Nordic Journal of Computing

Blood from Dahm's Turnip*

Kasey N. Klipsch and David S. Wise
Computer Science Dept.,
Indiana University
Bloomington, Indiana 47405–7104, USA
kklipsch@acm.org

Abstract. Dahm's compression of the table representing a COBOL data declaration by Knuth, is compressed still further.

CCS Classification: D.3.4 [Programming Languages]: Processors—Compilers, COBOL; E.1 [Data Structures]: Trees.
General Terms: design, algorithms.

Key words: Multilinked structures, COBOL.

1. Introduction

Section 2.4 of *The Art of Computer Programming* [1] presents a problem extracted from the design of a COBOL compiler. It is offered as an exercise in data-structure and algorithm design for multi-linked structures. Three algorithms are developed for building the structure, for checking qualified references, and for finding corresponding pairs. This data structure is then modified for a more efficient representation, with a discussion on the ramifications for the algorithms.

Knuth illustrates the tablular tree structure of the variables in a COBOL program with this example from his Figure 2.4(4):

```
1 A                 1 H
  3 B                 5 F
    7 C                 8 G
    7 D               5 B
  3 E                 5 C
  3 F                   9 E
    4 G                 9 D
                        9 G
```


Received September 1999.

As he recognizes, David M. Dahm created an ingenious representation [p. 433–434] to solve the problem. This paper presents a further space optimization either on Dahm's structure or on Knuth's illustrative one.

2. Changes in the Data Structure

Aside from the tree structure illustrated above, the additional data that tells "how much space each item of information occupies and in what format it appears" [bottom, p. 424] applies only to the terminal nodes. That is, the empty boxes at the right of Figure 2.4(5) [p. 428] are needed only for leaves.

Change the data structure, optimizing the structure for size, in one more step that has a minimal effect on the speed of the algorithms.

- If the additional data is small (*e.g.* an integer) provide one bit as a tag to identify a terminal node in the `SCOPE` fields of Dahm's representation (or in the `CHILD` fields). Use the rest of that field to store the atomic data in place of a null pointer (Knuth calls it Λ) or a reflexive (Dahm's) pointer.
- If the data is large, store it elsewhere and leave a pointer in the `SCOPE` (or `CHILD`) field to another data area, disjoint from the Data Table.

Any algorithm can check whether or not a node is a terminal by checking either the tag bit or the magnitude of this new pointer. If the Data Table is stored sequentially this pointer can be compared to the table's bounds. In particular, if the atomic information is stored below the entire Data Table, then a `SCOPE` pointer that points backwards identifies an atom, and one pointing forwards is as Dahm prescribes.

So, with ↑data representing either a tagged datum or a reference to it, Dahm's table [2.4–12, p. 434] is replaced by:

	PREV	SCOPE
A1:	Λ	G4
B3:	Λ	D7
C7:	Λ	↑data0
D7:	Λ	↑data1
E3:	Λ	↑data2

	PREV	SCOPE
F3:	Λ	G4
G4:	Λ	↑data3
H1:	Λ	G9
F5:	F3	G8
G8:	G4	↑data4

	PREV	SCOPE
B5:	B3	↑data5
C5:	C7	G9
E9:	E3	↑data6
D9:	D7	↑data7
G9:	G8	↑data8

3. Changes to the Algorithms

Every use of atomic `SCOPE` or `CHILD` changes but only Algorithm C is affected. So the aggregate time won't increase much because Algorithm C is used more rarely than A or B; in contrast, the compressed table may increase locality and save memory-access time.

Algorithm A [p. 429]: A `CHILD` pointer is always initialized to Λ at Step A6. Sometimes it is immediately updated while processing the next line at A4. When it is not, the terminal data should be newly installed.

Algorithm C [p. 431]: There is an inexpensive change to the algorithm used to find corresponding sets of data. In Step C2, instead of checking whether or not the `CHILD`$= \Lambda$, one should check whether it is $\uparrow$`data` or not. This is done by checking the tag bit or comparing its relative magnitude.

Dahm's Algorithm A [§2.4–12]: Atomic data is still stored at every leaf, now either in a tagged `SCOPE` field or in a different table.

Dahm's Algorithm C [§2.4–14]: Dahm's representation of the Data Table requires the use of a new algorithm for C [§2.4–14, p. 607]. In Step C2 change the conditional statement from `if SCOPE(P) = P or SCOPE(Q) = Q` to `if SCOPE(P) =` $\uparrow$`data or SCOPE(Q) =` $\uparrow$`data`. Similarly, the comparisons in C3 and C4 must detect $\uparrow$`data` as `SCOPE(Q)` or `SCOPE(P)`; if present, interpret these values as `Q` and `P`, respectively.

4. Conclusion

This optimization saves space for atomic information in every node of the Data Table, especially in non-terminal nodes where such information is unnecessary. When it fits in the tagged `SCOPE` field, a 33% space savings is realized. Even when the information must be moved to a new table, the space required for leaves is unchanged while the space for all non-terminal nodes shrinks.

Since the space for the Data Table *always* shrinks, the times for Algorithms A, B, and C will always improve as more of the Data Table fits in high-speed cache. This time improvement might be dramatic enough to allow the parent of `P` to be identified by serially scanning `P1` up the table until `SCOPE(P1)` $\geq$ `P`; that is, the stack of §2.4–14 might also be avoided.

While the problem was originally presented as a COBOL compiler, this data compression can be used elsewhere. Any application that uses a static tree with information only in leaves can use preorder sequential allocation [p. 349], Dahm's `SCOPE`s [§2.3.3(5)], and the compression described here to squeeze out any `INFO` fields.

Acknowledgements

Supported, in part, by the National Science Foundation under a grant numbered CCR-9711269.

References

[1] D. E. Knuth. *The Art of Computer Programming* **I,** *Fundamental Algorithms* (3rd ed.), Reading, MA: Addison–Wesley, (1997).