

Abstract OO Big O

Joan Krone
Denison University
Department of Math and CS
Granville, Ohio 43023
740-587-6484
krone@denison.edu

W. F. Ogden
The Ohio State University
Neal Avenue
Columbus, Ohio 43210
614-292-6007
ogden@cis.ohio-state.edu

SUMMARY

When traditional Big O analysis is rigorously applied to object oriented software, several deficiencies quickly manifest themselves. Because the traditional definition of Big O is expressed in terms of natural numbers, rich mathematical models of objects must be projected down to the natural numbers, which entails a significant loss of precision beyond that intrinsic to order of magnitude estimation. Moreover, given that larger objects are composed of smaller objects, the lack of a general method of formulating an appropriate natural number projection for a larger object from the projections for its constituent objects constitutes a barrier to compositional performance analysis.

We recast the definition of Big O in a form that is directly applicable to whatever mathematical model may have been used to describe the functional capabilities of a class of objects. This generalized definition retains the useful properties of the natural number based definition but offers increased precision as well as compositional properties appropriate for object based components. Because both share a common mathematical model, functional and durational specifications can now be included in the code for object operations and formally verified. With this approach, Big O specifications for software graduate from the status of hand waving claim to that of rigorous software characterization.

Categories and Subject Descriptors

D.2[Software Engineering], F.2[Analysis of Algorithms], F.3[Logics and Meanings of Programs]: Specifications, Models, Semantics – *performance specifications, performance analysis, performance proof rules.*

General Terms

Algorithms, Performance, Verification.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference '00, Month 1-2, 2000, City, State.
Copyright 2000 ACM 1-58113-000-0/00/0000...\$5.00.

Keywords

Performance, Formal Specification, Verification, Big O.

1. INTRODUCTION

The past forty years have seen a great deal of work on the rigorous specification and verification of programs' functional correctness properties [2] but relatively little on their performance characteristics. Currently performance "specifications" for programs commonly consist of reports on a few sample timings and a general order of magnitude claim formulated in a Big O notation borrowed from number theory. As we have discussed elsewhere [6], such an approach to the performance of reusable components is no more adequate than the test and patch approach is to their functionality.

As with functionality, problems with performance usually have their roots in the design phase of software development, and it's there that order of magnitude considerations are most appropriately encountered. This means our order of magnitude notations are generally applied in a somewhat rough and ready fashion (which is probably why the deficiencies of our current ones have escaped notice for so long). However, if their formulation doesn't reflect the ultimate performance of the components under design accurately and comprehensibly, then marginal designs become almost inevitable. So the way to get an appropriate order of magnitude definition is to formulate one that meshes smoothly with program verification.

With the advent of object oriented programming and a component based approach to software, formal specifications of a component's functionality are considered to be critical in order for clients to make good choices when putting together a piece of software from certified components.

To meet the need for reasoning about performance as well as functionality, we introduce a new object appropriate definition of Big O. Object Oriented Big O, or OO Big O for short, allows one to make sensitive comparisons of running times defined over complex object domains, thereby achieving much more realistic bounds than are possible with traditional big O.

We cast our approach in a framework that includes an assertive language with syntactic slots for specifying both functionality and performance, along with automatable proof rules that deal with both. Equally important is the need for the reasoning to be fully modular, i.e., once a component has been certified as

correct, it should not be necessary to reverify it when it is selected for use in a particular system.

Our approach is based on the software engineering philosophy that a component should be designed for reuse and thus include a general mathematical specification that admits several possible implementations – each with different performance characteristics [3, 7]. Of course, in order for a component to be reusable, it should include precise descriptions of both its functionality and its performance, so that prospective clients can be certain they are choosing components that fit their needs.

It is also important that all reasoning about constituent components – including reasoning about performance – be possible without knowing implementation details. In fact, if one is using generic components, it should be possible to reason about those components even before details such as entry types to a generic container type, are available.

2. OO Big O Definition

The traditional Big O is a relation between natural number based functions defined in the following way:

Given $f, g: \mathbb{N} \rightarrow \mathbb{R}$, $f(n)$ is $O(g(n))$ iff \exists constants c and n_0 such that $f(n) \leq c \cdot g(n)$ whenever $n \geq n_0$. A program whose running time is $O(g)$ is said to have growth rate $g(n)$ [1].

