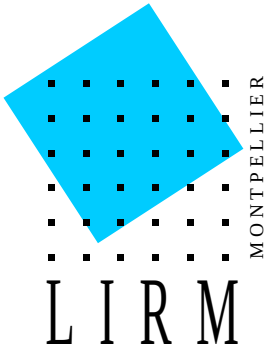
Implementing Statically Typed Object-Oriented Programming Languages

Roland Ducournau

pdfauthor



Laboratoire d'Informatique,
de Robotique et de Microélectronique
de Montpellier

Unité Mixte CNRS – Université Montpellier II
C 55060

Rapport de Recherche

Implementing Statically Typed Object-Oriented Programming Languages

Roland Ducournau

5 novembre 2002 R.R.LIRMM 2002–174

161, Rue Ada – 34392 Montpellier Cedex 5 – France
Tél: (33) 04 67 41 85 85 – Fax: (33) 04 67 41 85 00

Implementing Statically Typed Object-Oriented Programming Languages

ROLAND DUCOURNAU
LIRMM – CNRS et Université Montpellier II, France

In statically typed object-oriented languages, message sending, or late binding, is implemented with tables often called *virtual functions tables* (VFT). Those tables reduce method calls to function calls, with a small fixed number of extra indirections. The overhead is more or less important, whether inheritance and subtyping are single or multiple.

In this paper, we survey the various implementation schemes available in separate compilation, in the three cases of single inheritance, multiple inheritance, and single inheritance but multiple subtyping, e.g. JAVA. Many works have been done recently in the framework of global compilation, mostly for dynamically typed languages but also applied to the EIFFEL language in the SMALL EIFFEL compiler. We examine how global compilation can improve the implementation and propose some way to use global techniques—coloring and type analysis—in a separate compilation framework.

Categories and Subject Descriptors: D.3.2 [**Programming languages**]: Language classifications—*object-oriented languages*; C++; JAVA; EIFFEL; THETA; D.3.2 [**Programming languages**]: Processors—*compilers*; *memory management*

General Terms: Languages, Performance, Theory

Additional Key Words and Phrases: casting, coloring, contravariance, covariance, genericity, late binding, method dispatch, object-oriented languages, static typing, type analysis, separate compilation, virtual function tables

1. INTRODUCTION

In separate compilation of statically typed object-oriented languages, method calls—often named message sending or late binding—are generally implemented with tables called *virtual function tables* (VFT) in the C++ jargon. Method calls are then reduced to function calls with a small fixed number of extra indirections. An object is laid out as an attribute table, with a header pointing to the class table and some added information, e.g. for garbage collection. The cost of such an implementation depends on inheritance: with single inheritance, the overhead is reasonably small, but it is larger in the case of multiple inheritance. This paper aims to describe the various schemes generally used together with some alternatives, and to evaluate and compare them.

1.1 Object-oriented mechanisms

Two primary mechanisms are at the focus of implementation: read and write accesses to attributes and message sending, i.e. selection (also called dispatch) and call of the method corresponding to the receiver's dynamic type. A number of secondary mechanisms must also be considered. They are more or less explicit in the language design, but are all necessary and not so simple as one could expect:

Author's e-mail: ducournau@lirmm.fr

—inclusion polymorphism—i.e. the fact that an entity of a given static type may be bound to a value of a dynamic subtype—may need special attention as an object reference, e.g. `self`[1], may depends on its static type;

—attributes and methods overriding may be type invariant or not and, in the latter case, it can be type safe or unsafe (which is known as the covariance-contravariance problem);

—dynamic type checking is a need for constructs like *downcast* or `typecase`, as for unsafe overriding;

—class attributes, shared by all instances of a class, but reachable from the instance dynamic type in case of overriding, unlike `static` variables in C++ and JAVA;

—call to `super`, a way for the overriding method to call the overridden one;

—`null` value, for uninitialized variables and attributes;

—parameterized classes, in a bounded genericity framework.

Only method selection on a single receiver will be considered here: multiple selection, as it is realized in CLOS [Steele 1990] and theorized by Castagna [1997], can use techniques connected to single selection, but the combinatory in the cartesian product of types may be explosive.

