Efficient Data Structures For Multi-Mode Dispatching

[1]Ishan Jawa, [2]Gurpreet Singh, [3]Reena Sharma
[1,2,3]Doon Valley Institute of Engg. and Technology, Kurukshetra University, Haryana India

Abstract
The problem of dispatching in object oriented languages is the problem of determining the most specialized method to invoke for calls at run-time. This can be a critical component of execution performance. A number of results, including [Muthukrishnan and Muller SODA'96, Ferragina and Muthukrishnan ESA'96, Al-strup et al. FOCS'98], have studied this problem and in particular provided various efficient data structures for the monomethod dispatching problem. A paper of Ferragina, Muthukrishnan and de Berg [STOC'99] addresses the multi-method dispatching problem.
Our main result is a linear space data structure for binary dispatching that supports dispatching in logarithmic time. Using the same query time as Ferragina et al. this result improves the space bound with a logarithmic factor.

I. Introduction

In object oriented languages the modular units are abstract data types called classes and selectors. Each selector has possibly multiple implementations—denoted methods—each in a different class. The classes are arranged in a class hierarchy, and a class can inherit methods from its superclasses (classes above it in the class hierarchy). Therefore, when a selector s is invoked in a class c, the relevant method for s inherited by class c has to be determined. The dispatching problem for object oriented languages is to determine the most specialized method to invoke for a function call. This specialization depends on the actual arguments of the method call at run-time and can be a critical component of execution per formance in object oriented languages. Most of the commercial object oriented languages rely on dispatching of methods with only one argument, the so-called mono-method or unary dispatching problem. A number of papers, see e.g.,[10, 15] , have studied the unary dispatching problem, and Ferragina and Muthukrishnan [10] provide a linear space data structure that sup-ports unary dispatching in log-logarithmic time. However, the techniques in these papers do not apply to the more general multi-method dispatching problem in which more than one method argument is used for the dispatching. Multi-method dispatching has been identified as a powerful feature in object oriented languages supporting multi-methods such as Cecil [3], CLOS [2], Dylan [4]. Several recent results have attempted to deal with d-ary dispatching in practice (see [11] for an extensive list). Ferragina et al. [11] provided the first non-trivial data structures, and, quoting this paper, several experimental object oriented languages' "ultimately success and impact in practice depends, among other things, on whether multi-method dispatching can be supported ef-ficiently".

Our result is a linear space data structure for the binary dispatching problem, i.e., multi-method dispatching for methods with at most two arguments. Our data structure uses linear space and supports dispatching in logarithmic time. Using the same query time as Ferragina et al. [11], this result improves the space bound with a logarithmic factor. Before we provide a precise formulation of our result, we will formalize the general d-ary dispatching problem.

Definition (Multiple Dispatching Problem). Let T be a rooted tree denoting the class hierarchy. Each node in T corresponds to a class, and T defines a partial order on the set of classes:

$$A \preccurlyeq B \iff \text{A is an ancestor of B}$$

(not necessarily a proper ancestor).

If A is a proper ancestor of B we write $A \prec B$. Similarly, $B \succcurlyeq A$ $(B \succ A)$ if B is a (proper) descendant of A. Let M be the set of functions. Each function takes a number of classes as arguments. A function call is a query of the form $s(A_1, . . . , A_d)$ where s is the name of a function in M and $A_1, . . . , A_d$ are class instances/ objects. Let $s(A_1, . . . , A_d)$ be such a query. We say that s(B1, . . . , Bd) is applicable for $s(A_1, . . . , A_d) \iff B_i \succcurlyeq A_i$ for all $i \in \{1, . . . , d\}$.

The most specialized function for a query $s(A_1, . . . , A_d)$ is the function $s(B_1, . . . , B_d)$ such that

1. $s(B_1, . . ., B_d)$ is applicable for $s(A_1, . . ., A_d)$,
2. for every other function $s(C_1, . . ., C_d)$ applicable for $s(A_1, . . ., A_d)$ we have $C_i \preccurlyeq B_i$ for all i.