When the object classes central to modern programming are formally modeled, they present to clients a view of objects that could come from essentially arbitrary mathematical domains, since the point of introducing objects is to simplify reasoning about the functionality of components by providing a minimalist explanation that hides the details of their implementation. But the functionally simplest models may well have little to do with natural numbers. So the natural expression of the duration $f(x)$ of an operation on object x is as a function directly from the input domain used to model x to the real numbers. Clearly any gauging function g that we might want to use as an estimate for f should have this same domain. Accordingly, the Is_O relation between functions ($f(x) Is_O g(x)$) is defined by:

Definition: ($f: Dom \rightarrow \mathbb{R}$) Is_O ($g: Dom \rightarrow \mathbb{R}$): $B = (\exists A: \mathbb{R}^{>0}, \exists H: \mathbb{R} \exists \forall x: Dom, f(x) \leq A \cdot g(x) + H)$.

In other words, 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 g is not essentially faster than f . If f is run on a processor A times faster than g 's processor and also given a head start H , then f will beat g on all input data x .

Of course, in order to use this definition, it is necessary to have mathematical support in the form of theorems about the revised definition of Is_O . For example, we need an additive property so we can apply our 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 $f_1(x) + f_2(x) Is_O \text{Max}(g_1(x), g_2(x))$.

A development of appropriate theorems and definitions appears in [5].

3. ABSTRACT OBJECTS

If you want to produce rigorously specified and verified software components that support genericity, facilitate information hiding, and can be reasoned about in a modular fashion, it is necessary to adhere carefully to certain guidelines and principles [7].

A simple example of a general purpose component concept that can be used to make clear the need for a new definition of Big_O is the one that captures the “linked list.” Because one of our guidelines is to tailor a concept to simplify the client’s view, we call this concept a one-way list template and the objects it provides list positions. We describe list positions mathematically as pairs of strings over the entry type. The first string in a list position contains the list entries preceding the current position and is named *Prec*; the second string is the remainder of the list, *Rem*. Since the operations on list position (Insert, Advance, Reset, Remove, etc.) all have easy specifications in terms of this model, and since the underlying linking pointers are cleanly hidden, reasoning about client code is much simplified with this abstract model.

```

Concept One_Way_List_Template( type Entry;
                                evaluates Max_Total_Length: Integer );
                                uses Std_Integer_Fac, String_Theory;
                                requires Max_Total_Length > 0;

Type Family List_Position  $\subseteq$  Cart-Prod
                                Prec, Rem: Str(Entry)
                                end;

M
Operation Advance( updates P: List_Position );
requires P.Rem  $\neq \Lambda$ ;
ensures P.Prec  $\circ$  P.Rem = @P.Prec  $\circ$  @P.Rem and
                                |P.Prec| = |@P.Prec| + 1;

```

Although variations in list implementation details are usually insignificant, our system allows for the possibility of a multiplicity of different realizations (implementations) for any given concept. Each **Realization**, with its own potentially distinct performance characteristics, retains a generic character, since parameter values such as the entry type for lists have yet to be specified. The binding of such parameters only takes place when a client makes a **Facility**, which involves selecting the concept and one of its realizations along with identifying the appropriate parameters.

When designing concepts for maximal reusability, our guidelines prescribe that only the basic operations on a class of objects should be included, so for lists we only include Insert, Advance, etc., but not Search, Sort, etc. In order to have a rich enough Big O example, we will consider such a sorting operation, so we need to employ the **Enhancement** construct used to enrich basic concepts such as the one-way list.

Well-designed enhancements also retain to the extent possible the generality we seek in our concepts, but often they do add constraints that prevent their use in certain situations. Providing a Sort_List operation, for example, requires that list entries possess an ordering relation \preceq , so certain classes of entries would be precluded from lists if Sort_List were one an operation in the basic list concept.

EXAMPLE APPLICATION OF BIG O

The enhancement's name here is *Sort_Capability*, and it maintains the generic character of the concept (which allows entries to be of arbitrary type) by importing an ordering relation \preceq on whatever the entry type may be. A **requires** clause insists that any imported \preceq relation actually be a total preordering on whatever the entry type is.

The **uses** clause indicates that this component relies on a mathematical theory of order relations for the definitions and properties of notions such as total preordering. Note that an automated verifier needs such information.