We shall no more consider the various mechanisms which, under various names as encapsulation, protection, visibility or export, rule the access rights of some entity by the other ones: they actually are only access rights to existing implementation. Anyway, an exception will be done for SMALLTALK encapsulation which reserves for `self` all accesses to attributes [Goldberg and Robson 1983]: this will reveal of interest for implementation.

Primitive types issue will be also evaded, particularly concerning the question of calling a method on a receiver statically typed by the *universal type*, common super-type of primitive types and object classes. Such a universal type, `any` in EIFFEL, does not exist neither in C++, nor in JAVA. We consider also that a value is either an immediate value of a primitive type, or the address of an object instance of some class. Thus we exclude the fact that an attribute value could be the object itself, as in C++ or in EIFFEL with the keyword `expanded`: indeed, objects as immediate values forbids polymorphism for attributes.

As for types and subtyping, we shall keep a common point of view: type safety is assumed but we will also consider the effect of a covariant policy on implementation [Castagna 1995; Meyer 1997; Ducournau 2002b]. Static type checking, at compile-time, is out of the scope of this paper: only dynamic type checking will be examined.

1.2 Evaluating efficiency

There are two basic criteria for efficiency: time and space. Time efficiency can be judged on average but the ideal thing is constant time mechanisms. Space efficiency is evaluated by the complete memory needed for run-time. Space and time efficiencies vary in opposite directions: a single criterion is impossible and a compromise is always needed. Eventually, run-time efficiency is the main goal

[1] `Self` is the reserved word used in SMALLTALK to name the message receiver: it corresponds to `this` in C++ and JAVA and `current` in EIFFEL. An occurrence of `self` in a method of a given class has this class as its static type.

but compile-time efficiency must not be forgotten: attention should be payed to NP-hard optimizations.

1.2.1 *Space efficiency.* Three kinds of memory ressources must be considered. The dynamic part consists of the objects themselves, implemented as an attribute table, with one or more pointers to class tables: the use of a *garbage collector* must be envisaged for this part. The static part consists of the data structures associated to classes, which are read only and can be allocated in the code area, together with the code itself, where a given mechanism is handled by some number of instructions.

A static memory analysis has been done in the framework of global techniques which need to compact large tables [Driesen et al. 1995; Ducournau 1997], but it has not been done for separate compilation and static typing. As for dynamic memory, commonly used techniques may have a large overhead. First, some langages like JAVA may implement some mechanism through a constant dynamic overhead [Bacon et al. 2002]. Second, dynamic space may be sacrified for time efficiency, by putting part of static tables in dynamic tables: therefore dynamic overhead may need special optimizations [Eckel and Gil 2000].

1.2.2 *Time efficiency and processor architecture.* With classic processors, a measure of the time efficiency of a mechanism is the number of machine instructions needed for it. Modern processors make this measure obsolete since they have a pipeline architecture together with capabilities for parallelism (super-scalar processors) and branch prediction. In counterpart, memory access and unpredicted branch cause a multi-cycle latency. Thus the time spent for one instruction is no more one cycle. Moreover, composing two mechanisms may follow a law of the maximum instead of the sum. The instruction number remains only a space measure.

Implementing method calls with direct access in method tables has been for long considered as optimal: the effective overhead w.r.t. function calls seemed unavoidable. However, branching prediction of modern processors seems to have better performance with the technique, known as *inline cache* or *type prediction*, which consists in comparing the receiver's effective type with an expected type, the method of which is statically known. Such a test, statistically well predicted for whole programs, makes this technique very efficient [Driesen et al. 1995; Zendra et al. 1997]. Table-based techniques could be considered out of date. Nevertheless, two arguments are opposite to this thesis. Branching prediction favorizes type prediction as the former is restricted to conditional branching: but it could be extended, in future processors, to indirect branching, putting both techniques on an equal step [Driesen 1999]. Moreover, type prediction is not adapted to separate compilation.

