Report IAI-TR-94-3
October 1994
(revised April 1995)

Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems

Günter Kniesel
Institut für Informatik III
Universität Bonn
Römerstr. 164
D-53117 Bonn
Germany

e-mail: gk@cs.uni-bonn.de
phone: +49-228-550-276
fax: +49-228-550-382

Abstract

In this paper we introduce an object-oriented model that integrates class-based inheritance and object-based, dynamic delegation in the framework of a static type system and we show that implementation techniques for strongly typed, inheritance-based languages can be adapted to handle dynamic delegation efficiently. Our model and implementation scheme show how today's "production programming" systems can be smoothly extended to support object-based sharing and dynamically evolving objects, providing a degree of expressiveness and flexibility that was previously known only in the context of dynamically typed, prototype-based "exploratory programming" systems.

Keywords

inheritance, delegation, dynamic binding, static typing, roles, modifiable behaviour

Table of Contents

1. Introduction

Current object-oriented programming languages are not able to support both, rapid prototyping and production programming, equally well. Today the requirements of production programmers are best fulfilled by strongly typed, inheritance-based[1] languages ([Strou91], [Meye92], [MMN93]) whereas exploratory programmers prefer type-free ([GR89]) or even delegation-based ([LTP86], [Lieb86], [US87]) systems.

Delegation-based languages have introduced a uniform, type- and class-free object model and the concept of delegation, a dynamically modifiable inheritance relationship between *objects*. In the terms of the "Treaty of Orlando" ([SLU89]) delegation allows to express "unanticipated sharing on an object-by-object basis", providing a degree of flexibility and expressiveness that is lacking in the traditional object-oriented model based on static inheritance between classes.

The flexibility of object-based sharing by dynamic delegation is generally considered to be incompatible with static type checking and with the implementation techniques used in strongly typed, inheritance-based languages. This paper shows that delegation can be reconciled with static typing, yielding a basis for more powerful, yet safe, production programming systems and that static typing can be exploited to minimise the cost of method lookup in dynamically changing delegation hierarchies. The proposed implementation is compatible with "traditional" implementation techniques, showing how existing inheritance-based systems can benefit from the power of dynamic delegation.

The paper is structured as follows. In section 2 we clarify the apparent confusion that shows up in the literature about what delegation means. In section 3 we introduce a model that integrates inheritance, delegation, and static typing. The power of the integrated model is demonstrated on two examples, which are representative for two classes of problems that are not well supported by purely inheritance-based systems. The implementation is discussed in section 4, starting from the implementation scheme used in inheritance-based languages and gradually extending it to accommodate dynamic delegation. Section 5 describes the layout of the auxiliary runtime structures on which our implementation is based. Section 6 evaluates the space and runtime performance of our approach and section 7 compares it to other work in this domain.

2. Delegation ≠ Message Sending

Whereas nowadays the notion of *inheritance* is familiar even to beginners in object-oriented programming, the notion of *delegation* is overloaded in literature with three different meanings. In all cases an object, called the *child*, may have (dynamically modifiable) pointers to other objects, called its *parents*. E.g. in fig. 1a)-c) the object *employeeJohn* references its parent *personJohn* in the attribute **person*. Messages that cannot be answered by the child object are forwarded to the parent(s). Semantic differences arise due to the different methods of forwarding and the treatment of subsequent messages to *self*[2].

---

[1] Throughout the paper "inheritance-based" means "object-oriented languages based on classes and inheritance" whereas "delegation-based" means "object-oriented languages based on prototypical objects and delegation".

[2] In most object-oriented languages *self* denotes the receiver of the message that is being evaluated. The keywords *current* in Eiffel [Meye92]) and *this* in *Simula* ([DMN68]), *BETA* ([MMN93]) and *C++* ([Strou91]) have the same meaning.

Our understanding of delegation is the one initially introduced in actor-based languages (e.g. [Lieb86]) and used today in languages like *NewtonScript* ([Smit94a/b]), *Cecil* ([Cham93]), and *SELF* ([US87]). *Delegation* means that messages that cannot be answered using the receiver's message protocol are *automatically* forwarded to the parent(s) *without changing self*. When the forwarded message is answered by executing a parent's method, every subsequent message sent to *self* will be addressed to the receiver of the initial message. Hence the context of evaluation is automatically maintained to be the initial message receiver. E.g in fig. 1a, the message *employeeJohn.currentPhone#* cannot be answered by the object *employeeJohn*, since it contains no method for *currentPhone#*. Therefore the message is delegated to *personJohn*. When the method for *currentPhone#* found in *personJohn* is executed, and the message *self.phone#* is sent, *self* is still bound to *employeeJohn*, such that the phone number of John as employee (its office phone number) is returned. Intuitively, delegating a message means asking another object to do something *on behalf of the message receiver*, i.e. as if the message receiver would have done it.

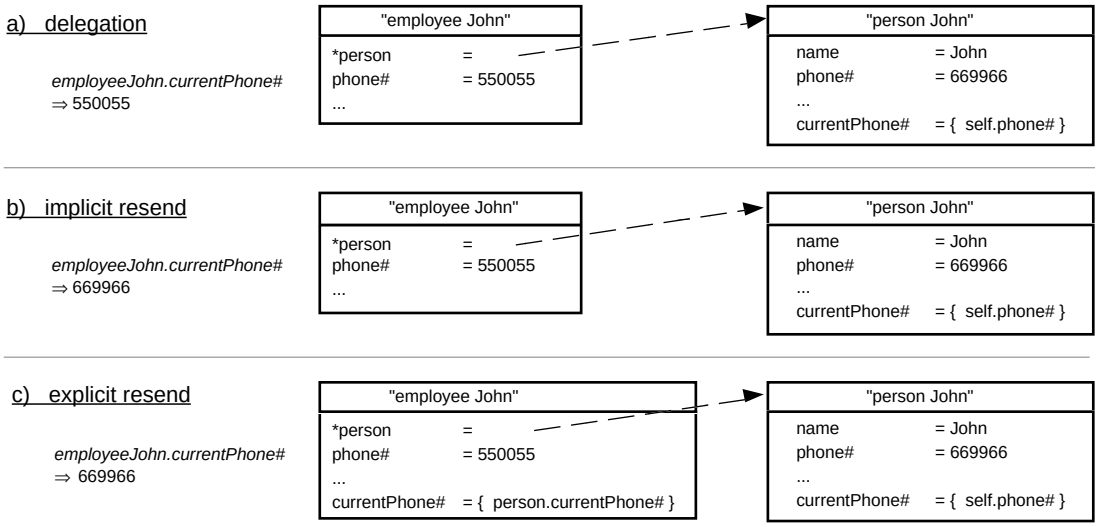


Fig. 1: Different results of the same message for delegation (a) and resends (b and c)

If *self* is changed, referencing the parent object during the evaluation of forwarded messages, all subsequent messages to *self* will ignore methods of the child (fig. 1b, 1c). Therefore the (private) phone number of John is returned, although the object which represents John as an employee was asked. Technically, we say that the receiver *resends* the message. Intuitively, resending means that the parent object will evaluate the message as *it* knows how, disregarding what the message receiver would have done.

There are two forms of resends, depending on whether message forwarding is done automatically (fig. 1.b) or requires explicitly adding to the child one method for every message that should be resent to the parent (fig. 1.c). In the former case we talk about *implicit resends*, in the latter about *explicit resends*. Unfortunately the term "delegation" is (mis)used in literature to denote both variants of resends, causing much confusion (delegation as implicit resends: [LTP86], [Strou87], [GSR94]; delegation as explicit resends: [RBP+91], p. 244]).

3. Combining Inheritance and Delegation

In *Delegation* ([Lieb86]), *SELF*, *Cecil*, and *NewtonScript* delegation is used in conjunction with a class-free object model, where every object is a self-contained entity (fig. 1), which may

define (and change) its own structure and behaviour. Especially, each object may define whether it has parents and who its parents are. We call this approach *unconstrained delegation*.

Integration of unconstrained delegation in inheritance-based object-oriented models would allow every instance to delegate, at any time, to any other object. In such a setting a compiler could not know whether a message that is not defined in the class of the receiver will produce a "message not understood" error at run time, or whether it will be answered by delegating to an object whose class contains a definition. If safety is a concern, all such messages have to be rejected by the type checker. This would have the effect of forbidding dynamic delegation altogether. On the other hand, if we were to allow such messages, the responsibility for program safety would be left to the programmer. Therefore requiring both, static type checking and dynamic delegation, sounds like a contradiction. Typically, implemented languages that combine delegation and typing restrict delegation to be static ([Cham93], [CL94]). Fortunately it is possible "to have one's cake and eat (much of) it too", by reverting to a slightly restricted form of dynamic delegation, that we call *typed dynamic delegation*.

3.1. Basic Notions

In this section we shall introduce the basic notions of typed dynamic delegation. The presentation is oversimplified, since we restrict ourselves to the aspects that are essential for understanding the special problems involved in the implementation of our integrated model. We assume that the reader is familiar with the standard notions of class, instance, and class-based inheritance that are common in most of today's object-oriented languages and systems ([Wegn89], [Snyd91], [Nier89], [GR89], [Stro91], [Meye92]). We also assume that inheritance is based on overriding [CoPa89]) rather than extension ([BrCo90], [MMN93]).

In our model, as in many other existing systems, objects are instances of classes, classes may inherit from each other, and all variables, parameters and method bodies have a statically declared type. Additionally, objects (instances) may delegate to other objects by referencing them in their *delegation attributes*. We talk about ***typed dynamic delegation*** if the existence and the type of all delegation attributes of instances of a class, *C*, is declared in *C*. If class *C* declares a delegation attribute of type *T*, we say that *C* ***delegates to*** *T* and that

- *C* is a ***declared child class*** of *T* and of each of *T*'s subclasses, and
- *T* is a ***declared parent class*** of *C* and of each of *C*'s subclasses.

Please note that "class *C* delegates to class *T*" is just a shorthand for saying "class *C* defines that its instances and the instances of its subclasses delegate to instances of *T* or of *T*'s subclasses"[1]. E.g. in fig. 2, class *B* delegates to class *C* and therefore the object obj_b_1, an instance of B_1, may delegate to the object obj_c_m, an instance of C_m.

Each subclass of a declared parent (child) class of a class, *X*, is a ***potential parent*** (***child***) ***class*** of *X*. The ***parent*** (***child***) ***classes*** of a class are the union of its declared and potential parent (child) classes. E.g. in fig. 2b, the class *D* is a declared parent of $C, C_1, ..., C_m$, the subclasses of *D* are potential parents of $C, C_1, ..., C_m$, and the subclasses of *C* are potential children of $D, D_1, ..., D_n$.

---

[1] Note that this is very different from saying that the object representing class *C* delegates to the object representing class *T*.

The ***ancestor*** (***descendant***) relation is the transitive closure of the parent (child) relation. The ***declared ancestor*** (***descendant***) relation is the transitive closure of the declared parent (child) relation. The ***potential ancestors*** (***descendants***) of a class are the difference between its ancestors (descendants) and its declared ancestors (descendants)[1]. E.g. in fig. 2, the classes *C*, C_1, ..., C_m, *D*, D_1, ..., D_n, are the ancestors of *B*, the classes *C* and *D* are the declared ancestors of *B*, and the classes C_1, ..., C_m, D_1, ..., D_n, are the potential ancestors of *B*. Note that any (declared or potential) new ancestor class of C_1, ..., C_m, D_1, ..., D_n, would also be a *potential* ancestor class of *B* (but not a *declared* ancestor).

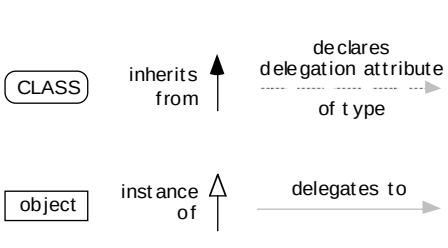


Fig 2a: Graphical notation

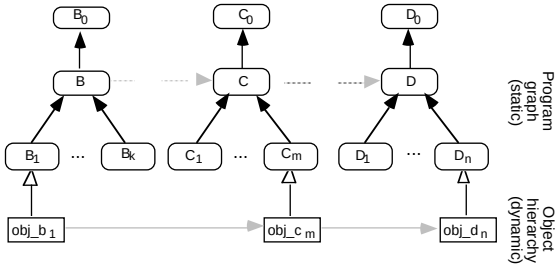


Fig. 2b: Example of program graph and object hierarchy

It will often be useful to regard a program as a directed graph consisting of class nodes connected by inheritance edges and delegation edges (upper part of fig. 2b). A path in the program graph that contains only inheritance edges is called an ***inheritance path***. A path in the program graph that *ends* with a delegation edge is called a ***delegation path***.

We say *A* ***is_a*** *B*, and call *A* a ***direct subclass*** of *B*, resp. *B* a ***direct superclass*** of *A*, if *A* is connected to *B* by an inheritance path of length 1. We say *A* ***is_a**** *B*, and call *A* a ***subclass*** of *B* resp. *B* a ***superclass*** of *A* if *A* is connected to *B* by an inheritance path of length ≥ 1. The ***inheritance hierarchy of a class*** is the subgraph of the program graph containing the class itself and all its superclasses and subclasses. The ***delegation hierarchy of a class*** is the subgraph containing the class itself and all its ancestors and descendants.
For a class *C* we denote its inheritance hierarchy by ***C.inherHierarchy,*** and its delegation hierarchy by ***C.delegHierarchy***. The inheritance hierarchy belonging to an *object* consists of its class, say *C*, and the superclasses of *C*.

All non-disjoint inheritance hierarchies of *classes* form an inheritance hierarchy of the *program graph*. More precisely, two classes, *X* and *Y*, are in the same ***inheritance (delegation) hierarchy*** *of the program* ***graph***, if there exists a sequence of classes, C_1, ..., C_n, with $C_1 := X$ and $C_n := Y$, such that for all $i \in \{1, ..., n-1\}$ the inheritance (delegation) hierarchies of *class* C_i and C_{i+1} have a non-empty intersection, or if *X* and *Y* are part of the inheritance (delegation) hierarchy of classes that are in the same inheritance (delegation) hierarchy of the *program graph*.

E.g. the program graph shown in fig. 2 contains three inheritance hierarchies (one of them is B_0, *B*, B_1, ..., B_n,) and one delegation hierarchy (containing all classes *except* B_0, C_0, and D_0).

The (own) ***protocol*** of a class is the set of all messages defined in the class and its superclasses. The ***extended protocol*** of a class is the union of its own and its ancestors' protocols. The ***delegated protocol*** is the difference of the extended protocol and the (own) protocol.

---

[1] Note that this is more then just the transitive closure of the potential parent relation. E.g. it also includes all *declared* ancestors of potential parents.

The ***abstract protocol*** of a class, *C*, is the extension of its protocol by the selectors of all messages that are sent to *self* in *C* or in *C*'s superclasses. The ***delegated abstract protocol*** and the ***extended abstract protocol*** are defined accordingly.

A class is ***concrete*** if all the *self*-messages sent in its methods are in its *extended* protocol. Only concrete classes may be instantiated.

The following table contains the formal definitions of the notions introduced so far. In the table, *P* denotes a program, and *C*, *X*, *Y* denote classes from *P*.

Notation	Definition
C.superclasses	$:= \{\ Y \mid C\ is\_a^*\ Y\ \}$
C.subclasses	$:= \{\ Y \mid Y\ is\_a^*\ C\ \}$
C.declParents	$:= \bigcup_{X \in \{C\} \cup C.superclasses} \{ Y \mid X\ delgates\_to\ Y \}$
C.declChildren	$:= \bigcup_{X \in \{C\} \cup C.superclasses} \{ Y \mid Y\ delgates\_to\ X \}$
C.potParents	$:= \bigcup_{X \in C.declParents} X.subclasses$
C.potChildren	$:= \bigcup_{X \in C.declChildren} X.subclasses$
C.parents	$:= C.declParents \cup C.potParents$
C.children	$:= C.declChildren \cup C.potChildren$
C.ancestors	$:= C.parents \cup \left(\bigcup_{X \in C.parents} X.ancestors\right)$
C.descendants	$:= C.children \cup \left(\bigcup_{X \in C.children} X.descendants\right)$
C.declAncestors	$:= C.declParents \cup \left(\bigcup_{X \in C.declParents} X.declAncestors\right)$
C.declDescendants	$:= C.declChildren \cup \left(\bigcup_{X \in C.declChildren} X.declDescendants\right)$
C.potAncestors	$:= C.ancestors \setminus C.declAncestors$
C.potDescendants	$:= C.descendants \setminus C.declDescendants$
C.inherHierarchy	$:= C.subclasses \cup \{C\} \cup C.superclasses$
C.delegHierarchy	$:= C.descendants \cup \{C\} \cup C.ancestors$
P.inherHierarchies	$:= \{ H \subseteq P \mid \forall X, Y \in H: (\exists \{C_1, ..., C_n\} \subseteq H: (C_1 = X \wedge C_n = Y \wedge \forall i = 1,...,n\text{-}1: C_i.inherHierarchy \cap C_{i+1}.inherHierarchy \neq \emptyset)) \}$
P.delegHierarchies	$:= \{ H \subseteq P \mid \forall X, Y \in H: (\exists \{C_1, ..., C_n\} \subseteq H: (C_1 = X \wedge C_n = Y \wedge \forall i = 1,...,n\text{-}1: C_i.delegHierarchy \cap C_{i+1}.delegHierarchy \neq \emptyset)) \}$

Table 1: Summary of notation and definitions.
P denotes a program (i.e. a set of classes), and *C*, *X*, *Y* denote classes from *P*.

3.2. Semantics and Use

A detailed discussion of the formal semantics of our model is outside the scope of this paper. In this section we shall confine ourselves to the discussion of the peculiarities of our model that are needed for understanding the implementation. We conclude the section with two examples that illustrate typical application scenarios and show the added power of our model compared to purely inheritance-based ones.

3.2.1. Overriding

In most inheritance-based systems (with the notable exception of *Simula* ([DMN68]) and *BETA* ([MMN93])) methods in subclasses override methods in superclasses. Similarly, in delegation-based systems, methods in child objects override methods in parent objects. The joint use of two different hierarchies requires an adaptation of the rules for overriding. The extension seems to be straightforward:

(a) a method defined in a class always overrides methods defined in its superclasses and

(b) a method defined in a class *or* in its superclasses always overrides methods defined in its ancestor classes.

This rule would preserve the semantics of each of the "pure" models. Unfortunately, in practice there is a catch in this intuitively and theoretically appealing solution. Application of the above rule to the message *obj_a.b* would produce the same result in both examples from fig. 3. After searching a definition for *b* in *A*, the message would be delegated to *obj_b* and the class B_1 would be searched. The method found there would be executed, and the message *self.x* would be sent next. The search would start again in class *A* (*self = obj_a*) leading to the execution of the local method for *x* instead of the one from B_1. The problem lies in the subtle difference between fig. 3a and fig. 3b.

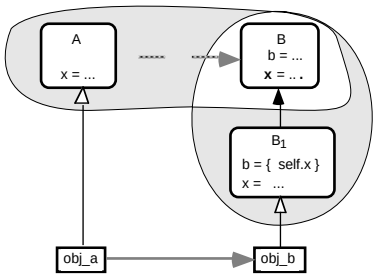


Fig. 3a: *A::x* overrides B_1*::x*

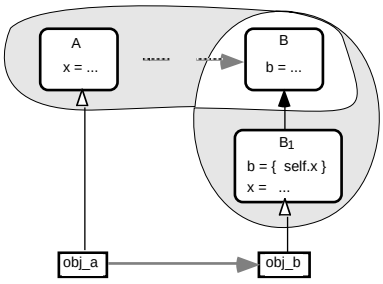


Fig. 3b) *A::x* may *not* override B_1*::x*

In fig. 3a, method *x* was first "introduced" in class *B*, and was then *independently redefined* in the descendant class *A* and in the subclass B_1. The two redefined versions of *x* most likely have a similar semantics, since they both modify the same original definition. Hence overriding simply selects the method whose semantics is better suited for objects of type *A*.

In contrast, the different definitions of *x* in figure 3b were *independently introduced* in two different classes, *A* and B_1, that may have been developed and compiled independently, maybe even by different programmers, or as parts of different libraries. It is at least very unlikely that they have the same semantics. Hence overriding would lead to a silent change of the meaning

of *x*, whose effect would propagate to all methods from B_1 that use *x*. This could produce very obscure and hard to locate run-time errors.

In order to avoid such a silent change of meaning when independently introduced methods accidentally have the same name, we use the following, modified ***rule for overriding***:

(a) a method defined in a class always overrides methods defined in its superclasses and

(b) a method defined in a class or its superclasses *only* overrides methods defined in ancestor classes, *if* the method's selector was already in the extended protocol of a *declared* ancestor class.

In our example, the declared ancestor class is *B*. The condition that the method's selector must be in the *extended* protocol of *B* says that the method may also be defined in a superclass of B or a declared ancestor class of *B*.

3.2.2. Static Typing

In our model it is possible to check statically that only messages that are safe for the *declared* type of a delegation attribute may be delegated via that attribute[1]. The validity of static type checking is based on the assumption that delegation attributes do not have the value *nil*, i.e. parent objects of the specified type, to which locally undefined messages can be delegated, always exist. Ensuring this invariant requires run-time checks of assignments to delegation attributes[2] and garbage collection. If an attribute is assigned a non-nil value, garbage collection will ensure that this value persists until the next assignment, i.e. the referenced object will not inadvertently be disallocated due to some programming error.

Defining at class level the type of objects to which instances may delegate, may, at a first glance, appear very restrictive compared to the use of delegation in untyped, prototype-based systems. However, many interesting applications of dynamic delegation can be expressed by typed delegation. In the next two sections we shall illustrate by typical examples how typed delegation can be applied to problems whose solution is not supported by traditional, purely inheritance-based object models.

