Fast and Compact Dispatching for Dynamic Object-Oriented Languages

Christian Queinnec

LIP6 & INRIA-Rocquencourt

Dynamic Object-Oriented languages allows for dynamic definition of new classes, new generic functions and new methods. This paper proposes a single and compact data structure to, at the same time, facilitate the addition of new classes, generic functions or methods, and still ensure a fast method selection.

Key words: data structures, programming languages, object-oriented dispatch.

Within a dynamic Object-Oriented language, new classes, new generic functions and new methods may be created at run-time. We assume the CLOS model [BDG$^+$88] where a generic function is a dispatching function over a set of methods. When invoked, a generic function executes its most appropriate method based on the class of its arguments. This dispatching process, or method selection, has been well studied [KR90,Ros91,HC92,AGS94,VH94,CT95] and its efficiency is based on the elaboration of a specific dispatch structure; see [VH94] for a detailed comparison.

The dynamic character of the language requires the implementation to be able to regenerate the dispatch structure whenever a class, a generic function or a method is added. Our paper proposes to implement the dispatch structure with a decision tree, respecting the inheritance tree thus making easy these additions and yet providing a fast dispatch mechanism based on a specific numbering of classes. While multiple dispatch is addressed in section 5, our solution is particularly useful for languages with single inheritance and single-method dispatch: the Smalltalk case [GR83], where the method is selected according to the class of a distinguished argument: the receiver. The examples of the rest of the paper are based on the tree of classes shown on figure 1

[1] Laboratoire d'Informatique de Paris 6, boîte 168, 4, place Jussieu, 75252 Paris Cedex 05 France – Email: `Christian.Queinnec@inria.fr`. This work has been partially funded by GDR-PRC de Programmation du CNRS.

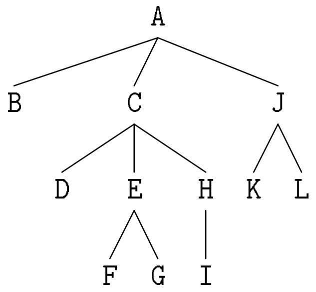

Fig. 1. An inheritance tree of classes

where `A` is the root of the inheritance tree. We will also use the four generic functions with methods defined as in table 1. In order to avoid cluttering the table, inherited methods will not be shown.

Table 1
Some generic functions and methods

	class											
generic	`A`	`B`	`C`	`D`	`E`	`F`	`G`	`H`	`I`	`J`	`K`	`L`
m	-	-	m_C	-	m_E	m_F	m_G	-	-	-	-	-
n	-	-	-	-	-	-	-	-	-	-	n_K	n_L
p	-	-	-	-	-	-	-	p_H	-	p_J	-	-
q	-	-	q_C	q_D	q_E	-	q_G	-	q_I	-	-	-

1 Dispatch structure

Method selection consists, given a class c (the class of the receiver) and a generic function g, to find quickly the most appropriate method for this class in this generic function. The original technique of Smalltalk was to obtain, from the class c, the dictionary of its methods (represented by a column of table 1) then to search in that dictionary for a method associated to the generic function g. If there is no such method, then method selection is restarted with the superclass of c. This technique works for single-method dispatch, it is rather compact but only offers a linear access time.

A second technique is to use an array with lines indexed by generic functions and columns indexed by classes. The difference with table 1 is that this array contains inherited methods: for instance at position $(p, \mathtt{I})$ is p_H since `I` inherits from `H`. Method selection is thus a simple constant-time array access. The disadvantage of this technique is the tremendous size of this array (16 MB for Smalltalk for instance). There exists a number of algorithms to compress this array [AR92,VH94], compress ratio are reported to be greater than 99% but this may take a dozen of seconds. Of course, the array does not need to be represented as an array; lines may be kept independently within generic functions.

In many applications, generic functions may be observed to belong to two types: universal or localized. A universal generic function such as `print` nearly contains a method for any class. A localized generic function (see m, n, p, q in table 1) is confined to one (or two) subtree(s) of classes.

