# Complexity Analysis of Late Binding in Dynamic Object-Oriented Languages*

Enrico Pontelli, Desh Ranjan, Gopal Gupta
Lab. for Logic, Databases, and Advanced Programming
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
{epontell,dranjan,gupta}@cs.nmsu.edu

September 12, 1999

**Abstract**

We study the algorithmic complexity of supporting *late binding* in *dynamic* object-oriented programming languages. *Dynamic* object-oriented languages (such as most Prototype-based languages and systems like CLOS) allow creation of new class definitions at runtime. *Late binding* means that procedure-call name resolution, i.e., mapping of procedure calls to method definitions, is done at run-time, rather than compile-time. Late binding is an essential feature of object-oriented programming and is present in most object-oriented languages, e.g., Java, CLOS, Smalltalk, C++ (virtual functions), etc. Name resolution for late binding is easily solved for static languages such as Java and C++; for dynamic languages, however, it is a complex problem. We propose an abstraction of the late binding problem for dynamic object-oriented languages in terms of operations on dynamic trees, and prove a time lower bound of $\Omega(\lg N)$ per operation,

*This work is an extended version of the paper "The Complexity of Late-binding in Dynamic Object-Oriented Languages" presented at the Symposium on Programming Languages, Implementations, Logics and Programs 1998, Springer Verlag.

where $N$ is the number of nodes in the tree. This result shows that efficient solutions—i.e., solutions that incur a constant time overhead per operation—of this problem are, in general, not possible. We also propose new data structures and algorithms for solving the late binding problem for dynamic object-oriented languages very efficiently, with a worst-case time complexity of $\tilde{O}(\sqrt[3]{N})$ per operation. This result is an improvement over frequently used approaches, that have a complexity of $\Omega(N)$ per operation.
**Keywords:** Complexity, Object-oriented Programming, Pointer machines.

# 1 Introduction

In this paper we study the algorithmic complexity of implementation mechanisms for supporting inheritance in *dynamic* object-oriented languages. The aim of this study is to model these implementation mechanisms in terms of operations on dynamic data structures, with the final goal of determining their complexity. Our complexity study is used to derive novel data structures for implementing inheritance mechanisms efficiently. In this work we limit ourselves to studying the complexity of implementing late binding inheritance in dynamic object-oriented systems, with particular focus on prototype-based languages and similar object-oriented systems with a dynamic class/object hierarchy. The inheritance considered is single inheritance, or multiple inheritance with linearization. Generalization of this work to more complex domains (tree and graph-based multiple inheritance) is still to be explored.

Inheritance refers to the property of object-oriented languages whereby new classes are defined by specializing existing classes. When a class D is defined as a subclass of another class B (sub-typing), it inherits all the attributes (both data and operations) from class B. *Late binding* means that procedure-call name resolution, i.e., mapping of a procedure call to a procedure definition, is done at run-time, rather than at compile-time. This is because, due to sub-typing and inheritance, it is not immediately clear which definition should be used corresponding to a procedure call, as procedure definitions may be multiply defined in a class hierarchy. Late binding is an essential feature of object-oriented programming and is present in most languages, e.g., Java, CLOS, Smalltalk, C++ (virtual functions), etc.
*Dynamic* object-oriented languages are languages that allow dynamic run-time creation of class definitions (as well as other run-time modifications of

the class hierarchy). Thus the complete class hierarchies are not known at compile-time or at any stage preceding the actual program execution. CLOS [20] and Smalltalk are two such languages. Additionally, prototype based languages [3, 5, 33] are also examples of such languages. Name resolution in presence of late binding is easily solved for static object-oriented languages (such as Java and C++) since the complete class hierarchy is known at compile-time; simple table lookups allow for constant time name resolution. For dynamic languages, due to absence of this information, it is not as easy. To study the complexity of the problems that arise in implementing name resolution in late binding dynamic object-oriented languages, we first abstract it and formalize it in terms of operations on *dynamic trees* (i.e., trees that grow and shrink with time). This formalization permits us to study the implementation problems rigorously and abstractly.

The computing model that we choose for our complexity study is the *Pointer Machine* [21, 22, 31, 35, 31, 2]. The main reason we choose pointer machines, instead of the more commonly used RAMs, is to achieve a finer level of analysis; the RAM model hides the actual cost of arithmetic operations, by allowing operations on numbers of size up to $\lg n$ ($n$ being the size of the input to the problem) to be treated as constant time operations. Pointer machines make these (relevant) costs explicit. Indeed, many lower bound results have been developed using the pointer machine model [4, 23, 35, 28]. Furthermore, pointer machines provide the operations needed for expressing computations over dynamic data structures. Besides, use of pointer machines enables us to use many existing results [23, 14, 17, 24, 27] which are useful in our study and have been developed on pointer machines.

Analysis of the name resolution problem in late binding dynamic languages on pointer machines gives us an insight into the sources of its implementation complexity. This can also be used to provide better solutions on different computational models (e.g., RAM). Most of the results proposed here can be transferred to other computational models; relationships between these two models have been studied elsewhere [31, 2] and are outside the scope of this paper.

Based on our study, we develop a lower bound time complexity of $\Omega(\lg N)$ per operation (where $N$ is the number of nodes in the tree/class hierarchy), for the name resolution problem in late binding dynamic object-oriented languages. This lower bound implies that the implementation of name resolution (creation, maintenance, member look-up, etc. of a class hierarchy) in late binding dynamic object-oriented languages will at least incur cost proportional to $\lg N$, where $N$ is the number of classes in a hierarchy. In particular, this means that *all the operations needed to support dynamic look-up in a class hierarchy cannot be made constant-time*. We believe this is an important result as it clarifies the limitations of any implementation mechanism devised for this problem. It also shows that the penalty for allowing dynamic behavior in the language cannot be removed. The result is not as straightforward as it may seem and, to our knowledge, has not been proved before.

We also propose efficient new data structures and algorithms for solving this problem which have a worst-case time complexity of $\tilde{O}(\sqrt[3]{N})$ per operation. This result is an improvement over existing approaches that typically have a complexity of $\Omega(N)$ per operation. The data structures are very practical and can be easily employed in actual implementations. Indeed the general idea resembles optimizations adopted in various object-oriented systems [1, 12, 18, 39].

The abstraction of the name resolution problem for dynamic object-oriented languages in terms of dynamic trees, the derivation of a lower bound on complexity of name resolution, and the development of an efficient, new implementation technique for it, are the main contributions of this paper. To the best of our knowledge, this is the first rigorous study of implementation of name resolution in late binding dynamic object-oriented languages. Our study is inspired by our observation that implementation mechanisms needed for supporting inheritance in dynamic object-oriented languages are similar to implementation mechanisms needed in various other programming paradigms (e.g., advanced forms of execution of logic programming [24, 27, 16], and certain Game-trees and Artificial Intelligence applications [26]). These observed similarities allow us to reuse techniques and results developed in one area to solve problems in other areas. All the results presented in this paper, to the best of our knowledge, are novel, and complement similar studies focused on more specialized languages or techniques [25, 29, 37, 1, 11]. The data structures adopted to develop our upper-bounds are also generalizations and extensions of the tabling and caching mechanisms proposed by various researchers [11, 10, 18].

We do not make any pretense to have completely solved the problems we consider. In fact, we raise numerous open questions. One of the goals of this paper is to bring these problems to the attention of both the data structures and programming languages communities, as well as to point out the similarities between the problems arising in object-oriented programming and those in different frameworks (e.g., logic programming). We hope this paper will initiate further research in the area of data structures for dynamic graphs that arise in implementation of declarative and object-oriented languages.

The rest of the paper is organized as follows. In the rest of this section we give a brief overview of object-oriented programming, along with the background information for the problems discussed in this paper. Section 2 presents an overview of the Pointer Machine model. Section 3 formalizes the problem in terms of operations on trees. The remaining sections present lower and upper bounds for the problems described. We conclude with some open problems and conjectures. The reader is assumed to be familiar with the basic concepts of object-oriented programming and with the basic notions of algorithm complexity.