There might not be a most specialized method, i.e., we might have two applicative methods $s(B_1, . . . , B_d)$ and $s(C_1, . . . , C_d)$ where $B_i \prec C_i$ and $C_j \prec B_j$ for some indices $1 \le i, j \le d$. That is, neither method is more specialized than the other. Multi-method dispatching is to find the most specialized applicable method in M if it exists. If it does not exist or in case of ambiguity, "no applicable method" resp. "ambiguity" is reported instead.

The d-ary dispatching problem is to construct a data structure that supports multi-method dispatching with functions having up to d arguments, where M is static but queries are online.

The cases d = 1 and d = 2 are called the unary and binary dispatching problems, respectively. Let N denote the number of classes in the class hierarchy, m the number of methods in M , and M the number of distinct function names in M.

In this paper we focus on the binary dispatching problem which is of "particular interest" quoting Ferragina et al. [11].

We assume that the size of T is O(m). If this is not the case we can map nodes that does not participate in any method to their closest ancestor that does participate in some method in O(n) time.

A. Results

Our main result is a data structure for the binary dispatching problem using O(m) space and query time O(log m) on a unit-cost RAM with word size logarithmic in N with O(N + m (loglog m)2) time for preprocessing. By the use of a reduction to a geometric problem, Ferragina et al. [11], obtain similar time bounds within space O(m log m). Furthermore they show how the case d = 2 can be generalized for d > 2 at the cost of factor logd−2 m in the time and space bounds.

Our result is obtained by a very different approach in which we employ a dynamic to static transformation technique. To solve the binary dispatching problem we turn it into a unary dispatching problem — a variant of the marked ancestor problem as defined by Alstrup et al. [1], in which we maintain a dynamic set of methods. The unary problem is then solved persistently. We solve the persistent unary problem combining the technique by Dietz [5] to make a data structure fully persistent and the technique from [1] to solve the tree color problem. The technique of using a persistent dynamic one-dimensional data structure to solve a static two-dimensional problem is a standard technique [17]. What is new in our technique is that we use the class hierarchy tree to denote the time (give the order on the versions) to get a fully persistent data structure. This gives a "branch-ing" notion for time, which is the same as what one has in a fully persistent data structure where it is called the version tree. This technique is different from the plane sweep technique where a plane-sweep is used to give a partially persistent data structure. A top-down tour of the tree corresponds to a plane-sweep in the partially persistent data structures.

C. Related and Previous Work

For the unary dispatching problem the best known bound is O(N + m) space, O(loglog N) query time and expected O(N + m) preprocessing time. The expectation in the preprocessing time is due to perfect hashing in a van Emde Boas predecessor data structure [15, 10].

For the d-ary dispatching, $d \geq 2$, the result of Ferragina et al. [11] is a data structure using space $O(m\ (t\ \log m/\log t)^{d-1})$ and query time $O((\log m/\log t)^{d-1} \log\log N\ )$, where t is a parameter $2 \leq t \leq m$. For the case t = 2 they are able to improve the query time to O(logd−1m) using fractional cascading. They obtain their results by reducing the d-ary dispatching problem to a point-enclosure problem in d dimensions: Given a point q, check whether there is a smallest rectangle containing q. In the context of the geometric problem, Ferragina et al. also present applications to approximate dictionary matching.

Eppstein and Muthukrishnan [9] looked at a similar problem called packet classification. Here there is a database of m filters available for preprocessing. A packet filter i in an IP network is a collection of d-dimensional ranges $[l^i, r^i] \times \cdots \times [l^i, r^i]$, an action A^i, and a priority p^i. An IP packet P is a d-dimensional vector ofvalues$[P_1,\ldots,P_d]$.AfilteriappliestopacketPif$P_j \in [l^i,r^i]$forj=1,...,d. In this case the packet classification problem is essentially the same as the multiple dispatching problem. For the cased = 2 Eppstein and Muthukrishnan gave an algorithm using space O(m1+o(1)) and query time O(loglog m), or O(m1+ε) and query time O(1). They reduced the problem to a geometric problem, very similar to the one in [11]. To solve the problem they used the plane-sweep approach to turn the static two-dimensional rectangle query problem into a partial persistent dynamic one-dimensional problem.