3.2.3. Roles of Objects

A main weakness of purely inheritance-based systems, which has often been criticised ([SLU89], [Pern90], [NRE 92], [ABGO93], [GSR94]), is their failure to model dynamic evolution of the world. One facet of dynamic evolution is the ability to acquire and *play* different *roles*. E.g. the fact that a person may become a student, and later, or simultaneously, an employee, is an example of two different roles that can be played by persons. In the following we will review the role modelling potential of three general-purpose mechanisms, multiple inheritance, multiple instantiation, and explicit manipulation of object references. Then we will show that typed delegation allows to do role modelling in a way that overcomes the difficulties

---

1 A detailed discussion of typing issues is contained in [Knie95].

2 These checks are not expected to have a significant effect on performance, since, even in dynamic delegation systems, like *Self*, changes of parent attributes represent a minuscule fraction of the overall computation done by a program.

of these approaches, while being more widely applicable. A comparison of our approach to specialised "role object models" will be done in chapter 6.

In languages where objects cannot change their class dynamically, multiple inheritance is often abused for role modelling but it provides no solution, due to the combinatorial explosion of the number of highly specialised "intersection subclasses" ([Scio89]) with ever decreasing generality and reusability. In order to be prepared for every situation that might occur during the lifetime of an object one always has to create instances of a "most specific" class, thus wasting storage and obscuring the semantics of the application (fig. 4a).

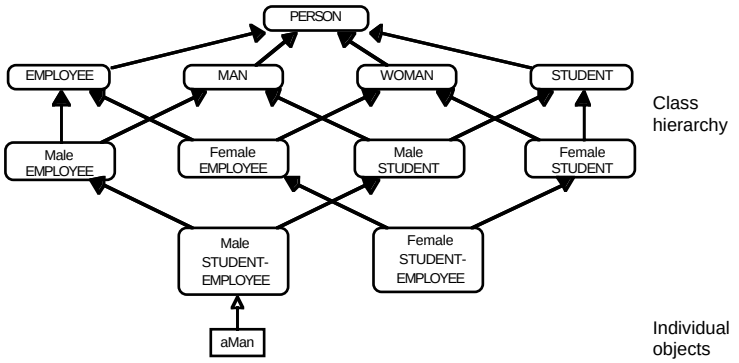


Fig. 4a: Extreme multiple inheritance-based modelling
of a man that *might become* a student and/or an employee

If one resorts to instances of more general classes, a situation like the one illustrated in fig. 4b could occur. A man that is simultaneously an employee and a student would be represented by two unrelated objects, which duplicate the information about him as a male person. Keeping this information consistent would be left as a burden to the application programmer, who would have to code two messages whenever the redundant information must be updated, e.g. *aMaleEmployee.setAddress(...)* and *aMaleStudent.setAddress(...)* for consistently changing the address of the same person.

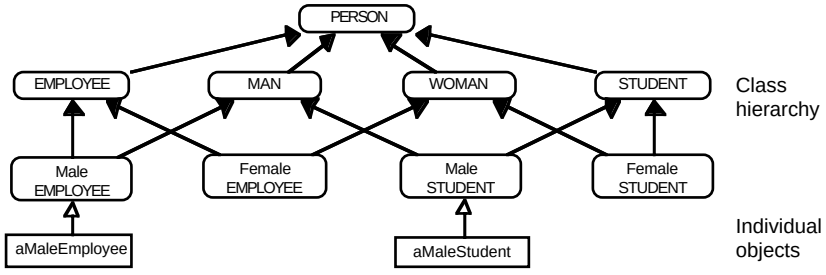


Fig. 4b: Alternative multiple inheritance-based modelling
of a man that *is* a student and an employee

Smalltalk allows any object to *become* any other object (*receiverObj become: otherObj*, [GR89]). This facility is based on an indirect reference model, where an object reference is an index into a global table that contains the physical addresses of all objects in the system (fig. 4c). By changing one entry in the global table, all references to a certain object can be redirected to another object.

This mechanism allows to achieve the desired effect only if the class of the receiver object is related by inheritance to the class of the argument object. E.g. assuming the inheritance hierarchy from fig 4c, one can model that a person becomes a student by creating a new *STUDENT* instance, *aStudent*, copying common information from the existing *PERSON*

instance, *aPerson*, to *aStudent*, and using the message *aPerson become: aStudent* to redirect all references from the old object to the new one.

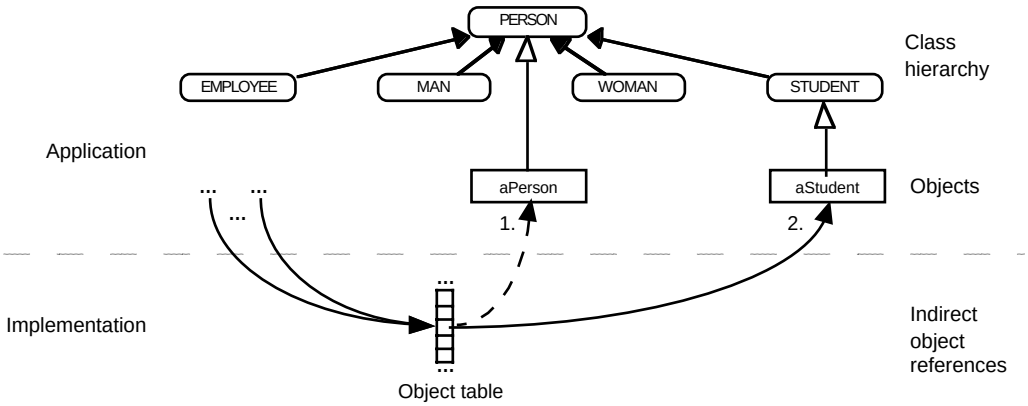


Fig. 4c: Object reference manipulation used for modelling a person that *became* a student

However, when the class of the old object is neither a sub- nor a superclass of the class of the new object, uncommon attributes and methods of the old object will be lost. E.g. with the inheritance hierarchy from fig. 4c, a *man* can only become an *asexual* student. There are two more drawbacks of Smalltalk's "*become*" method: Allowing any object to become any other object is conceptually unsound and the dependence on indirect addressing is generally considered an unacceptable efficiency bottleneck for compiled languages.

A more principled approach is taken in the database programming language "*Chimera*" ([CeMa 93]), which allows objects to be instances of different classes *simultaneously* (*multiple instantiation*), and to change some of their classes along the type hierarchy (*object migration*). In this model an instance of *MAN* may *additionally* become an instance of *EMPLOYEE* and of *STUDENT* (cf. the instantiations 1, 2, and 3 in fig. 4d). However, since all classes that have a common superclass may simultaneously contain the same object, there is no means to express mutually exclusive instantiations, e.g. that a man cannot simultaneously be a woman (the instantiation 4, shown by the dashed line in fig. 4d would be possible *in addition* to no. 1).

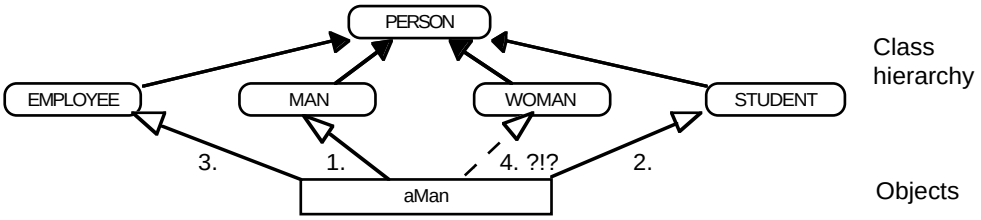


Fig. 4d: Multiple instantiation in Chimera

Other semantic problems of this approach are due to the need to resolve conflicts between methods with the same selector defined in different "most specific" classes. In [CeMa 93] it is requested that "one of [the most specific subclasses] has to be distinguished for type checking purposes, and determines which implementation of shared attributes is to be applied". Unfortunately, this criterion is only applicable if the "distinguished" class is among the classes defining the conflicting method. Otherwise it remains open how the conflict could be solved (e.g. if in fig. 4d *MAN* is the distinguished class but only *EMPLOYEE* and *STUDENT* contain a conflicting method). Further it is unclear how this approach can be implemented and compiled efficiently, since there is no way to anticipate which class memberships will be acquired at run-time by an object.

In contrast to the previous approaches, we represent an entity and each of its roles explicitly by individual objects[1]. E.g. in fig. 5, the conceptual object *John* is represented as a male person by an instance of *MAN* (*man1*), as an employee by an instance of *EMPLOYEE* (*emp1*), and as a student by an instance of *STUDENT* (*student1*). Since the classes *EMPLOYEE* and *STUDENT* do not inherit from *MAN*, no "male person" information, is contained in their instances. However, by delegating to *man1*, the objects *emp1* and *student1* can use the information of *man1* as if it were their own, e.g. the message *emp1.set_address(...)* would change the address attribute of the shared object *man_1* and this change would immediately be visible to the role *student1*.

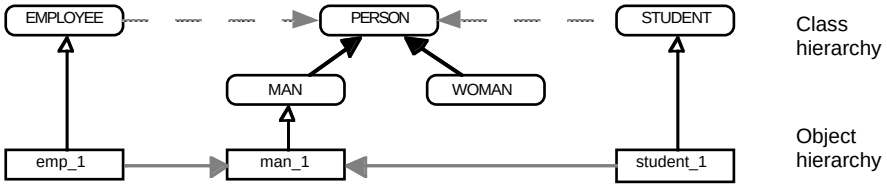


Fig. 5: Combined modelling using inheritance and typed delegation

Thus the solution depicted in fig. 5 avoids all the problems of the purely inheritance-based modelling (cf. fig. 4a,b): common information is factored out in a shared object that is transparently accessed by delegation (relieving the programmer from the burden of manually keeping track of object-specific sharing dependencies), objects are just as big as necessary, and no proliferation of multiply inheriting classes is needed at all, since the possible combinations of simultaneous roles are expressed by the *object-level* delegation hierarchies.

In addition, the *class-level* declaration of delegation (dashed grey lines in fig. 5, cf. fig. 2) allows to statically capture potential run-time dynamicity: immutable and mutually exclusive aspects of a concept (e.g. the sex of a person) are modelled by inheritance, whereas transient aspects (roles), and object-specific sharing are modelled by delegation. Thus the availability of two class-level hierarchies, inheritance and delegation, provides the technical basis for better conceptual modelling. E.g. the structure of our model of the person-student-employee application (fig. 5) implicitly expresses the constraint that a person cannot simultaneously be male and female.

Finally, the example also illustrates that our object model subsumes specialised "role object models" e.g. the ones presented in [WRS94] and [GSR94]. The modelling techniques proposed in these approaches are more powerful within our framework, since they benefit from the advantages of delegation over simple resend mechanisms (cf. chapter 2). A more detailed comparison of our approach to "role object models" is contained in chapter 6.

3.2.4. Dynamic Change of Behaviour

Another aspect of the added flexibility of our approach is the ability to model dynamic change of behaviour. Change of behaviour via *dynamic* delegation is within the realm of typed delegation since subtyping allows instances of a child class to delegate to objects that are instances of declared parent classes *or of their subclasses*. The following example illustrates an application of dynamic delegation.

---

[1] In their analysis of prototype-based systems, [DMC92] called this a *split-object* representation. We keep this terminology, although our system is not prototype-based.

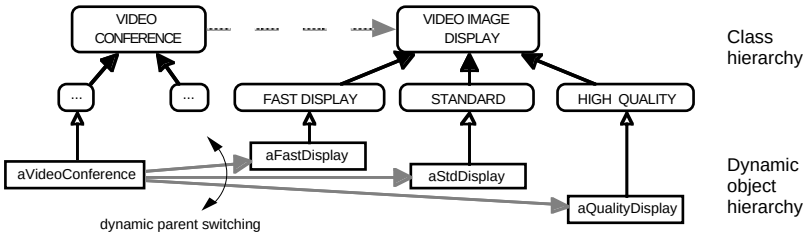


Fig. 6: Adaptation of behaviour to run-time conditions by switching of parent objects

In a video conferencing system, change of behaviour in response to run-time events is a common requirement; e.g. when heavy processing slows down the graphic display below a given rate, one would like to resort to a faster, though less accurate, video image display method and to return to the standard display or even high resolution display if the circumstances are favourable. However, one would not like to clutter the video conferencing code with permanent checks of the current conditions and case statements that explicitly call different locally defined display methods. Also, simply *resending* the display message to some parent object, *parent.display*, would often be inappropriate, since it would not allow to reuse the display method of another object but adapt it to own needs by redefining messages sent to *self* during execution of *display*.

Our model allows to use a single message, *self.display*, throughout the *VIDEO CONFERENCE* class, and to delegate calls of *display* to a parent object (fig. 6). The *VIDEO CONFERENCE* class only needs to provide methods that switch the parent reference between instances of different subclasses of *VIDEO_IMAGE_DISPLAY* that implement different versions of the *display* method. When necessary, a video conference object changes the value of its parent attribute, e.g. to a *FAST_DISPLAY* instance. Everything else is done by dynamic binding.

4. Compilation of Inheritance and Delegation

We describe our implementation scheme as a stepwise extension of the state-of-the-art compilation techniques for strongly typed, inheritance-based languages, which are briefly reviewed in section 4.1. Section 4.2. describes the extension of the compiled method format. The basic extensions of the dynamic binding scheme are introduced in section 4.3. Section 4.4. discusses the special treatment required for messages to *self* sent during the evaluation of delegated messages. Finally (4.5) we show that multiple delegation, multiple inheritance, and their joint use neatly fit into our framework, at conceptual as well as implementation level.

4.1. Compilation of Inheritance-based Languages

This section reviews the state-of-the-art compiled method format and dynamic binding scheme which is used in strongly typed, inheritance-based languages, e.g. in *C++* ([ES90]). Here we limit ourselves to single inheritance. Multiple inheritance will be discussed in section 4.5.

A method of the form

selector(params) = { ... source code ... }

is compiled as a function

renamed_selector(receiver, params) = { ... compiled code ... },

where *renamed_selector* guarantees uniqueness of compiled method code in the presence of selector overloading and *receiver* denotes the object for which the method will be called at run-time. The *receiver* argument is used as the environment for instance variable accesses and as the addressee of all *self* messages in the body of the method.

A class is represented at run-time by a *dispatch table* that contains one entry for each selector in its *abstract* protocol[1]. Every instance has a reference, *dTable*, to the dispatch table of its class (fig. 7b). Every entry in the dispatch table references the compiled code that is to be executed when instances of the class receive messages with the respective selector[2]. Inheritance is trivially implemented by referencing the same code in all subclasses that do not redefine a given method (e.g. method *C::a* in fig. 7b). If the method with selector *sel* is referenced at position *j* in the dispatch table of class *C*, ***index(sel, C)*** = *j*, then it is referenced at the same position in the tables of all subclasses of *C*, i.e.

$$\forall\ SC \in C.subclasses : index(sel, SC) = index(sel, C).$$

Thus the dispatch table structure of a class (type) is preserved in the dispatch tables of its subclasses (subtypes).

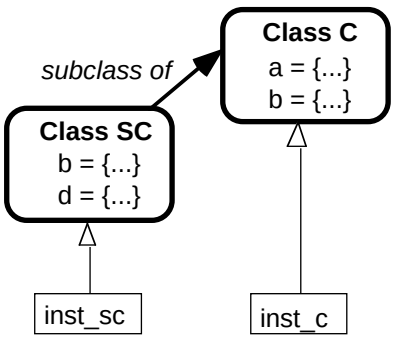


Fig. 7a) Classes and instances

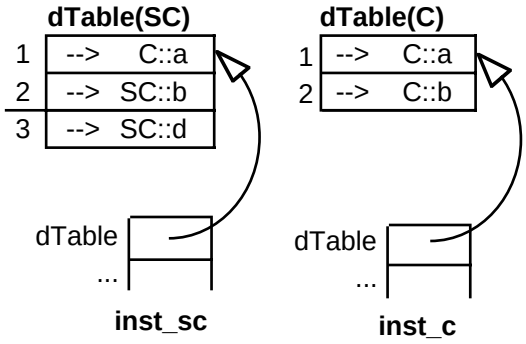


Fig. 7b) Corresponding dispatch tables and (part of) object layout

Since the declared type, *T*, of each message receiver is known at compile-time, the uniform dispatch table layout allows constant-time dynamic binding. A message of the form

receiver.selector(args)

is compiled as an indirect procedure call where the first statement corresponds to the dynamic binding of the message selector to the address of a compiled method and the second statement calls the determined code with the suitable arguments.

```
method := receiver.dTable[index(selector,T)[3]]        // dynamic binding
method(receiver,args)                                  // procedure call
```

Code template 1: "Standard" dispatch method in typed, (single) inheritance-based languages

---

1 Whenever we talk of protocols in conjunction with dispatch tables we always mean the corresponding abstract protocol version (cf 2.1.), even if this is not mentioned explicitly.

2 An entry of the form "*-->Class::selector*" represents a reference to the compiled code of the method with selector *selector* defined in class *Class*.

3 In all pseudo-code expressions underlining indicates the statically known parts (constants).

This scheme only involves

- reading the "dispatch table" reference of the receiver: `receiver.dTable`
- adding the statically known offset of the message selector: `[index(selector,T)]`
- reading the code reference at that address: `method := ...`
- and calling the referenced code: `method (receiver, args).`

It was our main design goal to preserve this scheme *exactly* for the case that delegation is not used and as closely as possible otherwise, for efficiency reasons as well as for compatibility with most widely-used languages.

4.2. Extension of Compiled Method Format

If delegation is added, messages delegated to an ancestor object will sometimes need to access variables of the ancestor (the *delegatee*), not of the message receiver. Therefore, in general, compiled methods must have one more system-generated parameter, the *delegatee*. A method of the form

selector(params) = {... source code ...}

is compiled as

*renamed_selector(receiver, **delegatee**, params)* = {... compiled code ...}.

The *receiver* argument denotes the initial receiver of the message, and is used as the addressee of *self* messages in the compiled code, thus maintaining the scope of message lookup.

The *delegatee* argument is used as the scope of instance variable accesses, whenever messages delegated to an ancestor object need to access variables of the ancestor. Note that we do not make any assumptions on whether variable accesses are statically bound to the delegatee or regarded as dynamically dispatched messages to *self*. The difference is only that in the first case variable references will *always* access local variables of the delegatee, whereas in the second case this will only happen for variables that are not contained in the receiver (resp. in objects on the path from the receiver to the delegatee).

A message *obj.selector(args)* that is bound to a method from the class of *obj*, will result in a function call of the form

*renamed_selector(**obj**, **obj**, args).*

When the message is delegated and bound to a method from an ancestor, *obj2*, the call will be

*renamed_selector(**obj**, **obj2**, args).*

Note that the *delegatee* parameter is only needed when a method is executed for a delegated message. Otherwise the "standard" compiled code format could be used. We will come back to this issue in section 4.4.3.

4.3. Extension of Dynamic Binding

In order to implement dynamic delegation in a typed, inheritance-based system, the "classic" dynamic binding scheme must be extended in various ways. In this section we shall discuss why dynamic binding in the presence of delegation is harder than in inheritance-based systems and we shall develop an efficient extension of the "classic" approach.

Consider the example from figure 8 and a variable *x* of declared type *C*. Clearly, the "classic" dynamic binding scheme (4.1.) will continue to work for messages from *C*'s *own* protocol, e.g. the message *x.b* will be bound correctly, no matter whether *x* references the object *child1* or *child2*. The question is, how to treat messages that must be delegated, e.g. *x.f*? The method *E::f*[1], which is the only definition for *f* that is statically known for type *C*, can be overridden by definitions in subclasses of *E*, *D*, or *C*. Which code is to be executed depends on the run-time type of the message receiver *and* on the run-time types of its ancestor objects. E.g.

- "*x := child1; x.f*" should call *SD::f* whereas
- "*x := child2; x.f*" should call *E::f*.

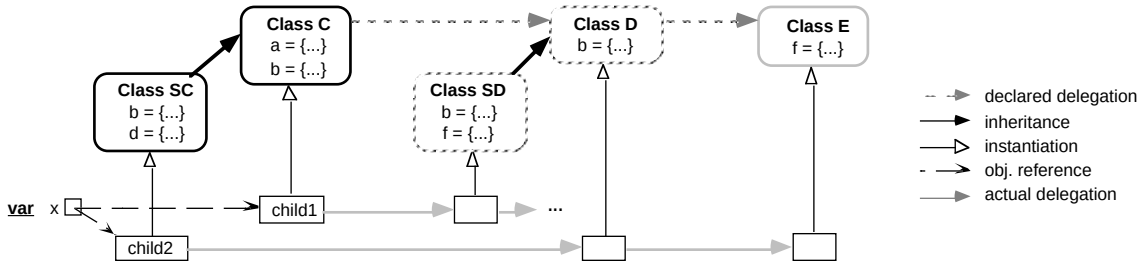


Fig. 8: A combined inheritance and delegation hierarchy

Note that the "right" code to be executed originates from different inheritance hierarchies, and the reference to it is located at different dispatch table positions within each hierarchy (*f* has index 1 in *E*'s hierarchy and index 2 in *SD*'s hierarchy).

4.3.1. Extended dispatch tables

The main limitation of the classic scheme, illustrated by the above example, is that the dispatch table of a class only contains entries for the selectors from its own protocol. Selectors from its delegated protocol have no unique, statically known dispatch table indices. Thus the invariant that assured dispatch table lookup in constant time does not hold for them. This problem can be solved by introducing *extended dispatch tables*, which contain entries for all selectors in the *extended protocol* of a class. Fig. 9 shows the extended dispatch tables created for the classes from the previous example. The dispatch table entries that have been left incomplete (with a question mark instead of a class name) will be treated in the next section.