## 1.1 Object Oriented Programming

In object-oriented programming solution of a problem is expressed in terms of creation and manipulation of *objects*. Each object encapsulates a data structure (a collection of *data fields* or *variables*) and its behavior (a collection of *member functions* or *methods*). We will use the general term *attributes* to refer to the components of an object, namely, its data fields and member functions. Objects are organized into *classes* (collections of objects with common properties) and classes of objects are organized into hierarchies (subclasses and superclasses). Powerful mechanisms, like inheritance, allows definition of classes/objects as refinements of other existing classes/objects. Hierarchies of classes can assume various formats, depending on the flexibility allowed by the language; e.g., they can be represented as a single tree (single rooted, single inheritance), as multiple trees (multi-rooted, single inheritance), or as arbitrary complex Direct Acyclic Graphs (DAGs) (multiple inheritance). Objects consist of instance variables (which define the state of the object) and methods (which define the behavior of the object). The structure of an object is defined in terms of the class it belongs to; furthermore the definition of each class is a result of a combination of the components of all its superclasses (via inheritance). Thus the presence of an attribute $\beta$ in an object of class $C$ depends on the presence of $\beta$ in $C$ or in any of its super-classes. Attributes can get redefined at different stages of the hierarchy, e.g., $\beta$ in class $C_1$ can be redefined in a subclass $C_2$.

Program execution takes place through execution of methods associated to the various objects—one object can request execution of a method associated with another.

## 1.2 Search Problems in Object-Oriented Languages

From the previous discussion it follows that determining the structure of an object (i.e., its attributes) requires a *search.* A static system where the hierarchy is statically defined allows this search to be completely performed at compile-time. Whenever a class definition is processed, the hierarchy can be studied to determine exactly all the variables and all the methods that characterize each instance of such class.

In more complex object oriented system, such as those that allow dynamic creation of classes and/or late binding, part of this search has to be performed at run-time. This search can be more complex, as the look-up table, or its equivalent, may have to be constructed at runtime since the structure of the class hierarchy may be modified at runtime. In the next subsections we illustrate situations where more complex search operations are required to determine attributes of a given object.

### 1.2.1 Late Binding and Dynamic Object-Oriented Languages

Self-reference and late binding are considered two essential components of the inheritance mechanism in an object-oriented system; in fact, various researchers [7, 8, 34] define inheritance as *"...an incremental modification mechanism in the presence of a late-bound self-reference"* [34]. The presence of polymorphism in object-oriented programming allows the assignments of objects belonging to a class $C$ to variables of type $B$, where $B$ is a base class (i.e., an ancestor class in the hierarchy) for $C$. Consider the following example (for the sake of simplicity we adopt a C++-like syntax, even if C++ is not a dynamic language.): given the classes square and circle, which are subclasses of the base class shape

```
class shape {           class square : public shape {      class circle : public shape {
 protected:              float side;                         float radius;
  char *name;            public:                             public:
 public:                   ....                               ...
  void display()          void display()                     void display()
   { ... }                 { ... }                            { ... }
};                      };                                  };
```

In an application program it is possible to define an array of pointers to shape:

```
shape *array[5];
circle little_circle(2.0);
square big_square(100.0);
```

and successively assign to it values taken from the subclasses circle and square, e.g.,

```
array[0] = &little_circle;
array[1] = &big_square;
```

The execution of the methods

$$\text{array[0]}\rightarrow\text{display()}$$

and

$$\text{array[1]}\rightarrow\text{display()}$$

should lead to the execution of the correct methods (i.e., those associated to the classes circle and square respectively). The impossibility of determining statically which method should be called implies the need to dynamically detect the correct applicable method at runtime. This process of dynamically detecting the correct method to execute is referred to as *late (or dynamic) binding*. Most of the current object-oriented languages (e.g., C++ and Java) rely on late binding (either implicitly, as in Java, or explicitly, as in the case of the *virtual* functions of C++).

Compilers of languages that allow only a static structuring of the class hierarchies can efficiently realize late binding. For example, in C++, a simple table of pointers to methods (*VTable*) is associated with each class instance; at runtime determining the applicable method just involves a simple table lookup. However, there are other languages which allow *dynamic* extension of the classes hierarchy at runtime (e.g., Dynamic Object Oriented languages such as CLOS and Smalltalk), allowing creation of new classes on the fly. In this framework, smarter mechanisms are needed to organize class methods so that the correct applicable method can be determined for a given call. This determination may now, however, involve search of the class hierarchy at runtime. How to efficiently perform this search is the problem we address and analyze in this paper. Similar situations occur (with greater frequency) in *prototype based object-oriented languages* [3, 5, 33]. In these frameworks the concept of class is not present, rather, it is replaced by the idea of *prototypes*. Objects and prototypes are created on the fly and each object/prototype can be cloned to create new objects; *parent* attributes are used to connect the objects into hierarchies; execution of methods is based either on *delegation*—i.e., messages that do not match a local method are delegated to the parent object (e.g., in Self [6])—or on inheritance (e.g., Omega [3]). The focus of this work is mainly towards these classes of languages, with the possibility of relatively frequent modifications to the hierarchy, and characterized by heavy run-time requirements and limited compiler intervention.

### 1.2.2 Multiple Inheritance

The problem becomes more complex when multiple inheritance is considered. In multiple inheritance a class is allowed to inherit attributes from different base classes. Various additional problems have now to be solved, in particular:

- what happens when the same attribute (e.g., same instance variable) is inherited multiple times through different paths in the hierarchy;
- how to resolve the name conflict between distinct inherited attributes that have the same names.

Current multiple inheritance systems adopt different methodologies to solve these problems, such as:

- *linearization* of the inheritance DAG, i.e., the DAG is topologically sorted and the list of nodes is traversed during name resolution (e.g., as in CLOS);
- *tree inheritance* where the inheritance DAG is converted into a tree by duplicating (some of the) nodes reachable via different paths;
- *graph-based approach* where the DAG is not transformed and, in presence of conflicts it is necessary for the programmer to specify how they should be resolved.

Approaches based on linearization will lead to situations that are analogous to those of single inheritance. Tree-inheritance generates a tree which is rooted at the most defined class and which has to be traversed in a predefined order.

In this work we will focus only on single inheritance, or on those forms of multiple inheritance that can be directly reduced to cases of single inheritance, such as linearization-based multiple inheritance. Generalization of our results to other forms of multiple inheritance is a topic of future research.

### 1.3 Search Problems in Other Areas

The situation illustrated in the previous subsections is not unique to object-oriented programming; similar types of search operations occur in various different contexts. Our interest in the problem of implementing inheritance mechanisms was in fact spawned by our previous research [24, 27] in the area of logic programming and search in Artificial Intelligence. Various forms of execution of logic programming (e.g., or-parallel execution of logic programming) progress by developing a computation tree (in parallel), where the different branches represent alternative computation paths, leading to possibly different solutions to the original query. Variables created before a branching point can be instantiated to distinct values along different branches; every time a variable is referenced, the correct value of the variable has to be determined—which is the value assigned to that variable along the current branch of execution. This problem closely resembles the one we are considering here for object-oriented programming—with the exception that in logic programming a variable can be assigned only once along a branch, while in object-oriented programming an attribute can be redefined multiple times along one branch of the hierarchy.

The close similarity between the two problems is of great importance, as it seems to allow transfer of knowledge between the two areas—e.g., providing implementation mechanisms to support parallel name look-up in object-oriented systems.

## 2 Pointer Machines

A *Pure Pointer machine* consists of a finite but expandable collection $R$ of *records*, called *global memory*, and a finite collection of *registers*. Each record is uniquely identified through an $\mathsf{address}$. We use $\mathcal{N}$ to denote the set of all addresses. A special address *nil* is used to denote an invalid address. Each record is a finite collection of named fields. All the records in the global memory have the same structure, i.e., they all contain the same number of fields, with the same names. Each field may contain either a *data* (from a generic data domain *Data*) or an address (i.e., and element from $\mathcal{N}$). If $\mathcal{F}$ is the finite collection of field names, then the following function $\mathsf{field\_access}$ : $\mathcal{F} \times \mathcal{N} \rightarrow Data \cup \mathcal{N}$ allows to access the value stored in a field of a record.