II. Preliminaries

In this section we give some basic concepts which are used throughout the paper. Definition 1. Let T be a rooted tree. The set of all nodes in T is denoted V (T). Let T(v) denote the sub-tree of T rooted at a node $v \in V(T)$. If $w \in V(T(v))$ then v is an ancestor of w, denoted $v \preceq w$, and if $w \in V(T(v)) \setminus \{v\}$ then v is a proper ancestor of w, denoted $v \prec w$. If v is a (proper) ancestor of w then w is a (proper) descendant of v. In the rest of the chapter all trees are rooted trees.

Let C be a set of colors. A labeling l(v) of a node $v \in V(T)$ is a subset of C, i.e., $l(v) \rightarrow C$. A labeling $l : V(T) \rightarrow 2^C$ of a treeT is a set of labelings for the nodes in T . Given a labeling of a tree T, the first ancestor of $w \in T$ with color c is the node $v \in T$ such that $v \in w$, $c \in l(v)$, and no node on the path between v and w is labeled c.

A. Persistent Data Structures

Data structures that one encounters in traditional algorithmic settings are ephemeral, i.e., previous states are lost when an update is made. In a persistent data structure also previous versions of the data structure can be queried. The concept of persistent data structures was introduced by Driscoll et al. [8].

Definition (Persistence). A data structure is partially persistent if all previous versions remain available for queries but only the newest version can be modi-fied. A data structure is fully persistent if it allows both queries and updates of previous versions. An update may operate only on a single version at a time, that is, combining two or more versions of the data structure to form a new one is not allowed.

In addition to its ephemeral arguments a persistent update or query takes as an argument the version of the data structure to which the query or update refers. Let the version graph be a directed graph where each node corresponds to a version and there is an edge from node v1 to a node v2 if and only if V2 was created by an update operation to V1. The version graph for a partially persistent data structure is a path, and for a fully persistent data structure it is a tree.

B. Known Results

Dietz [5] showed how to make any data structure fully persistent on a unit-cost RAM with logarithmic word size by an efficient implementation of the version tree. A data structure with worst case query time O(Q(n)) and update time O(F (n)) making worst case O(U (n)) memory modifications can be made fully persistent using O(Q(n) loglog n) worst case time per query and O(F (n) loglog n) expected amortized time per update using O(U(n) n) space.

III. The Tree Color Problem

Definition (Tree color problem). Let T be a rooted tree with n nodes, where we associate a set of colors with each node of T . The tree color problem is to maintain a data structure with the following operations:

color(v, c): add c to v's set of colors,

i.e., $l(v) \leftarrow l(v) \cup \{c\}$,

uncolor(v, c): remove c from v's set of colors,

i.e., $l(v) \leftarrow l(v) \setminus \{c\}$,

firstcolor(v, c): find the first ancestor of v with color c (this may be v itself).

The incremental version of this problem does not support uncolor, the decremental problem does not support color, and the fully dynamic problem supports both update operations.

A. Known Results

Alstrup et al. [1] showed how to solve the tree color problem on a unit cost RAM with logarithmic word size in expected update time O(loglog n) for both color and uncolor, and query time O(logn/loglogn), using linear space and preprocessing time. The expected update time is due to hashing. Thus the expectation can be removed at the cost of using more space. We need worst case time when we make the data structure persistent because data structures with amortized/expected time may perform poorly when made fully persistent, since expensive operations might be performed many times.

Querying and updating a version tree of a fully persistent data structure is an incremental version of the tree color problem. Dietz [5] showed how to solve the incremental tree color problem in O(loglog n) amortized time per operation using linear space, when the nodes are colored top-down and each node has at most one color.

Definition 4. We need a data structure to support insert and predecessor queries on a set of integers from {1..n} . This can be solved in worst case in O(loglogn) time per operation on a RAM using the data structure of van Emde boas [18] (VEB). We show

how to modify this data structure such that it uses only O(1) memory modifications per update.

IV. The Bridge Color Problem

The binary dispatching problem (d = 2) can be formulated as the following tree problem, which we call the bridge color problem.