The ***extended dispatch table*** of a class, *X*, contains an ***inherited section***, corresponding to the dispatch table of the superclass, a ***new section***, for the new messages defined in *X*, and different ***delegated sections***, corresponding to the dispatch tables of its different *declared* parent classes. E.g. in fig. 9 the extended dispatch table of class *SC* contains an inherited section (index 1 - 4) and an own section (index 5). The inherited section corresponds to the extended dispatch table

---

1 "*Class::selector*" denotes the method with selector *selector* defined in class *Class*.

of class *C*, and contains a delegated section corresponding to the extended dispatch table of *C*'s declared parent class *D* (index 3 - 4).

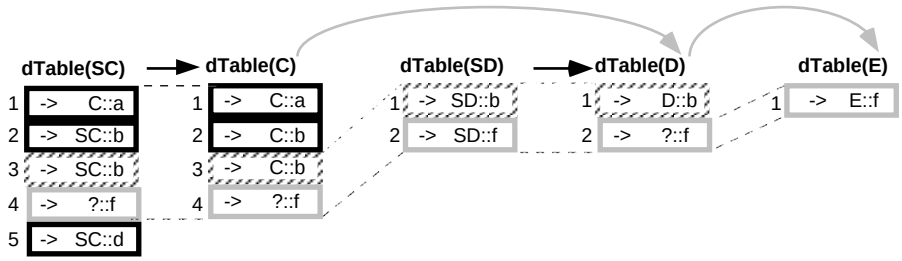
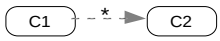


Fig. 9: Extended dispatch tables for the example from fig. 8. The patterns are the same as in fig. 8. Each pattern highlights the dispatch table section corresponding to the protocol of one declared ancestor class.

For describing the structure of extended dispatch tables, we introduce two functions, *offset(C1,C2)*, and *declaredOffset(C1,C2)*. They return the displacement of selector indices from the dispatch table of class *C2* within the dispatch table of class *C1*.

The partial function ***declaredOffset(C1,C2)*** is defined, if *C2* is a *declared* ancestor class of *C1*. It returns the last index before the beginning of *C2*'s section in *C1*'s dispatch table. (e.g. in fig. 9, *declaredOffset(C,D)* = 2, *declaredOffset(D,E)* = 1, *declaredOffset(C,E)* = 3).



The total function ***offset(C1,C2)*** extends the definition of *declaredOffset*:

offset(*C1, C2*) :=

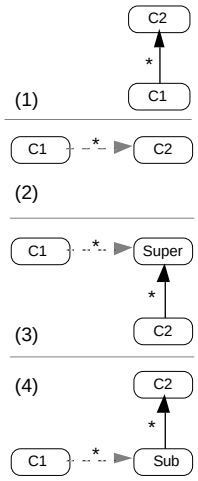
(1) if $(C2 \in C1.superclasses) \vee (C2 = C1)$
then 0

(2) else if $C2 \in C1.declAncestors$
then *declaredOffset*(*C1, C2*)

(3) else if $\exists\ Super := nearestSuperclass(C2, C1.declAncestors)$[1]
then *declaredOffset*(*C1, Super*)

(4) else if $\exists\ Sub := anySubclass(C2, C1.declAncestors)$
then *declaredOffset*(*C1, Sub*)

(5) else "undefined"



Case (1) expresses the trivial case that methods defined in *C1* itself or in one of its superclasses have offset zero within *C1*. Case (2) says that *offset*(*C1, C2*) = *declaredOffset*(*C1, C2*) if the latter is defined. Case (3) says that all subclasses of a declared ancestor class, *Super*, use the same delegated section within *C1*, hence their selectors have the same displacement as those of the declared ancestor class (e.g. in fig. 9 *offset(C,SD) = declaredOffset(C,D)* = 2 = *offset(C,D)*). Case (4) expresses that the methods inherited by a declared ancestor class, *Sub*, from its superclass, *C2*, have the same displacement within the dispatch table of *C1* as all other methods of *Sub*. If *C1* has different declared ancestors that are subclasses of *C2* then the methods inherited from *C2* will be contained in the dispatch table section of each of them. Therefore one can choose the offset of *C2* to be the offset of any of these classes. This is done by the function

[1] The function *nearestSuperclass(class,setOfClasses)* returns the class from *setOfClasses* that is reachable from *class* via the shortest inheritance path.

anySubclass. A detailed example is contained in section 5.1 (fig. 15 and 17, and table 3 and 4).

Unlike in purely inheritance-based systems, a selector may have multiple indices in the dispatch table of a class, due to the simultaneous existence of an index in the dispatch table of the superclass and of some parent class(es). The rule that locally / inherited methods generally override those of the delegation parents is trivially implemented by referencing the local / inherited code at *all* dispatch table positions assigned to the same selector[1]. E.g. in fig. 9 it was assumed that *b* was already defined in a superclass of *C*: there is an entry for *b* in the inherited dispatch table section and one in the delegated section. Both reference the local method for *b*.

In order to account for the existence of different indices for the same selector, the function ***index(sel,Class)*** has been redefined to return *sets* of indices. With the functions *index/2* and *offset/2* we can now specify the dependency between the structure of the extended dispatch table of a class, *C*, and the structure of the dispatch tables of its subclasses / descendant classes.

> If the selector *sel* has index *i* in the dispatch table of class *C*, $i \in$ *index(sel,C)*, then
>
> (1) the dispatch table of each *subclass* of *C* has an entry for *sel* at index *i*, and
>
> (2) the dispatch table of each *descendant* class, *Desc*, has an entry for *sel* at index *i+offset(Desc,C)*.

Condition (1) assures compatibility with the traditional dispatch table layout. It implies that each dispatch table starts with the *inherited section*. The *delegated sections* as well as the *new section* may be laid out in any order with respect to each other. Note however that, once fixed for a class, the order of the new and the delegated sections must be preserved in the tables of its descendants. This is implied by condition (2), which says that all indices of selectors from a declared ancestor class are shifted by the *same* offset in the layout of a specific descendant's dispatch table.

4.3.2. Customised Lookup Code

Extended dispatch tables assure that statically known indices exist for *all* selectors in the extended abstract protocol of a class. However, for selectors from the *delegated* protocol that are not locally redefined, there is no unique method definition that could be statically referenced in the extended dispatch table. Which method is appropriate depends on the classes of the receiver's *ancestor* objects. E.g. the message *child1.f* must be bound to *SD::f* whereas the message *child2.f* must be bound to *E::f* (fig. 8/9).

The solution is to reference a short code sequence that will dynamically determine the right method. It does so by "switching the lookup context" to the parent of the current object and by applying then the usual dynamic binding scheme, taking advantage of the fact that the index of the searched selector in the *parent's* extended dispatch table is statically known.

If no method for *selector* is defined in the class *C* (or its superclasses) the entries for *selector* in the dispatch table of *C* will reference the ***customised lookup code*** obtained by specialising the following procedure, *delegate*, on its statically known arguments (*selector* and *parentType*):

---

[1] For the implementation of the exception introduced in 3.2.1. see section 4.4.2.

```
delegate(receiver, delegatee, args, selector, parentType) = {
  delegatee := delegatee.parent                                  // get parent object
  method := delegatee.dTable[index(selector, parentType)1]       // standard dispatch
  method(receiver, delegatee, args)
}
```

Code template 2: Customised lookup code

The customised lookup code produced for the delegated messages from fig. 8. and 9. is shown in the following table.

the table of class	references at each of the positions	the customised lookup code
C, SC	*index(f,C)* = {4}	*delegatee**.parent.**dTable**[2]** (receiver, **delegatee.parent**, args)*
D	*index(f,D)* = {2}	*delegatee**.parent.**dTable**[1]** (receiver, **delegatee.parent**, args)*

Table 2: "Customised lookup code" referenced in dispatch tables of classes from fig. 8 / fig. 9

Summarising, the referenced code is either the compiled code of the searched method - if the method is defined within the current class or its superclass - or customised lookup code, if the searched method is delegated to a parent object. If the parent object does itself delegate the searched method, its dispatch table will again reference customised lookup code. So the "search" will proceed up the dynamic delegation hierarchy and stop at the first object whose dispatch table entry references a "real" method.

Note that no real searching, i.e. no check of stopping criteria or conditional branching is performed throughout. If native code compilation is performed the cost per customised lookup code dispatch can be further reduced. E.g. the *call* contained in each customised lookup code can be replaced by a *jump* and repeated passing of invariant arguments (*receiver, args*) and of useless return addresses can be eliminated. The *call* at the message sending site already does all necessary stack/register manipulation.

Compared to existing systems, our scheme implements dynamic delegation without the expense of full run-time search of method dictionaries. The customised lookup code reduces the lookup in each of the traversed dispatch tables to one access at a statically known index. The time complexity of this "search" is linear in the number of parents, whereas the search performed for dynamically delegated messages in SELF ([CUL89], [Hölz94a/c]), is linear in the total number of selectors within the parents' hierarchies - which may be orders of magnitude higher. Even if the SELF compiler used hashing instead of linear search, it would at least need to dynamically compute the hash function and check for collisions, where we can directly access a statically known index.

4.4. Messages to "self"

The basic schema presented in the last section needs to be extended for messages to *self*. In the presence of delegation *self* may be any object

---

[1] If *index(selector,parent_type)* contains more then one element, any of them may be used, since they always reference the same code. With "weak customization" any *negative* index can be used (cf. 4.4.3.).

- that *is* an instance of the current class or of one of its subclasses or
- that *delegates* to such an instance.

In the second case the receiver of a *self* message is different from the sender, unlike in inheritance-based systems. When this happens the receiver's type will be unknown at compile-time. Since the dispatch table index of a selector is only known if the receiver's type is known, the selector index must be determined at run-time.

The next two subsections describe how this is done and what precaution must be taken to guarantee the correctness of our method. The last subsection describes how any additional overhead can be avoided when the sender and the receiver of a *self* message are equal (which is always the case in purely inheritance-based systems).

4.4.1. Determination of selector indices

Thanks to the uniform structure of extended dispatch tables, a selector's index in the dispatch table of *self*'s class is the index in the dispatch table of some declared ancestor plus an offset (cf. 4.3.1). For each class, *C*, the offsets of selector positions relative to the dispatch tables of declared ancestor classes, *offset(C,Anc)*, can be statically determined. These values are stored in so called *offset arrays*, such that the offset determination at run-time is reduced to the constant time needed for an array access.

The ***offset array*** of a class, *C*, stores the values *offset*(C,C_1), ..., *offset*(C,C_n) for all classes, C_i, whose instances may send *self* messages that might need to access *C*'s dispatch table. Constant-time access to the stored values is based on the fact that each class, *Y*, has a *statically* known, *unique* index in the offset arrays of all classes, X_j, for which *offset*(X_j,*Y*) is defined. This unique index is called the *offsetArrayIndex* of the class *Y* and is denoted by *Y.offsetArrayIndex*.

The compilation of *self* messages is based on the use of offset arrays as an additional run-time structure. The compiled representation of each class has been extended by an *offset array reference*, which allows different classes to share the same offset array, and which is "nil" for classes that do not delegate. The offset array reference is always located at index 1 of the dispatch table. So we may use the dispatch table reference of instances to find the offset array, avoiding the extension of the basic object layout with a special offset array reference (cf. fig. 11). With this preparation, the message

self.selector(args)

in the body of a method defined in class *SC* is compiled as shown in the code template 3.

```
index  := index(selector,SC) +                    // add index in sender class table
          self.dTable[1].[SC.offsetArrayIndex]    // ... to offset in receiver table
method := self.dTable[index]                                  // standard dispatch
method(self,args)
```

Code template 3: Offset array based dispatch of *self* messages (cf. fig. 11)

After this initial step, the rest of the dynamic binding process is as described in 4.3., i.e. the called code is either the searched method code, or "customised lookup code".

4.4.2. Determination of access safety

A problem with messages to *self* occurs if the class of the receiver object is a *potential* descendant of the class of the message sender. In the example shown in fig. 10 the message *obj_a.b* leads to a submessage *self.x* while *self* is still bound to *obj_a*. The difficulty is that there is no index for *x* in the dispatch table of class *A*. If the above compilation scheme were used indiscriminately, *self* messages whose selectors are not in the extended protocol of the receiver's class, called (***access***) ***unsafe messages***, would access the wrong index. The determined index would either belong to some other method, or it would even point outside the dispatch table.

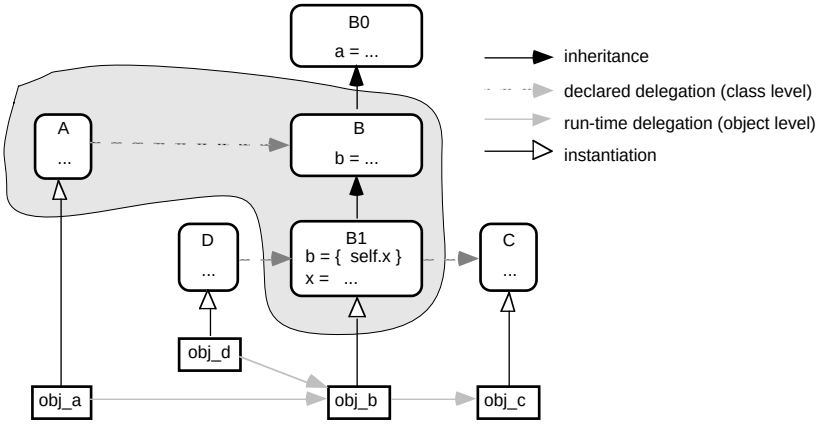


Fig. 10: *obj_a* may receive *self*-messages from *obj_b* and *obj_c* that are unknown to *A*

Please note, that access safety is *not* identical to type safety. Access-safety asks whether the selector of a message to *self* is statically known to the receiver object and whether its index in the receiver's dispatch table can be determined from its index in the sender's dispatch table; type safety asks whether the message will be answered without producing a "message not understood" error. In our model of typed delegation, these are two distinct issues, as can be easily checked in fig. 10: the selector *x* is statically unknown to class *A*; nevertheless, the message *self.x* sent from *obj_b* while *self = obj_a* will produce no "message not understood" error - after checking that it is access unsafe for *obj_a*, the message will be delegated to *obj_b*, whose class contains a method for *x*. Since only concrete classes can be instantiated, *every self* message is guaranteed to be defined either for the receiver object *or* for one of its ancestors. Thus, *self* messages will never produce "message not understood" errors. However, this does not assure access-safety.

Access-safety cannot be decided statically, since it depends on the run-time type of *self*. E.g. in fig. 10 *self.x* is access-unsafe with respect to *self = obj_a* but it is access-safe with respect to *self = obj_d*. So, again, we need an efficient run-time test. Please note that access-unsafety is an equivalent characterisation of the *self* messages to which the special overriding rule of section 3.2.2 applies (cf. fig 3). The detection and treatment of access-unsafe messages described in the following thus implements our special rule for method overriding.

The test of access safety is based on the observation that a message will access a wrong index iff its selector index in the sending class is bigger than the length of the accessed dispatch table's section that corresponds to the dispatch table of the sending class.

By differentiating what is the accessed dispatch table's "section that corresponds to the sending class", depending on the relative position of the sending and the accessed class in the

delegation hierarchy, we get the following definition of the function ***maxSafeIndex(AC, SC)***. It returns the highest selector index of the sender class *SC* that is safe wrt the accessed class *AC* (or "undefined", if *SC* cannot access *AC*). In the definition of *maxSafeIndex*(*AC, SC*) we use the auxiliary functions ***maxDTindex(class)***, which returns the maximum dispatch table index of *class*, and ***nearestSuperclass(class, setOfClasses)***, which returns the class from *setOfClasses* that is reachable from *class* via the shortest inheritance path. A detailed example is contained in chapter 5 (fig. 15 and 16, and table 3 and 4).

maxSafeIndex(*AC, SC*) :=

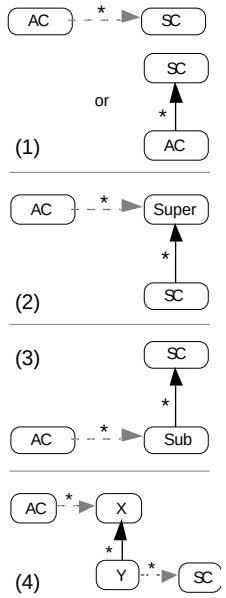
(1) if $SC \in (\{AC\} \cup AC.superclasses \cup AC.declAncestors)$
then *maxDTindex*(*SC*)

(2) else if $\exists\ Super = nearestSuperclass(SC, AC.declAncestors)$
then *maxDTindex*(*Super*)

(3) else if $(SC.subclasses \cap AC.declAncestors) \neq \emptyset$
then *maxDTindex*(*SC*)

(4) else if $(SC.inherHierarchy \cap AC.ancestors) \neq \emptyset$
then 0

(5) else "undefined"



Case (1) expresses that if the sender class is a *declared* ancestor of the accessed class, or identical to the accessed class, then the accessed dispatch table section corresponds exactly to the dispatch table of the sender class; hence the highest safe selector index is the highest index in the sender's dispatch table. E.g. in fig. 10, *maxSafeIndex(A,B)* = 2.

Case (2) expresses that if the sender class is not itself a declared ancestor, but has superclasses that are declared ancestors of the accessed class, then the accessed dispatch table section corresponds exactly to the dispatch table of the "nearest" superclass of the sender class; hence the highest safe selector index is the highest index in the dispatch table of that superclass. E.g. in fig. 10, *maxSafeIndex(A,B1)* = *maxSafeIndex(A,B)* = 2.

Case (3) expresses that if neither (1) nor (2) holds, and if the sender class has subclasses that are declared ancestors of the accessed class, then the accessed dispatch table section corresponds to the dispatch table of one of these subclasses; hence every method defined for the sender class will be safe with respect to the accessed class. Thus it suffices to set the highest safe selector index to the highest index in the sender's dispatch table. E.g. *maxSafeIndex(A,B0*) = 1 in fig. 10. Note that this part of the definition is needed, in order to enable execution of *compiled* superclass code on behalf of subclass instances (e.g. execution of *self.a* on behalf of *obj_b*).

Case (4) expresses, that all *self* messages from sender classes that are ancestors of the accessed class but are *not* in the inheritance hierarchy of a *declared* ancestor, are unsafe. E.g. in fig. 10, all *self* messages from *obj_c* to *obj_a* are access unsafe.

Case (5) expresses that all classes that are not ancestor classes can never send *self* messages to the "accessed class".

Using the function *maxSafeIndex*, the test whether the message *self.selector(args)* in the code of the sender class *SC* is access-unsafe wrt the dispatch table of the receiver class *AC* is:

$$\textbf{\textit{safe(selector,SC,AC)}} := \underline{index(selector,SC)} \leq maxSafeIndex(AC,SC).$$

In order to speed the run-time test, we compute and store all *maxSafeIndex* values at compile-time. For each class, *AC*, that is part of a delegation hierarchy, we define its ***safety array*** to contain the *maxSafeIndex* values wrt each of its ancestor classes. A reference to the safety array is contained at index 0 of *AC*'s dispatch table. Like for offset arrays, each class has a unique *safetyArrayIndex* in the safety arrays of all its descendant classes. Thus the above safety test can be implemented for an object *obj* of class *AC* as

$$\underline{index(selector,SC)} \leq obj.dTable[0].[\underline{SC.safetyArrayIndex}]$$

With this preparation, we can extend the compiled code generated for *self* messages by a constant-time test for unsafe messages. If the message is unsafe, it is immediately delegated to the next parent object and the check is repeated. Otherwise, the message is executed by accessing the offset array and the dispatch table, as described in the previous section.

```
delegatee := self
while ( index(selector,SC) > delegatee.dTable[0].[SC.safetyArrayIndex] )
   do delegatee := delegatee.parent;
```

Code template 4: Safety check before dispatch of *self* messages

The above code template only describes the *principle* of safety checks. The access to the parent (*delegatee := delegatee.parent*) requires knowledge of the offset at which the parent attribute is located within the delegatee object (*delegatee := delegatee[parentOffset]*). Since the type of the delegatee is statically unknown, the *parentOffset* must also be determined at run-time.

The standard solution "pattern" is to store the information at compile-time in a place where it can easily be found at run-time. We think that the right place to store the *parentOffset* is the offset array of a class. If offset arrays are extended to contain two-component entries,

```
offsetArrayEntry = { dTableOffset : integer; parentOffset : integer }
```

the *parentOffset* can be accessed using the dispatch table reference of the delegatee and the statically known *offsetArrayIndex* of the sending class (cf. *italic* part of code template 5). On superscalar processors this access is *free*: since there are no mutual data dependencies, it can be done in parallel to the computation required anyway for the safety check.

```
delegatee := self
while ( index(selector,SC) > delegatee.dTable[0].[SC.safetyArrayIndex] )
   do { parentOffset := delegatee.dTable[1].[SC.offsetArrayIndex].parentOffset
        delegatee := delegatee[parentOffset]
      }
```

Code template 5: Full safety check, including dynamic determination of the parent attribute offset

Note that our solution is based on the assumption that the dispatch table reference is located at the same statically known offset in all objects. This is a sensible assumption in pure object-

oriented languages. In hybrid languages that are constrained to be compatible with their non-object-oriented predecessors, like *C++*, the above assumption might be difficult to satisfy.

Fig. 11 gives an overview of all run-time structures involved in the dynamic binding of messages to *self*. The meaning of the negative indices of the shown dispatch table will be explained in the next section. A complete code template for the dynamic binding of *self* messages is contained in the appendix.