Enhancement Sort_Capability(**def. const** (x: Entry) \preceq (y: Entry): B

);
 for One_Way_List_Template;
 uses Basic_Ordering_theory;
 requires Is_Total_Preordering(\preceq);
Def. const In_Ascending_Order(α : Str(Entry)): B =
 (\forall x, y: Entry, **if** $\langle x \rangle \circ \langle y \rangle$ Is_Substring α , **then** $x \preceq y$);
Operation Sort_List(**updates** P: List_Position);
 ensures P.Prec = Λ **and** In_Ascending_Order(P.Rem) **and**
 P.Rem Is_Permutation @P.Prec \circ @P.Rem;
end Sort_Capability;

A client who wishes to order a list would be able to choose this list enhancement on the basis of these functional specifications. However, before choosing among the numerous realizations for it, a client should be able to see information about their performance. Rather than giving such timing (**duration**) information a separate ad hoc treatment, we introduce syntax for formally specifying **duration** as part of each **realization**. In short, we associate with every component not only a formal specification of its functionality but of its performance as well, so that a potential client can choose a component based on its formal specifications rather than on its detailed code.

To see how the new Big O definition can improve performance specifications, we consider an insertion sort realization for the Sort_List operation.

Because a realization for a concept enhancement relies upon the basic operations provided by the concept, its performance is clearly dependent on their performance, and that can vary with the realization chosen for the concept. Fortunately performance variations for a given concept's realizations seem to cluster into parameterizable categories, which we can capture in the **Duration Situation** syntactic slot. The normal situation for a one-way list realization, for example, is that the duration of each operation Is_O(1). Of course realizations of lists with much worse performance are possible, but we wouldn't ordinarily bother to set up a separate duration situation to support analyzing their impact on our sort realization.

Duration situations talk about the durations of supporting operations such as the Insert and Advance operations by using the notation **Dur**_{Insert}(E, P), **Dur**_{Advance}(P), etc. So we can use our Big O notation to indicate that the performance estimates labeled "normal" only hold when **Dur**_{Insert}(E, P) **Is_O** 1, etc.

Realization Insertion_Sort_Realiz(
 Oper Lss_or_Comp(**restores** E1, E2: Entry): Boolean;
 ensures Lss_or_Comp = (E1 \preceq E2);)

for Sorting_Capability;

Duration Situation Normal: **Dur**_{Insert}(P) **Is_O** 1 **and**
Dur_{Advance}(P) **Is_O** 1 **and** **Dur**_{=(i, j)} **Is_O** 1 **and** Λ

Inductive def. on α : Str(Entry) **of const**

Rank(E: Entry, α): \mathbb{N} **is**

(i) Rank(E, Λ) = 0;

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

M

Inductive def. on α : Str(Entry) **of const** P_Rank(α): \mathbb{N} **is**

(i) P_Rank(Λ) = 0;

(ii) P_Rank(ext(α , E)) = P_Rank(α) + Rank(E, α);

Theorem IS6: $\forall \beta$: Str(Entry), P_Rank(β) $\leq |\beta| \cdot (|\beta| - 1)/2$;

Def. const Len(P: List_Position): \mathbb{N} = (|P.Prec \circ P.Rem|);

Proc Sort_List(**updates** P: List_Position);

Duration Normal:

Is_O Max(Len(@P), P_Rank(@P.Prec \circ @P.Rem));

Var P_Entry, S_Entry: Entry;

M

While Length_of_Rem(P) \neq 0

affecting P, P_Entry, Sorted, S_Entry, Processed_P;

maintaining Sorted.Prec = Λ **and**

In_Ascending_Order(Sorted.Rem) **and**

Processed_P.Prec \circ P.Rem = @P.Prec \circ @P.Rem **and**

Sorted.Rem Is_Permutation Processed_P.Prec;

decreasing |P.Rem|;

elapsed_time Normal: **Is_O** P_Rank(Processed_P.Prec)

+ |Processed_P.Prec|;

do

Remove(P_Entry, P);

M

For each loop, we record the loop invariant that is used to establish the functional effect in its the **maintaining** clause, while the progress metric used to prove termination is recorded in the **decreasing** clause. The **elapsed_time** clause is used to specify on each pass through the loop how much time has elapsed since the loop was entered, which can vary according to the named situation ("Normal" in our example). If an elapsed time clause begins with the **Is_O** token, then the verifier must establish that the actual elapsed time function Is_O of the gauge function specified by the subsequent expression.

