETERMINACY PROOFS

Robert M. Keller
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Abstract

It is known that any parallel program graph composed of continuous operators itself represents a continuous function. In other words, the network is determinate in the sense that for a given input, the output is unique, independent of the timing of the constituent operators. This result is applied to some unusual data types, resulting in a determinacy proof for a parallel version of the alpha-beta minimax procedure. The notion of the *context* of a function is introduced, and its usefulness in constructing such proofs is demonstrated.

Introduction

This paper applies some previously-investigated concepts in a novel way. The concepts are: parallelism, as present in a class of graph models for parallel computation; data types, as defined in [Scott 76] and [Vuillemin 74]; and the alpha-beta minimax concept from game-playing programs (cf. [Nilsson 71], [Winston 77]). The result is that a certain parallel version of the alpha-beta procedure is determinate. The approach used in deriving the result will hopefully serve as a model for other correctness proofs.

The question of proving correctness of parallel programs is of significant concern. There are several appraoches to such correctness proofs, e.g. [Keller 76], [Owicki and Gries 76], [Lamport 77]. It is generally agreed that it is worthwhile to pursue techniques for reducing the number of interactions which need to be considered in constructing a proof for a parallel program. Examples may be found in [Lipton 75], [Kwong 77], [Doeppner and Keller 75], [Francez 76], and undoubtedly others. An extreme case of such techniques is to show that the program in question is *determinate*, i.e. produces the same result even in the presence of concurrency, and then prove correctness of a serialization of the program using sequential program proving techniques.

The aforementioned technique would have wider appeal, were it not for the fact that most programs of interest actually have *indeterminate* sub-programs at some level of their representation. Hence techniques which derive determinacy of a program based on determinacy of its sub-programs are inapplicable in such cases. Typical of this phenomenon are programs which use *cells* which are written into independently by several processes, thus having a non-unique sequence of values stored into them.

The alpha-beta procedure, in its most natural conception, has local indeterminacies as alluded to above. It is this form of indeterminacy which has lead some researchers, notably those in the area of "data flow" computations (cf. [Dennis 74]), to exclude cells from languages. To do so for the program of interest here would mean sacrificing an important technique for gaining efficiency.

We hope to show that such a program can be made determinate under a generalized notion of determinacy. This will done using a reasonably natural, though not immediately-obvious, program representation and an appropriate choice of data type. This then renders our parallel alpha-beta procedure amenable to sequential proof techniques, which will not be formally applied here.

Background

The basic idea of composing determinate programs to get larger determinate programs has been known for some time. Early discussion of this phenomenon appeared in [Patil 70] and [Karp and Miller 66]. The former presentation concerned itself with a "denotational" representation of program semantics, in contrast to the "operational" representation of the latter. That is, the former represented a sub-program as an operator on *sequence domains*. Later, [Kahn 74] observed that the same result could be derived by appealing to the "fixed point" reasoning being developed by Scott and others for operators on "data types" [Scott 76].

Recently, we have been investigating the usefulness of data types other than sequence domains and "flat" domains in representations of parallel programs. For example, "bag" domains were useful in [Keller 77a], and "tree" domains in [Keller 77b] for dealing with various kinds of computation. This approach is extended in the present paper, where another kind of domain will be shown useful, namely ordering the integers *numerically*, rather than giving them the customary flat ordering. It is not immediately obvious that this ordering is useful.

We begin by discussing the alpha-beta procedure and a concurrent version of it. Then, in order to make this paper complete, we state the definitions of "data type" and "continuous function", and present the "denotational determinacy theorem." We then present the data types which will be useful in our program, followed by application of the denotational determinacy theorem to it.

The Alpha Beta Procedure

We wish to devise a parallel program which will represent "minimax tree searching". For completeness, we present a brief sketch of the basic ideas and motivation for this problem. Further treatment can be found in [Nilson 71] and [Winston 77].

Throughout the remainder of this paper, we assume the existence of an underlying finite tree. The leaves of this tree are assigned integer values. The interior nodes are divided into "*max*" nodes and "*min*" nodes. Although not essential, we make the assumption that the root of the tree is a *max* node, its immediate descendants are *min* nodes, their immediate descendants are *max* nodes. etc. An example is shown in Figure 4, where *max* nodes are indicated by squares and *min* nodes by circles.

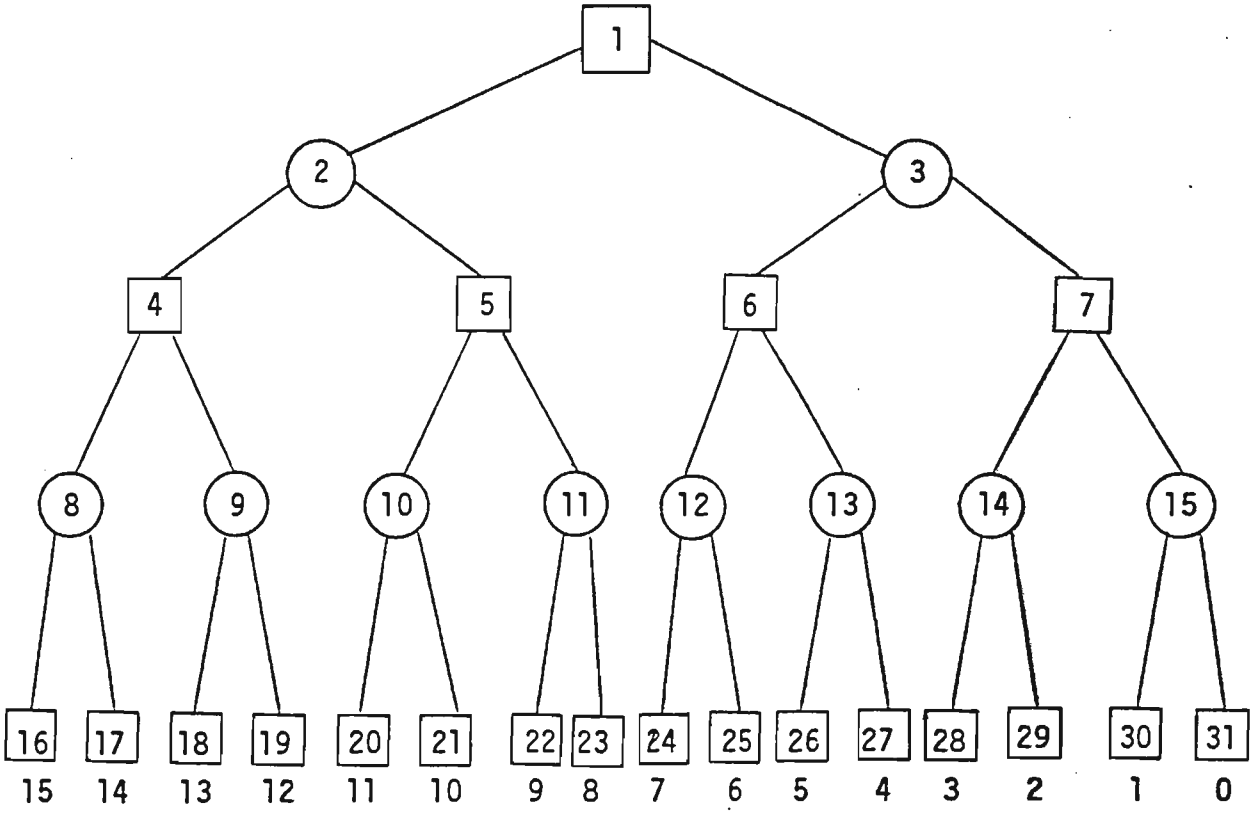

Figure 1 Example of a max-min tree. Numbers inside nodes are the names of those nodes. The numbers on the bottom row are the leaf values for this example.