The pointer machine is also supplied with two finite collections of registers, $d_1, d_2, \ldots$ (data registers) and $r_1, r_2, \ldots$ (pointer registers). Each register $d_i$ can contain an element from $Data$, while each register $r_i$ can contain one element of $\mathcal{N}$. The machine can execute *programs.* A program $P$ is a finite, numbered sequence of instructions, where each instruction in $P$ is uniquely identified by one number. These instructions allow addresses and data to be moved between registers and between registers and records' fields. The only "constant" which can be explicitly assigned to a register is *nil*. Special instructions are used to create a new record (returning its address) and to perform conditional jumps. The only conditions allowed in the jumps are true and equality comparison between two pointer registers. Observe that the content of the data fields will never affect the behavior of the computation. In terms of analysis of complexity, it is assumed that each instruction has a unit cost. For further details on the structure and behavior of the pure pointer machine the reader is referred to [2, 21, 35, 31]. Comparisons of the relative power of pointer machines and RAMs have been presented in [31].

Even though RAM is the most commonly used reference model in studies of complexity of sequential algorithms, the pointer machine model has received increasing attention as a viable model to study complexity of algorithms. The pointer machine model is particularly advantageous for complexity studies involving finer level of details [35]. The Pointer Machine model is simpler, thus making it more suitable for analysis of lower bounds of time complexity. The major difference between the RAM and the pointer machine model is the absence of arithmetic operations and tests in the pointer machine model. As a consequence, RAM commonly hides the actual cost of arithmetic operations, by allowing operations on numbers of size up to $\lg n$ ($n$ being the size of the input to the problem) to be treated as constant time operations. The pointer machine model, instead, makes these (*relevant*) costs explicit. From this point of view, the pointer machine provides a more realistic model for measure of time complexity. In fact, to address this weakness in the RAM model, variants of the model have been proposed (e.g., $\mathrm{RAM}_0$ and $\mathrm{RAM}_1$) which explicitly take the cost of arithmetic operations into account [31]. The pointer machine model is equivalent to these variants of the RAM model [31]. Many lower bound results have been developed by Tarjan and other researchers using the pointer machine model (e.g., [35, 4, 23]), and some of these results can be used to help us in studying the complexity of the problem at hand.

# 3 Abstraction of the Problem

In this paper we focus on the cost of performing efficient search as described in subsection 1.2.1. This immediately extends to the case of linearization-based multiple inheritance. Management of tree-based and graph-based multiple inheritance is not dealt with in this work and it is currently subject of research.

## 3.1 Preliminary Definitions

A tree $\mathcal{T} = \langle N, E \rangle$ is an acyclic, connected, rooted, and directed graph; $N$ is the set of all the nodes in the tree and $E$ is the set of the edges between the nodes. The tree contains a distinguished node, $\mathsf{root}$ which has no incoming edges. Each node in the tree has bounded degree. Trees are manipulated through three instructions:

1. $\mathsf{create\_tree}()$ which creates a tree containing only the $\mathsf{root}$;
2. $\mathsf{expand}(n, b_1, \ldots, b_k)$ which, given a node $n$ and labels $b_1, \ldots, b_k$, creates $k$ new nodes and adds them as children of $n$ ($b_1, \ldots, b_k$ can be thought of as the "names" of the new nodes);
3. $\mathsf{remove}(n)$ which, given a leaf $n$ of the tree, removes it from the tree.

These three operations are assumed to be the only ones available for modifying the "physical structure" of the tree.

The tree implements a partial ordering between the nodes in the tree. Given two nodes $n$ and $n'$, we write $n \preceq n'$ if $n$ is an ancestor of $n'$; $n \prec n'$ additionally says that $n \neq n'$.

## 3.2 Abstracting Inheritance

The abstraction of an object-oriented execution should account for the various issues present in object-oriented programming, (e.g., definition of new classes/objects, inheritance of attributes etc.). Previous research, as well as experience gained from the development of practical implementations [34, 30, 15, 38] shows that management of attributes is indeed one of the key issue in the implementation of an object-oriented system. The fact that the efficiency of attribute management has a significant impact on the performance of an object-oriented system has been experimentally demonstrated [19, 39, 12, 13]; furthermore, exploitation of parallelism and adoption of more complex approaches to object orientation make the problem of attribute management even more complex and difficult to implement. Thus, efficient implementation of attribute management is an important problem in implementation of object-oriented languages, if not the most fundamental one.

As described earlier, one of the main issues in dealing with inheritance in object-oriented programming is the search of a given attribute. This search may be needed when a class/object is created (in order to detect which variables should be included in each object) or during execution (in order to detect what is the correct method to execute in response to a given message). In the rest of the discussion we will adopt the following assumptions: the hierarchy is represented by a single-rooted tree which can dynamically grow and shrink; each node either introduces one attribute or redefines one. Each tree has a bounded degree—thus, without loss of generality, we can focus on binary trees. In section 5 we will illustrate how most of these assumptions can be lifted, without affecting the results obtained. The tree representing the hierarchy to be searched will be referred to as the *o-tree* in the rest of this paper.

The development of the object-oriented computation takes place as described in the previous section. A hierarchy—here assumed to be represented by a dynamic tree of either classes (class-based system), or objects (prototype-based system)—is dynamically created. Each new leaf created in the tree refines the object(s) defined at its parent node, by adding attributes and/or shadowing existing ones. The structure (set of attributes) of an object at node $n$ (denoted by $S(n)$) is thus determined by a combination $d(n, d(n_1, d(n_2, \ldots)))$ of the definitions present in the nodes $n_i$, where $n_i$ is the ancestor of $n$ at distance $i$ from node $n$. $d$ is a function with signature

$$d : \mathit{Nodes} \times 2^{\mathit{Nodes} \times \mathit{Attr}} \rightarrow 2^{\mathit{Nodes} \times \mathit{Attr}}$$

where *Attr* is the set of possible attributes and *Nodes* is the set of nodes in the hierarchy. Intuitively, each object is described by a set of pairs $\langle n, \psi \rangle$, where each pair identifies one attribute composing the object; $\psi$ determines the attribute and $n$ determines the node (class/prototype) from where we are going to inherit the definition of $\psi$. If $\gamma(n)$ is the attribute introduced in node $n$, then $d(n, \mathcal{S})$ is defined as

$$d(n, \mathcal{S}) = \{\langle n, \gamma(n)\rangle\} \cup \{\langle m, \psi\rangle \mid \langle m, \psi\rangle \in \mathcal{S} \ \wedge \ \psi \neq \gamma(n)\}$$

We can assume that a total of $M$ attributes are available—i.e., there are $M$ variables and methods. If the computation tree has size $N$, then we can assume that $M = O(N)$ (see also section 5).

If $\langle m, \psi\rangle \in S(m)$ and there are no further definitions of $\psi$ in the sequence of the path from $m$ to a descendent node $n$, then the path from $m$ to $n$ is said to be *$\psi$-free*.

At each node $n$ two operations on attributes are possible:

1. assign an attribute $\psi$ to a node $n$; as a consequence, every reference to $\psi$ in a node $m \succeq n$ (such that the path from $n$ to $m$ is $\psi$-free) will produce $n$ as result—i.e., identifies that $\langle n, \psi\rangle$ is present in $S(m)$.

2. dereference an attribute $\psi$ at a node $n$—that is identify the node

$$max_{\preceq}\{M \mid m \text{ defines } \psi \ \wedge \ m \preceq n\}$$

   equivalently, we want to determine the node $m$ such that $\langle m, \psi\rangle \in S(n)$.