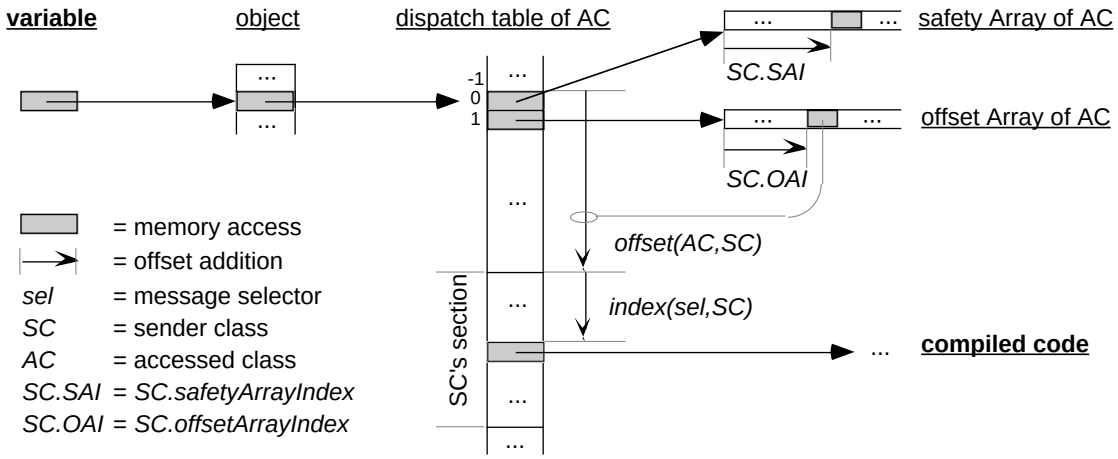


Fig. 11: Overview of run-time structures involved in the dynamic binding of messages to *self*

4.4.3. Weak Customisation

The main goal that shaped the design decisions described in the paper, was to avoid any kind of run-time or storage penalty for programs that do *not* use delegation. The dispatch of *self* messages described so far does not meet this goal, since the safety check and the offset calculation are performed for *all self* messages, including messages sent by *self*. The equality of the sender and receiver of *self* messages characterises the execution of non-delegated messages. In order to incur no loss of performance of non-delegated messages, customisation ([CUL89]) is used. *Two* versions of each method are generated,

- a *receiver version*, which is called for messages that have been *sent* to an object, and
- a *delegatee version*, which is called for messages that have been *delegated* to an object.

The receiver version can be compiled as in the classic approach (cf. 4.1.):

- Since it is statically known that *self* is the object on whose behalf the method is executed, the type of *self* is known, hence messages to *self* need no safety check and offset adjustment.
- Since it is statically known that the *delegatee* argument is equal to the *receiver* argument (cf. 4.2.), the *delegatee* parameter can be omitted from the parameter list of the receiver version *and* from every call to a receiver version.

The dispatch table of each class, *C*, now consists of two parts: one contains references to the specialised receiver versions the other contains references the delegatee versions. The receiver version of a method is referenced at position *index(sel,C)* in the table, whereas the delegatee version reference has the symmetric position, *-index(sel,C)*. Thus the final layout of dispatch tables is such that position 0 references the safety array, position 1 references the offset array,

the indices > 1 reference the receiver versions, and the indices < -1 reference the delegatee versions (fig. 11). This layout ensures compatibility with "inheritance-only" implementations, since the structure of the receiver part and the code referenced therein corresponds exactly to the outcome of the "standard" compilation scheme (4.1).

All messages in the receiver version, and explicit messages in the delegatee version of a method, are compiled to call only receiver versions. Delegatee versions are only called within code for *self* messages in delegatee versions and within customised lookup code (cf. 4.4.2. and 4.3.2). The only necessary adaptation of the previous compilation scheme is the use of negated dispatch table indices and of the additional delegatee argument when delegatee versions are called. The corresponding, final scheme of the generated code is contained in the appendix.

We call our approach *weak customisation* since only two versions are generated for each method. In contrast, strong customisation, e.g. in *SELF*, dynamically generates one specialised version of a method *for every receiver type*, provided that at least one object of that type has received a message with the corresponding selector. ([CUL89], p. 58: "Our SELF compiler [...] compiles a different machine code method *for each type of receiver* that runs a given source method.")

4.5. Multiple Inheritance and Delegation

The previous section concludes the introduction of new structures at the level of the run-time system. This section shows that the presented structures and techniques allow to implement additional functionality, especially that multiple inheritance and multiple delegation can be added without requiring any further extensions.

4.5.1. Ambiguity Resolution

Since methods in a class override those in its superclasses and methods in superclasses override methods in its parent classes (cf. 3.2.1.) conflicts can occur only between classes with the same status. A selector is ambiguous within the scope of a class, *C*, either if

- a corresponding method definition is not contained in *C* but in different superclasses or if
- a corresponding method definition is neither contained in *C* nor in its superclasses but in different *declared* parent classes of *C*.

Ambiguities are reported at compile-time and must be resolved by explicit local redefinitions *or* by renaming of conflicting selectors. If conflicting selectors from different source classes have the same semantics, a local redefinition, which overrides all conflicting definitions, is possible. On the other hand, renaming is the appropriate choice if conflicting selectors have different semantics. Compared to automatic conflict resolution techniques like linearization ([DuHa91) and points of view ([Duge91]), renaming does not hide potential ambiguities. It is also superior to explicit conflict resolution at the message sending site ([ES90], [Stro91]) in various respects: the user of the multiply inheriting class does not need to know its superclasses[1], he is not forced to statically bind a call to the implementation in a specific class, and

[1] The problems of the dependency on the inheritance structure of a class are described in detail in [Sny86].

he is not forced to introduce additional classes just in order to hand-code a renaming scheme, when needed (as proposed in [Stro94]).

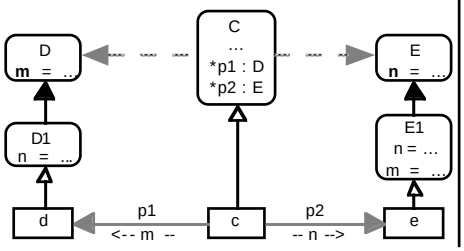
Renaming also offers the most general and handy solution to a problem that is specific to our model: when methods with the same name in a superclass and a parent class do not have the same semantics, there must be a way to disable the standard overriding of the parent class method by the superclass method. This can be easily achieved by renaming one of them.

When selector *m* from superclass[1] *A* is renamed as *ma*, and selector *m* from superclass *B* is renamed as *mb*, method definitions for the new names will consistently be used for *self.m* messages sent in *A* and *B*, i.e. *self.m* sent in *A* will always be bound to a definition of *ma*, whereas *self.m* sent in *B* will always be bound to a definition of *mb*. Thus each *self* message will automatically maintain its semantic context. There is no need to introduce an explicit "sender path tiebreaker rule" ([CUCH91]) for this purpose. No run-time overhead is involved in our scheme since each of the new names can be statically bound to the dispatch table position assigned to the old name in the corresponding superclass / parent class.

The next two sections describe how the dispatch table(s) of a class must be laid out in order to be able to reuse compiled code inherited from different superclasses and parent classes.

4.5.2. Multiple Delegation

In the case of multiple delegation the dispatch table of a child class contains a delegated section (cf. 4.3.1.) for *each* of its *declared* parent classes. An unambiguous message is always delegated via the *parent attribute* whose *declared* type contains a definition. E.g. in the following example, the message *c.m* is always delegated via the parent attribute *p1*, since *p1* is statically known (by its type declaration) to reference an object that possesses a method for *m*. Thus, for *C* instances, there is no conflict between the definition of *m* in *E1* and *D*. Similarly, *c.n* is always delegated via the parent attribute *p2*. In fig. 12. the definition of *n* from *E1* will be executed, since it is the most specific definition visible through *p2*.




```
C = {  *p1 : D,        // C defines multiply delegating instances.
       *p2 : E  }      // The asterisks denote parent attributes.
c : C; ...             // Declare object c, d, and e.
d = D1 new;            // Create instance of a subclass of D
e = E1 new;            // Create instance of a subclass of E
c = C new(d,e);        // Create C instance with parents d and e
c.n                    // Call c.dTable[index(n,E) + offset(C,E)]
c.m                    // Call c.dTable[index(m,D) + offset(C,D)]
```

Fig. 12: Multiple delegation hierarchy and ... corresponding example of source code and compiled code

The described behaviour is implemented by always dispatching unambiguous messages via the unique statically known index of the message selector in the receiver's dispatch table. E.g. the explicit receiver message *c.n* will always access the index for selector *n* within *E´*s section of C´s dispatch table, situated at position *index(n,E) + offset(C,E)*, whereas *c.m* will access the index *index(m,D) + offset(C,D)*. Note that the same index will be accessed by *self.n* messages sent from object *e* resp. object *d* (cf. 4.4).

---

1 The same is true for *parent* classes and any combination of superclasses and parent classes.

4.5.3. Multiple Inheritance

In this section we will sketch the implementation scheme for multiple inheritance initially developed by Krogdahl for *Simula* ([Krog84] [Krog85]) and adopted later, in a slightly modified form in *C++* ([Stro87], [ES90]) and other typed languages. The following summary just tries to convey the basic principles needed to understand the orthogonality of the implementation of multiple inheritance and delegation, illustrated in the next section.

The implementation of multiple inheritance is based on the object layout illustrated in fig. 13. The storage layout of instances of the multiply inheriting class, *C*, is determined by concatenating the instance layout defined by the superclasses, *B* and *D*, and the layout defined by *C*. The resulting object layout contains one dispatch table reference for each superclass, e.g. to table B_C for superclass *B* and to table D_C for superclass *D*. There is *no* separate dispatch table for the inheriting class *C*, but a section for the new selectors introduced in *C* is appended to one of the superclass tables. E.g. in fig. 13, the new messages from *C* have been added to the dispatch table B_C. Note that the referenced dispatch tables have the *same structure* but are *not identical* to the tables referenced by superclass instances. E.g. the D_C table of class *C* differs from the table of class *D* in its entries for the redefined method *d*.

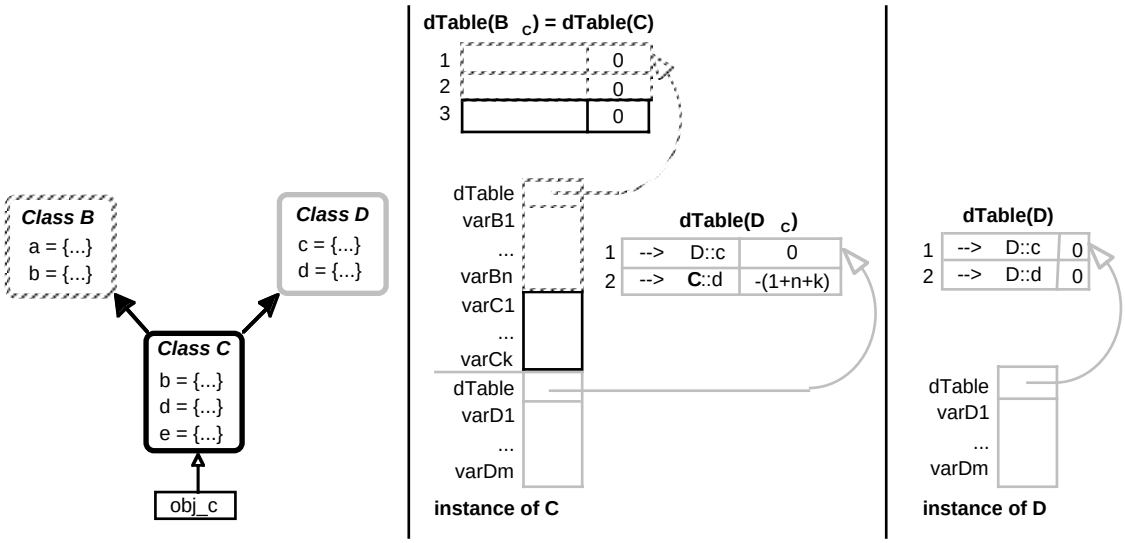


Fig. 13: Object and dispatch table layout for a multiply inheriting class *C*.
The layout of a *D* instance and of *D*'s dispatch table is included for comparison.

With single inheritance the value of a variable defined in a class can be accessed in all subclass instances by adding the same offset to the object reference. This is no longer true if the above instance layout is used. E.g. the value of *varD1* is located at offset 1 in *D* instances, whereas in *C* instances it has offset 1+n+k. Thus it is necessary to *adjust the object reference* whenever a *C* instance is regarded as an instance of *D*, i.e. when it is assigned to a variable or passed as a parameter of declared type *D*. This includes passing of the implicit *self* parameter to a method inherited from *D*. The adjustment is done by adding the relative position of the *D* part within *C* instances, usually called ***delta(C,D)***, to the object reference. The addition of this constant is included by the compiler in the code generated for assignments, parameter passings, and before the call of methods inherited from *D*.

When inherited methods are redefined in *C*, the adjustment has to be undone before calling the redefined code, which expects its *self* argument to be a *C* instance. The need to undo the adjustment cannot be predicted by the compiler. Therefore dispatch tables had been extended

by a second entry for every selector. If the adjustment does not need to be undone for the current *self* object the additional "delta" entry is zero. Otherwise it is *-delta(C,D)*. With this extension, the dispatch method introduced in 4.1 becomes:

```
method := receiver.dTable[indexOfMethod(selector,T)]          // dynamic binding
delta  := receiver.dTable[indexOfDelta(selector,T)]           // undo adjustment
object := receiver + delta                                 // ... of self reference
method(object,args)                                          // procedure call
```

Code template 5: "Standard" dispatch method (including multiple inheritance)

4.5.4. Multiple Inheritance and Delegation Combined

The above implementation scheme ensures that each access to an instance of a multiply inheriting subclass always "sees" the right subobject resp. dispatch table, depending on the declared type of the variable that references the instance. Therefore our limitation to one superclass / dispatch table in the previous sections represents no restriction. Everything that was said, equally applies to each superclass resp. *dispatch table* of a multiply inheriting class.

This orthogonality is illustrated in fig. 14a. Each instance of *C* will reference two dispatch tables, one corresponding to class *D* (which includes a delegated section for the parent class *E*), and one corresponding to class *F* (which includes a delegated section for the parent class *G*).

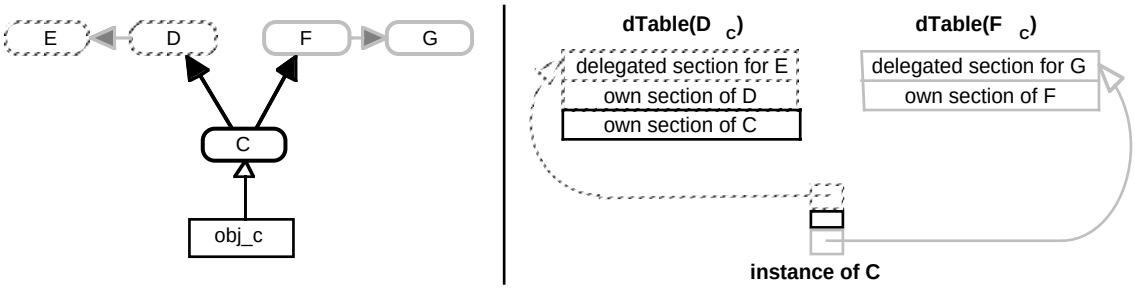


Fig. 14a: Mult. inher. from delegating classes ... and corresponding dispatch table layout

The *dispatch table of a class that* delegates *to a multiply inheriting class* contains one delegated section for each dispatch table of the parent class. E.g. in fig. 14b, class *C* has only one dispatch table, which contains two delegated sections, corresponding to the two dispatch tables of class *D*. Correspondingly, the offset array of *C* has one entry for *each dispatch table* of *D*.

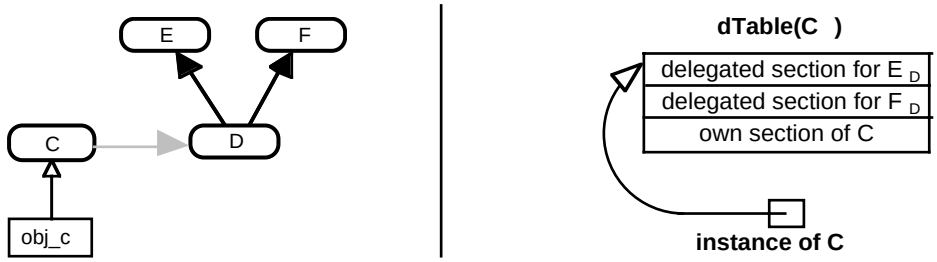


Fig. 14b: Delegation to multiply inheriting class ... and corresponding dispatch table layout

Finally, if the class *C* is defined using multiple inheritance and multiple delegation simultaneously, the delegated sections for the different parent classes are all included in the dispatch table that holds the new methods introduced in *C*. This is illustrated in fig. 14c.

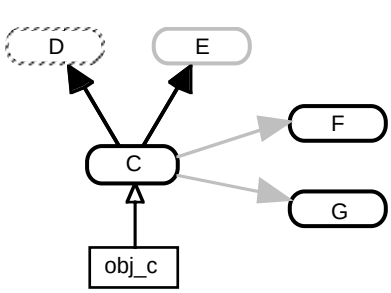
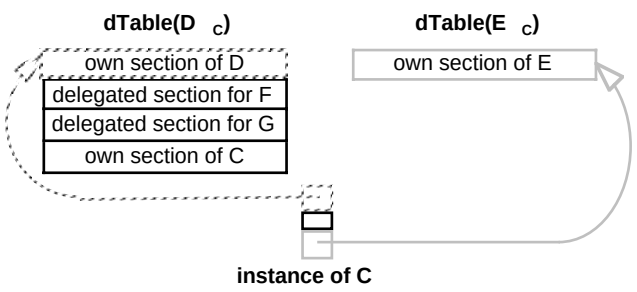


Fig. 14c: Mult. inher. and mult. deleg. together



... and corresponding dispatch table layout

Due to these minimal adaptations, all previously introduced techniques are applicable to any combination of (multiple) delegation and inheritance.

This section concludes the presentation of our proposal for the implementation of dynamic binding in "typed delegation"-based systems. The next chapter discusses how to achieve a storage efficient implementation of the additional run-time data structures on which our dynamic binding scheme is based. Readers interested mainly in run-time aspects might want to continue reading in chapter 6 and 7, where we evaluate the performance of our approach and compare it to related work.

5. Layout of Auxiliary Run-Time Structures

The compilation of messages to *self*, described in section 4.4., is based on the use of two auxiliary run-time structures, which have been added to the compiled representation of each class in a delegation hierarchy: safety arrays and offset arrays. Safety arrays are used to determine, whether a *self* message is access-safe with respect to a class (4.4.2.). If the message is safe, the index to be accessed in the dispatch table of the respective class is determined using the offset array (4.4.1).

In section 4.4 we simply assumed, that an implementation of auxiliary arrays exists that assures constant-time access to the stored values. Naturally, we would also like to have a space efficient implementation. So the requirements to be met are

1. access efficiency: for constant-time access, each class, *Y*, must have a *statically* known, *unique* index in the auxiliary arrays of all classes, X_j, for which $f(X_j,Y)$[1] is defined, and

2. space efficiency: each auxiliary array should contain neither unused nor redundant positions.

In this chapter we shall discuss to which extent the second requirement can be met without compromising the first one, on which our implementation scheme is built. We start with a brief discussion of the drawbacks of a straightforward array layout and introduce an example that will be used throughout this chapter for illustration of the discussed techniques. Then we develop a solution that achieves optimal results for a large class of programs.

---

[1] The safety array of a class, *X*, stores values of the function *maxSafeIndex*(*X*,*Y*) and its offset array stores values of the function *offset*(*X*,*Y*). When we do not differentiate between safety and offset arrays, we simply write *f(x,y)* instead of *maxSafeIndex(x,y)* or *offset(x,y)*.

5.1. What's the Problem?

In this section we assume a program that contains n classes, k delegation hierarchies of n_1, ..., n_k classes, and nd classes that are not involved in any delegation hierarchy.

The simplest approach to implement auxiliary arrays is to use two matrices of size n^2, where each $f(X,Y)$ value is indexed by unique row and column positions assigned to the argument classes, X and Y. This representation clearly ensures constant time access, but it wastes a lot of storage on undefined and redundant entries. If this naive approach were used in a large system like Smalltalk, two matrices of 500*500 entries, consuming about 1 megabyte of storage[1] would be needed for the representation of the (aprox. 500) predefined classes.

A first step to reduce the amount of wasted storage, is to create *one matrix for each delegation hierarchy* and to include in each matrix only the classes involved in the corresponding hierarchy. The use of many small matrices would reduce the total storage requirement from n^2 to $n_1^2 + ... + n_k^2$, which could make a significant difference, since

$$n_1^2 + ... + n_k^2 << (n_1 + ... + n_k)^2 = (n\text{-}nd)^2 \leq n^2.$$

As illustrated in fig. 15-17, much storage is still wasted, since most matrix entries are either undefined or redundant. Fig. 15 shows a sample program that will be used for reference throughout this chapter, and fig. 16 and 17 show the safety matrices resp. the offset matrices created for the two delegation hierarchies of the program. Each matrix row corresponds to an auxiliary array and each column to an array index. Empty fields represent undefined values. The *maxSafeIndex* and *offset* values have been calculated assuming that each of the classes from fig. 14 define as many new methods as shown in table 3. Table 4 contains the corresponding *maxDTindex* values (cf. 4.2.2. - for simplicity we ignored the dispatch table positions reserved for the auxiliary array references and assumed that dispatch table indices start at 1). The thick vertical lines separate the columns of classes from different inheritance hierarchies, for easier comparison with subsequent figures.