A third technique is to use decision trees for method selection. These decision trees respect the structure of the inheritance tree and are made of three different types of nodes. A `cst` node is a terminal node imposing the method to return; an `if` node is a binary choice node that orient the lookup to one particular subtree; an `xif` node (standing for indeXed IF) is a bag containing all the methods of a particular subtree. The rules for method selection are the following, where $c \leq class$ means that c is a (proper or not) subclass of *class*:

lookup c `[cst` *method*`]` $=$ *method*
lookup c `[if` *class no yes*`]` $=$
  if $c \leq class$ **then** lookup c *yes* **else** lookup c *no*
`lookup` c `[xif` *class no methods...*`] =`
  if $c \leq class$ **then** $methods_c$ **else** lookup c *no*

In the above rules, *methods...* stands for the methods associated to the classes of the subtree rooted at *class*; among them $methods_c$ represents the appropriate method for class c.

To concretize things, these are the decision trees for generic functions p and m where `-` means that there is no associated method. In the following `xif` node, methods are ranked by prefix order.

p `= [if H [if J - [cst` p_J`]] [cst` p_H`]]`
m `= [xif C -` m_C m_D m_E m_F m_G m_H m_I`]`
  where $m_C = m_D = m_H = m_I$

Different decision trees may exist for a generic function. If speed is preferred over space, then shallow `xif` nodes may be used to mimic the previously evoked array technique. If compaction is preferred then arbitrarily nested `if` nodes can be used. These `if` nodes should favor, in their *yes* part, the most heavily used subtrees. To choose the right decision tree depends on the desired criteria. One may rebalance them statically when adding methods, or one may profile them to rebalance them dynamically. A whole spectrum of decision exists here.

A new generic function is created with an indexed tree equals to `-`. A method m is added to a generic function g for class c by rearranging the node `[if` c g

[cst m]]. When a new class c is created, all xif nodes related to a superclass of c are extended to define by inheritance a method for c. We will not consider removal of classes because of possibly living instances. A generic function can be removed with all its methods without problem, a particular method can be removed if replacing it by its super-method.

The two problems to solve with these decision trees is to have a fast subclass test, $c \leq class$, and a fast class indexing for $methods_c$ allowing to retrieve the appropriate method out of an xif node. The first problem has a well known solution [Weg75,Coh91]; we propose an original solution to the second problem that nicely fits with the first.

2 Subclass test

Let's suppose that classes are numbered sequentially, for instance by order of creation. Let's note c.depth, the depth of class c in the inheritance tree, that is its number of proper superclasses; for example, A.depth is zero in figure 1.

A class is represented by some data structure containing its name, the description of its fields, and a link to its single superclass. We adjoin to this structure the sequence of all the numbers of its superclasses starting from the root class to the class itself included; see figure 2. The $c.\texttt{super}(i)$ notation designates the superclass of c at depth i. The number associated to a class c is given by $c.\texttt{number} = c.\texttt{super}(c.\texttt{depth})$.

To test whether c is a subclass of c' may be done in constant time with the following expression:

$$c.\texttt{depth} \geq c'.\texttt{depth} \quad \wedge \quad c.\texttt{super}(c'.\texttt{depth}) = c'.\texttt{number}$$

3 Relative numbering

The second problem is, given all the methods of a subtree, find the method associated to a given class of this subtree. If classes were numbered in prefix order then the solution would be simple. For a node [xif *class no methods...*] where *methods...* are ranked by prefix order, then $methods_c$ is just the j^{th} method of the xif node where $j = c.\texttt{number} - class.\texttt{number}$. Alas, this solution does not resist the introduction of new classes that requires to renumber classes in prefix order (even if that renumbering may be limited as in [Que93]) and to reorder the methods within all xif nodes.

