A Runnable Specification of AVL-Tree Insertion

by

M.H. van Emden

Research Report CS-81-14

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

April 1981

A RUNNABLE SPECIFICATION OF AVL-TREE INSERTION

M.H. van Emden
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

We argue that runnable specification of algorithms in logic have important practical advantages: they combine superior readability with a run-time efficiency which is acceptable in some situations. We point out that, in these situations, programs have become superfluous.

1. Introduction

Formal specification, as argued by Guttag and Horning [3], should play an important role in software system development: attaining an intuitive understanding of the problem and designing a system is harder than implementing a sound design. In view of this it is truly amazing that programming and programming languages have received so much more attention than specifying and specification languages.

In a properly executed system development the initial stages (understanding, design) are more important in the sense that they are less predictable and require more talented and experienced personnel than is the case with implementation. This should not obscure the fact that implementation is still far from negligible in cost. It is therefore appropriate to consider methods of formal specification where the result can be directly executed on a computer. Recently such a method has emerged, which is based on first-order predicate logic [8, 3, 10]. In this paper we report on our experience with a very small example, namely AVL-tree insertion.

We compare this example of logic specification with a Pascal program for the same computation. We choose Pascal for the comparison, as it is a modern language designed to meet the dual objectives of clarity and efficiency in terms of machine resources. We choose AVL-tree insertion because a Pascal program for it is easily accessible [17]. For us this has the advantage, besides the one of saving time, that the author of the Pascal program will be above any suspicion of not being sufficiently sympathetic to the Pascal language.

In recent years it has been widely recognized that it is important for computer programs to be as readable as possible. In the fifteen years between the inception of Fortran and Pascal considerable progress has been made in this respect. Unfortunately, progress seems to be slowing down: almost as much progress was made already in the five years between Fortran and Algol 60.

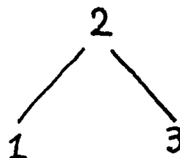
Concern for readability in the design of programming languages should not, however, obscure the fact that specifications are, by their very nature, more readable than programs. To emphasize this we have included in full a logic specification of AVL-tree insertion so that the reader can make a comparison in terms of readability with a description such as in [17]. We have attempted to give sufficient additional explanation to make the paper easily readable for those both unfamiliar with AVL-tree insertion and logic specifications.

Explicit use of first-order predicate logic as program specification language with emphasis on runnability originates with Davis [4]. Both our approach and Davis's is based on the Horn-clause subset of first-order predicate logic. OBJ [5] exemplifies an approach to runnable specifications which is typed, and based on another subset, namely the equations.

2. Trees Represented in Logic

The main linguistic categories in first-order predicate logic are terms and predicates. Terms denote objects; predicates denote relations between objects. Terms have a tree-like structure; our representation is indeed based on this likeness.

Formally speaking, a term is either a simple term or a composite term. A simple term is a constant or a variable. A composite term is $f(t_1, \ldots, t_n)$ where f is an n-place function symbol and $t_1, \ldots, t_n$ are terms. For example bst(bst(nil, 1, nil), 2, bst(nil, 3, nil)) is a term where bst is a 3-place function symbol and nil, 1, 2, 3 are constants. This term can be used to represent the binary search tree



The representation depends on the convention that nil stands for the empty tree and that the arguments of bst stand for, respectively, the left subtree, the root, and the right subtree. Notice that the function symbol bst is used here as a data constructor. It plays a role similar to the one of a record in Pascal. Of course the function symbol still denotes a function, in this case from (tree, node, tree) - tuples to trees.

Our example needs to be modified only slightly to accommodate AVL-trees, which have one additional component: a balance indicator, which indicates whether the right subtree has a height which is smaller than,

equal to, or larger than the left subtree. The value of the balance indicator is in these cases, respectively, equal to the constants "<", "-", ">". Think of "<" and ">" as arrows pointing to which subtree has greater height and of "-" as "neutral".

For AVL-trees we use the 4-place function symbol avl. Examples:

```
avl(nil, 1, -, nil)        and
avl(nil, 1, >, avl(nil, 2, -, nil))
```

are AVL-trees. Of course, there exist terms having avl as function symbol which are not AVL-trees and not even any kind of search tree.