In the remainder of this chapter we shall show that much better results can be achieved when exploiting the semantics of auxiliary array entries and the relationships between the classes of a program. We shall develop an algorithm that determines the layout of auxiliary arrays by incrementally formalising the defining characteristics of auxiliary arrays, and the requirements for access and storage efficiency stated above.

---

[1] The total size of the matrix depends on the respective hardware architecture; we assumed that a matrix entry is a short integer represented in two bytes.

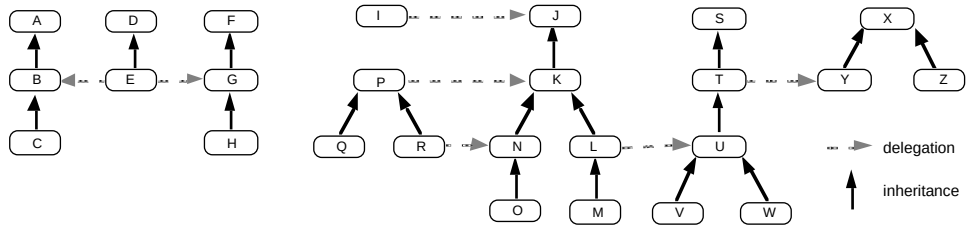


Fig. 15: Program graph containing two delegation hierarchies.
This example will be used for reference throughout this chapter.

new methods								
A	1	**J**	11	**S**	4
B	1	**K**	1	**T**	1
C	1	**L**	1	**U**	1
D	4	**M**	1	**V**	1
E	1	**N**	1	**W**	2
F	5	**O**	1	**X**	1
G	1	**P**	4	**Y**	1
H	1	**Q**	1	**Z**	2
I	8	**R**	1		

Table 3: Number of *new* methods in each class from fig. 15

maxDTindex					
A	1	**J**	11	**S**	4
B	2	**K**	12	**T**	7 = 4+1+2
C	3	**L**	21 = 12+1+8	**U**	8
D	4	**M**	22	**V**	9
E	13 = 4+1+2+6	**N**	13	**W**	10
F	5	**O**	14	**X**	1
G	6	**P**	16 = 4+12	**Y**	2
H	7	**Q**	17	**Z**	3
I	19 = 8+11	**R**	30 = 16+1+13		

Table 4: Highest dispatch table index of each class, *maxDTindex(C)*, (cf. 4.4.2) corresponding to fig. 15 and table 3.

	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
I	19	11	11	11	11	11	11				0	0	0	0	0	0	0	
J		11									0	0	0	0	0	0	0	
K		11	12								0	0	0	0	0	0	0	
L		11	12	21							4	7	8	8	8	1	2	
M		11	12	21	22						4	7	8	8	8	1	2	
N		11	12			13												
O		11	12			13	14											
P		11	12	12	12	12	12	16			0	0	0	0	0	0	0	
Q		11	12	12	12	12	12	16	17		0	0	0	0	0	0	0	
R		11	12	12	12	13	13	16		30	0	0	0	0	0	0	0	
S											4							
T											4	7				1	2	
U											4	7	8			1	2	
V											4	7	8	9		1	2	
W											4	7	8		10	1	2	
X																1		
Y																1	2	
Z																1		2

	A	B	C	D	E	F	G	H
A	1							
B	1	2						
C	1	2	3					
D				4				
E	1	2	2	4	13	5	6	6
F						5		
G						5	6	
H						5	6	7

Fig. 16: Naive implementation (5.1) of *safety* arrays for program graph from fig. 15.
Each matrix row corresponds to a safety array and each column to a safety array index.
The value *maxSafeIndex(AC,SC)* is contained in the array (row) of class *AC* at index (column) *SC*.
Empty fields represent undefined values.

	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
I	0	8	8	8	8	8	8											
J		0																
K		0	0															
L		0	0	0							13	13	13	13	13	18	18	
M		0	0	0	0						13	13	13	13	13	18	18	
N		0	0			0												
O		0	0			0	0											
P		4	4	4	4	4	4	0										
Q		4	4	4	4	4	4	0	0									
R		4	4	4	4	17	17	0	0	0								
S											0							
T											0	0				5	5	
U											0	0	0			5	5	
V											0	0	0	0		5	5	
W											0	0	0		0	5	5	
X																0		
Y																0	0	
Z																0		0

	A	B	C	D	E	F	G	H
A	0							
B	0	0						
C	0	0	0					
D				0				
E	5	5	5	0	0	7	7	7
F						0		
G						0	0	
H						0	0	0

Fig. 17: Naive implementation (5.1) of *offset* arrays for program graph from fig. 15.
Each matrix row corresponds to an offset array and each column to an offset array index.
The value *offset(AC,SC)* is contained in the array (row) of class *AC* at index (column) *SC*.
Empty fields represent undefined values.

5.2. Basic Definitions

5.2.1. Accessor Classes and Accessed Classes

In order to describe the relationship among the classes of a program wrt auxiliary arrays, we define for each auxiliary array of a class, *C*, two sets, its accessors and its accessed classes.

The ***accessors** of C's safety array*, ***C.SAaccessors***, are all classes whose methods, when executed, may send *self* messages to instances of *C*. According to the definition of the *maxSafeIndex* function (4.2.2.), these are all classes, *Y*, for which *maxSafeIndex*(*C*,***Y***) is not undefined, i.e. *C* itself, its superclasses, ancestors, and the superclasses of the ancestor classes:

C.SAaccessors := $\{C\} \cup C.superclasses \cup C.ancestors \cup C.ancestorSuperclasses$.

C.ancestorSuperclasses := $\bigcup_{A \in C.ancestors} A.superclasses$.

The ***accessors** of C's offset array*, ***C.OAaccessors,*** are all classes, *Y*, for which *offset*(*C*,***Y***) is not undefined, i.e. *C* itself, its superclasses, *declared* ancestor classes, and the *sub*classes of the declared ancestors:

C.OAaccessors := $\{C\} \cup C.superclasses \cup C.declAncestors \cup C.ancestorSubclasses.$

C.ancestorSubclasses := $\bigcup_{A \in C.declAncestors} A.subclasses$.

With respect to *safety* arrays, the ***accessed classes*** of *C* are all the classes whose safety array might be accessed by *self* messages sent from methods defined in *C*. According to the definition of the *maxSafeIndex* function, these are all classes, *X*, for which *maxSafeIndex*(*X*,*C*) is not undefined, i.e. *C* itself, its subclasses, descendant classes, and the descendants of its subclasses:

C.SAaccessedClasses := $\{C\} \cup C.subclasses \cup C.descendants \cup C.subclassDescendants.$

C.subclassDescendants := $\bigcup_{S \in C.subclasses} S.descendants$.

The ***accessed classes*** of *C* wrt *offset* arrays, are all classes, *X*, for which *offset*(*X*,*C*) is not undefined, i.e. the class itself, its subclasses, *declared* descendant classes, and the subclasses of its declared descendants:

C.OAaccessedClasses := $\{C\} \cup C.subclasses \cup C.declDescendants \cup C.descSubclasses.$

C.descSubclasses := $\bigcup_{D \in C.declDescendants} D.subclasses$.

When we do not differentiate between safety and offset arrays, we simply write ***C.accessors*** and ***C.accessedClasses***:

C.accessors := $C.SAaccessors \cup C.OAaccessors.$

C.accessedClasses := $C.SAaccessedClasses \cup C.OAaccessedClasses.$

5.2.2. Access Efficiency

The point in defining the above sets is that each class *C* must have a safety/offset array entry for each of its *accessor classes*, and, for constant-time access to the stored values, each of the *accessed classes* must have its entry for *C* at the same, unique index. Hence each class is *statically* assigned one *unique* index in the safety arrays (and one *unique* index in the offset arrays) of *all* its accessed classes. For safety (offset) arrays this index is called the ***safetyArrayIndex*** (***offsetArrayIndex***) of the class. For each class *C*,

(1a) ***C.safetyArrayIndex*** (***C.SAI***) is the position at which the value *maxSafeIndex(X,C)* is stored in the safety arrays of *C*'s accessed classes:

$$\forall\, C\colon \big(\forall\, X \in C.SAaccessedClasses\colon \quad X.safetyArray[C.SAI] = maxSafeIndex(X,C)\big).$$

(1b) ***C.offsetArrayIndex*** (***C.OAI***) is the position at which the value *offset*(*X,C*) is stored in the offset arrays of *C*'s accessed classes:

$$\forall\, C\colon \big(\forall\, X \in C.OAaccessedClasses\colon \quad X.offsetArray[C.OAI] = offset(X,C)\big).$$

When we do not distinguish between safety and offset arrays we write ***auxArrayIndex*** (***AAI***).

5.2.3. Significant Indices

In our implementation scheme (4.2.2.) each message accesses the auxiliary arrays of a class, *C*, at the *auxArrayIndex* assigned to the class that *textually contains* the message. Hence the indices that will be accessed at run-time, are the *auxArrayIndex* values of *C*'s accessors. These values define the *set of significant indices* of *C*'s auxiliary arrays:

$$\textbf{\textit{C.SAsignificantIndices}} := \{\, X.auxArrayIndex \mid X \in C.SAaccessors \,\}.$$

$$\textbf{\textit{C.OAsignificantIndices}} := \{\, X.auxArrayIndex \mid X \in C.OAaccessors \,\}.$$

The last two definitions show that, in principle, the *auxArrayIndex* values determine the array structure. However, for simplifying the following discussion, additional numbers, *MaxIndex* (*MaxI*), *MinIndex* (*MinI*), *MaxAccessorIndex* (*MaxAI*), and *MinAccessorIndex* (*MinAI*), are introduced for both kinds of arrays. The values *C.MinI* and *C.MaxI* delimit the range of significant auxiliary array indices of the class *C*. The values *C.MinAI* and *C.MaxAI* delimit the range of significant auxiliary array indices of the accessors that are *different* from *C*. The safety / offset array specific versions of the following definitions can be derived by substituting *AAI* by *SAI* resp. *OAI*, and *significantIndices* by *SAsignificantIndices* resp. *OAsignificantIndices*.

$$\textbf{\textit{C.MinI}} := \min(C.significantIndices). \quad \textbf{\textit{C.MinAI}} := \min(C.significantIndices \setminus \{C.AAI\}).$$

$$\textbf{\textit{C.MaxI}} := \max(C.significantIndices). \quad \textbf{\textit{C.MaxAI}} := \max(C.significantIndices \setminus \{C.AAI\}).$$

5.3. Storage Efficiency

Now the requirement for storage efficiency can be stated precisely. A representation of auxiliary arrays that achieves optimal storage efficiency must at least guarantee that for each class, *C*

(2) all indices of the allocated auxiliary arrays are from the interval [*C.MinI* ... *C.MaxI*]

(3) every index in the interval [*C.MinI* ... *C.MaxI*] is significant, i.e.

$\forall\, i \in [C.MinI \,...\, C.MaxI] : i \in C.significantIndices.$

(4) no index is redundant, i.e.

$$\forall\, C,C': \Big(\forall\, X \in C.accessedClasses \cap C'.accessedClasses: \big(f(X,C) = f(X,C') \big) \Rightarrow C.auxArrayIndex = C'.auxArrayIndex \Big).$$

Condition (2) and (3) say that auxiliary arrays should contain no positions that will never be accessed at run-time. Condition (4) says that two classes should have the same *auxArrayIndex*, if they have the same function values for all common accessed classes. In the following we shall discuss the implications of these conditions on the layout of auxiliary arrays and show that even stronger conditions are required sometimes.

5.3.1. Elimination of Redundant Indices

Condition (4) defines an equivalence relation on the classes of a program. All classes that may use the same *auxArrayIndex* are in the same equivalence class.

In our example the equivalence classes for *offset* arrays are {*A, B, C*}, {*D, E*}, {*F, G, H*}, {*I*}, {*J, K, L, M*}, {*N, O*}, {*P, Q, R*}, {*S, T, U, V, W*}, {*X, Y, Z*}. In each equivalence class there is always one "top" element that is a superclass of all other elements. We use these "top" elements as representatives and write the equivalence classes of our example as $[A]_R$, $[D]_R$, $[F]_R$, $[I]_R$, $[J]_R$, $[N]_R$, $[P]_R$, $[S]_R$, $[X]_R$. The index $_R$ indicates that these are equivalence classes induced by the condition for elimination of redundancy, i.e. condition (4).

The equivalence classes for *safety* arrays are {*A*}, {*B*}, ..., {*P*}, {*Q, R*}, {*S*}, {*T*}, {*U*}, {*V, W*}, {*X*}, {*Y, Z*}. These classes have *no* "top" element, so we choose representatives randomly.

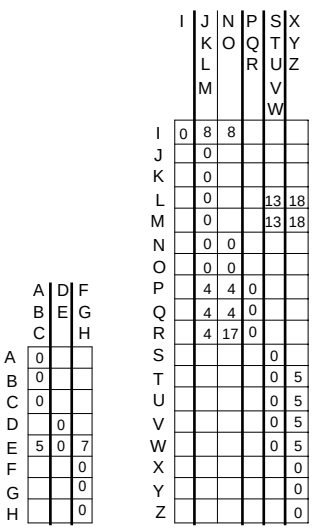


Fig. 18: Redundancy-free *offset* arrays

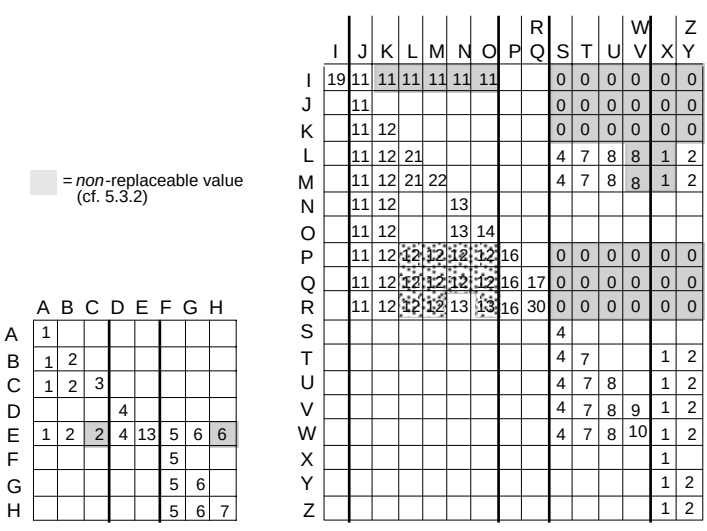


Fig. 19: Redundancy-free *safety* arrays

Fig. 18 and 19 illustrate the different effect of the elimination of redundancy on offset arrays and on safety arrays. Each column corresponds to one of the equivalence classes induced by condition (4). For *offset* arrays, this first compactification step already results in a significant reduction of storage consumption. For *safety* arrays, however, only a minimal reduction is achieved.

5.3.2. Elimination of Subsumed Indices

The unsatisfactory results for *safety* arrays, suggest that a stronger criterion than (4) is needed for reducing their storage demand. Indeed, such a criterion exists. It is based on the semantic of safety arrays, resp. of the *maxSafeIndex* values contained in them.

In order to understand the basic idea of the next compactification step, please recall that the test whether a message is access safe wrt an accessed class, *AC*, is done by checking whether the index of the message selector in the sending class, *SC*, is *smaller* than the *maxSafeIndex* of the accessed class and the sending class (cf. 4.4.2.):

$$\underline{index(selector,SC)} \leq maxSafeIndex(AC,SC).$$

If this test succeeds for a certain selector and a given *maxSafeIndex* value, then it will also succeed for the same selector and any bigger *maxSafeIndex* value. Thus one can replace the "correct" *maxSafeIndex* value by a bigger one, provided that this change does not lead to wrong results for other selectors from the sending class. This is always guaranteed, if all selectors of the sending class are access safe wrt the accessed class, i.e. if the condition

(5) $$\textbf{\textit{replaceable}}(AC,SC) := maxSafeIndex(AC,SC) = maxDTindex(SC)$$

holds. We say that the *maxSafeIndex* value for *SC* is replaceable in the array of *AC*. In fig. 19 the *non*-replaceable values are marked with a grey pattern. All other values are replaceable.

We are interested in a replacement that allows to map different classes to the same *safetyArray-Index*. Such a replacement is defined in the following, based on the notion of *subsumption*. Two classes, *X* and *Y*, ***subsume*** *each other* if their *maxSafeIndex* values are equal or if the smaller of both values is replaceable for *all* classes that are accessed either by *X* or by *Y*:

(6) $$\textbf{\textit{subsumes}}(X,Y) := \forall\, AC \in X.accessedClasses \cup Y.accessedClasses:$$
$$\big(maxSafeIndex(AC,X) = maxSafeIndex(AC,Y) \big)$$
$$\vee \big(maxSafeIndex(AC,X) < maxSafeIndex(AC,Y) \wedge replaceable(AC,X) \big)$$
$$\vee \big(maxSafeIndex(AC,X) > maxSafeIndex(AC,Y) \wedge replaceable(AC,Y) \big).$$

If *X* and *Y* subsume each other, their respective columns in the safety matrix can be melted into one, which contains the bigger one of the respective *maxSafeIndex* values, for each of the accessed classes, *AC*. We call this value the *subsumingMaxSafeIndex* of *X* and *Y* wrt *AC*:

(7) $$\textbf{\textit{subsumingMaxSafeIndex}}(AC,X,Y) :=$$
$$\text{if} \quad subsumes(X,Y) \wedge AC \in X.accessedClasses \cup Y.accessedClasses$$
$$\text{then} \quad max\big(\{maxSafeIndex(AC,X), maxSafeIndex(AC,Y)\} \big)$$
$$\text{else} \quad \text{"undefined"}.$$

Now we can specify the desired replacement condition, ***sameIndex***(*X*,*Y*). Two classes may share the same safety array index, if they subsume each other *and* if each other class subsumed by one of them may also share the same index:

(8) ***sameIndex***(*X*,*Y*) := *subsumes*(*X*,*Y*) ∧ (∀ *Z*: *subsumes*(*X*,*Z*) ⇒ *sameIndex(Z,Y)*)
∧ (∀ *Z'*: *subsumes*(*Y*,*Z'*) ⇒ *sameIndex(Z',X)*)

In general, there are different equivalence relations that can be built such that *sameIndex*(*X*,*Y*) holds for all pairs of elements in every equivalence class. E.g. we can partition the classes from our example into the sets

- {*A, B, C*}, {*D, E*}, {*F, G, H*}, {*I*}, {*J, K,* <u>*L, M*</u>}, {<u>*N, O*</u>}, {*P, Q, R*}, {*S, T, U, V, W*}, {*X, Y, Z*} or
- {*A, B, C*}, {*D, E*}, {*F, G, H*}, {*I*}, {*J, K,* <u>*N, O*</u>}, {<u>*L, M*</u>}, {*P, Q, R*}, {*S, T, U, V, W*}, {*X, Y, Z*}.

The reason is that the *subsumes* (and the *sameIndex*) relation is reflexive and symmetric but not transitive. E.g. *subsumes*(*O*,*J*) ∧ *subsumes*(*J*,*M*), but ¬*subsumes*(*O*,*M*), since the safety array of *R* contains different non-replaceable values for *O* and *M* (fig. 19). Thus, in our example, we have to decide whether *J* should be grouped together with *M* or with *O*.

The criterion for deciding which equivalence relation to choose is based on the fact that

(9) for every program, *one* of the partitions of its classes induced by (8) wrt safety arrays is identical to *the* partition induced by (4) wrt offset arrays.

This can be easily proved by expanding the definition of *offset*/2 in (4) and the definition of *subsumes*/2 and *maxSafeIndex*/2 in (8). In our example, the first one of the possible partitions induced by (8) is identical to the partition induced by (4) (cf. 5.3.1.).

Lemma (9) is of high practical value, since it allows to reduce the elimination of subsumed indices from safety arrays to the elimination of redundant indices from offset arrays. This enables a drastic simplification of the compiler, since only condition (4) must be checked.

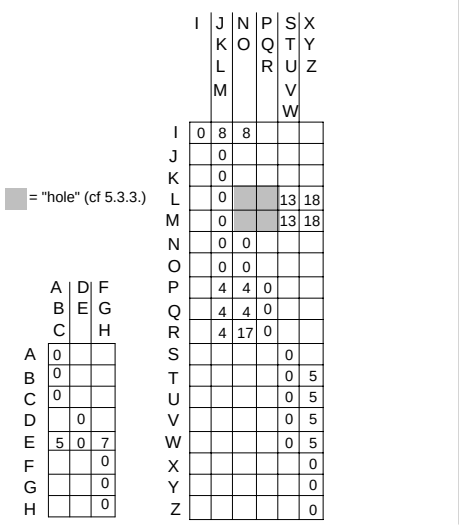


Fig. 18 (repeated): Redundancy-free *offset* arrays

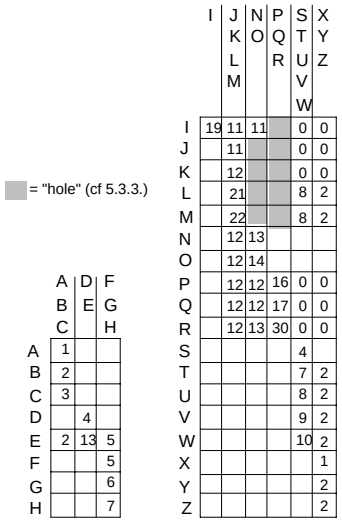


Fig. 20: Subsumption-free *safety* arrays

Figure 20 illustrates the result of eliminating subsumption in our example on the basis of condition (4). Note the significant improvement over the state of the safety matrix in fig. 19.

Figure 18 is repeated to illustrate the structural similarity of the offset matrix after elimination of redundancy and the safety matrix after elimination of subsumption.