The task is to compute the *minimax value* of the root node, where the minimax value of an arbitrary node n is denoted $\mu(n)$ and is defined by the following recurrence:

$$\mu(n) = \begin{cases} \text{the value assigned to n if n is a leaf} \\ \max(\mu(n_1),\mu(n_2),\ldots) \text{ if n is a } \textit{max} \text{ with immediate descendants } n_1,n_2,\ldots \\ \min(\mu(n_1),\mu(n_2),\ldots) \text{ if n is a } \textit{min} \text{ with immediate descendants } n_1,n_2,\ldots \end{cases}$$

The alpha-beta technique is one which computes μ of the root in a manner which can be considerably more efficient than the recurrence above indicates. The basic idea can be explained as follows. Suppose that n is a *max* node. To compute $\mu(n)$, we can compute $\mu(n_1)$, $\mu(n_2)$,..., where n_1, n_2,... are the immediate descendants of n, and then compute the max of those values. This can be done by keeping a "running" max of the $\mu(n_i)$. If during the computation of one of the $\mu(n_i)$, it can be ascertained that the value of $\mu(n_i)$ cannot possibly exceed the running max, despite the fact that $\mu(n_i)$ may not yet be exactly known, then the computation of $\mu(n_i)$ can be aborted; i.e. it might as well be taken as some "don't care" value ($-\infty$ will do) since it cannot "contribute" to the maximum. Since n_i is a *min* node, it need only be known that $\mu(n_i)$ is less than the running max for this type of "cutoff" to occur. If $\mu(n_i)$ is computed by a running min, then whenever the latter becomes less than the running max involved in computing $\mu(n)$, the condition necessary for a cutoff is satisfied.

The type of cutoff described above is called β *cutoff*. A similar type of cutoff can occur in computing μ of a min node, and we call it α *cutoff*.

We can summarize the technique by the procedures given in Figure 2, which are hopefully self-explanatory. To compute $\mu(n)$, we call maxnode(n, ∞).

```
maxnode(n, β):
    if leaf(n)
        then return(val(n))
        else
            allocate runmax;
            runmax ← -∞;
            for each immediate descendant n' of n
                do
                runmax ← max(runmax, minnode(n', runmax));
                if runmax > β then return (∞) fi
                od;
            return(runmax)
        fi

minnode(n, α):
    if leaf(n)
        then return(val(n))
        else
            allocate runmin;
            runmin ← ∞;
            for each immediate descendant n' of n
                do
                runmin ← min(runmin, maxnode(n', runmin));
                if runmin < α then return (-∞) fi
                od;
            return(runmin)
        fi
```

Figure 2 Program for the α-β procedure.

A summary of the calls for the tree of Figure 1 is shown in Figure 3.

It should be mentioned that the procedures shown do not take full advantage of all possible cutoffs, but rather only of "shallow" cutoffs. "Deep" cutoffs are possible, in the sense that a running max at one level can be used to cut off min computations several levels below. This requires both α and β parameters for *both* procedures. We have not yet successfully applied our method to this more extensive case. Additionally, the explanation of the method would appear to be simplified by consideration of the simpler procedures first.

It has been assumed in the preceding paragraph that the for each iterations are done sequentially. What happens when they are done currently will be discussed in the next section.

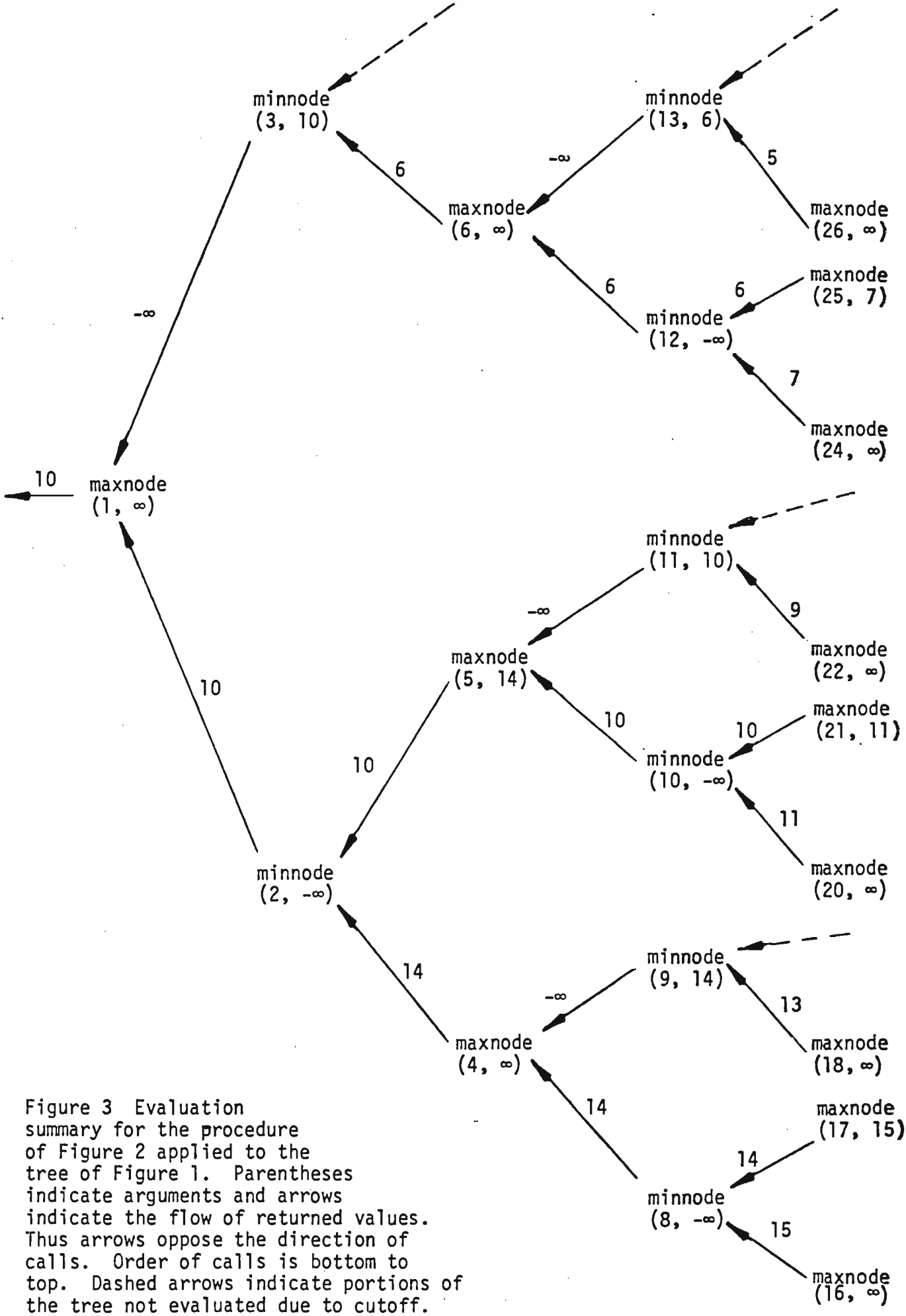

Figure 3 Evaluation summary for the procedure of Figure 2 applied to the tree of Figure 1. Parentheses indicate arguments and arrows indicate the flow of returned values. Thus arrows oppose the direction of calls. Order of calls is bottom to top. Dashed arrows indicate portions of the tree not evaluated due to cutoff.