In the formalization proposed here we assume an additional operation:

3. alias two attributes $\psi_1$ and $\psi_2$; This means that every reference to $\psi_1$ ($\psi_2$) in a node $m \succeq n$ will produce the same result as $\psi_2$ ($\psi_1$).

The alias operation is slightly unusual; we introduce it in this abstraction in order to allow the representation of constructions like the ability to access redefined properties by using superclass names as qualifiers (as in C++), and the *super* (Smalltalk) and *resend* (Self), as well as the "inverted" behavior of languages like Beta (with the *inner* construct). The alias operation *subsumes* both these behaviors. In the rest of this work we will deal briefly with this operation; most of the complexity results are *independent* of the presence of the alias operation—i.e., the cost of the alias operation is *negligible* with respect to the complexity of the remaining operations. The alias operation is introduced simply to show that operations like *super* and *resend* are not the real "source" of problems in the efficient management of dynamic object-oriented languages.

In the previous abstraction we have assumed the presence of one attribute per node. This restriction can be made without loss of generality if we can assume that the number of attributes in the node is bound by a program dependent constant. Section 5 considers possible relaxations of this condition. Let us refer to the problem of supporting these dynamic tree operations (assign, dereference, and alias) together with the operations to manipulate the physical structure of the o-tree as the $\mathcal{OOP}$ problem. The problems of *aliasing* and *dereferencing* are discussed later on separately. Figure 1 illustrates an example of an o-tree with various operations performed on it.

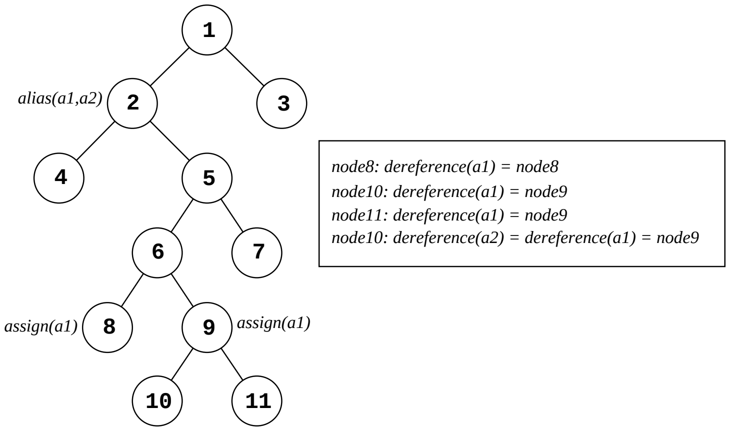


Figure 1: Operations on o-tree

All the operations need to be implemented efficiently to guarantee adequate performance. Our abstraction of the problem properly accounts for *all* the costs incurred during execution and is directly related to the computation in the hierarchy. Thus, even schemes which move part of the cost of managing search during object creation are properly represented in our abstraction—such costs will be accounted for in the abstraction during the expansion of a node. Some naive schemes for solving $\mathcal{OOP}$, with relative complexity study, are presented in section 4.2.

# 4 Complexity on Pointer Machines

In this section we will explore the lower and upper bounds for some of the most important operations required to support object-oriented languages. The section is organized in two parts: the first one describes the computation of the lower bound for the $\mathcal{OOP}$, while the second one focuses on the development of an upper bound for the problem.

## 4.1 Lower Bound for $\mathcal{OOP}$

In this section we will develop a lower bound on the complexity of the operations required in our abstraction of object-oriented languages. The initial goal was to demonstrate that a dynamic late binding object-oriented system cannot be maintained with a constant-time cost per operation. This result, as shown below, can be easily proved just based on the behavior of the alias operation alone. Nevertheless, the real source of complexity arises from the need to maintain dynamic references that, in each branch of the hierarchy, allow to resolve every attribute reference. This produces a considerably higher lower bound.

It is quite easy to observe that along each branch of the tree, attributes are managed in a way very similar to in the *union-find* problem [9]. Initially all attributes are distinct; given an attribute $X$, let us denote with $\phi(X)$ the set of all the attributes that are currently (on the branch at hand) aliased to $X$. If a new aliasing between $X_1$ and $X_2$ takes place, then the set $\phi$ is modified as $\phi(X_1) = \phi(X_1) \cup \phi(X_2)$. If we assume that the value of an attribute $X$ is always associated to the representative of $\phi(X)$, then any dereference operation is based on a find operation in order to detect the representative. We also assume (as addition to the union-find model) that the representative is selected as the node in $\phi(X)$ which is closer to the root.

The above observation can be used to provide a proof of the non-constant time nature of the computation involved in the management of attributes in a class hierarchy.

**Theorem 4.1** *The amortized time complexity of $\mathcal{OOP}$ is*

$$\Omega(N + M\alpha(M, N))$$

*where $N$ is the number of attributes and $M$ is the number of tree operations. $\alpha(M, N)$ denotes the very slowly growing inverse of the Ackermann function.*

**Proof.** *(Sketch)* The proof is based on the idea that $\mathcal{OOP}$ subsumes the union-find problem. Let us consider a generic union-find problem with $N$ elements and $M$ union-find operations. From the results in [35, 23] we know that this problem has a lower bound of $\Omega(N + M\alpha(M, N))$.

Let us show how we can use the mechanisms of $\mathcal{OOP}$ to implement a solution to the union-find problem. Each of the $N$ elements is encoded as one of the attributes. Each time a create$(X)$ operation is performed an expand operation is performed on the leftmost leaf and the operation $\mathsf{assign}(X)$ is performed on the leftmost leaf. Each $\mathsf{union}(X, Y)$ operation is translated into an aliasing between the two attributes $X$ and $Y$; the aliasing is performed to the leftmost leaf of the tree. Each $\mathsf{find}(X)$ operation corresponds to the dereferencing of the attribute $X$, starting from the leftmost leaf in the tree. Thus a set is represented by a collection of attributes which have been aliased to each other; the fact that the dereference operation returns the same result for aliased attributes guarantees the existence of a unique representative for the set. Clearly each operation in the union-find problem can be encoded using the $\mathcal{OOP}$ support and a constant number of additional operations. Thus, each solution to $\mathcal{OOP}$ cannot be better than the lower bound for the union-find problem. □

This result, as can be seen from the proof, relies mainly on the execution of aliasing operations. Nevertheless, it is possible to show that the result holds even when we disallow aliasing during execution of object-oriented programs. In fact, the inability to achieve constant-time behavior is inherent in the management of the attributes in the tree and *does not* depend on the presence of the alias operation.

**Theorem 4.2** *On pointer machines, the worst case time complexity of* $\mathcal{OOP}$ *is* $\Omega(\lg N)$ *per operation even without aliasing.*

**Proof.** The basic idea of the proof is that since there is no direct addressing in the pointer machines starting from a particular node only a "small" number of nodes can be accessed in a small number of steps. Without loss of generality, assume that each record has at most two pointers. Assume that we have a pointer machine program that performs the dereference operation in worst–case $c(N)$ time for some function $c$. The dereference procedure receives only two pointers as arguments, one to the record that contains the name of the attribute whose value is being searched, and the other to the leaf node for which this attribute value is being searched. Starting from either of these records, we can access at most $2^{c(N)+1}$ different records within $c(N)$ steps. We use this fact to prove that $c(N) = \Omega(\lg N)$.
Suppose $c(N)$ is not $\Omega(\lg N)$. Then there exist arbitrary large values of $N$ such that $c(N) < \frac{1}{8} \lg N$.
Consider the tree with $N$ nodes that has the following structure (see figure 2). It has depth $t_1(N) + t_2(N)$. The first $t_1(N)$ levels constitute a complete binary tree (let us indicate with $\Gamma_1$ the first $t_1(N)$ levels of the tree); the

