A Sensitive Order Notation for Performance Expression[1]

Joan Krone, Mathematics and Computer Science, Denison University, Granville, OH 43023.

William F. Ogden, Computer and Information Science, Ohio State University, Columbus, OH 43210.

Murali Sitaraman, Computer Science, Clemson University, Clemson, SC 29634.

Performance of algorithms is expressed typically in terms of the sizes of parameters using big-O notation to compare functions over natural numbers. We present a more general notation that allows the domains of the functions used to document performance to be arbitrary mathematical spaces instead of natural numbers. The generality permits presentation of more sensitive performance expressions naturally in terms of arbitrary values of objects without a need for metricization. The proposed notation is useful in describing performance of generic components and subtle variations in performance of common algorithms, such as sorting. It is compositional and it is compatible with standard big-O notation.

Keywords – analysis of algorithms, formal methods, performance evaluation, program correctness, programming languages, program specification, software engineering, and specification languages.

1. A SENSITIVE ORDER NOTATION

The standard approach to express the running times of algorithms is in terms of input sizes. Using the big-O notation, an algorithm that takes time f(n) on input size n, can be documented to take time O(g(n)) where the domains of f and g are the set of natural numbers. When the input to an algorithm is a number, then the big-O notation is both adequate and appropriate. In object-oriented computing, however, the inputs to computations are often complex container types whose entries are complex types themselves. For an example, consider the time to copy a list of arbitrary entries, such as trees. If the

[1] This work is supported in part by the National Science Foundation under grants CCR-0081596 and CCR-0113181. Correspondence: krone@denison.edu

performance is expressed as O(n) as is typically done in terms of n – the size of the list – then the expression is inadequate. This is because a list containing a few large trees may take much longer to copy than a much longer list containing empty trees. To provide more meaningful expressions using big-O notation, it is necessary to define a metricizing function that converts complex objects such as lists of trees of arbitrary information to a number. It is useful to have a notation that allows performance to be expressed using functions on arbitrary mathematical spaces that characterize the abstract values of programming objects. A straightforward generalization of the big-O notation[2] to functions over arbitrary spaces is not possible, because the definition relies on natural number ordering to define an exclusionary set. We propose the following more sensitive definition for performance expression for computations over non-trivial objects whose value spaces are arbitrary.

Definition: Given f, g: Dom $\rightarrow \mathbb{R}$, f(x) is O(g(x)) iff ($\exists$ A: $\mathbb{R}^{>0}$, $\exists$ H: $\mathbb{R}$ $\ni$ $\forall$ x: Dom, $f(x) \leq A \cdot g(x) + H$).

For two timing functions f and g mapping a computational domain *Dom* to the real numbers, to say that f(x) is O(g(x)) is to say that there is some positive acceleration factor A and some handicap H such that for every domain value x, $f(x) \leq A \cdot g(x) + H$. If we think of f and g as representing competing processes, f being big O of g means that f is not essentially faster than g. If g is run on a processor A times faster than f's processor and also given a head start H, then g will beat f on all input data x.

The new notation is upward compatible with big-O in [1], and for problems where size-oriented performance expressions are sufficient, the new notation yields the same expressions. In an electronic appendix, we prove that the new definition has desirable compositional properties. We have defined corresponding notations for Ω, θ, and o, and established their properties as well. The rest of the paper

[2] The big-O relation between two natural number functions is defined in [1]: Given f, g: $\mathbb{N} \rightarrow \mathbb{R}$, f(n) is O(g(n)) iff $\exists$ positive constants c and n_0 $\ni$ $f(n) \leq c \cdot g(n)$ whenever $n \geq n_0$.

illustrates the use the new notation considering list copying and sorting (whose performance can vary from linear to quadratic depending on the permutation of the entries in the initial list).

2. DAT ABSTRACTION MOTIVATION FOR GENERALIZING BIG-O RELATION

To motivate performance analysis when objects take their values from different mathematical spaces, we begin with a formal specification of a List component. The specification in Figure 1 is given in the RESOLVE notation [2], though any model-based specification language can be used. Specification of different abstract data types may use other standard mathematical models such as integers, sets, functions, and relations, in addition to tuples and strings.