Concurrent Alpha-Beta Procedure

Examination of the procedures in Figure 2 suggests that it is possible to perform the "for each" iterations concurrently. Moreover, if parameters are passed by name or by reference, it is possible for each parallel sub-computation to use the most current values of runmax and runmin to achieve an earlier cutoff, thereby improving efficiency. The savings can be proportional to the number of immediate descendants. An implementation detail, which we do not dwell on here, is that to realize this savings, when a procedure, say maxnode, returns ∞ because of the value returned by a minnode, it must be possible to abort other concurrent minnode calls. Failure to do so cannot effect the value returned by maxnode (because ∞ can't be exceeded), but will result in unnecessary computation.

Unfortunately, the parallel execution of these procedures seems to require cells (namely for runmax, runmin) in an essential way. By "essential" here we mean that concurrent updating is permitted, contrary to "data flow" type of communication. Put another way, if one were to examine the data values "flowing" from the cell as they are read out, one would not see a unique sequence as would be required for the computation to be determinate in the sense of, e.g., [Patil 70] or [Karp and Miller 66]. However, there would be a unique ultimate value, so that the procedure is determinate in a weaker sense.

In the next section, we indicate a generalized notion of determinacy which can account for this behavior. Subsequently, we use the notion to deduce the desired property of the α-β procedures.

We repeat that languages oriented to the more conservative concept of "data flow" do not appear able to accomodate the sort of computation present in these procedures.

The Denotational Determinacy Theorem

By a data type we mean a set D of data objects together with a partial ordering $\sqsubseteq$ on D and a bottom (i.e. least) object $\perp \in D$ with respect to the partial ordering, such that any chain of objects

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$$

has a limit, i.e. a least upper bound object, in D. The latter is denoted

$$\sqcup \{d_0, d_1, d_2, \ldots\}$$

and its defining property is, letting $\tilde{d}$ denote the above,

(i) $(\forall i)\ d_i \sqsubseteq \tilde{d}$ ($\tilde{d}$ is an upper bound)

(ii) $(\forall d \in D)\ [(\forall i)\ d_i \sqsubseteq d] \Rightarrow \tilde{d} \sqsubseteq d$ ($\tilde{d}$ is the least such)

Intuitively, a chain represents a sequence of partial results. Each partial result is a better approximation to some ultimate result than its predecessor. The ultimate result is the limit of the sequence of partial results.

The abstract notion of a data type does not tell us how to interpret the ordering and the meaning of partial results. The interpretation is embedded in one or more concrete instances of the abstraction.

For example, the *stream* data type has been discussed extensively in connection with "data flow" computation models [Kahn 74], [Kosinski 76], etc. In this data type, the data objects are streams (finite or infinite sequences) of elements. The ordering is the *prefix* ordering, i.e. $x \sqsubseteq y$ means x is a prefix of y. Such a data type is useful, for example, in representing communication between two processes by message passing. A partial result is the entire sequence of messages which has been transmitted from one process to the other since the beginning of the computation. A limit is the sequence of messages which is transmitted throughout the computation, and may be infinite. Other data types which are related to streams are *bags* [Keller 77a] and *trees* [Keller 77b].

Most models for programming languages without parallelism or non-determinism start with a base consisting of a flat data type, i.e. one having the property that

$$x \sqsubseteq y \quad \text{iff} \quad [x \neq \perp \Rightarrow x = y]$$

in which any chain has at most two distinct elements, representing approximations of no information and perfect information, respectively. An example is the integers with undefined ("?") as $\perp$, as shown in Figure 4. The arrows in that figure indicate the ordering $\sqsubseteq$, in the sense that $\sqsubseteq$ is the transitive closure of $\rightarrow$. One can then get non-flat data types from a base data type by considering the data type of "continuous functions" on data types, Cartesian products of data types, etc.

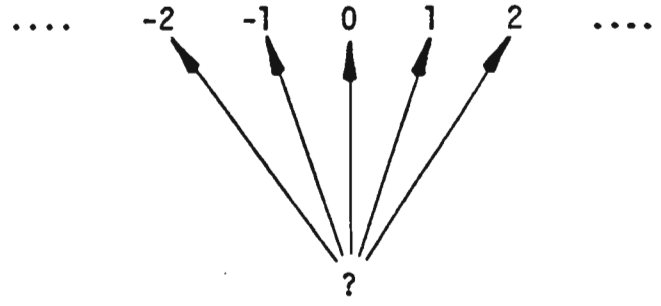

Figure 4 The flat data type Z.

Figure 5 illustrates four non-flat data types with integers as elements. These will be used in the main example of this paper, and the reason that some integers appear both with and without a ^ will become evident when that example is explained.

We call a chain of data objects a trajectory and a consecutive pair of objects in that chain a change. Intuitively, not all trajectories will occur in a particular program of interest. The set of trajectories which can will be called a context.

A function $f: D_1 \rightarrow D_2$ is called continuous with respect to a context if for any trajectory in that context,