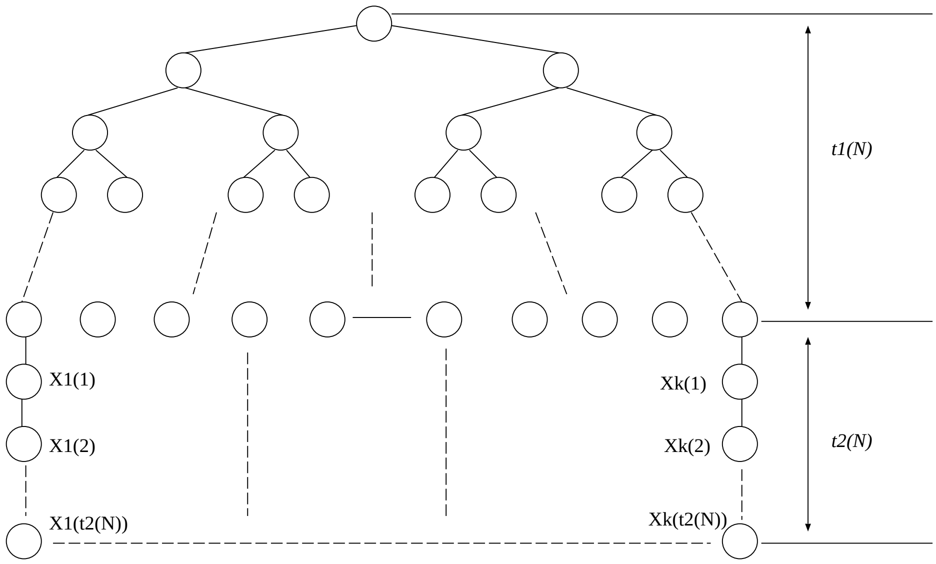


Figure 2: Complex Tree

remaining $t_2(N)$ levels consist of $2^{t_1(N)}$ lists of nodes, each of length $t_2(N)$ and each attached to a different leaf of $\Gamma_1$. Clearly, we must have that $2^{t_1(N)+1} - 1 + t_2(N) * 2^{t_1(N)} = N$ or that $t_2(N) = (N+1)/2^{t_1(N)} - 2$.

Let us also assume that we have $t_2(N)$ different attributes,

$$\{X_1, \ldots, X_{t_2(N)}\}$$

The nodes in each branch segment going from depth $t_1(N)+1$ to $t_2(N)$ are arbitrarily assigned the attributes $\{X_1, \ldots, X_{t_2(N)}\}$. The tree structure is represented in figure 2; in the figure, $k = 2^{t_1(N)}$, and $\langle Xi(1), \ldots, Xi(t_2(N))\rangle$ is a permutation of $\langle X_1, \ldots, X_{t_2(N)}\rangle$.
Suppose we choose $t_1(N)$ and $t_2(N)$ such that

$$2^{c(N)+1} t_2(N) < 2^{t_1(N)} \tag{1}$$

We should observe that, since the procedure can perform at most $c(N)$ steps, then starting from the records representing the attribute we can explore at most

$$2^{c(N)+1} t_2(N)$$

records—since there are $t_2(N)$ attribute records. From the choice of $t_1(N)$ and $t_2(N)$ in the formula (1), and from the fact that there is a total of

$$2^{t_1(N)}t_2(N)$$

records to be searched, there exists a leaf node $l$ in the tree such that none of the last $t_2(N)$ nodes on the path from root to $l$ is accessed starting from the pointer to the attribute node in any of the computations $\mathsf{dereference}(X_i, l)$, for $i$ ranging from 1 to $t_2(N)$. This implies that for any of the $t_2(N)$ different $\mathsf{dereference}(X_i, l)$ calls, the appropriate node on the path must be accessed starting from the leaf node. Since the dereference procedure works in time $c(N)$ we must have that

$$2^{c(N)+1} \geq t_2(N)$$

This gives us that $c(N) \geq \lg t_2(N) - 1$. Since, $c(N) < \frac{1}{8}\lg N$, $2^{c(N)+1} < 2N^{1/8}$. Let us choose $t_1(N) = \frac{3}{4}\lg N$. Then $2^{t_1(N)} = N^{3/4}$ and $t_2(N) = (N+1)/N^{3/4} - 2 \approx N^{1/4}$. These $t_1(N), t_2(N)$ satisfy the inequality (1) above as $2^{c(N)+1}t_2(N) \approx 2N^{3/8} < N^{3/4}$. Hence we can say that $c(N) \geq \lg t_2(N) - 1$, i.e. $c(N) > 1/4 \lg N - 1$ which contradicts our assumption that $c(N) < \frac{1}{8}\lg N$. Hence $c(N) = \Omega(\lg N)$.

Observe that the proof holds independently of the representation of the o-tree in the pointer machine. □

The above result is much stronger than the result in theorem 4.1, and gives us a greater insight into why it is impossible to support object-oriented executions with a constant-time overhead.

## 4.2 Upper Bound for $\mathcal{OOP}$

In this section we develop an upper bound for the $\mathcal{OOP}$ problem. Most of the implementation schemes proposed in the literature either restrict the capabilities of the object-oriented system (allowing to collect sufficient information during compile-time and reduce run-time search to simple table lookups) or can be shown to have a worst case complexity of $\Omega(N)$ per operation. Currently, the best result we have been able to achieve is the following:

**Theorem 4.3** *The $\mathcal{OOP}$ problem with no aliasing can be solved on a pointer machine with a single operation worst–case time complexity of $O(\sqrt[3]{N}(\lg N)^k)$ for a small $k$.*

For the sake of simplicity, we will recast the $\mathcal{OCP}$ problem in the following terms. The computation is described in terms of operations on a dynamic tree. We will focus on a growing only tree. Thus, the only operation allowed on the tree is $\mathsf{expand}(m, X_1, X_2)$ which expands the leaf $m$ by appending two successors $n_1$ and $n_2$, respectively labeled with $X_1$ and $X_2$. Here we are assuming, as mentioned above, that each node in the tree has bounded degree; thus, without loss of generality, we can as well restrict our attention to binary trees. Section 5 considers generalizations of this case. Also, for simplicity we assume that $X_1$ and $X_2$ are the actual attributes associated to such nodes—i.e., we avoid explicit use of assign. Only leaves can be expanded. The tree is *labeled*; thus a tree is coupled with a labeling function:

$$l\colon\ \mathit{Nodes} \rightarrow \Gamma$$

where *Nodes* is the set of all the nodes in the tree and $\Gamma$ is a finite set of labels. Given this, we can define the function

$$\mathsf{dereference}\colon\ \Gamma \times \mathit{Nodes} \rightarrow \mathit{Nodes} \cup \{\bot\}$$

which given a label $X$ and a node $n$ returns the lowest (i.e., closer to $n$) node $m$ (if any) on the path (from the root) ending in $n$ which is labeled $X$. Our problem now reduces to the efficient implementation of the two operation expand and dereference.

Let us start developing a solution under the assumption that the number $N$ of nodes to be inserted in the tree is known in advance. For our purposes, we can think the labels to be stored directly in the nodes of the tree. Let us first consider the naive scheme. The expand operation simply expands a leaf attaching the two new successors. The dereference operation starts with a label and a leaf, and walks back towards the root searching for the desired label. Clearly, in this naive implementation, the expand operation is $O(1)$, but the dereference operation can take time $O(N)$ in the worst case, since the tree may be completely unbalanced. Alternatively, on the other extreme, we could require expand to create a complete summary of the branch up to that point ($\tilde{O}(N)$), allowing a very fast dereference operation ($\tilde{O}(1)$). In both cases, a sequence of $N$ operations will lead to a worst–case complexity of $\tilde{O}(N^2)$.

The general idea in our improved schemes is to balance the amount of work done during the expand and the dereference operations, by making the expand operation more complex in order to make the search more efficient. We first start by providing a simple $\tilde{O}(\sqrt{N})$ solution; we later improve the solution to $\tilde{O}(\sqrt[3]{N})$. This last solution is considerably more involved than the first one, but uses some of the ideas in the first one.