The List component provides a List type and operations to manipulate objects of List type. It is parameterized by the type of entries in a list. We can view a List abstractly as a pair of strings over the type of entries in the list, denoted by the cross product space Str(Entry) X Str(Entry) in the figure. Here, the first string contains the list entries preceding the current position and it is named *Prec*; the second string is the remainder of the list, *Rem*. Initially, a List is a pair of empty strings, i.e., (Λ, Λ). In the specification of an operation, the **requires** clause, when present, specifies an obligation for the caller. The **ensures** clause is a guarantee from a correct implementation, and it describes how the list *P* is updated. Here, *#P* denotes the incoming value of *P* and *P* denotes the outgoing value.

Conceptualizing a List object as a pair of strings makes it easy to explain insertion and removal from the "middle". For example, suppose that (<3, 5>, <4, 1, 9>) is the abstract value of an Integer list. When the Insert operation is called to a add 7, the new element is inserted at the beginning of the *Rem* string of the list and the resulting list becomes (<3, 5>, <7, 4, 1, 9>). A call to the Remove operation reverts the list to (<3, 5>, <4, 1, 9>). Neither of these operations changes the insertion position. To change the position, Advance or Reset needs to be called. Advance moves the list position so that the list

(<3, 5>, <4, 1, 9>) becomes (<3, 5, 4>, <1, 9>). Reset sets the insertion point to the beginning by making the list (Λ, <3, 5, 4, 1, 9>). Neither of these operations changes the overall contents of the list. Since the operations all have easy specifications in terms of the mathematical model, and since the underlying linking pointers in the implementation are cleanly hidden, reasoning about client code is much simplified with this abstract model [2].

```
Concept List_Template (type Entry);
    uses Number_Theory, String_theory;

    Type List ⊆ Prec: Str(Entry) X Rem: Str(Entry);
        exemplar  P
        initialization ensures |P.Prec| = 0 and |P.Rem| = 0;

      Operation Insert ( alters E: Entry; updates P: List);
        ensures P.Prec = #P.Prec and P.Rem = <#E> ◦ #P.Rem;

      Operation Remove ( replaces R: Entry; updates     P: List );
        requires |P.Rem| > 0;
        ensures P.Prec = #P.Prec and #P.Rem = <R> ◦ P.Rem;

      Operation Advance ( updates P: List );
        requires |P.Rem| > 0;
        ensures P.Prec ◦ P.Rem = #P.Prec ◦ #P.Rem and |P.Prec| = |#P.Prec| + 1;

      Operation Reset …
      Operation Advance_To_End …
      Operation Prec_Length …
      Operation Rem_Length …
end List_Template;
```

Figure 1: Specification of a List Sort Operation

Using the List abstraction and the new notation, the time to copy a list containing arbitrary entries can be expressed naturally as $O(\sum_{E:\text{Entry}} \text{Occurs\_Ct}(E, P.\text{Prec} \circ P.\text{Rem}) \cdot \mathbf{Dur}_{\text{Copy\_Entry}(E)})$ where the domain of the function is the cross product of the mathematical space of list objects (i.e., Str(Entry) X Str(Entry)) and the time to copy an arbitrary entry (i.e, a function from Entry to Real numbers). In the expression *Occurs_Ct* denotes the number of occurrences of a particular entry in a given string. The expressive power necessary for this example is typical of what is needed to express the performance of typical computations on generic objects with mathematical abstractions.

3. EXAMPLE ANALYSIS

The second example illustrates how the new notation makes it possible to capture the sensitivity needed for more precise expressions that depend on values of objects, not just sizes. We consider an algorithm for sorting a “list”, though most other algorithms discussed in data structures courses raise similar issues. The running time complexity of the insertion sort procedure is given usually as O (n^2) where n is the number of elements in the list. If the expression is based merely on the length of the list, it cannot make any distinction between the sorting times for different lists of the same length. The time for insertion sort is linear on a sorted list, and it increases to quadratic time complexity gradually depending on how unsorted the list is. To address this issue partially, a best case estimate, such as $\Omega(n)$ is given in analyzing the algorithm. The new notation makes it more apparent that the insertion sorting takes linear time in some cases and quadratic time in some cases, showing a direct correspondence between the particular permutation of entries in a list and the running time of insertion sort.