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$$

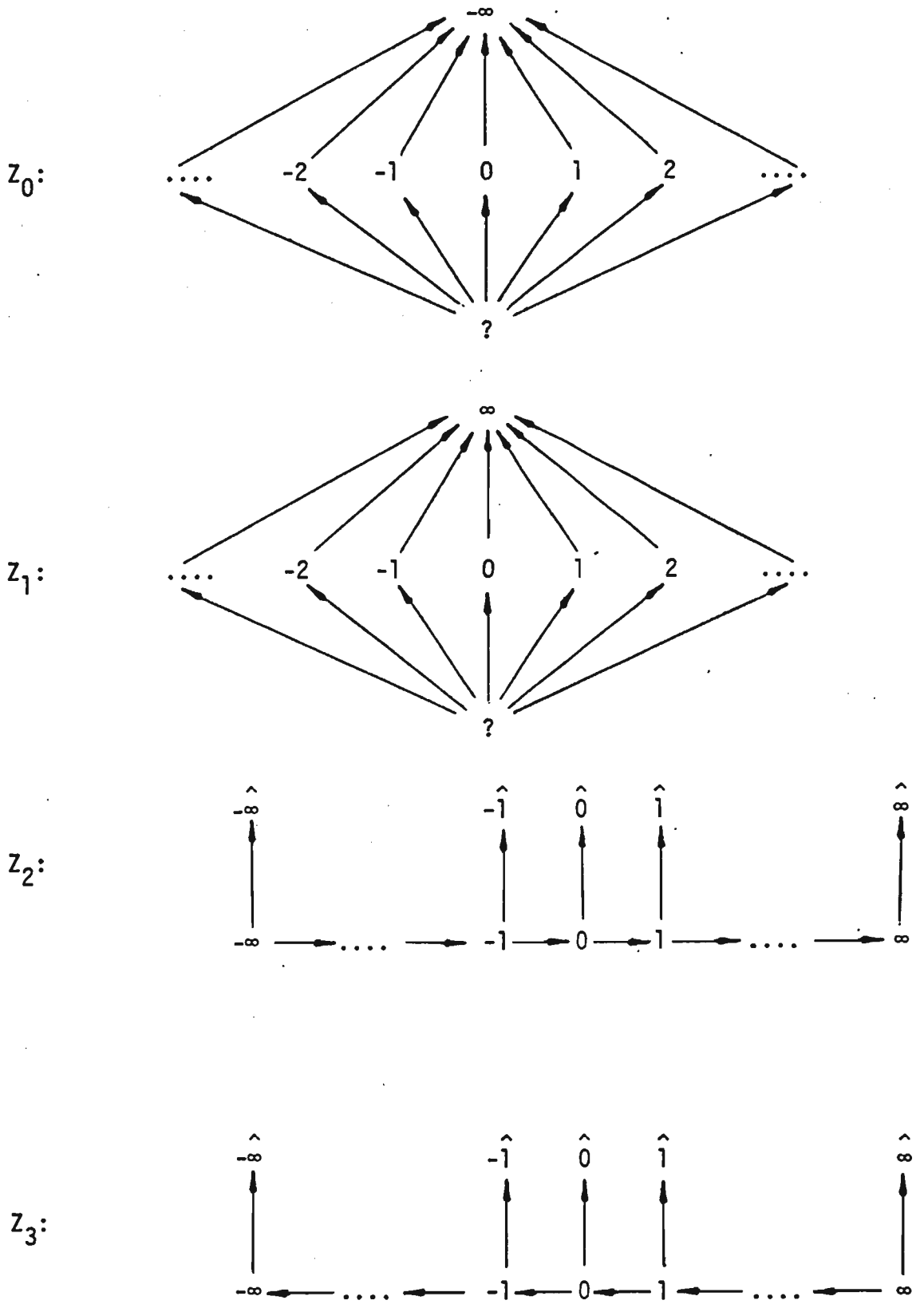

Figure 5 Other data types for the integers.

we have

(i) $f(d_0) \sqsubseteq f(d_1) \sqsubseteq f(d_2) \sqsubseteq \ldots$ (the monotonicity condition)

and

(ii) $f(\sqcup\{d_0,d_1,d_2,\ldots\}) = \sqcup\{f(d_0),f(d_1),f(d_2),\ldots\}$ (the limit condition)

Here the subscripts for $\sqsubseteq$ and $\sqcup$ (i.e. whether they apply to D_1 or to D_2) are implicit from context.

As has been discussed before, e.g. [Scott 70], it is natural to expect "computable" functions to be continuous, for this implies that every *finite* partial result occurs in response to some *finite* input segment (to put it sketchily). We do not further motivate this expectation here. The notion of continuity with respect to a context seems not to have been explored much. A similar concept is mentioned in [Patil 70].

Evidently, any Cartesian product of data types is itself a data type, with the obvious ordering and bottom element. Hence an n-ary function is continuous if it is continuous by the above definition on the Cartesian product of the n constituent data types.

Definition If every trajectory in a context has finitely many distinct elements, then we call the context a finite trajectory context.

In many cases of interest, e.g. when the data type is flat, we deal only with finite trajectory contexts. In such cases, the monotonicity condition alone suffices to demonstrate continuity. Indeed, if there are only finitely many elements in a chain,

$$d_0 \sqsubset d_1 \sqsubset d_2 \quad \ldots \sqsubset d_n$$

then requiring

$$f(d_0) \sqsubseteq f(d_1) \sqsubseteq f(d_2) \sqsubseteq \ldots \sqsubseteq f(d_n)$$

means that

$$\sqcup\{f(d_0),\ f(d_1),\ \ldots,\ f(d_n)\} = f(d_n)$$

and since

$$\sqcup\{d_0,\ d_1,\ldots,\ d_n\} = d_n$$

this is exactly what we need for continuity.

To see the use of the concepts described above in computing, imagine a program graph consisting of nodes connected by arcs. To each arc there corresponds a data type and to each node a continuous function on the data type which is the Cartesian product of the data types of its input arcs. Some arcs are only connected to a node on one end, and these constitute the input and output arcs to the graph, in the obvious manner. Furthermore, we include recursion by allowing some of the nodes, rather than being pre-defined functions, to be antecedents of unique productions. The consequences of such productions are graphs which can be viewed as replacing the antecendent nodes. Further examples of recursion appear in [Kahn 74], [Keller 77a], [Keller 77b].

The denotational determinacy theorem simply states that such a graph itself represents a continuous function. In particular, the output of a network is uniquely determined by its inputs. A sketch of the proof may be found in [Keller 77a]. Here we are mainly interested in a novel application, involving the data types described earlier.

A Simple Example

Before presenting the example of main interest, we present an auxiliary example which illustrates the use of recursion as discussed above. In this example, we assume an underlying binary tree. A node n in the tree is either an atom, in which case it has a numeric value, val(n), or a non-atom, in which case it has two immediate successors 1st(n) and 2nd(n). The recursive program shown in Figure 6 is designed to compute the maximum value in the tree. It does this by computing maxima of sub-trees concurrently. The labels on arcs in the diagram indicate the data type. Here Z is the flat data type of integers (see Figure 4), N is the flat data type of node names, and B is the flat data type of truth values. We leave the precise definition of the constituent functions to the reader, pointing out that continuity is equivalent to monotonicity, because we have only finite-trajectory contexts and hence verification of continuity is easy. Figure 7 illustrates an example tree and Figure 8 illustrates the computation of the maximum of that tree using the program of Figure 6.