3. The Insert Relation Specified in Logic

Data structures may be represented by terms of logic. Compatibly with this choice, operations on data structures may be represented by relations between terms. The arguments of the relation then include the data structure as it was before the operation and as it is afterwards.

Take, for example, the insert relation among binary search trees. The following assertion is true of the relation we have in mind:

```
insertl(nil, *elt, bst(nil, *elt, nil))        ...(1)
```

Note that an identifier preceded by an asterisk is a variable. Assertions are (implicitly) universally quantified over all variables occurring in them. The first argument of "insertl" is the tree before insertion, the last argument is the tree afterwards, and the middle argument in the element inserted.

We have only covered insertion into the empty binary search tree. The following assertion states a useful fact about insertion into non-empty trees:

```
insertl(bst(*lst, *root, *rst), *elt, bst(*lstl, *root, *rst))
        ←le(*elt, *root) & insert (*lst, *elt, *lstl)        ...(2)
```

Read "←" as "if" and "&" as "and". The predicate symbol "le" stands for the relation "less than or equal to". The assertion (2) says that insertion is achieved by insertion into the left subtree provided that the element to be inserted is less than or equal to the root. Similarly we have

```
insertl(bst(*lst, *root, *rst), *elt, bst(*lst, *root, *rstl))
        ←ge(*elt, *root) & insert (*rst, *elt, *rstl)         ...(3)
```

The predicate symbol "ge" stands for the relation "greater than or equal to".

The assertions (1), (2), and (3) in conjunction are a complete specification of binary tree insertion in the sense that whenever t_2 is the result of inserting e into a binary tree t_1, the assertions logically imply insertl(t_1, e, t_2).

The insert relation for AVL-trees is similar to the one for binary search trees. AVL-trees are binary search trees where for every node the left and right subtrees do not differ more than one in height. Insertion into AVL-trees proceeds in two phases. The first is the same as insertion for binary search trees. As a result of such insertion it may happen that at one or more nodes subtrees differ in height more than one. In the second phase, if necessary, an operation called "rebalancing" is then carried out which restructures the tree so that it becomes an AVL-tree. This second phase is represented by the relation "adjust" which has as arguments the old and new forms of the tree. Note that to adjust a tree may be to leave it as it is, in case insertion did not destroy the property of being an AVL-tree.

A typical assertion concerning the insert relation for AVL trees is:

```
insert(avl(*lst, *root, *bi, *rst), *elt, *nt)
← le(*elt, *root) & insert (*lst, *elt, *lstl)          ...(4)
& adjust (avl(*lst1, *root, *bi, *rst), *nt)
```

The variable *bi is the balance indicator, *nt is the new tree after adjustment. The first argument of "adjust" is not necessarily an AVL-tree. The second argument ("nt" is short for new tree) is either the same as or a rebalanced version of the first argument; and in either case an AVL-tree.

The assertion (4) gives the right general idea, but it still needs some improvement. It is of course in theory possible for "adjust" to check whether its first argument is balanced (by completely traversing it, for example). It is much more attractive to make use of the fact that before insertion the tree was balanced and then we need only as additional information whether the height of the subtree, in which insertion took place, increased. And, of course, whether this subtree was on the left or on the right. This additional information comes as additional arguments for "adjust". Part of it has to be supplied by "insert", which therefore also gets an additional argument. Thus the three major assertions for AVL-tree insertion become as shown below. They are the counterparts of assertions (1), (2), and (3) for unbalanced insertion.

```
INSERT(NIL,*ELT,AVL(NIL,*ELT,-,NIL),YES).
                                                          /*      ...(5) */
INSERT(AVL(*LST,*ROOT,*BI,*RST),*ELT,*NT,*ISCHANGED)
<- LE(*ELT,*ROOT) & INSERT(*LST,*ELT,*LST1,*LSTISCHANGED)
 & ADJUST(AVL(*LST1,*ROOT,*BI,*RST),*LSTISCHANGED,LEFT,*NT,*ISCHANGED).
                                                          /*      ...(6) */
INSERT(AVL(*LST,*ROOT,*BI,*RST),*ELT,*NT,*ISCHANGED)
<- GE(*ELT,*ROOT) & INSERT(*RST,*ELT,*RST1,*RSTISCHANGED)
 & ADJUST(AVL(*LST,*ROOT,*BI,*RST1),*RSTISCHANGED,RIGHT,*NT,*ISCHANGED).
                                                          /*      ...(7) */
```

4. The Relations Auxiliary to AVL-tree Insertion

The arguments of the adjust relation are as follows:

adjust(oldtree, ischanged1, leftorright, newtree, ischanged2)

"newtree" is either the same as "oldtree", or is the result of rebalancing. Of "oldtree" it is given that it is the result of doing one insertion into an AVL-tree. As a result it may have happened that one of the subtrees increased in height. Whether this is the case is indicated by "ischanged1"; "left or right" indicates in that case which subtree was inserted into.