Figure 2 contains the specification of an operation to sort a list. The operation is specified to reset the List, and guarantee that the remaining part of the list is sorted and is a permutation of the entire incoming list which is *#P.Prec* concatenated with *#P.Rem*. The definition of *Is_Ascending_Order* (or more precisely *Is_Non_Descending_Order*) should be given in terms of the definition used for sorting entries and it is omitted here due to space considerations. Figure 3 contains a procedure to implement the list sorting operation.

```
Operation Sort_List( updates P: List );
    ensures P.Prec = Λ and In_Ascending_Order( P.Rem ) and
        P.Rem Is_Permutation #P.Prec o #P.Rem;
```

Figure 2: Specification of a List Sort Operation

```
Procedure Sort_List( updates P: List );
```

```
    Var P_Entry, S_Entry: Entry;
    Var Sorted: List;
    Reset ( P );
    While Length_of_Rem( P ) ≠ 0 do
        Remove( P_Entry, P );
        Iterate
            When Length_of_Rem( Sorted ) = 0
                do exit;
            Remove( S_Entry, Sorted );
            When Lss_or_Comp( P_Entry, S_Entry )
                do Insert( S_Entry, Sorted ) exit;
            Insert( S_Entry, Sorted );
            Advance( Sorted );
        repeat;
        Insert( P_Entry, Sorted );
        Reset( Sorted );
    end;
    P :=: Sorted;
end Sort_List;
```

Figure 3: Insertion Sort Procedure

The code in Figure 3 uses a local list *Sorted*. Both the inner and outer loops maintain the invariant of keeping the elements in the *Sorted* list sorted. We have omitted a formal statement of the invariant in the figure because it is not necessary for the present discussion, though it is useful to present it at least at an informal level. The strategy of the outer loop is to place successive values *P_Entry* from the list *P* into their proper place in the list *Sorted*. The **Iterate**…**repeat** construct is a "for ever" loop that is terminated when one of the exit conditions is satisfied. The task of the inner loop is to position the *Sorted* list appropriately for the insertion of *P_Entry*. In the inner loop, *Lss_or_Comp* is a generic procedure to compare two entries. In the last statement of the procedure, a swap statement, denoted by :=:, is used to transfer the result from the *Sorted* list back to the parameter *P*, the list that is to be sorted.

Swap operator can be implemented in constant time by exchanging references, without deep copying and without introducing aliasing.

To analyze the efficiency of the insertion sorting algorithm, we examine first the inner loop and then the outer loop to get a duration estimate for each. As is the case in the classical performance analysis, we assume that all List operations and the *Lss_Or_Cmp* operation to compare entries take a constant time, though this assumption is not necessary when using the revised notation. Based on the assumption, in classical analysis we note that the outer loop may execute at most n times (where n is the length of the input list). The inner loop might execute a maximum of n times if the next entry that is to be inserted into the sorted list needs to be compared with all the previously inserted entries into that list. This leads us to an overall worst-case complexity of $O(n^2)$ for the insertion sort procedure.

For a more precise analysis, note that whenever the inner loop is entered, the preceding string of the Sorted list is empty because it is reset in the outer loop. The inner loop compares each entry *S_Entry* in the remaining string of the sorted list with the entry *P_Entry*. After comparison, the entry is inserted back into the list and the list is advanced. The time for the inner loop depends upon the number of entries to be compared with the next item before finding the correct place for insertion, in particular, the number of entries in Sorted list that are less than or comparable to *P_Entry*. To get the increased precision, we need to define a function on strings of entries α that counts how many entries in α are less than an entry *E* and hence would be “advanced” over when positioning *E* after α has been sorted[3]. For this reason, we define a *Rank(E,* α *)* function and states some of its properties.

Inductive definition on α: Str(Entry) **of**

Rank(E: Entry, α): $\mathbb{N}$ **is**

(i) Rank(E, Λ) = 0;

[3] We have chosen to count the number of calls to Advance operation at the end of the inner loop rather than the number of calls to the comparison operation. In order of magnitude analysis, however, this does not make any difference.