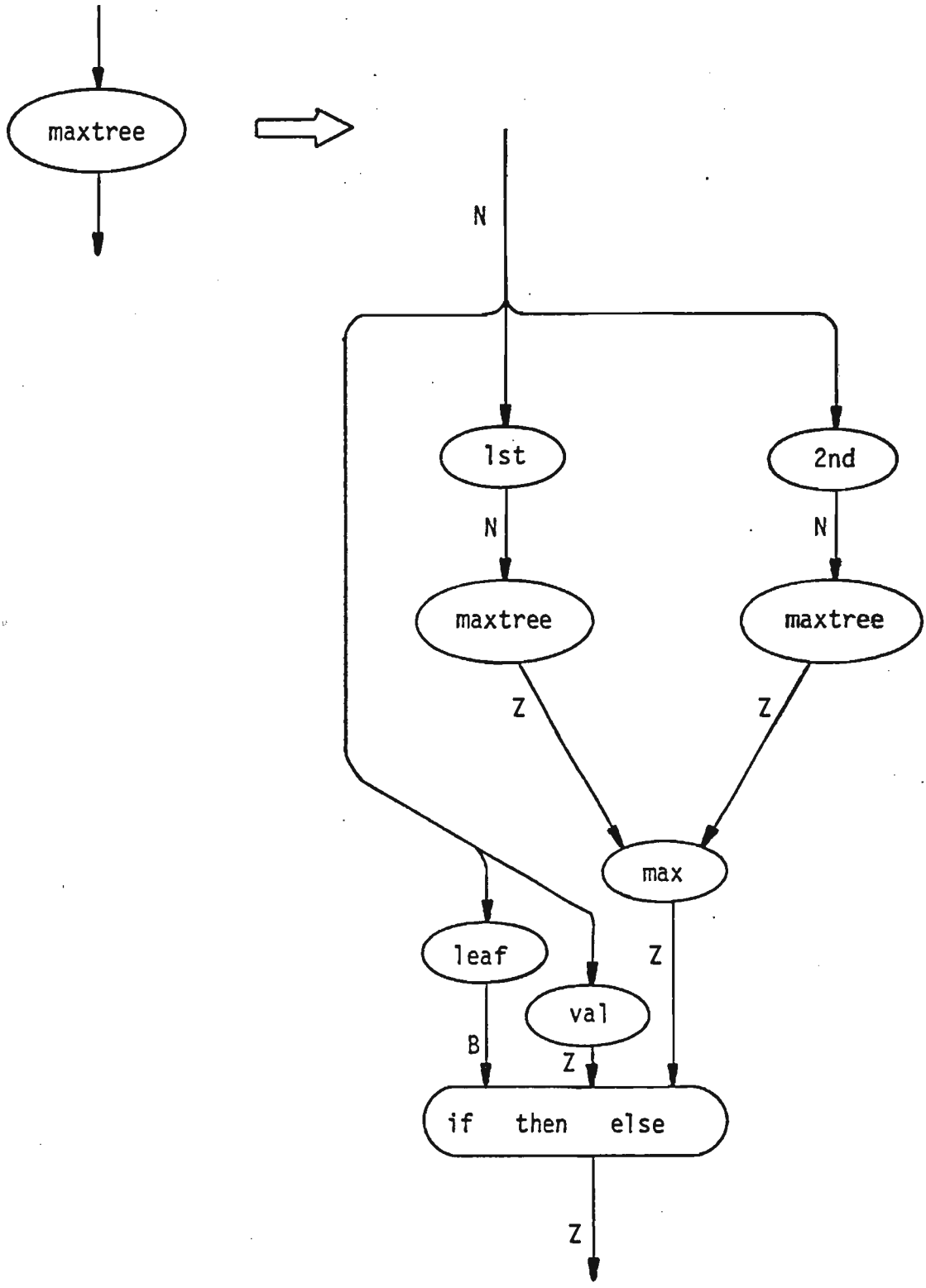

Figure 6 The program graph for maxtree.

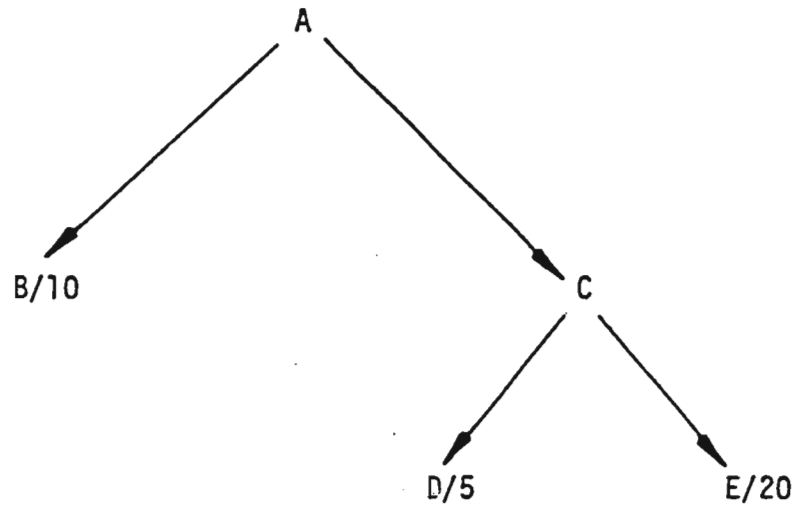

Figure 7 An argument tree for the function represented by maxtree. A, B, C, D, E are node names and 10, 5, 20 are leaf values.