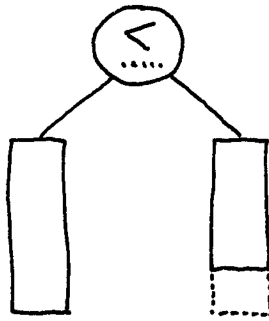
It may happen that "newtree" is of greater height than "oldtree"; this is indicated by "ischanged2".

The first fact about "adjust" is simple:

```
ADJUST(*OLDTREE,NO,*,*OLDTREE,NO).                        /*      ...(8) */
```

This says that if the subtree did not change in height as the result of an assertion, then adjustment does not change the tree. A variable name * is one that has a name different from any other in its assertion. This assertion is true for any value of its third argument.

Let us now consider the case where insertion did change the height of a subtree. Whether this makes rebalancing necessary depends on two things: What the balance of the tree was before insertion and whether the height-increasing insertion happened on the left or the right. For example, here is a tree which was originally tilted to the left and where a height-increasing insertion occurred on the right:



The new tree does not have to be rebalanced. The balance indicator changes from "<" to "-". However, if the height-increasing insertion would have happened on the left, rebalancing would have been necessary. The various possibilities are summarized in the following table:

BALANCE BEFORE	WHERE INSERTED	BALANCE AFTER	WHOLE TREE INCREASED	TO BE REBALANCED
-	LEFT	<	YES	NO
-	RIGHT	>	YES	NO
<	LEFT	-	NO	YES
<	RIGHT	-	NO	NO
>	LEFT	-	NO	NO
>	RIGHT	-	NO	YES

This table can be regarded as the specification of a 5-argument relation: the table states which 5-tuples belong to the relation. Let us call the relation "table". Then the following assertions of logic specify the relation "table", and therewith the contents of the above table.

```
/*     BALANCE    WHERE      BALANCE   WHOLE TREE   TO BE          */
       BEFORE     INSERTED   AFTER     INCREASED    REBALANCED

TABLE(   -    ,   LEFT   ,    <    ,    YES    ,    NO    ).
TABLE(   -    ,   RIGHT  ,    >    ,    YES    ,    NO    ).
TABLE(   <    ,   LEFT   ,    -    ,    NO     ,    YES   ).  /*...(9)*/
TABLE(   <    ,   RIGHT  ,    -    ,    NO     ,    NO    ).
TABLE(   >    ,   LEFT   ,    -    ,    NO     ,    NO    ).
TABLE(   >    ,   RIGHT  ,    -    ,    NO     ,    YES   ).
```

... assertion (8) says all about the "adjust" relation for the case where no subtree changed in height. In the other case the "adjust" relation depends on "table" and "rebalance", as expressed in the following assertion:

```
ADJUST(AVL(*LST,*ROOT,*BI,*RST),YES,*LOR,*NT,*ISCHANGED)
<- TABLE(*BI,*LOR,*BI1,*ISCHANGED,*TOBEREBALANCED)
 & REBALANCE(AVL(*LST,*ROOT,*BI,*RST),*BI1,*TOBEREBALANCED,*NT).
```

The first argument of "rebalance" is the tree which possibly needs re-balancing; the last argument is an AVL-tree which contains the same keys as the first argument. The second argument is the balance indicator of the tree in the last argument. This information is specified in "table".

Again the first assertion about "rebalance" is simple: it covers the case where "table" has specified that no rebalancing is needed.