If we used condition (4) as the basis of an algorithm, we would need to compute the full *offset* matrix first, before starting to compute equivalence classes. Of course, we are interested in a more direct and efficient way of deciding which classes may use the same *auxArrayIndex*. Fortunately, the following equivalent version of condition (4) can be checked directly on the program graph:

(4') Two classes, *X* and *Y*, may use the same *auxArrayIndex* if
- a) there is an inheritance path $X = C_1, ..., C_n = Y$ of length ≥ 1, *and*
- b) no class C_i in this path has a declared descendant class from which there are at least two different paths to a class $C_j \neq C_i$.

In our algorithm we employ (4') to construct a simplified graph, in which all class nodes that use the same *auxArrayIndex* are collected into one (hyper)node. The (hyper)nodes are linked by edges corresponding to the edges in the original graph. Fig. 21 shows the hypergraph produced for our example. Note that the inheritance hierarchy *J, K, L, M, N, O* has been split between *N* and *K* into two hypernodes, because there are two different paths ($R \rightarrow P \rightarrow K$ and $R \rightarrow N \rightarrow K$) from *N*'s descendant *R* to *N*'s superclass *K*.

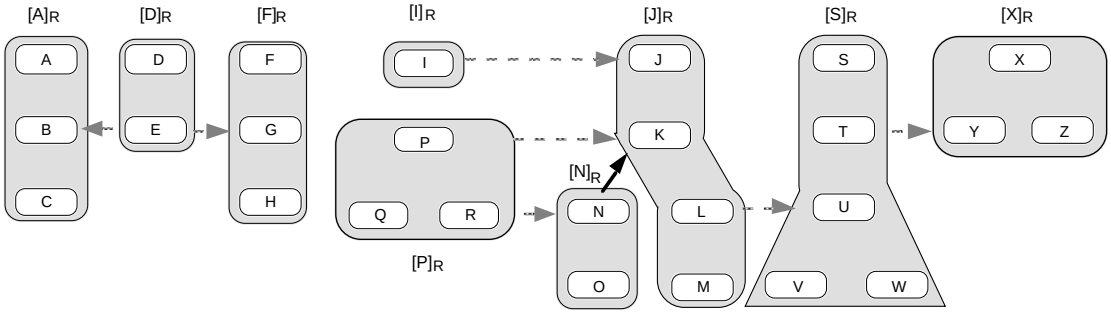


Fig. 21: Hypergraph constructed for the program graph from fig 15.

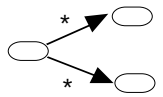
Note that each hypernode corresponds to an equivalence class induced by (4), resp. to a matrix column in fig. 18 and 21 (equivalence class ≈ hypernode ≈ matrix column ≈ *auxArrayIndex*).

5.3.3. Elimination of Insignificant Inner Indices ("holes")

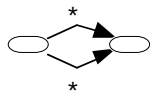
After elimination of redundant and subsuming indices, auxiliary arrays may still contain "holes", i.e. insignificant indices *between* the lowest and the highest significant index (*MinI* and *MaxI*). E.g. in fig. 20, the safety arrays of the classes *I, J, K, L,* and *M* contain undefined values at the position $[N]_R$ and/or $[P]_R$.

In order to eliminate these holes, as required by (3), the columns of the matrices must be reordered, resp. the hypernodes must be numbered with other *auxArrayIndices*. However, it is not obvious, whether a numbering algorithm that satisfies (3) exists and how it works. In order to answer this question we first need some additional terminology describing the structure of program graphs.

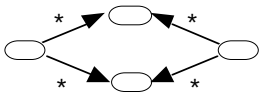
Different paths in the program (hyper)graph are ***potentially conflicting***, if they have the same start node, and at most one path starts with an inheritance edge[1].



Potentially conflicting paths are ***confluent*** if they have the same end node.



A group of potentially conflicting paths is ***hidden confluent*** with another group of potentially conflicting paths if their end nodes are pairwise equal.



Conflicting paths are either confluent paths or hidden confluent paths. It can be shown that

(10) in a program whose (hyper)graph contains *no* conflicting paths, full elimination of holes is *always* possible,

(11) in a program whose (hyper)graph contains conflicting paths, full elimination of holes is *not* possible, in general (only in some special cases).

Lemma (11) can be trivially proved by the counterexamples illustrated in fig. 22a) and 22b). Each example shows one possible numbering and the array that would contain holes if this numbering were used (i.e. the array of node *D* in fig. 22a and the array of node *F2* in fig. 22b). For obvious reasons we cannot show all possible numberings of the two examples. However, it should be easy for the reader to check that no hole-free numbering exists in either case, as there are only a few structurally different numberings of each example. The examples apply to both kinds of auxiliary arrays, since they contain only *declared* parent classes, such that the accessors of safety arrays and of offset arrays are identical.

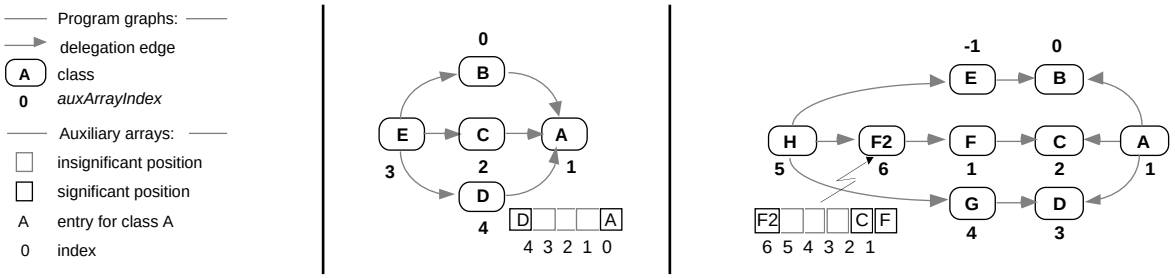


Fig. 22: Counterexample for ... a) confluent paths b) hidden confluent paths

The proof of (10) is done in two steps. The idea is to show that

- in the absence of conflicting paths, a program's hypergraph can always be numbered such that condition (12) and (13) hold, and that
- conditions (12) and (13) imply absence of holes (i.e. condition (3)).

We shall not go into the details of the proof, but limit ourselves to introduce condition (12) and (13) and sketch an algorithm that produces a numbering that satisfies these conditions.

---

1 In the examples on the right hand an edge marked with an asterisk (*) represents a path of arbitrary length.

(12) For all classes, *X*, involved in a delegation hierarchy *X.auxArrayIndex* is at most by 1 smaller / bigger than the minimum / maximum *auxArrayIndex* of *X*'s accessors different from *X* itself (cf. 5.2, definition of *minAI* and *maxAI*):

$$X.minAI - 1 \leq X.auxArrayIndex \leq X.maxAI + 1.$$

The following figure gives some examples of numberings ruled out / allowed by (12). Only the arrays containing holes are shown; "lightning" symbols indicate the places where (12) has been violated.

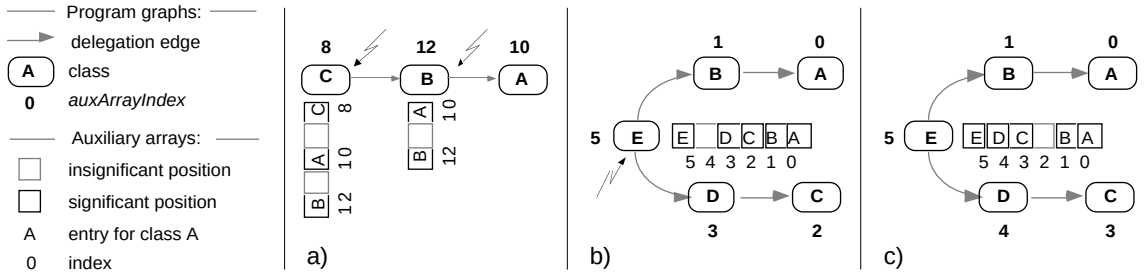


Fig. 23: Example of numberings... a,b) ... ruled out by (12) c) ... allowed by (12)

Note that the numbering shown in 23c) is *not* ruled out by condition (12) although it produces a hole at position 2 in the safety array of class *E*. Obviously condition (12) alone is too weak, since it expresses only the dependency of a class from its accessor classes. The mutual dependency of accessor classes that have a common accessed class is expressed by condition (13):

(13) For all classes, *X*, involved in a delegation hierarchy, the *auxArrayIndex* of all end nodes of maximal[1] potentially conflicting paths that start at *X* is either
- a) smaller than the *auxArrayIndex* of any other node on these paths, or
- b) by 1 bigger than the *auxArrayIndex* of *X*, or
- c) by 1 bigger than the *auxArrayIndex* of some parent node of *X*.

The following figure shows the three numberings of the graph from fig. 23c) that satisfy (12) *and* (13). The maximal potentially conflicting paths are $E \rightarrow B \rightarrow A$ and $E \rightarrow D \rightarrow C$. In all three cases condition (13a) holds for the end node *A*. The numbering of end node *C* satisfies condition (13b) in fig. 24a,c), and condition (13c) in fig. 24b).

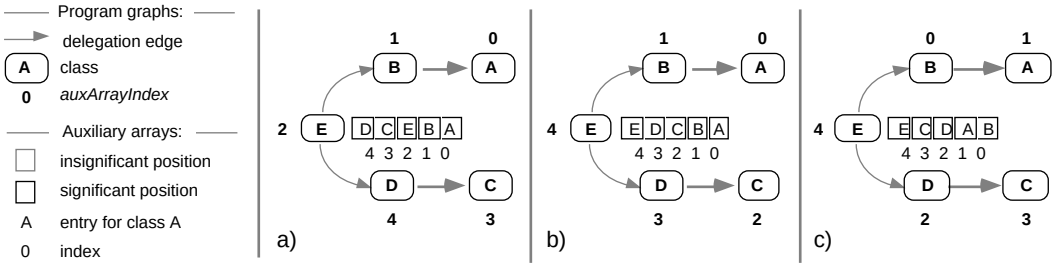


Fig. 24: Example of numberings allowed by (12) ∧ (13)

[1] A path is maximal if its last node has no outgoing edge.

Algorithm

From (12) and (13) we can derive a numbering algorithm that works on the hypergraph constructed according to (4'). For ease of understanding we shall first describe its simplest version, which assigns *safetyArrayIndices* to the nodes of a hypergraph that contains no conflicting path. Then *offsetArrayIndices* will be included into our numbering scheme. Finally the modifications required for the treatment of graphs with conflicting paths will be added.

The algorithm iterates over all delegation hierarchies of the *hyper*graph. It maintains a counter, *nextSAI*, which always contains the lowest *safetyArrayIndex* that has not yet been assigned to an accessor class or an accessed class of the current hypernode. For every delegation hierarchy of the hypergraph the counter is initialised to 0 and the numbering starts at the end node of a maximal path (i.e. at a node with no outgoing edges)[1]. In our example (cf. fig. 21 and 25), index 0 is assigned to the nodes $[A]_R$ and $[X]_R$.

The numbering of any hypernode, *H*, starts by assigning the current value of *nextSAI* to it and incrementing the counter. If the node multiply delegates, e.g. $H = [D]_R$, its yet unvisited ancestors are visited next, starting again with an end node[2]. In our example $[F]_R$ gets the number 2. When the numbering of *H* and of all its ancestors is completed, the current value of *nextSAI* is taken as the initial value for the recursive numbering of all children of *H*[3]. Note that, when no conflicting paths exist, all children get the same number. The process is repeated until all nodes have been numbered.

For including *offsetArrayIndices* into our numbering scheme, we need a second counter, *nextOAI*, which always contains the lowest *offsetArrayIndex* that has not yet been assigned to an accessor class or an accessed class of the current node. Since, in general, the accessors of the offset array of a class are just a subset of the accessors of the safety array, we need to slightly extend the algorithm. The subprocedure that numbers the ancestors of a multiply delegating node, *H*, keeps track, which of the visited nodes are / are not accessors of *H*'s offset array, and returns the successor of the highest *offsetArrayIndex* assigned to an accessor of *H*. This value is assigned to *nextOAI* before starting the numbering of child nodes.

Adaptation of the algorithm to graphs containing confluent paths requires that *subclass* nodes are numbered before child nodes. E.g. in fig. 25 the nodes $[I]_R$ and $[P]_R$ were numbered only after $[J]_R$ and $[N]_R$. Thus their number is the successor of the biggest number assigned to any node from the inheritance hierarchy of $[J]_R$ and $[N]_R$ (4 = 1+ max({2,3})). When a child node that has already been numbered is visited again, and its number is smaller than the current *nextSAI* value, the node is renumbered with the current value.

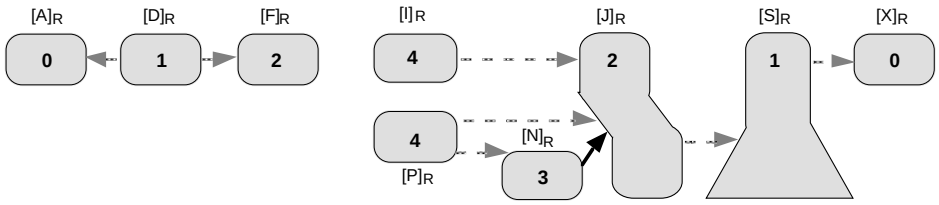


Fig. 25 Numbering of hypernodes / equivalence classes (cf. fig. 19)

---

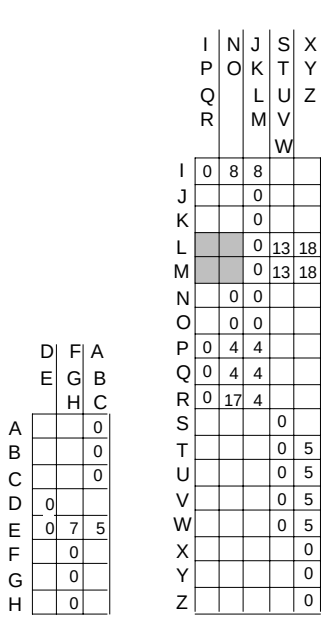
1 This node will be the only one in the current hierarchy for which condition (13a) holds.

2 This step enforces that condition (13c) holds for all end nodes whose number is not 0.

3 This step enforces that condition (12) holds for all nodes that are not end nodes.

Figure 25 shows the numbering of the equivalence classes / hypernodes produced by our algorithm for the hypergraph from fig. 21. The state of the auxiliary array matrices after (re)numbering[1] is shown in fig. 26. Note that the same *auxArrayIndex* has been assigned to $[I]_S$, and $[P]_S$, further reducing the size of the matrices, and that there are no more "holes" within the range of significant indices of any of the auxiliary arrays. The positions that were "holes" in fig. 18 and fig. 20 (marked by a grey pattern), have been moved outside the significant part of the respective arrays.

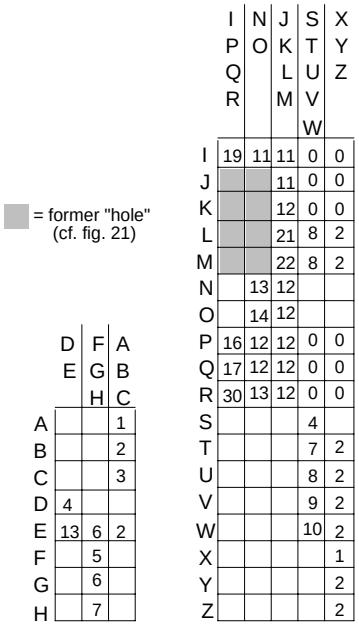
A version of the above algorithm that is correct wrt multiple inheritance can simply be derived by replacing every occurrence of the term "class" by the term "dispatch table", since there is a one to one correspondence between the dispatch tables of a multiply inheriting class and the tables of its superclasses. Each table of a multiply inheriting class gets the *auxArrayIndex* of the corresponding superclass table. Treatment of graphs containing *hidden* confluent paths will not be discussed here, for the sake of brevity, since it can be derived from what has been presented so far.



	D E	F G H	A B C
A			0
B			0
C			0
D	0		
E	0	7	5
F		0	
G		0	
H		0	

	I P Q R	N O	J K L M	S T U V W	X Y Z
I	0	8	8		
J			0		
K			0		
L			0	13	18
M			0	13	18
N		0	0		
O		0	0		
P	0	4	4		
Q	0	4	4		
R	0	17	4		
S				0	
T				0	5
U				0	5
V				0	5
W				0	5
X					0
Y					0
Z					0

Fig. 26a: *Offset* arrays after (re)numbering



	D E	F G H	A B C
A			1
B			2
C			3
D	4		
E	13	6	2
F		5	
G		6	
H		7	

	I P Q R	N O	J K L M	S T U V W	X Y Z
I	19	11	11	0	0
J			11	0	0
K			12	0	0
L			21	8	2
M			22	8	2
N		13	12		
O		14	12		
P	16	12	12	0	0
Q	17	12	12	0	0
R	30	13	12	0	0
S				4	
T				7	2
U				8	2
V				9	2
W				10	2
X					1
Y					2
Z					2

Fig. 26b: *Safety* arrays after (re)numbering

5.3.4. Elimination of Insignificant Leading Indices

In fig. 26 the arrays of the classes *D*, *F*, *G*, *H* and *S* still contain insignificant initial sections, thus violating (2). Since arrays usually start at index 0, fulfilment of (2) requires a numbering scheme that guarantees that for all classes *MinI* = 0. Unfortunately, in the presence of multiple delegation, there exists no numbering that satisfies (1) and (2) simultaneously. Figure 27 shows the minimal counterexample: no matter in which way the *auxArrayIndex* values 0, 1, and 2 are assigned to the classes *A*, *B*, and *C*, the *MinI* of either *A* or *C* will be non-zero.

---

1 The matrix columns in our examples are numbered from *right to left*.

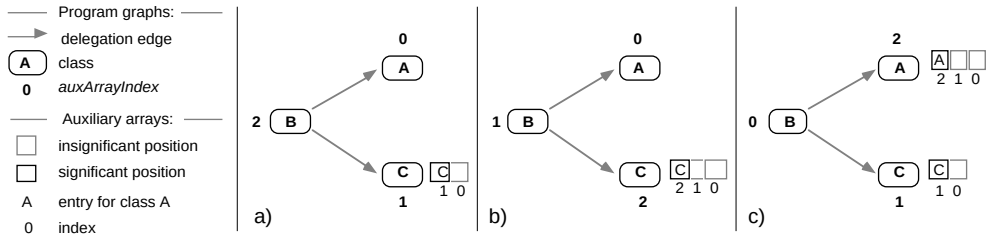


Fig. 27: In the presence of multiple delegation no numbering scheme exists that avoids arrays with an insignificant initial part.

It is, nevertheless, possible to satisfy (2) since we know in advance that the indices between 0 and *MinI* will never be accessed at run-time – they can safely be ignored. Indeed, only the positions *MinI* to *MaxI* are physically allocated. All the range of significant indices is shifted "downwards" by *MinI* positions: the entry that should be at *MinI* becomes the first entry (at index 0), and so on, until the entry that should be at *MaxI* becomes the entry at index *MaxI* - *MinI*. Each reference to the allocated array is statically "adjusted" to point *MinI* positions *before* the first physically allocated element. Thus array accesses using the computed "virtual" *auxArrayIndices* between *MinI* and *MaxI* are correct and require no index adjustment at run-time. E.g. in fig. 27b the array of class *C* will be shifted by two positions, such that the only significant entry will be allocated at index 0. The array reference of *C* will be set to point to position -2 such that array accesses using the *auxArrayIndex* of *C*, 2, will find the correct value.

Fig. 28 shows the state of auxiliary arrays from fig. 26 after removal of insignificant initial (and trailing) indices.

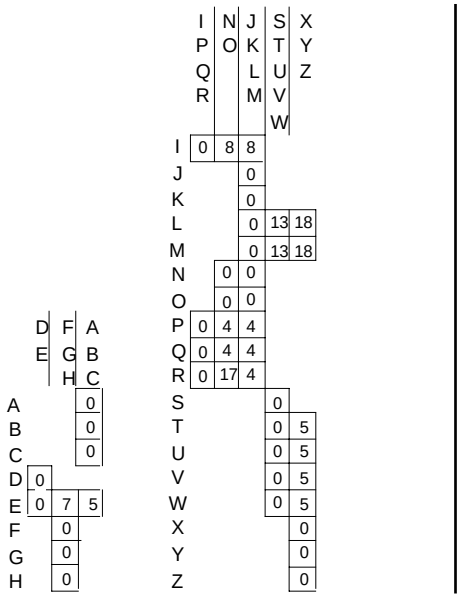


Fig. 28a: *Offset* arrays after elimination of insignificant initial sections

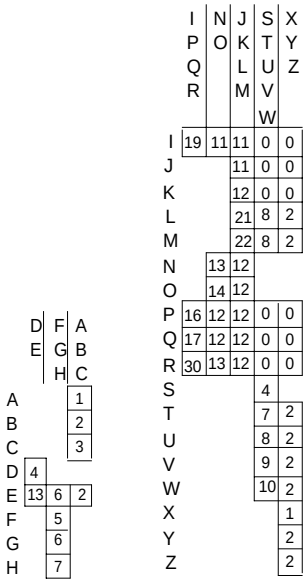


Fig. 28b: *Safety* arrays after elimination of insignificant initial sections

5.3.5. Sharing of Auxiliary Arrays

After elimination of redundancy and subsumption, elimination of holes, and elimination of insignificant initial / trailing sections, there is no way to further reduce the size of *individual* arrays. However, we can still reduce their *total* size, by sharing the same array among different classes, and by not allocating unused arrays.

Sharing of *offset* arrays is possible between a declared parent class and all its *subclasses* that are not themselves declared parent classes, since their offset arrays are *identical*. E.g. in figure 28a the arrays of *T*, *U*, *V* and *W* are identical.