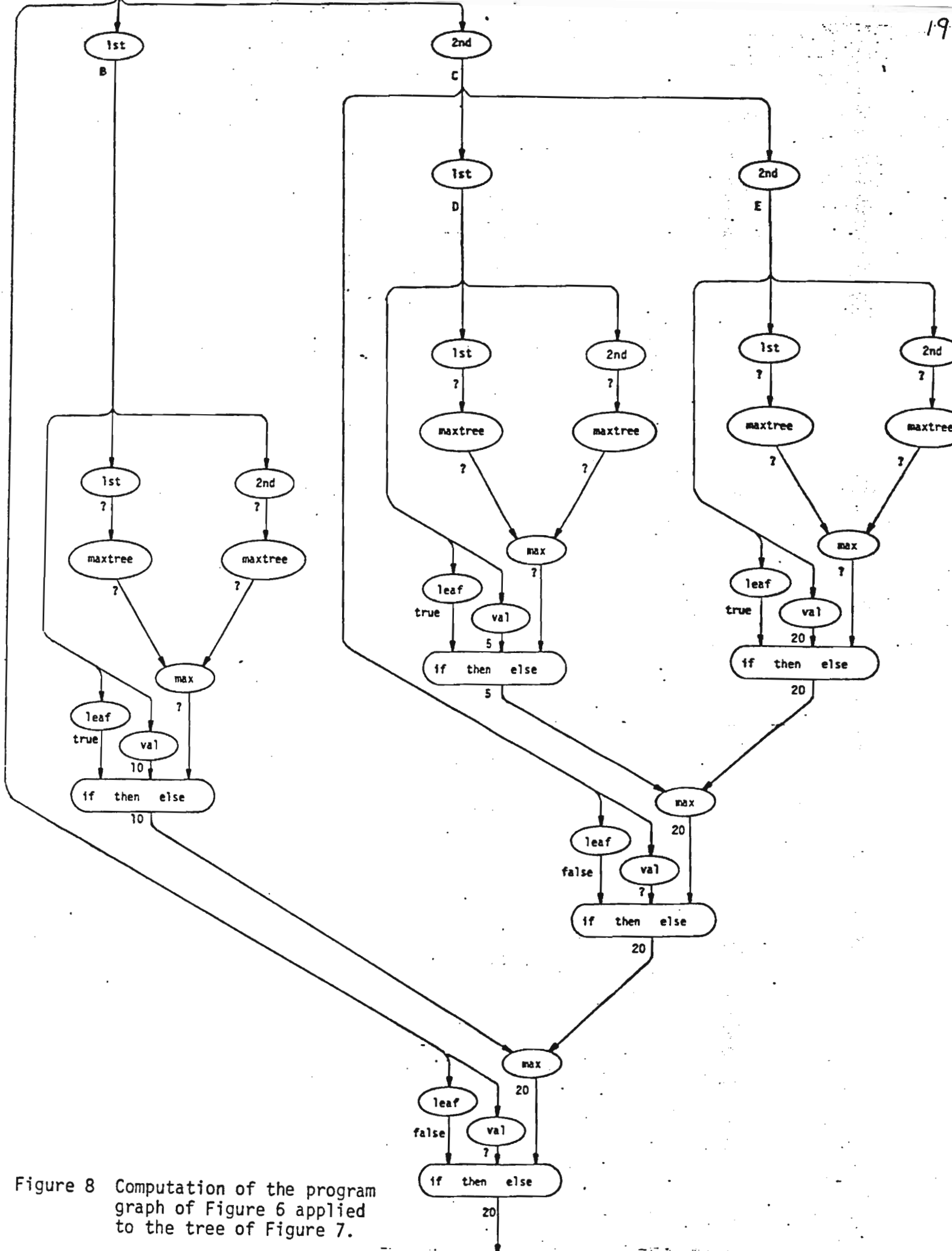

Figure 8 Computation of the program graph of Figure 6 applied to the tree of Figure 7.

Determinacy of the Alpha Beta Procedure

We now show how the denotational determinacy theorem can be used to prove determinacy of the concurrent alpha-beta procedure of Figure 9, which is a computation graph version of the procedures of Figure 2. This graph involves mutually recursive productions for the nodes maxnode and minnode. We assume that each non-leaf node n has exactly three immediate successors, 1st(n), 2nd(n), and 3rd(n).

The key idea is that, although some of the operators are not determinate in the usual data-flow sense, they are determinate under an appropriate choice of data type ordering. Accordingly, the arcs of the computation graph are labelled with names of data types, Z_0, Z_1, Z_2, Z_3 being as defined in Figure 5. N is the flat data type of node numbers, as in the example of Figure 6. The parenthesized expressions in Figure 9 indicate correspondences with the program of Figure 2.

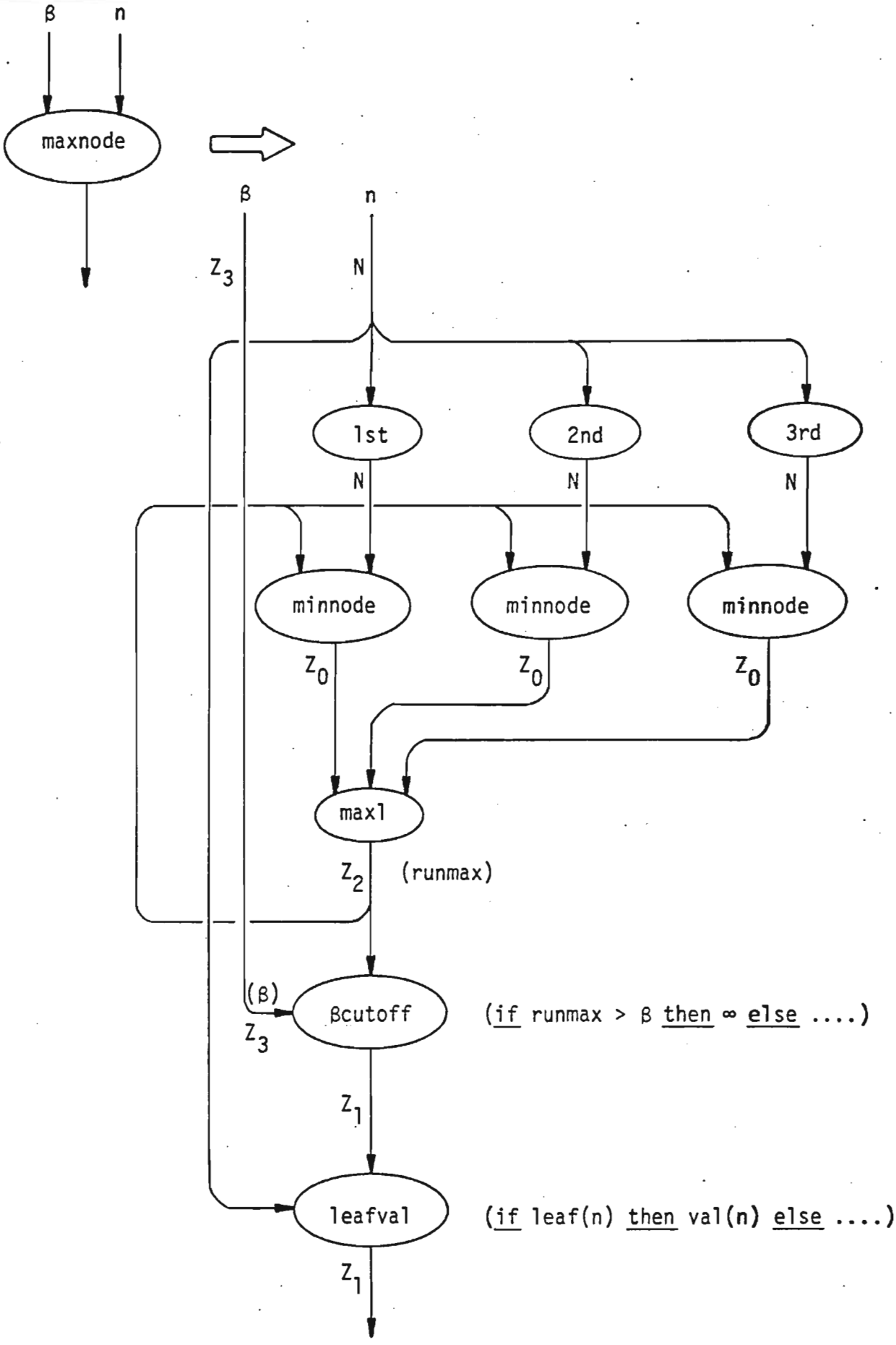

Figure 9a Program graph for maxnode.