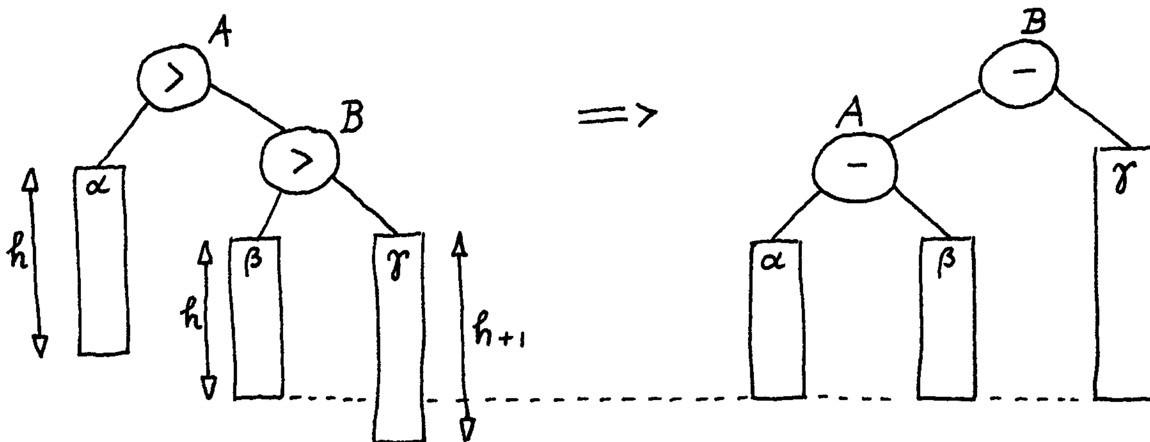
```
REBALANCE(AVL(*LST,*ROOT,*BI,*RST)
          ,*BI1,NO,AVL(*LST,*ROOT,*BI1,*RST)).
```

The only difference in this case between the old tree and the new tree is possibly in the balance indicator.

In case rebalancing is necessary, "rebalance" is defined in terms of "=>", which denotes the relation for the required operations on the tree. The first (second) argument is the tree before (after) the operation. Note, that unlike the other predicate symbols, "=>" is used in infix notation.

```
REBALANCE(*OLDTREE,*,YES,*NEWTREE)
          <- (*OLDTREE => *NEWTREE).
```

Finally the time has come to consider the actual tree transformations. This is best explained by means of diagrams, as in Knuth [7].



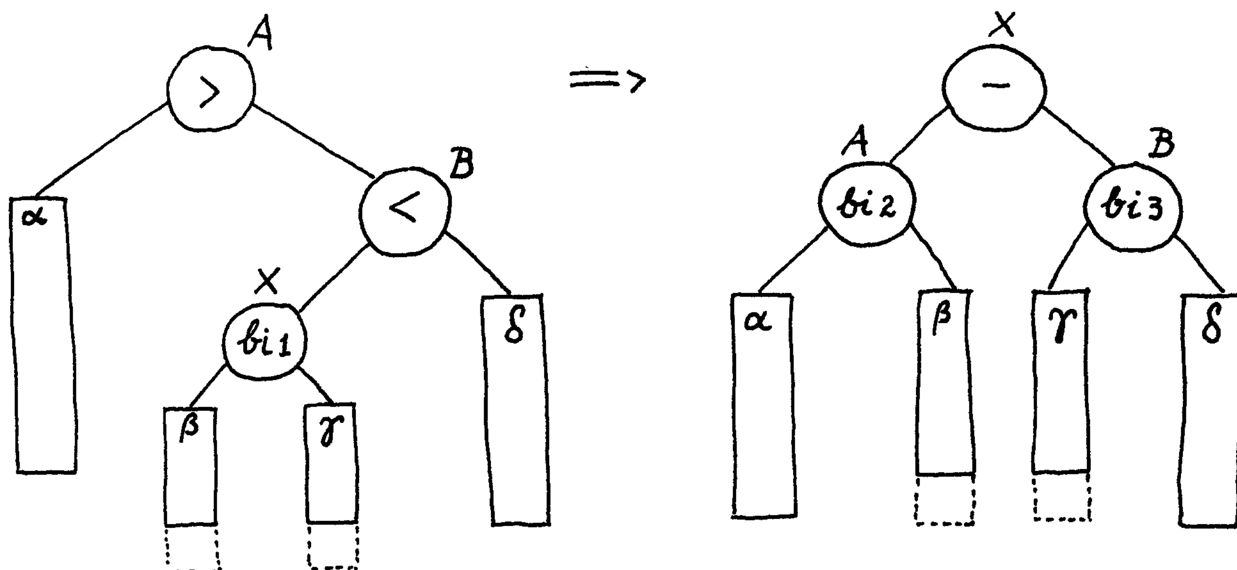
The tree on the left is not an AVL-tree because the subtrees of node A differ more than 1 in height. The indicated operation then yields an equivalent AVL-tree. An attractive feature of logic is that the above diagram can be translated literally (figuratively speaking) into a logic assertion, as follows.

```
AVL(*ALFA,*A,>,AVL(*BETA,*B,>,*GAMMA)) =>
AVL(AVL(*ALFA,*A,-,*BETA),*B,-,*GAMMA).
```