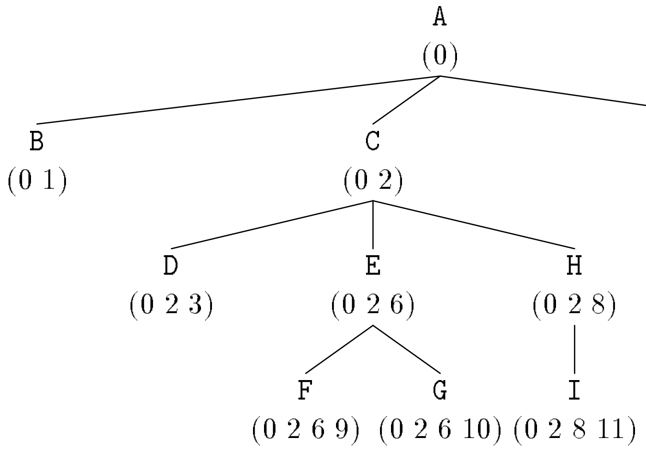

Fig. 2. A numbered inheritance tree of classes

Let's then suppose that a class is numbered with respect to all its superclasses, for instance by order of creation; see figure 3. The relative number of a subclass c with respect to one of its superclass *class* is noted c.`relnum`(*class*.`depth`). Of course, c.`number` $=$ c.`relnum`(0) and c.`relnum`(c.`depth`) $= 0$. For instance, in figure 3, `G` is the 10th class with respect to `A` (10 is its number), is the 5th class with respect to `C`, the second class with respect to `E` and the zeroth with respect to itself.

A `xif` node holding the methods for a subtree of classes rooted in *class*, places $method_c$, the method associated to c, according to the relative number of c with respect to *class*, therefore method selection within the yes part of a `xif` node is done in constant time:

lookup c `[xif` *class no methods*...`]` =
  if c.`depth` $\geq$ *class*.`depth`
    $\wedge$ c.`super`(*class*.`depth`) $=$ *class*.`number`
  then *methods*(c.`relnum`(*class*.`depth`))
  else lookup c *no*

4 Implementation

When a class is created, it has to be numbered with respect to all its superclasses. This is simple if we keep in every class c, a c.`next` field counting the number of its subclasses including itself. When a class is created with a given inheritance path, the value of these counters are the relative numbers of the new class with respect to these superclasses; these counters are incremented right after. For example, were we to add a class `M` inheriting from `E`, it will be numbered as shown on figure 4.

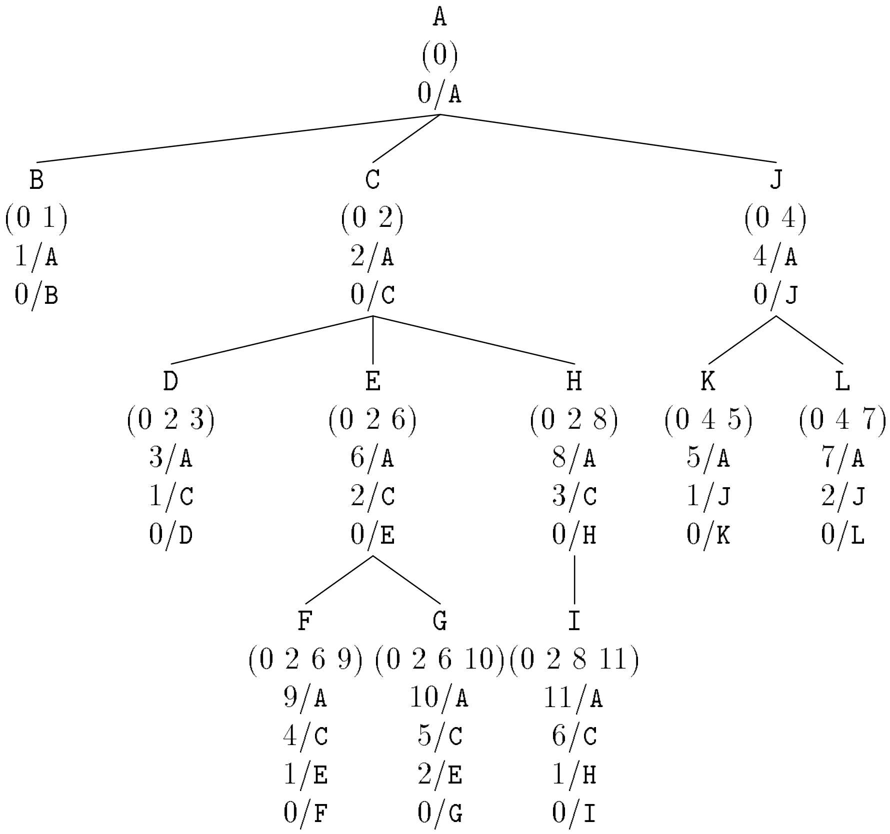

Fig. 3. A completely numbered inheritance tree of classes

When a new class c is created, any `xif` node discriminating on a superclass of c must be extended so c inherits the method of its superclass. To easily find these `xif` nodes, they may be linked from the class they discriminate on. Furthermore, to avoid extending again and again `xif` nodes, they may be allocated with some spare room.