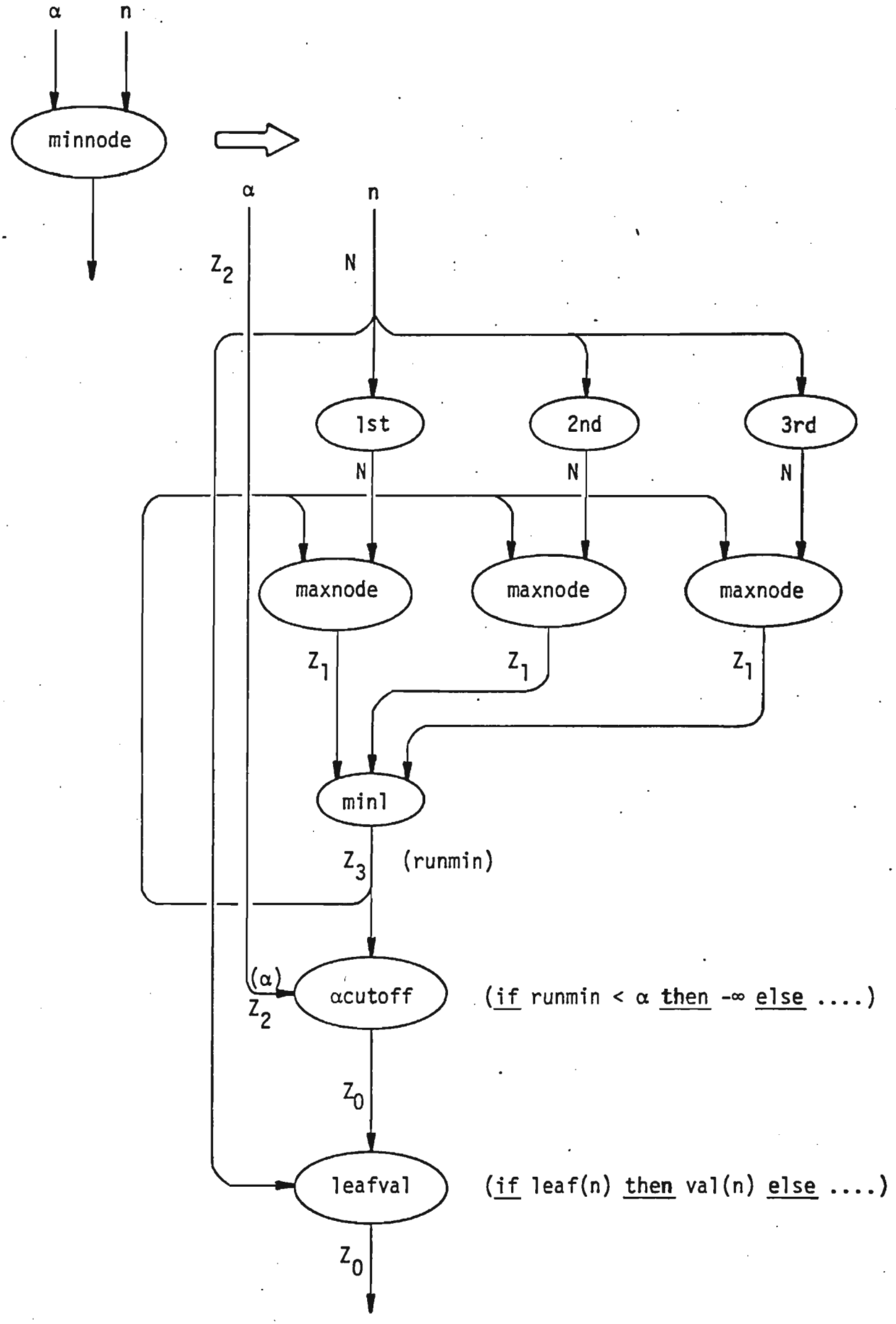

Figure 9b Program graph for minnode.

We must now define the constituent functions

$$\text{1st: } N \to N$$

$$\text{2nd: } N \to N$$

$$\text{3rd: } N \to N$$

$$\text{leafval: } N \times (Z_0 \cup Z_1) \to (Z_0 \cup Z_1)$$

$$\text{max1: } Z_0 \times Z_0 \times Z_0 \to Z_2$$

$$\beta\text{cutoff: } Z_3 \times Z_2 \to Z_1$$

$$\text{min1: } Z_1 \times Z_1 \times Z_1 \to Z_3$$

$$\alpha\text{cutoff: } Z_2 \times Z_3 \to Z_0$$

and show that they are continuous.

The functions 1st, 2nd, and 3rd simply give the respective immediate successors of their arguments. If no such successors exist, they produce the value "?". These functions are therefore monotone, and since their domain is flat, they are also continuous. The function leafval tests whether its first argument is an atom. If so, it gives the value of that atom, and otherwise it gives its second argument. Since both arguments are from a domain with chains of length at most 3, the monotonicity of leafval implies its continuity.

The function βcutoff serves to monitor the value runmax. When and if it determines that runmax > β, it returns ∞. If it gets a "final" value of runmax $\leq$ β, it returns that value. We define "final" as follows (see Figure 5):

$(\forall z \in Z_2 \cup Z_3)$ final(z) is true iff the value of z has a ^.

In summary, letting <z> denote the numeric value of z, we have

$$\beta\text{cutoff}(\beta, z) = \begin{cases} ? \text{ if } \neg\text{final}(z) \text{ and } \langle z\rangle \leq \langle\beta\rangle \\ z \text{ if final}(z) \text{ and } \langle z\rangle \leq \langle\beta\rangle \\ \infty \text{ if } \langle z\rangle > \langle\beta\rangle \end{cases}$$

Lemma 1 βcutoff is monotone.

Proof We must show that

1. $\beta \sqsubset \beta' \Rightarrow \beta\text{cutoff}(\beta, z) \sqsubseteq \beta\text{cutoff}(\beta', z)$
2. $z \sqsubset z' \Rightarrow \beta\text{cutoff}(\beta, z) \sqsubseteq \beta\text{cutoff}(\beta, z')$

First, assume $\beta \sqsubset \beta'$. Since $\beta, \beta' \in Z_3$, we have, according to the definition of Z_3, $\langle\beta'\rangle \leq \langle\beta\rangle$. If $\langle z\rangle \leq \langle\beta'\rangle$, then $\langle z\rangle \leq \langle\beta\rangle$, so that $\beta\text{cutoff}(\beta, z) = \beta\text{cutoff}(\beta', z)$. On the other hand, if $\langle z\rangle > \langle\beta'\rangle$, then $\beta\text{cutoff}(\beta', z) = \infty \in Z_1$, and irrespective of its value, $\beta\text{cutoff}(\beta, z) \sqsubseteq \infty$.

Next, assume $z \sqsubset z'$. From definition of Z_2, $\neg\text{final}(z)$ and $\langle z\rangle \leq \langle z'\rangle$. Suppose first that $\langle z'\rangle \leq \langle\beta\rangle$. Then $\langle z\rangle \leq \langle\beta\rangle$ and hence $\beta\text{cutoff}(\beta, z) = ? \sqsubseteq \beta\text{cutoff}(\beta, z')$. On the other hand, if $\langle z'\rangle > \beta$ then $\beta\text{cutoff}(\beta, z) \sqsubseteq \infty = \beta\text{cutoff}(\beta, z')$.

The definition of αcutoff is similar:

$$\alpha\text{cutoff}(\alpha, z) = \begin{cases} ? \text{ if } \neg\text{final}(z) \text{ and } \langle z\rangle \geq \langle\alpha\rangle \\ z \text{ if final}(z) \text{ and } \langle z\rangle \geq \langle\alpha\rangle \\ -\infty \text{ if } \langle z\rangle < \langle\alpha\rangle \end{cases}$$

The proof of the continuity of αcutoff is analogous to that of βcutoff.

A claim which we will later justify is the following.

Lemma 2 In the example being considered, βcutoff has a finite-trajectory context.

Consequently, continuity of βcutoff follows from monotonicity, which we proved in Lemma 1.

We now define

maxl(x,y,z) = the numeric maximum of x,y,z with a ^ if none of x, y, or z is ?, and without a ^ otherwise.

Because each argument is a member of Z_0, maxl must have a finite-trajectory context, since each chain has at most 7 distinct elements, e.g.