The symmetrically similar case is given by

```
AVL(AVL(*ALFA,*A,<,*BETA),*B,<,*GAMMA) =>
AVL(*ALFA,*A,-,AVL(*BETA,*B,-,*GAMMA)).
```

The other type of tree transformation is represented in the diagram below.



There are in fact two different, but very similar, subtypes here, depending on the balance at node X initially. If it is "<", then afterwards the balance at A is "-" and at B it is ">". If initially the balance at X is ">", then afterwards the balance at A is "<" and at B it is "-". Initially the balance at X cannot be "-", because the element, which causes the unbalance at A, can only have been inserted in one of the subtrees of X. These contingencies are summarized in the following small table

```
           /*  BI1   BI2   BI3  */
TABLE2(     <  ,  -  ,  >    ).
TABLE2(     >  ,  <  ,  -    ).
```

The transformation is represented in the following assertion.

```
AVL(*ALFA,*A,>,AVL(AVL(*BETA,*X,*BI1,*GAMMA),*B,<,*DELTA)) =>
AVL(AVL(*ALFA,*A,*BI2,*BETA),*X,-,AVL(*GAMMA,*B,*BI3,*DELTA))
<- TABLE2(*BI1,*BI2,*BI3).
```

The symmetrically similar case is

```
AVL(AVL(*ALFA,*A,>,AVL(*BETA,*X,*BI1,*GAMMA)),*B,<,*DELTA) =>
AVL(AVL(*ALFA,*A,*BI2,*BETA),*X,-,AVL(*GAMMA,*BI3,*DELTA))
<- TABLE2(*BI1,*BI2,*BI3).
```

```
INSERT(NIL,*ELT,AVL(NIL,*ELT,-,NIL),YES).
                                                              /*     ...(5) */
INSERT(AVL(*LST,*ROOT,*BI,*RST),*ELT,*NT,*HTISCHANGED)
<- LE(*ELT,*ROOT) & INSERT(*LST,*ELT,*LST1,*LSTISCHANGED)
 & ADJUST(AVL(*LST1,*ROOT,*BI,*RST),*LSTISCHANGED,LEFT,*NT,*HTISCHANGED).
                                                              /*     ...(6) */
INSERT(AVL(*LST,*ROOT,*BI,*RST),*ELT,*NT,*HTISCHANGED)
<- GE(*ELT,*ROOT) & INSERT(*RST,*ELT,*RST1,*RSTISCHANGED)
 & ADJUST(AVL(*LST,*ROOT,*BI,*RST1),*RSTISCHANGED,RIGHT,*NT,*HTISCHANGED).
                                                              /*     ...(7) */

ADJUST(*OLDTREE,NO,*,*OLDTREE,NO).                            /*     ...(8) */

ADJUST(AVL(*LST,*ROOT,*BI,*RST),YES,*LOR,*NT,*HTISCHANGED)
<- TABLE(*BI,*LOR,*BI1,*HTISCHANGED,*TOBEREBALANCED)
 & REBALANCE(AVL(*LST,*ROOT,*BI,*RST),*BI1,*TOBEREBALANCED,*NT).


/*    BALANCE     WHERE      BALANCE    WHOLE TREE   TO BE
       BEFORE    INSERTED     AFTER      INCREASED   REBALANCED     */

TABLE(   -     ,   LEFT    ,    <    ,     YES     ,    NO     ).
TABLE(   -     ,   RIGHT   ,    >    ,     YES     ,    NO     ).
TABLE(   <     ,   LEFT    ,    -    ,      NO     ,   YES     ). /*...(9)*/
TABLE(   <     ,   RIGHT   ,    -    ,      NO     ,    NO     ).
TABLE(   >     ,   LEFT    ,    -    ,      NO     ,    NO     ).
TABLE(   >     ,   RIGHT   ,    -    ,      NO     ,   YES     ).


REBALANCE(AVL(*LST,*ROOT,*BI,*RST)
         ,*BI1,NO,AVL(*LST,*ROOT,*BI1,*RST)).

REBALANCE(*OLDTREE,*,YES,*NEWTREE)
         <- (*OLDTREE => *NEWTREE).


AVL(*ALPHA,*A,>,AVL(*BETA,*B,>,*GAMMA)) =>
AVL(AVL(*ALPHA,*A,-,*BETA),*B,-,*GAMMA).

AVL(AVL(*ALPHA,*A,<,*BETA),*B,<,*GAMMA) =>
AVL(*ALPHA,*A,-,AVL(*BETA,*B,-,*GAMMA)).

AVL(*ALPHA,*A,>,AVL(AVL(*BETA,*X,*BI1,*GAMMA),*B,<,*DELTA)) =>
AVL(AVL(*ALPHA,*A,*BI2,*BETA),*X,-,AVL(*GAMMA,*B,*BI3,*DELTA))
<- TABLE2(*BI1,*BI2,*BI3).

AVL(AVL(*ALPHA,*A,>,AVL(*BETA,*X,*BI1,*GAMMA)),*B,<,*DELTA) =>
AVL(AVL(*ALPHA,*A,*BI2,*BETA),*X,-,AVL(*GAMMA,*B,*BI3,*DELTA))
<- TABLE2(*BI1,*BI2,*BI3).


      /*  BI1   BI2   BI3   */
TABLE2(    <  ,  -  ,  >    ).
TABLE2(    >  ,  <  ,  -    ).
```