#### 4.2.1 An $\tilde{O}(N^{\frac{1}{2}})$ scheme

The first algorithm we describe will lead to a complexity $O(\sqrt{N}(\lg N)^k)$.[1]

The behavior of dereference can be improved by creating at various points of each branch a "table" that summarizes the actions performed on that segment of the branch. This is illustrated in figure 3. Table *T1* summarizes the assignment of labels to nodes for the part of the branch between nodes `n1` and `n2`; table *T2* summarizes the assignment of labels to nodes for the part of the branch between nodes `n3` and `n4`. Nodes that are associated to tables are linked together. Note that, although we talk conceptually about tables, these tables are actually implemented as binary search trees on pointer machines. We can do this because we can assume the existence of a total order between the labels (we can remove this assumption by paying an $O(\lg N)$ penalty).

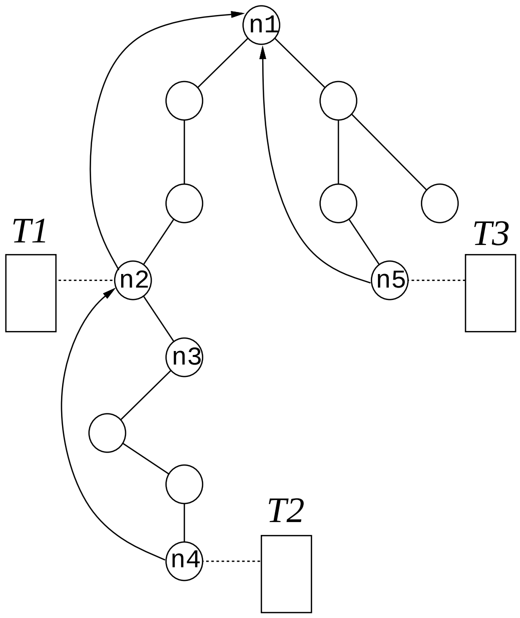


Figure 3: Use of Tables

The execution of dereference requires linearly scanning the branch from the leaf up to the first table node. If the desired label is not found in such part of the branch, then the table is examined. If the label is not found,

[1]For the sake of simplicity in most of the following analysis we will ignore the $(\lg N)^k$ factor.

then the search is repeated in the preceding table, and so on. If the root is reached, then the searched label has not been assigned to any node on that branch.

The complexity of this process is the following: let us assume that the tables are created every $t(N)$ nodes.

- the initial linear search will not visit more than $t(N)$ nodes;
- searching inside a table will require $O(\lg t(N))$;
- since there are at most $O(N)$ nodes on the branch, then the search process will visit at most $O(\frac{N}{t(N)})$ tables.

Combining these elements, we can state that the complexity of this implementation of dereference is

$$O\left(t(N) + \lg t(N)\frac{N}{t(N)}\right)$$

The expand operation, on the other hand, has a worst-case time complexity $\tilde{O}(t(N))$ per operation (occurring at the time of table creation).

Choosing $t(N) = \sqrt{N}$ the total cost[2] is $O(\sqrt{N}\lg N)$.

#### 4.2.2 An $\tilde{O}(N^{\frac{1}{3}})$ Solution:

The above scheme can be improved by making use of the solution to the ancestor problem—the problem of determining if one node is ancestor of the other in a dynamically growing tree. This is a well studied problem with very efficient solutions [14, 17, 21, 32, 36]. Let $\beta(N)$ denote the complexity of solving the ancestor problem for a dynamic tree with $N$ nodes. We know from the literature [36, 17] that it is possible to solve the ancestor problem for two nodes $x, y$ in time $\approx O(\lg\min(\mathit{depth}(x), \mathit{depth}(y)))$. Thus we can assume that $\beta(N) = O(\lg N)$.

The intuition behind this second solution is the following: if a certain label $X$ has been assigned to very few nodes—let's say $k$ nodes $m_1, \ldots, m_k$—it is possible to solve the problem dereference($X$,$n$) by simply considering each of the $k$ nodes $m_i$ and verifying whether $m_i$ is an ancestor of $n$. This is possible (in an efficient way) only if $k$ is sufficiently small. Observe that, in the worst case, the number of nodes associated to $X$ can be $O(N)$. On the

[2]We choose to ignore the polylog factors in making an optimum choice for $t(N)$.

other hand, only "few" labels will be used very frequently. We can try to take advantage of these facts by collecting the assignments of labels to nodes into *groups*. Each group is supposed to have sufficiently small cardinality to allow its exhaustive search.

The general idea in the search scheme (dereference) would be to first determine the group of the label being searched, and then perform an exhaustive search within the group, using the solution to the ancestor problem. Let $t(N)$ be the cardinality of a group. Thus, we associate to each label $X$ a set of groups, containing all the nodes $n$ in the tree such that $l(n) = X$. During the construction of the tree, every time we create a node $m$ with label $X$, we insert $m$ in the current group for $X$. If the current group is full—i.e., it contains $t(N)$ elements—then a new group for $X$ is created, and $m$ is added to such group. Nodes which lead to the creation of a new group are called *boundary nodes*. Each boundary node is properly marked, and contains a pointer to the new group.

Additionally, if the node $n$ is inserted for the label $X$, and $X$ has ever before received a new group, then $n$ is a *used node*, and it is also marked. More formally:

**Definition 4.4** *Given a label $X$, we indicate with $\#(X)$ the number of groups currently existing for $X$. $G(X,i)$ indicates the $i^{th}$ group for $X$ (if $1 \leq i \leq \#(X)$). $|G(X,i)|$ indicates the cardinality of the $i^{th}$ group of $X$.*

The used and boundary nodes can be defined as follows.

**Definition 4.5** *A node $n$ such that $l(n) = X$ is said to be a* **boundary node** *if, at the moment of its creation, $|G(X, \#(X))| = t(N)$. A node $n$ such that $l(n) = X$ is said to be a* **used node** *if $\#(X) \geq 2$.*

Figure 4(i) schematically illustrates these concepts with a simple example. The example assumes $t(N) = 2$. The numbers indicate the order in which the indicated nodes have been inserted. Node 4 is a boundary node as it requires the creation of a new group for `Z`; for this reason it has been marked. Node 6 is also a boundary node, for label `X`. Node 7 is a used node, since it is inserting a label for which there has already been a group change (due to node 6); for this reason the node is also marked.

Observe that, since there are $N$ nodes in the tree, and a group has size $t(N)$, during the insertion there will be at most $\frac{N}{t(N)}$ full groups. Observe also that given that there are $N$ nodes, there can be at most $\frac{N}{t(N)}$ labels having more than one full group. This means that along each branch of the tree, we cannot have more than $O(N/t(N))$ nodes that have been marked (either for being used or boundary).

Considering the organization of label assignments into groups, a strategy to implement the dereference operation would be to simply visit the marked nodes along the current branch (we can assume that they are linked together). If a node with the correct label is found, then we have an immediate solution—determined in $O(N/t(N))$. If a node with the correct label is not found, then there are only two possibilities left:

- the label has been used in the branch and the assignment was generated before filling the first group of that label; thus the assignment has been recorded in the first group for that label. The correct node can be found in time $O(t(N)\beta(N))$.
- the label has not been used at all in the branch. Again, if we search the nodes in the first group $(O(t(N)\beta(N)))$ we are going to determine this situation.

Thus, in general, the process will lead to a solution in $O(t(N)\beta(N)+N/t(N))$ steps.

The search can be improved by adding, as in the scheme described in the previous section, tables to summarize the group numbers associated to the used/boundary nodes in the previous section of the branch. Considering that we need to keep track of at most $N/t(N)$ nodes per branch—the maximum number of used and boundary nodes—using the results from the previous section we can determine that the best behavior can be achieved by placing a table at every $\sqrt{N/t(N)}^{th}$ used/boundary node along a branch. The effect of the table will be to summarize the previous $\sqrt{N/t(N)}$ used and boundary nodes generated along the current branch. Figure 4(ii) illustrates the situation. Marked nodes represent used/boundary nodes.