Sharing of *safety* arrays is possible between a declared child class and its declared ancestors, since the safety array of a declared descendant *contains* the arrays of all its declared ancestors. E.g. in figure 28b the array of *J* is contained in the array of *I*, and the arrays of *B*, and *G* are contained in the array of *E*.

Both kinds of auxiliary arrays are only required for classes that are part of a delegation hierarchy. Thus we do not need to allocate any arrays for the classes *A*, *D*, *F*, *S*, *X*, and *Z*, which are outside the two delegation hierarchies of our example (cf. definition in 3.1).

The next figure shows the final state, containing only the physically allocated arrays. Each of the classes in brackets shares the offset array of its superclass (fig. 29a), resp. the highlighted part of the safety array of its declared parent class (fig 29b).

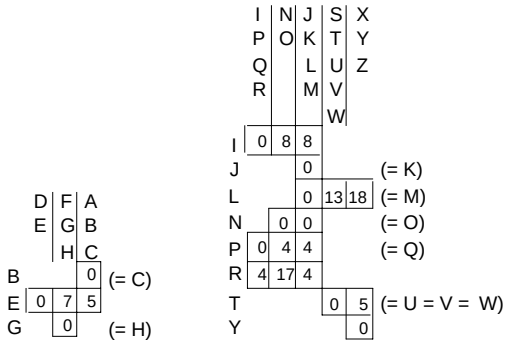


Fig. 29a: Final state of *offset* arrays

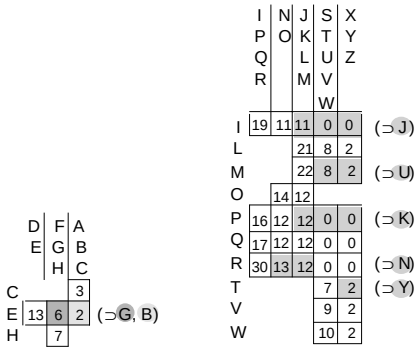


Fig. 29b: Final state of *safety* arrays.

5.4. Summary

Summarising, our auxiliary array layout guarantees constant-time access and almost optimum space efficiency. The auxiliary arrays of *each* class have *minimal* size, except for one type of programs (5.2.1). Their *total* size is minimised too, since classes share their auxiliary arrays whenever possible: each declared child class shares its *safety* array with all its *parent classes*, and each *declared parent* class shares its *offset* arrays with all its *subclasses* that are not themselves declared child classes.

Even for programs that contain conflicting paths, which in general do not allow optimal results, our layout scheme achieves a significant reduction of array size. For our example with 26 classes, the total size of safety arrays was reduced to 39 and the total size of offset arrays to just 23 elements, giving a total of 62 elements, which is just 4,7% of the 1352 ($= 2*26^2$) elements needed by the naive, quadratic approach.

6. Evaluation: The Price to Pay

In this section we analyse the overall space and run-time costs of our implementation. The discussion is of formal nature, since we currently have no complete prototype that can be used for concrete measurements.

6.1. Run-Time

In our analysis we distinguish ***implicit receiver messages***, *self.selector(args)*, and ***explicit receiver messages***, *obj.selector(args)*. Implicit receiver messages may either be sent by *self* (as in purely inheritance-based systems) or by one of its ancestors. Each message may either be bound to a method in (the class of) the receiver or in (the class of) one of its ancestor objects. These distinctions give rise to the six cases summarised in table 5 and discussed in the following. In the table, *VTBL* stands for the cost of the "standard" dispatch technique (cf. 4.1, 4.5.2, and code template A.1 in the appendix), and *L* is the cost of a memory access (*L*oad)

message	**method definition** found in dispatch table of the receiver	the receiver's n-th ancestor
explicit message	*VTBL*	*VTBL + n*(VTBL + L)*
self message sent by *self*	*VTBL*	*VTBL + n*(VTBL + L)*
self message sent by an ancestor of *self*	$VTBL_{AA}$	$VTBL_{AA}$ *+ n*(VTBL + L)*

Table 5: Run-time costs of dynamic binding compared to the costs of purely inheritance-based languages. The costs of dynamic binding in inheritance-based languages is denoted by *VTBL*, the cost of dynamically binding *self* messages using auxiliary arrays is denoted by $VTBL_{AA}$ (cf. text).

The two highlighted cases are the only ones that can occur in inheritance-based languages. In both cases, the dynamic binding costs of our approach are identical to *VTBL*. Thus we have achieved our main goal, to implement dynamic delegation in a way that does not involve any run-time overheads for programs that do *not* use delegation.

When messages must be delegated the "customised lookup code" (cf. 4.3.2 and code template A.2 in the appendix) iterates the *VTBL* dynamic binding on the chain of ancestor objects that is statically known to contain a method definition. The cost of *one* customised lookup code execution is *VTBL + L*, where *L* is the memory access for switching to the next parent object. Thus the *additional* cost of a message delegated to the *n*-th ancestor object is *n**(*VTBL + L*).

For *self* messages that are sent by an ancestor of *self*, an extension of the *VTBL* method was required in order to enable safe and efficient dynamic binding when the type of *self* is statically unknown (cf. 4.4. and code template A.3 in the appendix). The proposed extension, which we call $VTBL_{AA}$ (for *VTBL* with *A*uxiliary *A*rrays), can be divided into five phases. The *initialisation* part involves a single instruction. The *safety check loop* consists of the *s*afety *c*heck (***SC***) itself and the code for getting the next parent object (***getParent***), which includes the dynamic determination of the position of the parent attribute within the current object. Both operations are executed as often as the safety check *fails*. This number appears as ***fsc*** (number of *f*ailed *s*afety *c*hecks) in the following formulas. The loop ends with the first successful safety check (*SC*). It is followed by the *determination of local* d*ispatch* t*able* i*ndices* (***DTIndices***), the decision whether to access the *r*eceiver or *d*elegatee version of the target method (***R?D***), and the application of the *VTBL* dynamic binding (***bind***).

Thus the cost of getting the compiled code from the first dispatch table for which the message is access safe is

$(VTBL_{AA})$ $\quad 1 + fsc*(SC+getParent) + SC + DTIndices + R?D + bind.$

Expressing the above costs in terms of executed instructions would be a neglect of modern hardware architectures. On pipelined processors ([SPARC92]) only the total number of cycles can be an approximate measure, due to latencies involved in memory accesses and especially in branch instructions. Refilling the processors pipeline with code from an unpredictable branch target address involves a multi-cycle execution delay. E.g. [CaGr94] reported that the elimination of the branch instruction in the *VTBL* method could lead to a performance improvement of up to 66% for *C++* programs. Recently, [DHV95] concluded a detailed comparison of different dispatch methods. They showed that methods that contain no unpredictable branches and whose instructions have few data dependencies can outperform *VTBL* inspite of executing more instructions, since they do not incur branch penalties and can execute independent instructions in parallel.

We have analysed our approach with the method described in [DHV95] and with respect to the *P97* class of processors. The assumed characteristic of *P97*, which models the next generation of processors ([DHV95]), is the ability to execute up to four instructions in parallel (but at most two loads or one branch). All instructions, except memory accesses and branches, execute in one cycle. In the following formulae *L* is the load latency and *B* is the branch latency. The assembler code for the dispatch of *self* messages whose sender is an ancestor of *self*, and the corresponding instruction schedule on *P97* are contained in the appendix. From the schedule we can see that the cycle costs of our code are

$$SC = 3L + 2, \quad getParent = L - 1, \quad DTIndices = 0, \quad R?D = 2, \quad bind = B + L + 1.$$

At a first glance, these results are astonishing:

- inspite of missing knowledge about *self's* type, the net cost for determining the dispatch table indices (*DTIndices*) is *zero*,
- the net cost of getting the parent reference (*getParent*) is even *less* then a load instruction,
- and, although the *VTBL* method is used, the cost for the final binding to the target code (*bind*) is *less* then the cost of *VTBL*, which is $B + 2L + 1$ ([DHV95]).

These results stem from the good exploitation of parallelism possible in our scheme due to the relatively small number of data dependencies. *DTIndices* and the determination of the parent reference offset can be done in parallel to the safety check. Scheduling of the remaining access to the parent reference into the delay slot of the safety check branch instruction, produces the surprising cycle count of less then a load latency for *getParent*. The additional subtraction for accessing delegatee code during *bind* can be executed in parallel to the safety check branch. Since the first step of *VTBL*, the access to the dispatch table, is already part of the safety check loop, the cost of *bind* is $VTBL\text{-}L = B+L+1$.

Substituting these values into the above formula we get the cost of $VTBL_{AA}$ on *P97*:

$$(VTBL_{AAP97}) \qquad 1 + fsc*(3L + 2 + L - 1) + 3L + 2 + 0 + 2 + B + L + 1$$

$$= fsc*(4L + 1) + B + 4L + 6.$$

Since *VTBL* is not applicable to cases that would produce failed safety checks, we have to compare the cost of $VTBL_{AA}$ and *VTBL* when $fsc = 0$. It is

$(VTBL_{AAP97}, fsc = 0)$ $B + 4L + 6.$

Thus, compared to *VTBL*, the dynamic binding of *self* messages when the type of *self* is statically unknown involves a constant additional cost of $2L+5 = 11$ cycles on a *P97* processor ([DHV95]):

$$VTBL_{AAP97} = VTBL_{P97} + 2L + 5 = 1{,}85 * VTBL_{P97}$$

Summarising, even in programs that use delegation additional costs are only incurred by delegated messages and subsequent *self* messages. For *self* messages sent during delegated messages a constant overhead (of 11 cycles on *P97*) is incurred. For delegated messages the overhead is linear in the number of searched parent objects, which, in general[1], is statically bounded. Note that the complexitiy of accessing a variable in an outer block by following the *origin* / *StaticFather* reference in *BETA* / *Simula* ([MMN93], [KLLM93]) is also linear in the number of parents / *StaticFather*s. Supporting the block concept of *BETA* requires a step from constant to linear complexity, whereas supporting the much powerful concept of dynamic delegation does not change the complexity class again.

6.2. Space

The size of a compiled program is the compiled code size plus the size of run-time data structures.

In our approach the size of data structures is the size of dispatch tables plus the size of auxiliary arrays. As shown in chapter 5, the space costs involved in auxiliary arrays is small compared to the costs of dispatch tables, since the former only depends on the number of *classes* in a program whereas the latter depends on the number of *methods*. The theoretical upper bound of auxiliary array size, $totalNrOfClasses^2$, is unlikely to ever occur in non-trivial programs, since the pathologic quadratic case requires the program graph to consist of one single chain of delegating classes. In practice our dispatch table layout achieves order of magnitude improvements, e.g. in the example used throughout chapter 5, less then 5% of the worst case quadratic costs were needed.

Regarding the size of dispatch tables, the addition of the methods defined in the inheritance hierarchies of parent classes to the dictionary of a delegating class has the same space complexity as the implementation scheme for multiple inheritance (cf. 4.5.2.). If a class, *C*, has *n* immediate superclasses and/or parent classes, whose dispatch tables have size $S_1, ..., S_n$, and if S_0 is the number of new methods defined in *C* itself, then its dispatch table will have size

$$S_C = 3 + 2*(S_0 + S_1 + ... + S_n,).$$

Three words are required for the reserved positions of the dispatch table (safety array reference, offset array reference and a yet unused position). The factor 2 represents the duplication of the dispatch table size due to weak customisation. The sum $S_0 + S_1 + ... + S_n$, is the size of dispatch tables that corresponds to multiple inheritance. Note that in our system, this formula

---

[1] For explicit receiver messages and *self* messages sent by *self*, the number of customized lookup code executions, *n*, is statically bounded by the length of the delegation path from the class of the receiver to the first *declared* ancestor class that contains a method definition. For *self* messages that are sent by an ancestor of *self*, *n* is dynamically bounded by the length of the delegation path from *self* to the *self* message sender. Even in the case of multiple delegation, no other paths need to be explored at run-time.

equally applies to pure multiple inheritance, pure multiple delegation, and any combination of them.

It is especially interesting to compare the last case to the purely inheritance-based modelling used to simulate the same functionality (cf. fig. 4a and fig. 5). The dispatch tables of the "intersection classes" needlessly duplicate the tables of their superclasses. Thus, when comparing applications of *equal functionality*, the apparent double table size of our approach is outweight by the size of the additional dispatch tables needed otherwise. Put differently, weak customisation just "spends" the storage wasted otherwise on the dispatch tables of intersection subclasses.

In our approach the compiled code is

- the code referenced by the "receiver part" of dispatch tables (which is exactly the code generated by the *VTBL* method),
- the additional code referenced by the "delegatee part" of dispatch tables, and
- the additional customised lookup code (shared by delegatee and receiver parts).

The size of customised lookup code is only five instructions, which are generated for each delegated method, and are reused throughout an inheritance hierarchy. It is hard to predict how much this will contribute to overall program size, since it strongly depends on the degree to which messages in the delegated protocol of a class are redefined locally.

The replacement of *VTBL* code by $VTBL_{AA}$ code, for each *self* message, increases the size of code generated for *self* messages in the delegatee version of a method almost by a factor of 5 (22 instead of 5 instructions, cf. appendix). Assuming that 60-80% of the messages in a program are *self* messages, the dispatch code size in delegatee versions increases 3,4 to 4,2 times (3,4 = 0,6*5 + 0,4 and 4,2 = 0,8*5 + 0,2). Thus the total dispatch code size, which is the size of dispatch code in receiver versions, ***codeSize(VTBL)***, plus the size of code in delegatee versions, ranges between *4,4*codeSize(VTBL)* and *5,2*codeSize(VTBL)*.

Summarising, our approach involves only a small increase of data structure size introduced by the use of auxiliary arrays. This is in sharp contrast to the strong increase of code size. However, since for *VTBL* data structure size dominates dispatch code size by a ratio of approx. 3:1 ([DHV95]), the code size increase will not increase the total compiled program size as much as one might expect. If *x* is the overall dispatch related storage costs of a program compiled with the *VTBL* method, the corresponding costs of our approach will be

$$dataSize + codeSize$$

$$= auxArraySize + dTableSize + receiverCode + delegateeCode + customizedLookupCodeSize$$

$$= auxArraySize + 0{,}75x + 0{,}25x + 5{,}2*0{,}25x + customizedLookupCodeSize$$

$$= 2{,}3x + auxArraySize + customizedLookupCodeSize.$$

Thus, assuming that the results reported in [DHV95] can be generalised, we expect a maximum increase of dispatch related storage costs by a factor of 2,5 over the costs involved in purely inheritance based programs of similar functionality.

7. Related Work

In the domain of strongly typed, inheritance-based programming languages we are not aware of any previous attempt to implement dynamic delegation, except the failed experiment with "delegation through a pointer" described in [Stro87] and [Stro94]. "Delegation through a pointer" was not really delegation but an implicit resending mechanism (cf. 2) with a bunch of additional restrictions, imposed by the aim to squeeze the concept into the implementation scheme for multiple inheritance. In his retrospective evaluation ([Stro94], p. 272) Stroustrup gives the following reasons for not including this concept into *C++*:

"Unfortunately, every user of this delegation mechanism suffered serious bugs and confusion. Two problems appeared to be the cause [...]:
- Functions in the delegating class do not override functions of the class delegated to.
- The function delegated to cannot use functions from the delegating class or in other ways 'get back' to the delegating object.

[...] We also found examples where we wanted to have two objects delegate to the same 'shared' object. Similarly, we found examples where we needed to delegate [...] to an object of a derived class[1]".

Our approach includes all the missing features, without compromising the "don't pay for what you don't use" design rule promoted as one of the guiding design principles in [Stro94].

In the domain of untyped, delegation-based languages the most advanced implementation techniques have been developed in the framework of the *SELF* project ([US87], [CUL89], [CU91], [HCU91], [USCH92], [Hölz94a]). The implementors of *SELF* have demonstrated, that pure object-oriented languages based on prototypes and delegation can be implemented efficiently and that optimisation of compiled code quality and of compile time need not contradict each other ([Hölz94b]). The work done in the design and implementation of the *SELF* system was one main motivation and inspiration for the integration of delegation in more conventional languages.

The *SELF* compiler is strongly based on customisation, inlining and dynamic recompilation. These techniques are tuned for rapid prototyping within the *SELF* development environment but they seem incompatible with production programming, especially with delivery of libraries of reusable compiled code and of turn-key applications[2]. All optimizations are only applied if parent slots are declared to be static. There is *no* compilation / optimisation of *dynamic* delegation in *SELF*. To the best of our knowledge, the same is true for all other delegation-based languages, ranging from efficiently compiled languages like *Cecil* ([Cham93])[3], to experimental language frameworks based on reflective architectures ([MC93], [DMC92])[4].

---

1 In our terminology, it was not possible to "delegate" to an instance of a *potential* parent class (cf 3.1).

2 Although first steps towards the solution of the latter problem are described in [AgUn94].

3 In *Cecil* delegation is generally required to be static.

4 Reflection is a very powerful tool for object-oriented programming ([Coin87], [Ferb89]) but implementation techniques for reflective architectures that achieve similar performance as *C++*, *Self* or *Cecil* are still unknown.

Compared to *SELF*, our scheme implements dynamic delegation without the expense of full run-time search of method dictionaries. The customised lookup code reduces the lookup in each of the traversed dispatch tables to *one* access at a statically known index. The time complexity of this search is linear in the number of *parents*, whereas the search performed for dynamically delegated messages in *SELF* ([CUL89], [Hölz94a], [Hölz94c]) is linear in the total number of *selectors* within the parents' hierarchies – which may be orders of magnitude higher. Even if the *SELF* compiler used hashing instead of linear search, it would at least need to dynamically compute the hash function and check for collisions, where we can directly access a statically known index. Note also, that in our approach only one delegation path will be explored at run-time, even in the presence of multiple delegation, and the number of parent objects on this path is, in general, statically bounded (cf. 6.1. and 4.5.1). Due to the missing static type information there is no way how to achieve in *SELF* a similar reduction of the search space, without additional run-time overhead[1].

In the domain of object oriented database systems (OODB) there has been considerable work on "role object models". The approach of [Scio89] "integrates" delegation and classes by simulating classes in a prototype-based OODB environment. "Instances" are represented by multiple delegating objects, one for the instantiated "class" and one for each "superclass", forcing an extreme fragmentation of objects and use of delegation, where it is not necessary. Our integrated model avoids this problem by allowing all features that should in the end be permanently attached to *one* object to be modelled in an inheritance hierarchy, whereas delegation is only used to express transient properties, as dynamic sharing relations between different objects. Sciore also offers no hints how to achieve an efficient implementation.

Role object models that are conceptually much closer to our integrated model are presented in [WRS94] and [GSR94]. Both allow two hierarchies, subclasses and role classes. In both approaches the role hierarchy is based on implicit resends instead of delegation and there are no considerations how an efficient implementation could be achieved[2]. Note that the

The only role model known to us that has a time-efficient implementation is the one underlying the database language Fibonacci ([ABGO93]). The implementation is based on creating a dispatch table *for every object* and dynamically modifying it when an object acquires a new role. The storage demand and the need for dynamic dispatch table restructuring of this approach is out of line with the needs of production programming languages. It also appears that the model of Fibonacci is more restricted than ours, since it is based on only *one* static (type) hierarchy. There is no equivalent to our statically declared delegation hierarchy, which controls the roles that can be acquired at run-time by an object. The lack of this structuring mechanism allows semantically incompatible roles to be played simultaneously.

We may thus conclude that our model is the first one that marries object-based delegation and class-based inheritance in the framework of a static type system, and that our implementation scheme is the first one that allows an implementation of *dynamic* delegation, in a way that is compatible with the state-of-the art implementation techniques for object-oriented "production programming" languages.

---

1 In fact, the "sender path tiebreaker rule" ([CUCH91]) introduced in *SELF* version 2.0 achieved the same effect, but it had to be enforced at run-time.

2 The model of [GSR94] is implemented in Smalltalk ([GR89]), based on the ability to create classes at run-time and to treat messages as first-class objects. The model of [WRS94] is formulated in modal logic.

8. Conclusions

In this paper we have shown how "mainstream", statically type-checked, inheritance-based languages can be extended to include dynamic delegation. Delegation allows instances to "inherit" from other instances. Hierarchies of delegating objects can be constructed an modified dynamically. Especially, objects can vary their behaviour at run-time by delegating to another "parent" object (without compromising type safety). Combined delegation and inheritance hierarchies can express dynamically changing roles of objects, obviating the need for special purpose "role object models".

In spite of this added functionality, delegation can be implemented in a way that is compatible with the efficient implementation techniques currently used in typed, inheritance-based languages. *Extended dispatch tables* preserve constant time dynamic binding within inheritance hierarchies, even in the presence of delegation. For delegated messages, the full run-time search of each method dictionary along a delegation hierarchy, which is used in untyped languages in the presence of dynamic delegation, is reduced to only one access to each dispatch table along a path (in the worst case). No comparisons or conditional branches are performed during this "search". A special problem introduced by delegation is that no receiver type is statically known for messages to *self* sent during the evaluation of *delegated* messages. The proposed solution using *safety arrays* and *offset arrays* only adds a constant cost to these messages.

Due to *weak customisation*, added costs only occur if messages are really delegated. If delegation is not needed for a given message, it will incur no run-time penalty, even in classes that are defined using delegation. All our techniques are extensions of the "classic" dispatch table based dynamic binding scheme, such that they should fit well into existing compilers. Hence, in principle, the way is open for the extension of typed, inheritance-based languages with a powerful feature previously known only in the context of rapid prototyping systems based on type- and class-free object models.

Nevertheless, there are still many open questions left. Is it possible to further reduce the costs of delegated messages, e.g. by replacing the customised lookup code with a method that does not depend on expensive indirect branches? What role can other dispatch techniques, e.g. hashing or variants of inline caching ([HCU91], [DHV95]), play in a system like ours? What improvements in space an time efficiency could be achieved by relaxing the constraint of compatibility with standard techniques? Are they worth the price?