To speed up method selection and spare some memory references and since the only use of a class in an `if` or `xif` node is to check whether the class of the receiver is one of its subclass, only its depth and number are required. This is safe because these numbers never vary. Therefore a `xif` node may be implemented as:

$$\{\texttt{xif}\ \textit{class}.\texttt{depth}\ \textit{class}.\texttt{number}\ \textit{size}\ \textit{no}\ \ \overbrace{\underbrace{\textit{methods}\ldots}_{\textit{class}.\texttt{next}}\ \textit{spare room}\ldots}^{\textit{size}}\}$$

Of course, rather than being interpreted, decision trees may also be compiled into native code for efficiency reason.

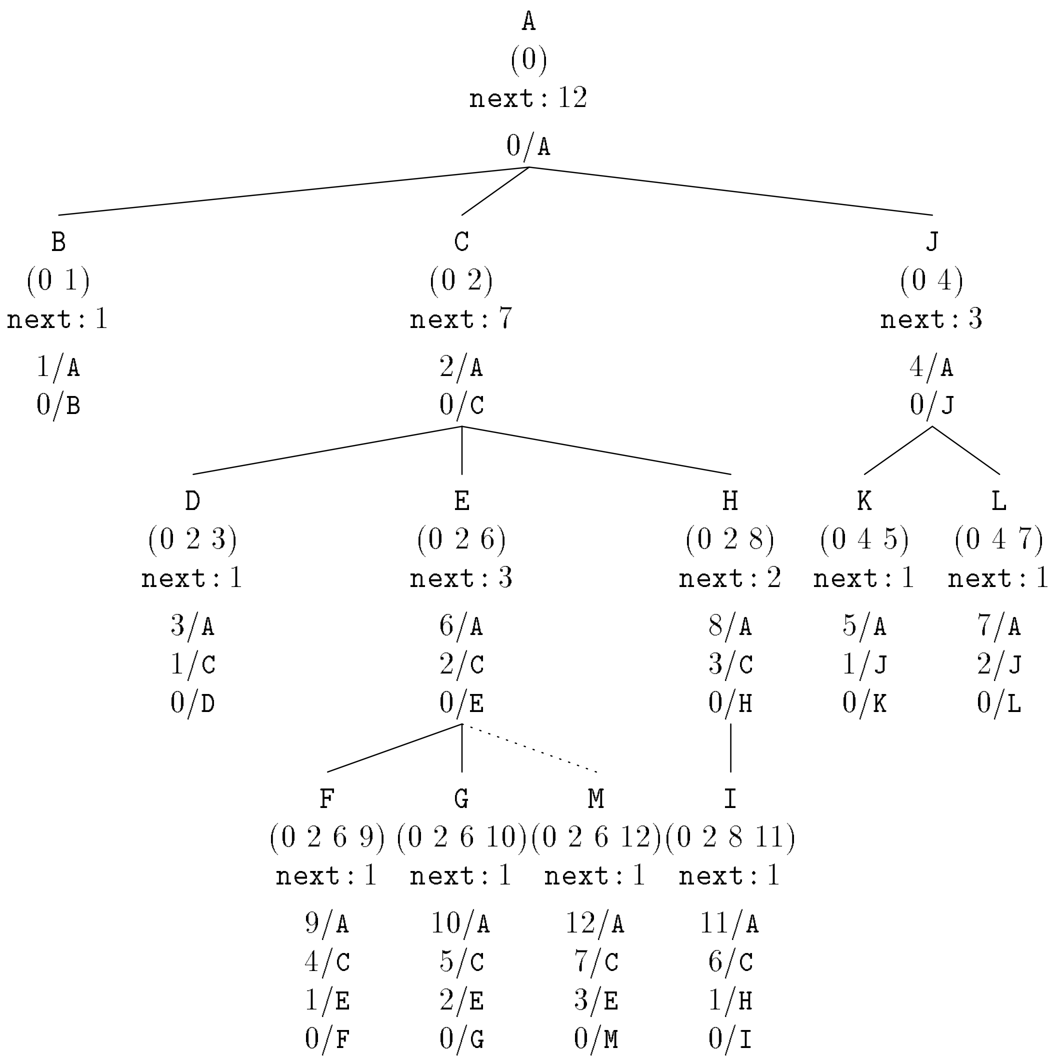

Fig. 4. Introduction of class M

The numbers that qualify a class c may overlap since $c.\texttt{super}(c.\texttt{depth}) = c.\texttt{relnum}(0)$. A class is therefore qualified by its depth, the numbers of its superclasses and its relative numbers with respect to all its superclasses. The overhead is $2(\texttt{depth}+1)$ numbers. For example, M is qualified by the following numbers:

$$\begin{array}{cc} \texttt{depth} & \\ 3 & \overbrace{0\quad 2\quad 6\quad \rlap{$\underbrace{}_{\texttt{relnum}(.)}$}12}^{\texttt{super}(.)}\quad 7\quad 3\quad 0 \end{array}$$

5 Multiple dispatch

Multiple dispatch occurs when a generic function selects the method to execute based on the classes of two or more arguments. Often a generic function is still localized on a small subtree of classes. Since the case of multiple dispatch on more than two variables is rare [KR90], we focus on the case of two discriminating variables i.e. double dispatch. Nodes `cst` and `if` may still be used in our decision tree to orient the method selection but we need another node for double dispatch. A `xif` node introduced a one-dimensional vector of methods, a `xxif` may well introduce a two-dimensional matrix of methods. Our previous solution can easily be adapted. Here is the shape of a `xxif` node discriminating on `E` and `J`:

$$\texttt{[xxif E J } \mathit{no} \; \boxed{\begin{array}{c|ccc} & \texttt{J} & \texttt{K} & \texttt{L} \\ \hline \texttt{E} & - & - & - \\ \texttt{F} & - & - & - \\ \texttt{G} & - & - & - \end{array}} \; \texttt{]}$$

The problem with n-dimensional matrix is their decreasing density as n grows. We nevertheless suspect that generic functions are sufficiently localized for our solution to be worthwhile.

6 Conclusion

Dynamic object-oriented languages with single-inheritance and single-dispatch may number classes according to our scheme and use a dispatch structure for method selection based on decision trees respecting the class inheritance tree. The depth of these decision trees allow to control the speed/space ratio. This scheme has been implemented in MEROON [Que93], an object system for the Scheme programming language.

7 Acknowledgments

The original class numbering scheme was discovered during a stay at University of Montreal in summer 1994. Thanks to Jan Vitek for his encouragement.

References

[AGS94] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *Int. Conf. on OOPSLA*, Portland, Octobre 1994. ACM.

[AR92] Pascal André and Jean-Claude Royer. Optimizing method search with lookup caches and incremental coloring. In *OOPSLA '92 — Object-Oriented Programming Systems and LAnguages*, pages 110–126, Vancouver (British Columbia, Canada), October 1992. ACM Press. Also Sigplan Notices 27(10).

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification. *SIGPLAN Notices*, 23, September 1988. special issue.

[Coh91] Norman H Cohen. Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems*, 13(4):626–629, October 1991.

[CT95] Weimin Chen and Volker Turau. Multiple-dispatching based on automata. *Theory and Practice of Objects Systems*, 1(1):41–60, 1995.

[GR83] Adèle Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation.* Addison Wesley, 1983.

[HC92] Shih-Kun Huang and Deng-Jyi Chen. Efficient algorithms for method dispatch in object-oriented programming systems. *Journal of Object-Oriented Programming*, 5(5):43–54, September 1992.

[KR90] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in pcl. In *LFP '90 - ACM Symposium on Lisp and Functional Programming*, pages 99–1052, Nice (France), June 1990.

[Que93] Christian Queinnec. Designing MEROON v3. In Christian Rathke, Jürgen Kopp, Hubertus Hohl, and Harry Bretthauer, editors, *Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP'93 Workshop*, number 788, Sankt Augustin (Germany), September 1993.

[Ros91] J Rose. A minimal metaobject protocol for dynamic dispatch. In *Proceedings of the OOPSLA '91 Workshop on Reflection and Metalevel Architectures in Objecct-Oriented Programming*, October 1991.

[VH94] Jan Vitek and R Nigel Horspool. Taming message passing: Efficient method look-up for dynamically typed languages. In *ECOOP '94 — 8th European Conference on Object-Oriented Programming*, Bologna (Italy), 1994.

[Weg75] Ben Wegbreit. Retrieval from context trees. *Information Processing Letters*, 3(4):119–120, March 75.