5. Running the Specification

In Figure 1 we have collected together all assertions specifying insertion for AVL-trees. It is also a listing of the code run by a PROLOG interpreter [14]. The run time was compared to the Pascal program for AVL-tree insertion published by N. Wirth [17]. The runs were performed on an IBM 4341 computer under the VM/CMS operating system. The Pascal program was run by two language processors. One was "Waterloo Pascal" which is a processor for student use intended to economize on translation and loading, without regard for efficiency in the resulting code. The other is IBM's Pascal VS optimizing compiler.

The results are summarized in the tables below.

Processor	Mode	Virtual CPU time in seconds
Waterloo Pascal	no printing	.99
Waterloo Pascal	output printed	1.40
Prolog	no printing	.22
Prolog	output printed	.28
Pascal VS	no printing	.04
Pascal VS	output printed	.11

Figure 2

Time required to run the translated
program for AVL-tree insertion
inserting 50 nodes

Processor	Processor Size (blocks of 800 bytes)	Minimum machine size required (Kbytes)	Translate and Load Time (seconds virtual CPU time
Waterloo Pascal	17	640	2.67
Prolog	29	301	.32
Pascal VS	336	784	6.16

Figure 3

Resources required for translation

6. Conclusions

In this example logic has proved to be a near-ideal interface between a human and a computer.

For the human, logic is a medium allowing concise and versatile expression of facts about relations. Note how the assertions for "insert" express recursive definitions. Note how directly tabular material can be represented. And note also how directly the diagrams for the tree transformations are represented in the text. The precision and rigor of logic is guaranteed by a mathematically defined semantics. More importantly, the meaning of logic assertions is usually intuitively ascertainable, without reference either to formally defined semantics or proof theory.

For a machine, logic is apparently interpreted with an efficiency which is in some situations sufficient to make the writing of programs superfluous. In other situations the advantages, claimed by Guttag and Horning [6] for their unrunnable specifications, still apply.

After these euphoric remarks a few sobering thoughts are in order; thoughts concerning the state of the art in systems running logic specifications.

1. Only runtime has been compared, not space utilization. One reason is that such a comparison is a good deal more difficult to make, requiring some additional instrumentation for both the Pascal and logic systems used. A more important reason is that the logic system used in this comparison is based on an early design which is not as economical in space as the newer implementations [11, 13].