It seems that the borderline between overconstraining delegation such that it is useless and unchaining it such that it becomes unmanageable with "traditional" implementation techniques is quite narrow. We think that, for the time being, we have found a good compromise. However, we would like to study how far the strictly static type-checking scheme assumed in this paper can be relaxed. Another still open field is the application of global optimisation techniques e.g. in the spirit of [DGC94] and [VHU92]. It seems that within our integrated model, static analysis bears a high potential for optimisations of run-time and compiled code size.

We hope that this paper will contribute to rid delegation of the aura of being overly expensive, conceptually unmanageable, and incompatible with static typing. If, one day, implementors of typed, inheritance-based languages will start considering the integration of dynamic delegation into their systems we will have achieved our goal.

Acknowledgements

I thank Thomas Lemke and Wolfgang Reddig for long discussions, constructive criticism, and helpful suggestions regarding the contents of this paper as well as its presentation. Herrmann Stamm shared with me his detailed knowledge of graph theory and efficient graph algorithms, which helped in the design of the numbering algorithm that determines the layout of auxiliary arrays. Thanks are also due to Jan Vitek for exchange of ideas and useful references.

References

[ABGO93]

A. Albano, R. Bergamini, G. Ghelli, R. Orsini: "An Object Data Model with Roles". In Proceedings of the 19th VLDB Conference, Dublin, Ireland, 1993.

[AgUn 94]

O. Agesen, D. Ungar: “Sifting Out the Gold: Delivery of Compact Applications from an Exploratory Object-Oriented Programming Environment”. In OOPSLA 94 Proceeding, *SIGPLAN Notices, **29**(10), October 1994*, ACM Press, pp. 355-370.

[BrCo90]

G. Bracha, W. Cook: “Mixin-based Inheritance”. In OOPSLA / ECOOP ‘90 Proceedings, Oct 1990, Ottawa, Canada *SIGPLAN Notices, **25** (10),* Oct. 1990, ACM Press., pp. 303-312.

[CaGr94]

Brad Calder, Dirk Grunwald: "Reducing indirect function call overhead in *C++* Programs". In Conference Rocord of *POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, pp. 397-408.

[CeMa93]

Stefano Ceri, Rainer Manthey: "Consolidated specification of Chimera". Report IDEA.DE.2P.006.01, ESPRIT Project 633 IDEA, Nov. 1993.

[Cham93]

C. Chambers: "The *Cecil* Language: Specification and Rationale". Dept. of Computer Science and Engineering, Univ. of Washington, TR 93-03-05, March 1993 (available on the www and by ftp from {www,ftp}.cs.washington.edu.

[CL94]

C. Chambers, G T. Leavens: "Type-checking and Modules for Multi-Methods". In OOPSLA 94 Proceedings, *SIGPLAN Notices **29**, 10* (October 1994), ACM Press.

[CoPa89]

W. Cook, J. Palsberg: “A Denotational Semantics of Inheritance and its Correctness”. In OOPSLA ‘89 Conference Proceedings, *SIGPLAN Notices, **24** (10),* Oct. 1989, ACM Press, pp. 433-444.

[CUCH91]

Craig. Chambers, David. Ungar, Bay-Wei. Chang, Urs. Hölzle: “Parents are Shared Parts of Objects: Inheritence and Encapsulation in *SELF*”. In *Lisp and Symbolic Computation: An International Journal, **4**, 3,* 1991, Kluwer Academic Publishers, pp. 21-36.

[CUL89]

C. Chambers, D. Ungar, E. Lee: “An Efficient Implementation of *SELF* a Dynamically-Typed Object-Oriented Language Based on Prototypes”. In OOPSLA ‘89 Conference Proceedings, *SIGPLAN Notices, **24** (10),* Oct. 1989, ACM Press, pp. 49-70.

[CU91]

C. Chambers, D. Ungar: “Making Pure Object-Oriented Languages Practical”. In OOPSLA ‘91 Conference Proceedings, *SIGPLAN Notices, **26** (11),* Oct. 1991, ACM Press, pp. 1-15.

[Coin87]

Pierre. Cointe: “Metaclasses are First Class: The ObjVlisp Model”. In OOPSLA 87 Proceedings, Special issue of *SIGPLAN Notices* ***22*** *(12),* December 1987, ACM Press, pp. 156-167.

[DGC94]

Jeffrey Dean, David Grove, Craig Chambers: "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis". University of Washington, Dept. of Computer Science and Engineering, Techical Report 94-12-01, December 1994. [DHV95]

Karel Driesen, Urs Hölzle, Jan Vitek: "Message Dispatch on Pipelined Processors". Technical Report, University of California at Santa Barbara, 1995, accepted at ECOOP'95, to appear in Lecture Notes in Computer Science, Springer Verlag, 1995.

[DMC92]

C. Dony, J. Malenfant, P. Cointe: “Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation”. In OOPSLA 92 Proceedings, *SIGPLAN Notices*, ***28*** *(10),* Oct. 1993, ACM Press, pp. 201-217.

[DMN68]

O.J. Dahl, B. Myrhaug, K. Nygaard: *"SIMULA 67 Common Base Language"*. Norwegian Computing Center, Oslo, 1968.

[DuHa91]R. Ducournau, M. Habib: “Masking and Conflicts, or To Inherit is Not to Own!“. In [LNS91], pp. 223-242.

[Duge91]

P. Dugerdil: “Inheritance Mechanisms in the OBJLOG Language: Multiple Selective and Multiple Vertical with Points of View”. In: [LNS91], pp. 245-256.

[ES90]

Margret A. Ellis, Bjarne Stroustrup: *"The Annotated C++ Reference Manual"*. Addison-Wesley, Reading, MA, 1990.

[GSR94]

G. Gottlob, M. Schrefl, B. Röck: "Extending Object-Oriented Systems with Roles". To appear in *Journal of the ACM*.

[GR89]

Adele Goldberg, David. Robson: "*Smalltalk-80: The Language*". Addison-Wessley, 1989.

[HCU91]

Urs Hölzle, Craig Chambers, David Ungar: "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches". In Proceedings of ECOOP '91, Lecture Notes in Computer Science 512, Springer Verlag, 1991, pp. 21-38.

[Hölz94a]

U. Hölzle: "Adaptive Optimization for *SELF*: Reconciling Hihg Performance With Exploratory Programming". PhD thesis, Stanford University. Available electronically from file://self.stanford.edu/pub /*SELF*-3.0/papers/hoelzle_thesis.ps.Z.

[Hölz94b]

U. Hölzle: "A Third Generation *SELF* Implementation: Reconciling Responsiveness with Performance". In OOPSLA 94 Proceeding, *SIGPLAN Notices, **29**(10), October 1994,* ACM Press, pp. 229-243.

[Hölz94c]

Urs Hölzle: Private e-mail exchange, Sept. 1994.

[KL89]

Won Kim, Frederick H. Lochovsky (Eds.): *“Object-oriented Concepts, Databases, and Applications”*. ACM Press, Addison-Wesley 1989.

[KLL+93]

J.L. Knudsen, M. Löfgren, O. Lehrmann-Madsen, B. Magnusson: *"Object-Oriented Environments - The Mjølner Approach"*. Prentice Hall International Ltd., Hemel Hempstead, UK, 1993.

[Knie95]
Günter Kniesel: "A Uniform Model of Sharing in Object-Oriented Programming". Ph.D.-thesis, University of Bonn, 1995 (in preparation).

[Krog84]
Stein Krogdahl: "An Efficient Implementation of *Simula* Classes with Multiple Prefixing". *Research Report No. 83, June 1984, Institute of Informatics, University of Oslo*. ISBN 82-90230-82-6.

[Krog85]
Stein Krogdahl: "Multiple inheritance in *Simula*-like languages". In *BIT 25* (1985), pp. 318-326.

[LNS91]
Maurizio Lenzerini, Daniele Nardi, Maria Simi (Eds.): *"Inheritance Hierarchies in Knowledge Representation and Programming Lanfuages"*. Wiley 1991.

[LTP86]
W. R. LaLonde, D. A. Thomas, J R. Pugh: "An Exemplar Based Smalltalk". In OOPSLA 86 Proceedings, *SIGPLAN Notices, **21**(11), 1986*, ACM Press, pp. 322-330.

[Lieb86]
H. Liebermann: "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems". In OOPSLA 86 Proceedings, *SIGPLAN Notices, **21**(11), 1986*, ACM Press, pp. 214-223.

[Meye92]
B. Meyer: *"Eiffel: The Language"*. Prentice-Hall, 1992.

[MC93]
P. Mullet, P. Cointe: "Definition of a Reflective Kernel for a Prototype-Based Language". In [Nishio & Yonezawa 93], pp. 128-144.

[MMN93]
Ole Lehrmann Madsen, Birger Moller-Pedersen, Kristen Nygaard: *"Object-Oriented Programming in the BETA Programming Language"*. Addison-Wessley and ACM Press 1993.

[NRE92]
G.T. Nguyen, D. Rieu, J. Escamilla: "An Object Model for Engineering Design". In [ECOOP 92], pp. 233-251.

[Nier89]
Oscar Nierstrasz: "A Survey of Object-Oriented Concepts". In [KL89], pp. 3-22.

[Pern90]
Barbara Pernici: "Objects with Roles". In *Proceedings of the Int. Conf. on Office Information Systems*, Boston (Ma), ACM Press, 1990.

[PS94]
Jens Palsberg, Michael. I. Schwartzbach: *"Object-Oriented Type Systems"*. Wiley & Sons, 1994.

[RBP+91]
J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *"Object-Oriented Modeling and Design"*. Prentice Hall, 1991.

[Scio89]
E. Sciore: "Object Specialization". *ACM Transaction on Information Systems, 7(2), 1989*, pp. 103-122.

[SLU89]
L. A. Stein, H. Liebermann, D. Ungar: "A Shared View of Sharing: The Treaty of Orlando". In [KL89], pp. 31-48.

[Snyd86]A. Snyder: "Encapsulation and Inheritance in Object-Oriented Programming Languages". In OOPSLA 86 Proceedings, *SIGPLAN Notices, **21**(11), 1986*, ACM Press, pp. 38-45.

[Snyd91]
A. Snyder: "Inheritance in Object-oriented Programming Languages". In [LNS91], pp. 153-172.

[SPARC92]
SPARC International Inc.: "*The SPARC Architecture Manual, Version 8*". Prentice Hall, 1992.

[Stro87]
B. Stroustrup: "Multiple Inheritance for *C++*". Proc. of European Unix User's Group Conference, Helsinki, May 1987.

[Stro91]
B. Stroustrup: "The *C++* Programming Language". Addison-Wessley Publishing, 1991, Second Edition.

[Stro94]
Bjarne Stroustrup: "The Design and Evolution of *C++*". Addison-Wesley, Reading, MA, 1994.

[US87|
D. Ungar, R.B. Smith: "*SELF*: The Power of Simplicity". In OOPSLA 87 Proceedings, Special issue of *SIGPLAN Notices **22** (12)*, December 1987, ACM Press, pp. 227-242.

[USCH92|
D. Ungar, R.B. Smith, C. Chambers, Urs. Hölzle: "Object, Message and Performance: How they coexist in *SELF*". In *IEEE Computer, **25**, 10,* (October 1992).

[VHU92]
Jan Vitek, R. Nigel Horspool, James S. Uhl: "Compile-time analysis of object-oriented programs". In Proceedings CC'92, 4th International Conference on Compiler Construction, Paderborn, Germany, pp. 236-250. Springer Verlag (LNCS 641), 1992.

[Wegn89]
P. Wegner: "Concepts and paradigms of Object-Oriented Programming" (Expansion of OOPSLA 89 keynote talk). *OOPS Messenger, **1**,1 (August 1990)*, pp. 7-87.

[WRS94]
R. Wieringa, W. de Jonge, P. Spruit: "Roles and Dynamic Subclasses: A Modal Logic Approach". In ECOOP 94 Proceedings, Springer-Verlag, LNCS 821, 1994.

Appendix

Notation

In the following code templates we use the meta-variable SC for the class that textually contains the message (the "Source Class"), and RT for the declared type of the receiver object (the "Receiver Type"). Underlining highlights statically known parts of an expression. Note that underlining of a component name, as in *x.c*, indicates that the *offset* (*not the value*) of component *c* within the storage area referenced by variable *x* is statically known. Since the compiler assigns own, statically determined indices to the different components of structures strored in an array, we write

- `offsetArray[SC.parentOffsetIndex]` for `offsetArray[SC.offsetArrayIndex].parentOffset`
- `offsetArray[SC.dTableOffsetIndex]` for `offsetArray[SC.offsetArrayIndex].dTableOffset`
- `dTable[methodIndex(selector,SC)]` for `dTable[index(selector,SC)].methodRef`
- `dTable[deltaIndex(selector,SC)]` for `dTable[index(selector,SC)].delta`.

Explicit receiver message, *receiver.selector(args)*, and
implicit receiver message, *self.selector(args)*, when *self* is the sender of the message

```
method := receiver.dTable[methodIndex(selector,RT)]          // dynamic binding
delta  := receiver.dTable[deltaIndex(selector,RT)]           // (re)adjustment ...
object := receiver+delta                                     // ... of self reference
method(object, args)                                         // procedure call with adjusted self reference
```

Code template *VTBL*: "Standard" dispatch method, including treatment of multiple inheritance

Customized lookup code

```
delegate(receiver, delegatee, args, selector, parentType) = {
  delegatee := delegatee.parent                                  // get parent object
  method    := delegatee.dTable[-indexOfMethod(selector,parentType)]  // get delegatee code
  delta     := delegatee.dTable[indexOfDelta(selector,parentType)]    // (re)adjustment ...
  delegatee := delegatee + delta                                 // ... of delegatee reference
  method(receiver, delegatee, args)                              // call delegatee code
}
```

Code template *CLC*: Customized lookup code,
including treatment of multiple inheritance and "weak customisation"

Implicit receiver message, *self.selector(args)*, when *self* is *not* the sender of the message

```
delegatee := self

// While not access safe: delegate message to parent
while ( methodIndex(selector,SC) > delegatee.dTable[0].[SC.safetyArrayIndex] )
   do { parentOffset := delegatee.dTable[1].[SC.parentOffsetIndex]
        delegatee := delegatee[parentOffset]
      }

// Calculate local indices using offset array
offset      := delegatee.dTable[1].[SC.dTableOffsetIndex]
methodIndex := methodIndex(selector,SC) + offset
deltaIndex  := deltaIndex(selector,SC)  + offset

// (Almost) normal dispatch to compiled code
if delegatee == self
   // if the message was safe for self
   then { // execute standard "receiver version"
          method := delegatee.dTable[methodIndex]
          delta  := delegatee.dTable[deltaIndex]
          method (self+delta, args)
        }
   else { // execute full "delegatee version"
          method := delegatee.dTable[-methodIndex]
          delta  := delegatee.dTable[-deltaIndex]
          method (self, delegatee+delta, args)
        }
```

Code template $VTBL_{AA}$: Pseudocode version of auxiliary array based dispatch of *self* messages (cf. fig. 11), including safety check with dynamic determination of the parent attribute offset, treatment of multiple inheritance, and "weak customisation"

Implicit receiver message, *self.selector(args)*, when *self* is *not* the sender of the message

```
(1)             load  self, delegatee
        loop:
(2)             load  [delegatee + #dTtableOffset], table
(3)             load  [table + 0], safetyArray
(4)             load  [table + 1], offsetArray
(5)             load  [safetyArray + #SC_safetyArrayIndex],  maxSafeIndex
(6)             load  [offsetArray + #SC_dTableOffsetIndex], dTableOffset
(7)             load  [offsetArray + #SC_parentOffsetIndex], parentOffset
(8)             add   dtableOffset, #SC_methodIndex, methodIndex
(9)             add   dtableOffset, #SC_deltaIndex, deltaIndex
(10)            cmp   maxSafeIndex, #SC_methodIndex
(11)            load  [table, deltaIndex], delta
(12)            sub   table, methodIndex, methodReference
(13)            ble   #loop
(14)            load  [delegatee, parentOffset] delegatee              // delay slot
        execute:
(15)            cmp   self, delegatee
(16)            bne   #callDelegateeVersion
        callReceiverVersion:
(17)            load  [table, methodIndex], method
(18)            add   self, delta, self
(19)            call  method
        callDelegateeVersion:
(20)            load  [methodReference], method
(21)            add   delegatee, delta, delegatee
(22)            call  method
        continue: ... first instruction after self message ...
```

Assembler version of code template $VTBL_{AA}$.
The instruction sequence corresponds to the schedule illustrated in fig. A1

```
R1                          a register (any argument without #)
R#1                         a register or an immediate (prefix R#)
#immediate                  a constant value (prefix #)
load    [R1, R#2], R3       set R3 to word at adress R1+R#2
-------------------------------------------------------------------
add     R1, R#2, R3         set R3 to R1+R#2
sub     R1, R#2, R3         set R3 to R1-R#2
comp    R1, R#2             compare value in register R1 and R#2
-------------------------------------------------------------------
bne     #imm                if last compare is not equal, jump to #imm
ble     #imm                if last compare is less or equal, jump to #imm
call    R#1                 jump to address in R#1, saving return address
```

Abstract assembler instruction set (based on [DHU95])

In the following figure, nodes represent instructions and arcs represent data and control dependencies. Nodes are labeled with the number of the corresponding instruction from the assembler code for $VTBL_{AA}$. Each instruction is annotated with the cycle in which it starts the phase of its execution that produces its result. All instructions in the same row start executing in the same cycle (i.e. they execute in parallel if they are not part of mutually exclusive branches).

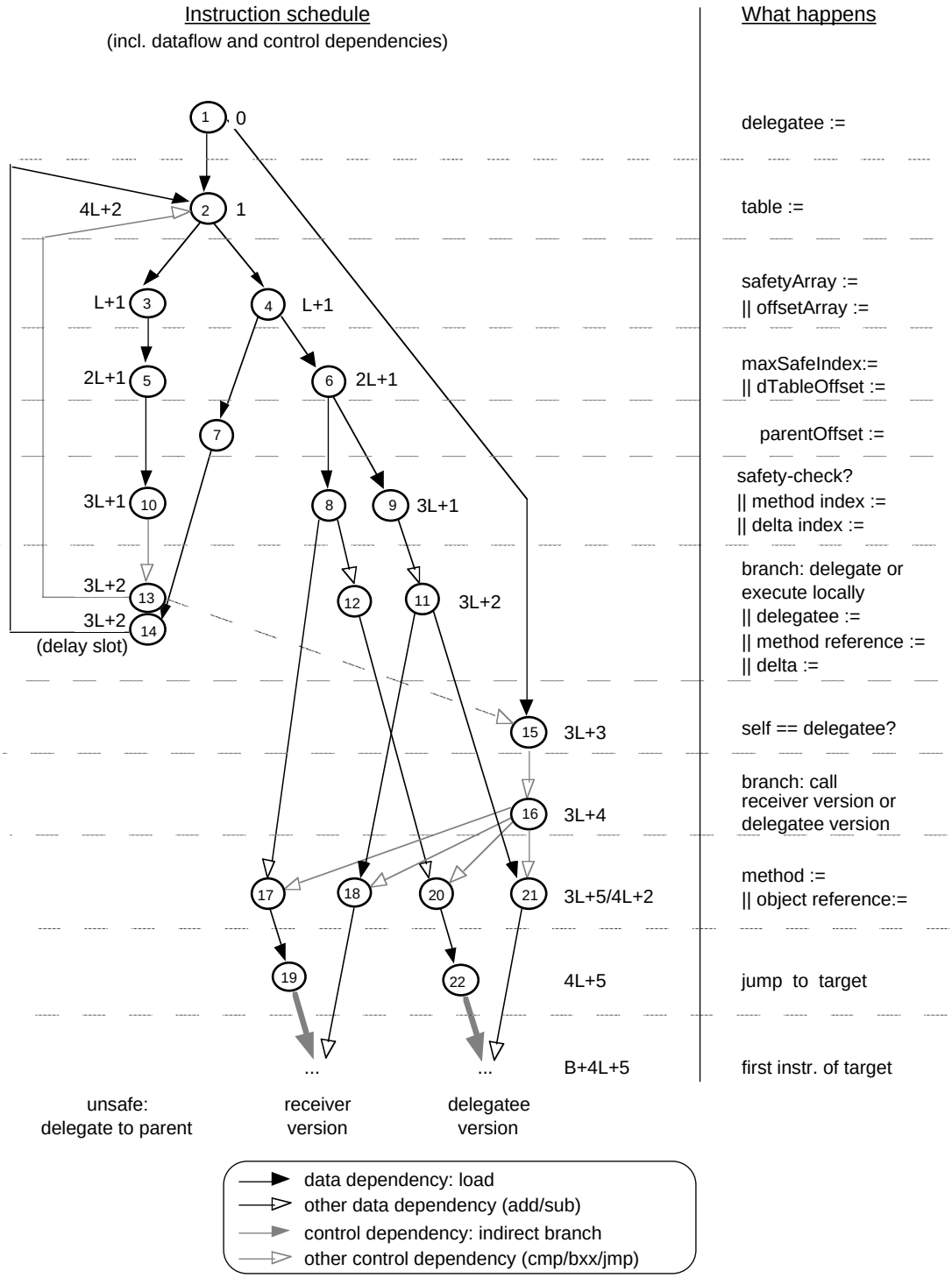


Fig. A1 : Schedule of $VTBL_{AA}$ on *P97* processors
(body of safety check loop eliminated using delay slot,
part of offset array access and final binding scheduled in parallel to the safety check)