Using this scheme, a search for a label starting from a given leaf will proceed as follows:

- starting from the leaf, the various used/boundary nodes are accessed up to the first table. If the label is found within these nodes, then we have immediately an answer to the problem. Considering the assumption made before, this first phase will traverse at most $O(\sqrt{N/t(N)})$ nodes.

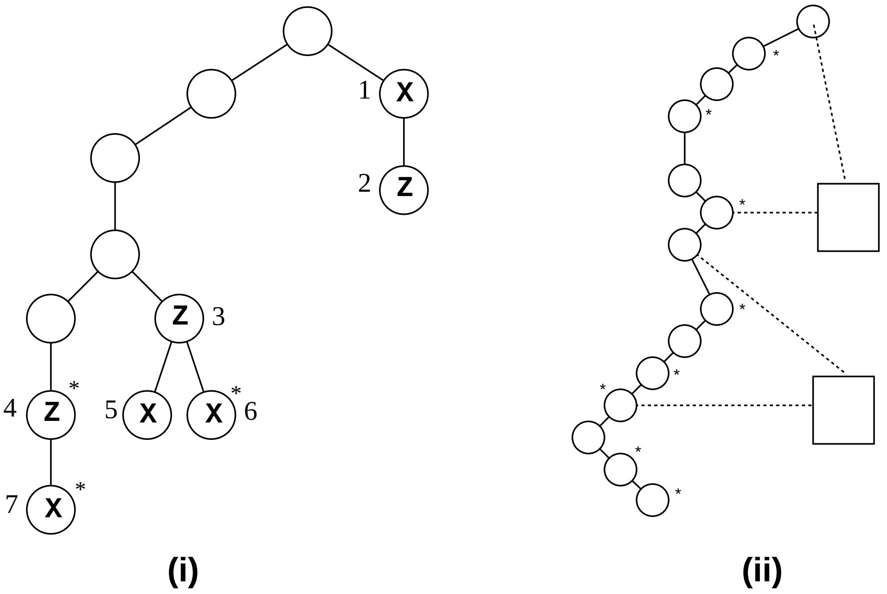


Figure 4: Examples of Trees

- if the label is not found in the used/boundary nodes at the end of the branch, then the first table is searched for the considered label. This search will cost $O(\lg(N/t(N)))$.
- if the label is found in the table, then we can assume that the entry will directly refer to the correct group. Otherwise the search has to be repeated in the preceding table. At most $\sqrt{N/t(N)}$ tables are going to be searched, leading to a total time of $O(\sqrt{N/t(N)}\lg(N/t(N)))$.
- finally, if the label is not found in any of tables, then the group of this label never changed (hence is 1). A final search has to be performed in the initial group for the label, which will imply a cost of $O(t(N)\beta(N))$.

Thus, we obtain a total complexity for the dereference operation of:

$$O\left(\sqrt{N/t(N)} + \sqrt{N/t(N)}\lg(N/t(N)) + t(N)\beta(N)\right)$$

Choosing $t(N) = \sqrt[3]{N}$, noting that $\beta(N)$ is $O(lgN)$, we get that the complexity of dereference is $\tilde{O}(\sqrt[3]{N})$.

To conclude the argument, we need to tackle the two assumptions made at the beginning of the proof:

1. the argument was focused only on growing trees, i.e., we ignored the remove operation. It is relatively easy to observe that this operation is applied only to leaves of the tree and hence does not affect the validity of the results.

2. we assumed at the beginning that the number of nodes $N$ is known beforehand. One can show that the use of the *current* number of nodes in place of the *total* number of nodes is going to produce variations only in the polylog part of the complexity. We made this assumption to keep the presentation of the proof simple and easy to follow—the use of the current number of nodes in place of $N$ leads to a tedious amount of detail.

Observe also that the model presented can be extended in a straightforward manner to support also the alias operation—which can be supported with the same kind of data structures used for the union-find problem, thus adding a worst–case complexity of $O(\frac{\lg N}{\lg \lg N})$ [4] per operation.

It is interesting to observe that the average case behavior can be improved by adopting a better structuring of the nodes in each group. In particular the nodes in a group can be organized as a tree which reflects the ancestor relationship present in the original dynamic tree—which would, on average, improve the search in a group. Also, a different criteria can be used to switch to a new group: a new group is created whenever the tree present in the current one has reached a specified height. This criteria would guarantee the same worst case behavior as before but may allow to place a larger number of nodes within the same group, potentially leading to a reduction in the time required to scan the main tree. □

The above scheme allows us to gain good efficiency on all the requested operations. The scheme increases memory consumption; nevertheless the space complexity remains $O(N)$.

The previous discussion focused on improving the cost of the dereference operation. In order to make sure that the model has an acceptable low complexity we need to guarantee that also the other main operations for manipulating the tree structure, i.e., expand/assign and remove can be performed at low cost.

In all the schemes described above, removal of a node can be performed in constant time without endangering the behavior of the scheme. The execution of expand/assign requires some more attention. Let us focus on the last scheme described. One of various cases may occur during creation of a new node:

1. the node is created and is assigned a variable which is not heavily used: in this case no special action is required (the cost is $O(1)$);

2. the node is created and assigned an heavily used variable, but the node is not going to contain a summarizing table. As in the previous case, no special action is required, except for marking the node and linking it to its predecessor marked node (the cost is $O(1)$);

3. the node is created, assigned an heavily used variable, and the node requires the creation of a table of marked nodes. The cost for the construction of the table is $\tilde{O}(\sqrt[3]{N})$.

Thus, the combined expand and assign operation can be performed with a worst case time complexity $\tilde{O}(\sqrt[3]{N})$, which is consistent with the cost of the other operations.

It is also interesting to observe that the amortized complexity of the combined expand/assign operation is $\tilde{O}(1)$, since the non-constant time cost ($\tilde{O}(\sqrt[3]{N})$) is paid only once every $\sqrt[3]{N}$ operations.

# 5 Generalizing the Lower and Upper Bounds

There are various assumptions that we made in the beginning. These assumptions were made to keep the presentation simple, and to derive the lower and upper bounds. Relaxing these assumptions will not affect the lower bound, as the problem will become more complex if an assumption is relaxed. However, relaxing an assumption can increase the upper bound. The effect of relaxing the assumptions is discussed next.

**Number of Attributes per Node**

The treatment of the previous sections was based on the assumption that a single attribute is associated to each node in the tree. As long as the total number of attributes defining a class/object is bounded by a given constant, the lower and upper bounds change only up to a constant.

Introducing arbitrary numbers of attributes in a node will add a penalty for searching the attribute; if we assume $k$ to be a measure of the number of attributes per node, then a $O(\lg k)$ penalty per operation will be incurred.

### Degree of the Nodes

In most of the previous treatment we have assumed the use of a bounded degree tree (in particular most of the discussion focused on binary trees). Nevertheless, by looking at the structure of the various proofs we can observe that the issue does not affect the final result. The various formulae proposed are based on the independent structure of each separate path, and it is not affected by the degree of the various nodes. Thus this assumption can be easily relaxed without affecting the final results.

### Arbitrary Insertion Points

We assumed in deriving the upper bound that the o-tree grows only at the leaf nodes. This assumption is related to the previous one. However, note that adding new children to any node in the tree (even internal nodes) does not affect the existing paths in the tree. Thus, the schemes here provided can be easily modified to accommodate a more general version of the expand operation—the only change required in the $O(\sqrt[3]{N})$ scheme is keeping an additional pointer with each node which points to the most recent boundary/used node. This can be easily maintained with a constant additional overhead.

### Modification of Internal Nodes

Various dynamic object oriented systems allows modification of existing classes/prototypes, by adding new attributes or changing existing ones. This would imply, in our model, the ability of modifying attributes of internal nodes of the tree. If we allow such operations, then the upper bound is affected.

The operation of replacing an individual attribute $\beta$ in an internal node $v$ with an attribute $\gamma$ can be implemented by updating the information stored in node $v$ and by maintaining an external table which records the replacement operations performed. A search for $\gamma$ will require two traversal of the data structure, one searching directly for $\gamma$, and one for $\beta$ (where $\beta$ has been detected from the global replacement table).