2. The example is particularly favourable for current logic systems, which are at their best with tree-structured data and with recursion as opposed to iteration. The Waterloo logic system, for example, has no iteration. When iteration has to be done by recursion (certainly not always necessary) it is done at a considerable penalty in terms of space. However, Warren's system [16] has the tail-recursion optimization. Here iteration is still done by recursion but in a way which combines the important semantic advantages of recursion with most of the economy of iteration.

   Terms are more flexible than LISP's lists; this already carves out a sizable niche of applications for logic specifications. Still, the problem of including arrays remains to be addressed, although there do not seem to be any fundamental difficulties.

3. The nature and role of specifications is not yet clear. The logic systems have been used primarily as a neat way of programming, with little concern for whether the code is usable as a specification. McDermott's note [12] is symptomatic for this orientation. Even

within specifications, widely differing objectives are possible. The example in this paper is specification of the logic of an algorithm (in the sense of Kowalski [9]) for AVL-tree insertion. It would also be possible to specify when a binary search tree is ordered, what the height of a tree is and then just to specify that insertion must result in an ordered tree of which the heights of the subtrees differ by 0 or 1. No system is in sight which would run such a specification with anything but monstrous inefficiency. Yet such a specification, without any commitment to a specific algorithm, is closer to most people's intuition than the one exhibited in this paper. The algorithmic specification is in need of justification with respect to the non-algorithmic one. Such justification is facilitated by the fact that both are in first-order predicate logic, which has a well-developed proof theory. See Clark [1, 2] for examples of such justification.

4. For Pascal it is apparently possible for a computer to produce code which runs about 20 times faster than an interpreter. Such speed-ups are of course not necessary for logic processors, as they are not as slow as Pascal interpreters. But it is not at all clear whether the equivalent factor of about 10 is achievable by compilers for logic. Some encouraging results have been obtained for logic compilation by Warren [15]; only time can tell how far this can be taken.

7. Acknowledgements

Thanks are due to Virgil Chan for help with writing the specification and to Ronald Ferguson for the timing comparisons. The National Science and Engineering Research Council financed facilities supporting this work.

8. References

1. Clark, K.L., The verification and synthesis of logic programs. Department of Computation & Control, Imperial College, 1977.

2. Clark, K.L., Predicate Logic as a Computational Formalism. Springer (to appear).

3. Colmerauer, A., Metamorphosis Grammars. In Natural Language Communication with Computers, L. Bolc (ed.), Springer Lecture Notes in Computer Science, 1978.

4. Davis, R., Runnable specifications as a design tool. In Logic Programming, K. Clark and S. Tärnlund (eds.), to appear.

5. Goguen, J.A., and Tardo, J.J., An introduction to OBJ, Specification of Reliable Software, (Conference Proceedings, MIT, April, 1979).

6. Guttag, J., and Horning, J.J., Formal specification as a design tool. Seventh Annual ACM Symposium on Principles of Programming Languages, SIGACT/SIGPLAN, 1980.

7. Knuth, D., The Art of Computer Programming, Vol. I, Addison-Wesley, 1968.

8. Kowalski, R.A., Predicate logic as a programming language; Proc. IFIP 74, North Holland, 1974, pp. 556-574.

9. Kowalski, R.A., Algorithms = Logic + Control, Comm. ACM 22(1979), pp. 424-436.

10. Kowalski, R.A., Logic for Problem-Solving, North Holland, Elsevier, 1979.

11. McCabe, F.G., Micro Prolog. In Logic Programming, K. Clark and S. Tärnlund (eds.), to appear.

12. McDermott, D., The Prolog phenomenon, SIGART Newsletter, July 1980. pp. 16-20.

13. Mellish, C.S., An alternative to structure-sharing in the implementation of a PROLOG interpreter. In Logic Programming, K. Clark and S. Tärnlund (eds.), to appear.

14. Roberts, G.M., An implementation of PROLOG; M.Sc. Thesis, Dept. of Computer Science, University of Waterloo, 1977.

15. Warren, D.H.D., Implementing PROLOG-compiling predicate logic programs. DAI Research Reports 30 and 40, Dept. of Artificial Intelligence, University of Edinburgh, 1977.

16. Warren, D.H.D., An improved PROLOG implementation which optimizes tail recursion. In K. Clark and S. Tärnlund (eds.), Logic Programming (to appear).

17. Wirth, N., Algorithms + Data Structures = Programs, Prentice Hall, 1975.