The portion of our proof rules that deals with verifying duration estimates for loops accomplishes its objective by checking that the duration of each loop body Is_O of the difference between the value of the gauge function at the end of the loop body and its value at the beginning.

Clearly the elapsed time of an insertion sort depends heavily upon the order of the elements in the original list @P, but traditional natural number based Big_O analysis would require that we project the @P list onto a natural number "n" and express our gauge function in terms of that n (e.g. n³). Typically that n would be the length of a list (what we've formally defined as Len(P) so that n = Len(@P)). Since Len(@P) is totally insensitive to the order of the entries in @P, we could at best end up with a duration estimate for Sort_List of n².

To exploit the increased precision of the OO Big O definition, 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 skipped over when positioning E after α has been sorted, and that's why our realization includes the definition of the $\text{Rank}(E, \alpha)$ function. Since the elapsed time of the outer loop depends upon the cumulative effect of positioning successive entries in $@P$, we also need to define a "preceding rank" function $P_Rank(\alpha)$.

Using these definitions, we can express **elapsed time** bounds for the two loops in the code and the overall **Duration** bound:

$$\text{Max}(\text{Len}(@P), P_Rank(@P.Prec @P.Rem)).$$

Now one of the theorems about P_Rank is that $P_Rank(\alpha) \leq |\alpha| \cdot (|\alpha| - 1) / 2$, so it follows that $\text{Dur}_{\text{Sort_List}}(P) \text{ Is_O } \text{Len}(P)^2$ too, and we can get the much less exacting estimate produced by traditional Big O analysis if we wish. We're just not forced to when we need a sharper estimate. Another point to note is that besides being compatible with correctness proofs for components, the direct style of performance specification is much more natural than the old style using the often ill defined "n" as an intermediary.

4. THE CALCULUS FOR OO BIG O

Our `Sort_List` example illustrates how we can use the new Big O notation in performance specifications and indicates how such specifications could fit into a formal program verification system. The success of such a verification system depends upon having a high level calculus for Big O that allows verification of performance correctness to proceed without direct reference to the detailed definition of Big O.

Of course making such a calculus possible is one of the primary motivations for the new Big O definition, and in [4] we have developed a number of results like the earlier theorem OM1 to support this calculus. Another simple illustration of a property of the new Big O important for verification is dimensional insensitivity.

Theorem OM2: If $f(x) \text{ Is_O } g(x)$ and $F(x, y) = f(x)$ and

$$G(x, y) = g(x), \text{ then } F(x, y) \text{ Is_O } G(x, y).$$

Taken together, these results must justify both the proof rules for our program verification system and the expression simplification rules for the resulting verification conditions.

5. CONCLUSION

A critical aspect of reusable components is assured correctness, an attribute attainable only with formal specifications and an accompanying proof system. Here, we claim that while functional correctness is absolutely necessary for any component that is to be reused, it is not sufficient. Reusable components need formally specified performance characteristics as well.

Traditional Big O order of magnitude estimates are inadequate because they deal only with the domain of natural numbers and offer no support for modularity and scalability.

If we want to design software components that can be reused, such components must have formal specifications for both

functionality and performance associated with them and there must be a proof system that addresses both. Moreover, to avoid intractable complexity, it must be possible to reason about these components in a modular fashion, so that one can put together hierarchically structured programs, each part of which can be reasoned about using only the abstract specifications for its constituent parts. 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 OO Big O.

To develop maximally reusable components, it is necessary to be able to reason about them in a generic form, without knowing what parametric values may be filled in when the component is put into use.

OO Big O satisfies all these criteria, supporting complete genericity, performance analysis of programs over any domain, and modular reasoning.

6. REFERENCES

1. Aho, A., Hopcroft, J., Ullman, J., *Data Structures and Algorithms*, Addison-Wesley, 1983.
2. de Roever, W., Engelhardt, K. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, Cambridge University Press, 1998.
3. Krone, "The Role of Verification in Software Reusability." Dissertation, The Ohio State University, 1988.
4. J. Krone, W. F. Ogden, and, M. Sitaraman, *Modular Verification of Performance Constraints*, Technical Report RSRG-03-04, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, May 2003, 25 pages.
5. Ogden, W. F., *CIS680 Coursenotes*, Spring 2002.
6. 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.
7. Weide, B., Ogden, W., Zweben, S., "Reusable Software Components," in M.C. Yovits, editor, **Advances in Computers**, Vol 33, Academic Press, 1991, pp. 1 – 65