$(?, ?, ?) \sqsubset (5, ?, ?) \sqsubset (5, 10, ?) \sqsubset (-\infty, 10, ?) \sqsubset$

$(-\infty, 10, 20) \sqsubset (-\infty, -\infty, 20) \sqsubset (-\infty, -\infty, -\infty)$

As we shall see next, the last change in this trajectory could not possibly occur, so that an input trajectory can actually have at most 6 distinct elements. Furthermore, the impossibility of the last change in the above trajectory is essential if maxl is to be monotone.

Lemma 3 The context of maxl is restricted in the following way:

For any change of one of the variables x to $x' = -\infty$, x is not the numeric maximum of the other two variables y, z.

Proof For a variable x to change to $-\infty$, it must be the output of a minnode with $x < \alpha$, the α input of that minnode. This follows from the definition of the program in Figure 9. But also by this definition, α is the numeric maximum of x, y, z, so that $x < \alpha$ is impossible.

Given that it is impossible for the largest of the three inputs of maxl to change to $-\infty$, it is clear that maxl is monotone. Since it has a finite-trajectory context, it is also continuous.

We next define

minl(x,y,z) = the numeric minimum of x, y, z, with a ^ if none of x, y, or z is ?, and without a ^ otherwise.

We just mention that the same sort of context constraint holds for minl and the proof of its continuity is analogous to the proof for maxl.

We now return to justify the claim of Lemma 2, that βcutoff has a finite-trajectory context.

Proof of Lemma 2 It suffices to show that the trajectory of each of the two inputs to βcutoff is finite. For the second input, since it is the output of maxl, this follows from the definition of maxl and the fact that maxl has a finite-trajectory input context. The first input is finite-trajectory because it is in Z_1.

Of course, a similar restriction and proof holds for αcutoff.

One point remains to be clarified. If we again compare the program of Figure 2 to that of Figure 9, once a maxnode call in Figure 2 returns the value ∞ due to a cutoff, other subordinate minnode calls are ignorable. However, a maxnode call in Figure 9 does not, by the semantics of program graphs, *ignore* such subordinate calls. It just happens that such calls do not change the ultimate value produced by the maxnode call. It is in this sense which the two programs correspond.

The reason for the inclusion of the elements with ^'s in Z_2 and Z_3 should now be apparent. These symbols are used so that the functions αcutoff and βcutoff know when it is safe to report a value other than "?". That is, these functions cannot report the "running" values, because those values are subject to change in a way which would adversely affect other instances of maxnode or minnode. Of course, Z_0 and Z_1 reflect how these values can change.

On the other hand, the running values are useful for the α and β inputs, which is precisely why their domains are Z_2 and Z_3 instead of Z_0 and Z_1.

Conclusions and Future Efforts

We have presented a parallel alpha-beta procedure and proved its determinacy using the denotational determinacy theorem. Such a proof successfully demonstrates the desirable aspect of decomposing the proof of a system into manageable parts. In this case, what needed to be done was to choose appropriate *data types* and then show that the constituent operators are continuous functions on those data types. Ideally, once the data type is chosen, the functions can be proved *in vacuo*. However our example did not quite meet this ideal requirement.

Consequently, we refined the notion of continuity to *continuity with respect to a context*. We then proved simple lemmas which showed that contexts were such that the functions became continuous, while these functions are not continuous with respect to the "universal" context. Of course, such lemmas have analogues in "invariants" required to prove "operational" programs.

It is important to notice that the ordering aspect of data types has little to do with the physical realization of it. Indeed, we could use a single realization for all four of the different types of integers discussed here. Rather, the ordering reflects the way in which the data will be modified during a computation. In order to construct a proof as done here, we might start with the underlying domain being given by the program, but are free to discover the appropriate ordering which will allow the constituent functions to be continuous. In general, this may require substantial intellectual effort, or even be impossible.

The technique used here suggests a paradigm for constructing provable concurrent programs, namely selecting data types and operators so that the latter are continuous. Determinacy then follows automatically and

sequential proof techniques can then be applied to show total correctness. We hope to find more examples which can be solved by such techniques. One already found is the "leaf count" example of [Keller 77a]. We feel confident that the orderings used in the present paper will work with little modification for parallel versions of other "cut off" algorithms, such as those in [Burkhard and Keller 73]. Whether there are other significant classes of examples amenable to this kind of treatment remains to be seen.

Acknowledgements I thank Professor Gary Lindstrom for kindling my interest in the alpha beta procedure. Thanks also to Karen Evans for her assistance in preparing the manuscript.

References

[Burkhard and Keller 73] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. CACM, 16, 4, 230-236 (April 1973).

[Dennis 74] J. B. Dennis. First version of a data flow procedure language in B. Robinet (ed.), Programming Symposium, Lecture Notes in Computer Science, 19, 362-376 (1974).

[Doeppner and Keller 75] T. W. Doeppner, Jr. and R. M. Keller. On the relevance of abstract models in modeling semaphore implementations. Princeton University, Department of Electrical Engineering, Computer Science Laboratory Tech. Rept. TR193 (Oct. 1975).

[Francez 76] N. Francez. An application of a method for analysis of cyclic programs. Submitted for publication (May 1976).

[Kahn 74] G. Kahn. The semantics of a simple language for parallel programming. Proc. IFIP '74, 471-475 (1974).

[Karp and Miller 66] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queuing. SIAM J. Appl. Math, 14, 6, 1390-1411 (Nov. 1966).

[Keller 76] R. M. Keller. Formal verification of parallel programs. CACM, 19, 7, 371-384 (July 1976).

[Keller 77a] R. M. Keller. Denotational models for parallel programs with indeterminate operators. Proc. IFIP Working Conference on Formal Description of Programming Concepts, to be published by North-Holland (1978).

[Keller 77b] R. M. Keller. Semantics of parallel program graphs. University of Utah, Dept. of Computer Science, Tech. Rept. UUCS-77-110 (July 1977).

[Kwong 77] Y. S. Kwong. On reductions of parallel programs. Theoretical Computer Science 5, 1, 25-50 (1977).

[Lamport 77] L. Lamport. Proving the correctness of multiprocess programs. IEEE Trans., SE-3, 2, 125-143 (March 1977).

[Lipton 75] R. J. Lipton. Reduction: a new method of proving properties of parallel programs. CACM, 18, 12, 717-721 (Dec. 1975).

[Nilsson 71] N. J. Nilsson. Problem-solving methods in artificial intelligence. McGraw-Hill (1971).

[Owicki and Gries 76] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. CACM, 19, 5, 279-285 (May 1976).

[Patil 70] S. Patil. Closure properties of interconnections of determinate systems. Proc. Project MAC Conference on Concurrent Systems and Parallel Computation, 107-116 (June 1970).

[Scott 70] Outline of a mathematical theory of computation. Fourth Annual Princeton Conference on Information and Systems Sciences, 169-176 (March 1970).

[Scott 76] D. Scott. Data types as lattices. SIAM J. Comput., 5, 3, 522-587 (Sept. 1976).

[Vuillemin 74] J. Vuillemin. Correct and optimal implementations of recursion in a simple programming language. JCSS, 9, 332-354 (1974).

[Winston 77] P. H. Winston. Artificial intelligence. Addision-Wesley (1977).