Definition 5 (Bridge Color Problem). Let T1 and T2 be two rooted trees. Between T_1 and T_2 there are a number of edges—called bridges—of different colors. Let C be the set of colors. A bridge is a triple $(c, v_1, v_2) \in C \times V(T_1) \times V(T_2)$ and is

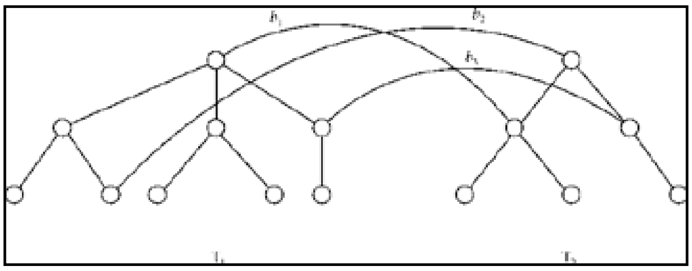

Fig. 1: An Example of the Bridge Color Problem. The solid lines are tree edges and the dashed and dotted lines are bridges of color c and c′, respectively. firstcolorbridge(c,v,u) returns b_3. firstcolorbridge(c′,r,s) returns ambiguity since neither b_1 or b^2 is closer than the other. denoted by $c(v_1, v_2)$. The bridge color problem is to construct a data structure which supports the query firstcolorbridge(c,v_1,v_2).

firstcolorbridge(c, v_1, v_2) Find a bridge $c(w_1, w_2)$ such that:

1. $w_1 \preccurlyeq v_1$ and $w_2 \preccurlyeq v_2$.
2. There is no other bridge $c(w_1, w_2)$ such that $w_1 \prec w_1 \preccurlyeq v_1$ or $w_2 \prec w_2 \preccurlyeq$ v2.

If there is no bridge satisfying the first condition return NIL. If there is a bridge satisfying the first condition but not the second then return "ambi-guity".

See fig. 1 for an example of the bridge color problem. The binary dispatching problem can be reduced to the bridge color problem the following way. Let T_1 and T_2 be copies of the tree T in the binary dispatching problem. For every method $s(v_1, v_2) \in M$ make a bridge of color s between $v_1 \in V(T_1)$ and $v_2 \in V(T_2)$.

The problem is now to construct a data structure that supports firstcolorbridge.

The objective of the remaining of this paper is to show the following theorem:

Theorem 1. Using expected O(m loglog m) time for preprocessing and O(m) space, the query firstcolorbridge can be supported in worst case time O(log m) per operation, where m is the number of bridges.

V. A Data Structure for the Bridge Color Problem

Let B be a set of bridges (|B| = m). As mentioned in the introduction we can assume that the number of nodes in the trees involved in the bridge color problem is O(m), i.e., $|V(T_1)| + |V(T_2)| = O(m)$. In this section we present a data structure that supports firstcolorbridge in O(log m) time per query using O(m) space for the bridge color problem.

We first reduce the static bridge color problem to the dynamic tree color problem. For each node $v \in V(T_1)$ we define the labeling lv of T2 as follows. The labeling of a node $w \in V(T_2)$ contains color c if w is the endpoint of a bridge of color c with the other endpoint among ancestors of v. Formally, $c \in l_v(w)$ if and only if there exists a node $u \prec v$ such that $c(u, w) \in B$. In addition to each labeling l_v, we need to keep the following extra information stored in a sparse array H(v): For each pair $(w, c) \in V(T_2) \times C$, where lv(w) contains color c, we keep the first ancestor v′ of v from which there is a bridge $c(v', w) \in B$. Note that this set is sparse, i.e., we can use a sparse array to store it. Similar define the symmetric labelings for T1. See fig. 2. for an example.

For each labeling lv of T_2, where $v \in V(T_1)$, we will construct a data structure for the static tree color problem. The query firstcolorbridge(c, u, w) can then be answered by the following queries in this data structure.

First perform the query firstcolor(w, c) in the data structure for the labeling lu of the tree T_2. If this query reports NIL there is no bridge to report, and we return NIL. Otherwise let w′ be the reported node. We make a lookup in H(u) to determine the bridge b such that $b = c(u', w') \in B$. By definition b is the bridge over (u, w′) with minimal distance between w and w′. However, it is possible that there is a bridge (u′′, w′′) over (u, w) where dist(u,u") < dist(u,u'). By a symmetric computation with the data structure for the labeling l(w) of T_1 we can detect this. If so we return "ambiguity". Otherwise we return the unique first bridge b. See fig. 2 for an example.