(ii) Rank(E, α ∘ <D>) = $\begin{cases} \text{Rank}(E, \alpha) + 1 & \text{if } D \prec E \\ \text{Rank}(E, \alpha) & \text{otherwise} \end{cases}$;

In the definition "∘" denotes concatenation and ≺ denotes a generic notion of "less than" on type Entry. Given this definition, it is easy to see that the duration for the inner loop is proportional to *Rank(P_Entry, Sorted.Rem)* at the beginning of the loop (or *Rank(P_Entry, Sorted.Prec)* at the end of the loop). Rank counts the number of calls to *Advance* that are necessary to position the list properly. The outer loop inserts the next entry into the *Sorted* list. Since the time of the outer loop depends upon the cumulative effect of positioning successive entries from the list *P* into the *Sorted* list, we define a "preceding rank" function *P_Rank(α)*. For an example, suppose that we are sorting a list with the abstract value (Λ, <3, 5, 4, 1, 9>) in the ascending order. After two iterations of the outer loop, the *Sorted* list becomes (Λ, <3, 5>). The time to insert the next entry 4 depends upon the "rank of 4" in the string <3, 5>. To compute the cumulative time to add each entry, we define *P_Rank* as below:

Inductive def. on α: Str(Entry) **of** P_Rank(α): ℕ **is**
(i) P_Rank(Λ) = 0;
(ii) P_Rank(α ∘ <E>) = P_Rank(α) + Rank(E, α);

Using the formula *P_Rank*, we can see that, it takes a lot less time to sort the list (Λ, <9, 5, 4, 3, 1>) in ascending order than the list (Λ, <1, 3, 4, 5, 9>), though the two lists have the same 5 entries. In the first case, no advances are necessary. In the second case, 10 advances are necessary. Using the new notation, we can give a precise estimate of the procedure as O(Max(|#P.Prec ∘ #P.Rem|, P_Rank(#P.Prec∘#P.Rem)), where the domain of the function is the mathematical space of lists (i.e., Str(Entry) X Str(Entry)). The maximum is necessary is this expression, because even when no advances are necessary, the insertion sort procedure given in Figure 3 takes at least linear time because every entry is moved from the initial list to the sorted list in the process.

An important theorem about *P_Rank* is that $P\_Rank(\alpha) \le |\alpha|\cdot(|\alpha|-1)/2$, so it follows that the duration of the *Sort_List* procedure can be expressed using a natural number function more classically, as: $O(\ |\#P.Prec \circ \#P.Rem|^2\ )$. Thus the new notation can be used to specify the much less exacting estimate based on sizes if we wish. But it allows more sensitivity when we need a sharper estimate is needed. Of course, in order to use this definition, it is necessary to have mathematical support in the form of theorems about the proposed definition. For example, we have proved the additive property below (in the electronic Appendix) so we can apply the analysis to a succession of operation invocations:

Theorem OM1: If $f_1(x)$ is $(O\ g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then $f_1(x) + f_2(x)$ is $O(Max(g_1(x), g_2(x)))$.

4. DISCUSSION

Big-O notation is designed to deal with comparison of functions whose domains are natural numbers. It is necessary to generalize the notation to functions over arbitrary mathematical spaces, such as those might be used in specification of generic, abstract data type objects. The generality of the proposed notation is especially useful in the context of reusable components [3], where the context of use is not known in advance, and hence, performance needs to be expressed making few assumptions about component usage. Performance specifications are necessary for components to reason about the performance of component-based systems in a modular fashion. To avoid the rapid compounding of imprecision that otherwise happens in such systems, it is also essential to use high precision performance specification mechanisms, such as the one presented here. A significant benefit from an educational perspective is that the new notation permits a better understanding of the underlying algorithms. For example, understanding the performance estimate for insertion sorting given in this paper clarifies for a student how it is intrinsically different from a bubble sorting algorithm, though both algorithms have

$O(n^2)$ worst case complexity. The direct style of performance specification is much more natural than using an intermediary natural number.

5. REFERENCES

[1] Aho, A., Hopcroft, J., Ullman, J., *Data Structures and Algorithms,* Addison-Wesley, 1983.

[2] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E., “Reasoning About Software-Component Behavior,” in Frakes, W.B., ed., *Software Reuse: Advances in Software Reusability (Proceedings Sixth International Conference on Software Reuse),* Springer-Verlag LNCS 1844, 2000, 266-283.

[3] Sitaraman, M., Krone, J., Kulczycki, G., Ogden, W., and Reddy, A. L. N., “Performance Specification of Software Components,” *ACM SIGSOFT Symposium on Software Reuse*, May 2001.

[4] J. M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 29(9), Sep. 1990, 8-24.