In general, if $k$ replacements have been performed in the whole data structure, then we have a degradation of complexity to $O(k\sqrt[3]{N})$ per operation.

# 6 Conclusions

In this paper we studied the complexity of supporting single inheritance (or multiple inheritance with linearization), in late binding dynamic object-oriented systems. We focused on the complexity of the operations associated with determining the correct attributes to be used in association with a given class/object. We developed lower and upper bounds for the $\mathcal{OOP}$ problem, i.e., the problem of implementing the basic operations required for correctly managing attributes detection.

The lower bound produced, $\Omega(\lg N)$, is a confirmation that an ideal situation where all the required operations can be implemented in constant-time is not feasible. This seems to confirm the results achieved for similar problems by other authors [29, 25]. Observe that the apparent capability of the RAM model to achieve $O(1)$ time complexity for all the operations is indeed misleading, since it relies on the incorrect assumption of constant-time cost for arithmetic operations on large numbers. Our analysis on pointer machines is thus producing a more precise evaluation of the complexity of the problem.

The upper bound, $\tilde{O}(\sqrt[3]{N})$, even if far from the lower bound, is of great importance, as it suggests that current implementation mechanisms are sub-optimal and better implementation schemes are feasible. In fact, from the implementation descriptions proposed for various systems, it seems that mostly naive schemes (typically requiring $O(N)$ time per operation) have been adopted. The scheme presented handle the $\mathcal{OOP}$ without the use of the alias operation. Nevertheless, from [4] we can infer that the alias operation can be implemented with worst–case complexity $O(\frac{\lg N}{\lg \lg N})$, thus adding just a lower order component to $\sqrt[3]{N}$.

The results presented in this work are mainly related to worst–case analysis of the problem. Nevertheless, all the worst–case situations depicted are reasonable and we expect the average–case complexity (which is heavily dependent on the specific application) to be sufficiently close to the results derived here.

The results have been developed using pointer machines as computation model. The pointer machine model allowed us to perform a very *precise* analysis avoiding the unrealistic assumptions made by other models.

An open problem of interest is the development of better upper bounds for the $\mathcal{OOP}$ in case of no aliasing between variables. We have shown that this problem has a non-constant time lower bound, but an optimal upper bound is still a subject of research. Our conjecture is that the problem admits a solution with a worst case time complexity $O((\lg N)^k)$ (for a small $k$) per operation.

Finally, it is important to observe that the $\mathcal{OOP}$ problem is actually a very general problem, which occurs in almost the same form in various different frameworks. For example, [26, 24, 27] illustrates how the same problem arises in the context of parallel execution of logic programming and in various artificial intelligence applications (e.g., game trees). Thus, the results presented here are general and can be successfully applied to problems outside of object-oriented programming systems.

## Acknowledgments

We would like to thank Luc Longpre and Kurt Mehlhorn for the comments on this work. We would also like to thank the anonymous referees for the many helpful suggestions.

The work has been partially supported by NSF grants CCR 98-75279, CCR 99-00320, CDA 97-29848, EIA 98-10732, CCR 96-25358, HRD 96-28450, and by a grant from the US-Spain Research Commission.

## References

[1] E. Amiel, O. Gruber, and E. Simon. Optimizing Multi-method Dispatch using Compressed Dispatch Tables. In *Conf. on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 1994.

[2] A.M. Ben-Amram. What is a Pointer Machine? Technical report, DIKU, University of Copenhagen, 1995.

[3] G. Blaschek. *Object Oriented Programming with Prototypes*. Springer Verlag, 1994.

[4] N. Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM Journal on Computing*, 15(4):1021–1024, 1986.

[5] A. Borning. Classes versus Prototypes in Object Oriented Languages. In *Fall Joint Computer Conference*. ACM/IEEE, 1986.

[6] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF. In *OOPSLA*. ACM, 1989.

[7] W.R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[8] W.R. Cook and J. Palsberg. A Denotational Semantics of Inheritance and its Correctness. In *Proc. OOPSLA*. ACM, 1989.

[9] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[10] P. Deutch and A. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings POPL*. ACM Press, 1984.

[11] K. Driesen. Selector Table Indexing and Sparse Arrays. In *Proceedings OOPSLA*. ACM Press, 1993.

[12] K. Driesen and U. Holzle. Minimizing Row Displacement Dispatch Tables. In *Conf. on Object Oriented Programming Systems, Languages, and Applications*. ACM Press, 1995.

[13] K. Driesen and U. Holzle. The Direct Cost of Virtual Function Calls in C++. In *Conf. on Object Oriented Programming Systems, Languages, and Applications*. ACM Press, 1996.

[14] H.N. Gabow. Data structures for weighted matching and nearest common ancestor. In *ACM Symp. on Discrete Algorithms*, pages 434–443, 1990.

[15] A. Goldberg and D. Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, 1983.

[16] G. Gupta and B. Jayaraman. Analysis of or-parallel execution models. *ACM TOPLAS*, 15(4):659–680, 1993.

[17] D. Harel and R.E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestor. *SIAM Journal of Computing*, 13(2):338–355, 1984.

[18] U. Holzle, C. Chambers, and D. Ungar. Optimizing Dynamically-typed Object-Oriented Languages with Polymorphic Inline Caches. In *Proceedings ECOOP*. Springer Verlag, 1991.

[19] U. Holzle and D. Ungar. A Third Generation SELF Implementation. In *Conf. on Object Oriented Programming Systems, Languages, and Applications*. ACM Press, 1994.

[20] S.E. Keene. *Object-oriented Programming in Common Lisp*. Addison Wesley, 1988.

[21] D.E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968.

[22] A.N. Kolmogorov and V.A. Uspenskii. On the Definition of an Algorithm. *Uspehi Mat. Nauk*, 13, 1958.

[23] H. LaPoutré. Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines. *Journal of Computer and System Sciences*, 52:87–99, 1996.

[24] E. Pontelli, D. Ranjan, and G. Gupta. On the Complexity of Parallel Implementation of Logic Programs. In *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 123–137. Springer Verlag, 1997.

[25] G. Ramalingam and H. Srinivasan. A Member Lookup Algorithm for C++. In *Programming Languages Design and Implementation*. ACM Press, 1997.

[26] D. Ranjan, E. Pontelli, and G. Gupta. Dynamic Data Structures in Advanced Programming Languages Implementation. Technical report, Dept. Computer Science, New Mexico State University, 1997.

[27] D. Ranjan, E. Pontelli, and G. Gupta. On the Complexity of Or-Parallelism. *New Generation Computing*, 17(3):285–308, 1999.

[28] D. Ranjan, E. Pontelli, L. Longpre, and G. Gupta. The Temporal Precedence Problem. *Algorithmica*, 1999. (to appear).

[29] J.G. Rossie and D.P. Friedman. An Algebraic Semantics of Subobjects. In *Conf. on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 1995.

[30] J.H. Saunders. A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming*, 1(6), 1989.

[31] A. Schönhage. Storage Modification Machines. *SIAM Journal of Computing*, 9(3):490–508, August 1980.

[32] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

[33] L. Stein. Delegation is Inheritance. In *OOPSLA*. ACM, 1987.

[34] A. Taivalsaari. On the Notion of Inheritance. *Computing Surveys*, 28(3), 1996.

[35] R.E. Tarjan. A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets. *Journal of Computer and System Sciences*, 2(18):110–127, 1979.

[36] A.K. Tsakalidis. The Nearest Common Ancestor in a Dynamic Tree. *ACTA Informatica*, 25, 1988.

[37] J. Vitek, R. Nigel Horspool, and A. Krall. Efficient Type Inclusion Tests. In *Conf. on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 1997.

[38] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1), 1990.

[39] O. Zendra, D. Colnet, and S. Collin. Efficient Dynamic Dispatch without Virtual Function Tables. In *Conf. on Object Oriented Programming Systems, Languages, and Applications*. ACM Press, 1997.