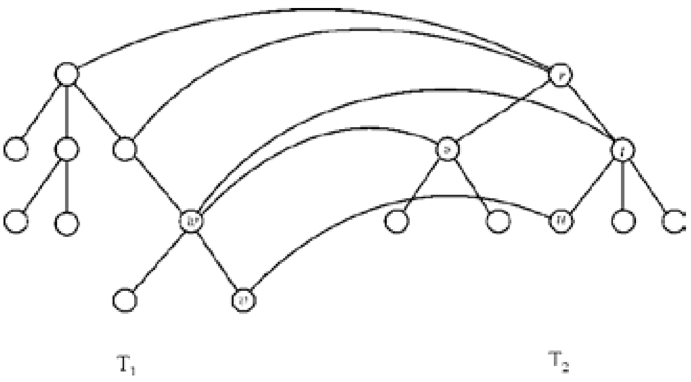

Fig. 2: Example of labeling. The labeling for $v \in V(T_1)$, $l_v : l_v(r) = \{c_1, c_2\}$, $l_v(s) = \{c_3\}$, $l_v(t) = \{c_1\}$, $l_v(u) = \{c_2\}$. The labeling lw for $w \in V(T_1)$ is the same as l_v except that lw(u) is empty.

Explicit representation of the tree color data structures for each of the labelings l_v for all nodes v in T_1 and T_2 would take up space $\Omega(m^2)$. Fortunately, the data structures overlap a lot: Let $v, w \in V(T_1)$, $u \in V(T_2)$, and let $v \preccurlyeq w$. Then $l_v(u) \subseteq l_w(u)$. We take advantage of this in a simple way. We make a fully persistent version of the dynamic tree color data structure. The idea is that the above set of O(m) tree color data structures corresponds to a persistent version, each created by one of O(m) updates in total. Formally, suppose we have generated the data structure for the labeling lv, for v in T1. Let w be the child of node v in T1. We construct the data structure for the labeling lw by updating the persistent structure for lv by inserting the color corresponding to all bridges with endpoint w (including updating H(v)). Since the data structure is fully persistent, we can repeat this for each child of v, and hence obtain data structures for all the labelings for children of v. In other words, we can form all the data structures for the labeling l_v for nodes $v \in V(T_1)$, by updates in the persistent structures according to a top-down traversal of T_1. Another way to see this, is that T_1 is denoting the time (giving the order of the versions). That is, the version tree has the same structure as T_1. Similarly, we construct the labelings for T_1 by a traversal of T_2. We conclude with the following lemma:

Lemma 1. A static data structure for the bridge color problem can be constructed by O(m) updates to a fully persistent version of the dynamic tree color problem.

A. Reducing the Number of Memory Modifications in the Tree Color Problem

Alstrup et al. [1] gives the following upper bounds for the tree color problem for a tree of size m. Expected O(loglog m) update time for both color and uncolor, and query time O(log m/loglog m), with linear space and preprocessing time.

For our purposes we need a slightly stronger result, i.e., updates that only make worst case O(1) memory modifications. By inspection of the dynamic tree color algorithm, the bottle-neck in order to achieve this, is the use of the van Emde Boas predecessor data structure [18] (VEB). Using a standard technique by Dietz and Raman [6] to implement a fast predecessor structure we get the fol-lowing result.

Theorem 2. Insert and predecessor queries on a set of integers from {1, . . . , n} can be performed in O(loglog n) worst case time per operation using worst case O(1) memory modifications per update.

To prove the theorem we first show an amortized result1. The elements in our predecessor data structure is grouped into buckets $S_1, \dots, S_k$, where we maintain the following invariants:

(1) max S_i < min S_{i+1} for i = 1, . . . k − 1, and

(2)1/2 log n < $|S_i| \leq 2$ log n for all i.

We have k ∊ O(n/ log n). Each S_i is represented by a balanced search tree with O(1) worst case update time once the position of the inserted or deleted element is known and query time O(log m), where m is the number of nodes in the tree [12-13]. This gives us update time O(loglog n) in a bucket, but only O(1) memory modifications per update. The minimum element si of each bucket Si is stored in a VEB.

When a new element x is inserted it is placed in the bucket Si such that $s_i < x < s_{i+1}$,or in S_1 if no such bucket exists. Finding the correct bucket is done by a predecessor query in the VEB. This takes O(log logn) time. Inserting the element in the bucket also takes O(log logn) time, but only O(1) memory modifications. When a bucket Si becomes to large it is split into two buckets of half size. This causes a new element to be inserted into the VEB and the binary trees for the two new buckets have to be build. An insertion into the VEB takes O(loglog n) time and uses the same number of memory modifications. Building the binary search trees uses O(log n) time and the same number of memory modifications. When a bucket is split there must have been at least log n insertions into this bucket since it last was involved in a split. That is, splitting and inserting uses O(1) amortized memory modifications per insertion.

Lemma 2. Insert and predecessor queries on a set of integers from {1, . . ., n} can be performed in O(loglogn) worst case time for predecessor and O(log logn) amortized time for insert using O(1) amortized number of memory modifications per update.

We can remove the amortization by the following technique by Raman [100] called thinning at the cost of making the bucket sizes $\Theta(\log^2 n)$.

Let α > 0 be a sufficiently small constant. Define the criticality of a bucket to be:

$\rho(b) = (1/\alpha \log n) \max\{0, \text{size}(b) - 1.8 \log_2 n\}$.1

A bucket b is called critical if ρ(b) > 0. We want to ensure that size(b) ≤ 2 $\log_2$n. To maintain the size of the buckets every α log n updates take the most critical bucket (if there is any) and move logn elements to a newly created empty adjacent bucket. A bucket rebalancing uses O(log n) memory modifications and we can thus perform it with O(1) memory modifications per update spread over no more than α log n updates.

We now show that the buckets never get too big. The criticality of all buckets can only increase by 1 between bucket rebalancings. We see that the criticality of the bucket being rebalanced is decreased, and no other bucket has its criticality increased by the rebalancing operations. We make use of the following lemma due to Raman:

Lemma 3 (Raman). Let $x_1, \dots, x_n$ be real-valued variables, all initially zero. Repeatedly do the following:

(1) Choose n non-negative real numbers $a_1, \dots, a_n$ such that

$\sum$n i=1 a_i = 1, and set

$x_i \leftarrow x_i + a_i$ for $1 \leq i \leq n$.

(2) Choose an x_i such that $x_i = \max_j \{x_j\}$, and set $x_i \leftarrow \max\{x^i - c, 0\}$ for some constant c ≥ 1.

Then each x_i will always be less than ln (n + 1), even when c = 1.

Apply the lemma as follows: Let the variables of Lemma 3 be the criti-calities of the buckets. The reals ai are the increases in the criticalities between rebalancings and c = 1/α. We see that if α ≤ 1 the criticality of a bucket will never exceed ln n + 1 = O(log n). Thus for sufficiently small α the size of the buckets will never exceed 2 $\log_2$ n. This completes the proof of Theorem 2.

We need worst case update time for color in the tree color problem in order to make it persistent. The expected update time is due to hashing, and can be removed at the cost of using more space. We now use Theorem 2 to get the following lemma.

Lemma 4. Using linear time for preprocessing, we can maintain a tree with complexity O(loglog m) for color and complexity O(log m/loglog m) for firstcolor, using O(1) memory modifications per update, where m is the number of nodes in the tree.

B. Making the Data Structure Persistent

Using Dietz' method [5] to make a data structure fully persistent on the data structure from Lemma 4, we can construct a fully persistent version of the tree color data structure.

Taking advantage of the fact that the structure of the version tree is known from the beginning, we can get faster preprocessing time for our bridge color data structure.

Since the structure of the version tree is known from the beginning we can construct the tree color data structure for the version tree, by first to running through the whole version tree in an Euler tour, and remembering which updates are made in which node. Then we can construct the tree color data structure for the version tree using the data structure for the static tree color problem by Muthukrishnan and Muller [15]. This uses expected linear preprocessing time and linear space using worst-case O(loglog m) time per query. As mentioned earlier the expectation in the preprocessing time can be removed.

Another possibility is to use a modified version of Dietz' method to make a data structure fully persistent. Recall that this method gives an expected amortized slowdown of O(loglog m). The amortization comes from the problem of maintaining order in a list. Since we know the structure of the version tree from the beginning we can get rid of this amortization. This gives a significant simplification of the construction of our bridge color data structure. This data structure uses expected O(loglog m) time per update, and the preprocessing time for our bridge color problem is thus a factor of O(loglog m) bigger than in the approach using a static tree color data structure for the version tree.

References

[1] S. Alstrup, T. Husfeldt, T. Rauhe,"Marked ancestor problems (extended abstract)", In IEEE Symposium on Foundations of Computer Science (FOCS), pp. 534-543, 1998.

[2] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon,"Common LISP object system specification X3J13 document 88-002R", ACM SIGPLAN Notices, 23, 1988. Special Issue, September 1988.

[3] C. Chambers,"Object-oriented multi-methods in Cecil", In O. L. Madsen, editor, ECOOP '92", European Conference on Object-Oriented Programming, Utrecht, The Netherlands, volume 615 of Lecture Notes in Computer Science, pp. 33-56. Springer-Verlag, New York, NY, 1992.

[4] Inc. Apple Computer. Dylan Interim Reference Manual.1994

[5] P. F. Dietz,"Fully persistent arrays. In F. Dehne, J.-R. Sack, and N. San-toro, editors, Proceedings of the Workshop on Algorithms and Data Structures, volume 382 of Lecture Notes in Computer Science, pp. 67-74, Berlin, Aug. 1989. Springer-Verlag.

[6] P. F. Dietz, R. Raman,"Persistence, amortization and randomization", In Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 78-88, 1991.

[7] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, R. E. Tarjan,"Dynamic perfect hashing: Upper and lower bounds", In 29th Annual Symposium on Foundations of Computer Science (FOCS), pp. 524-531. IEEE Computer Society Press, 1988.

[8] J. R. Driscoll, N. Sarnak, D. D. Sleator, R. E. Tarjan,"Making data structures persistent", Journal of Computer and Systems Sciences, 38(1), pp. 86-124, 1989.

[9] D. Eppstein, S. Muthukrishnan,"Internet packet filter manegement and rectangle geometry", In Proceedings of the 12th annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2001

[10] P. Ferragina, S. Muthukrishnan,"Efficient dynamic method-lookup for object oriented languages. In European Symposium on Algorithms, Vol. 1136 of Lecture Notes in Computer Science, pp. 107-120, 1996.

[11] P. Ferragina, S. Muthukrishnan, M. de Berg,"Multi-method dispatching: A geometric approach with applications to string matching problems", In Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, pp. 483-491, May 1999.

[12] R. Fleischer,"A simple balanced search tree with O(1) worst-case update time", International Journal of Foundations of Computer Science, 7, pp. 137-149, 1996.

[13] C. Levcopoulos, M. Overmars,"A balanced search tree with O(1) worstcase update time", Acta Informatica, 26, pp. 269-277, 1988.

[14] K. Mehlhorn, S. Näher,"Bounded ordered dictionaries in O(log log n) time and O(n) space", Information Processing Letters, 35, pp. 183-189, 1990

[15] S. Muthukrishnan, M. M üller,"Time and space efficient method-lookup for object-oriented programs (extended abstract)", In Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 42-51, Jan.1996.

[16] R. Raman. Eliminating Amortization: On Data Structures with Guaranteed Response Time. Ph.D thesis, University of Rochester, Computer Science Department, October 1992. Technical Report TR439.

[17] N. Sarnak, R. E. Tarjan. Planar point location using persistent search trees. Communications of the ACM, 29, pp. 669-679, 1986

[18] P. van Emde Boas,"Preserving order in a forest in less than logarithmic time and linear space", Information Processing Letters, 6, pp. 80-82, 1978.