We base our evaluation of time efficiency on an intuitive pseudo-code, mostly borrowed to [Driesen and Hölzle 1995; Driesen et al. 1995; Driesen 1999]. Each code sequence will be measured by an estimate of the number of cycles, parameterized by memory latency—L whose value is 2 or 3—and branching latency—B whose value may run from 3 to 15. More details can be found in the works by K. Driesen.

1.3 Notations and conventions

Uppercase letters will denote classes or types, according to the context. The class specialization relationship is denoted by $\prec_d$: $B \prec_d A$ means that B is a direct subclass of A. One assumes that $\prec_d$ has no transitive edges and its transitive (resp. and

reflexive) closure is denoted by $\prec$ (resp. $\preceq$): the latter is a partial order. Subtyping is noted $<:$, this is a pre-order. We consider that classes are types and that class specialization is subtyping (i.e. $\preceq$ is a subset of $<:$): even if type theory distinguishes both relationships, this is a common simplification in most languages. τ_s and τ_d, where $\tau_d <: \tau_s$, denote respectively the static and dynamic types of an entity: the static type is an annotation in the program text, whereas the dynamic type is the class instanciating the value currently bound to the entity. At last, a *root* is a class (resp. type) without superclass (resp. supertype). As for properties, i.e. methods and attributes, we adopt the following vocabulary: a class *has* or *knows* a property if the property is *defined* in the class or in one of its super-classes. A class *introduces* a property when the property is defined in the class, not in its superclasses. We assume that all ambiguities caused by *static overloading* (à la C++ or JAVA) have been solved by renaming [Meyer 2001; Ducournau 2002b]. Thus, the notation for a method (e.g. m) will denote both the method name and the *generic property*, invariant by inheritance and overriding, that this name unambiguously denotes[2]. Moreover, we are not concerned here by the inheritance problem itself, especially when multiple—i.e. which is the method inherited by some dynamic type?—but only by an efficient way of calling the appropriate method, whatever it is. Therefore, we will use the terms attribute and method to denote generic properties and, in some rare cases, we will qualify them with *definition* to denote the occurrence of the generic property in some class.

1.4 Context and plan

This paper aims to survey implementation techniques, whether they are effectively implemented in some language, described in the literature, or merely imagined as a point in the state space of possible implementations. A difficulty arises as most effective implementations are not described in the literature, either because they are not assumed to be novel, or for confidentiality reasons, because they are. Conversely, many techniques are theoretically described in the literature, without any known implementation. Language specifications could help to determine if a technique is adapted to some language, but currently, complete specifications are not always implemented. Thus, the schemes that we describe are more likely than real, but principles should not be too far. Moreover, it cannot be excluded that some techniques described here are novel, though it is not the paper goal.

Language implementation depends also on the way a program runs. Is it interpreted or compiled? Is compilation separate or global? Is linking static or dynamic? Is loading incremental? Our main objective is separate compilation, as it is common in most of the considered languages, but we will examine the effect of this choice on the possible techniques.

Structure of the paper. One can classify statically typed languages by their inheritance and subtyping relationships, according to whether they are single or multiple. Moreover, when inheritance is multiple, it may be arborescent, i.e. the superclasses

[2] This notion of *generic property* is missing in most languages: the term was coined on the model of CLOS *generic function*. In the literature on method dispatch, the SMALLTALK term of *method selector* is often used, whereas Zibin and Gil [2002] use *method family*. No term applies to attributes.

Table I. Four kinds of languages

inheritance	subtyping: single		subtyping: multiple	
single	Simula, C++ (si)	SST	MST	Java, Theta, Eiffel#, C#
arborescent	—		AI	C++ (non virtual)
multiple	—		MI	C++ (virtual), Eiffel

of every class form a tree. Each of the four resulting categories uses different techniques (Table I). The next two sections present the standard implementation principles in the two extreme cases: single subtyping (SST) then multiple inheritance (MI). Various notions are precisely introduced on the way. In section 4, we examine some alternatives for multiple inheritance, among which C++ "non virtual" implementation (NVI) for arborescent inheritance (AI). The next section describes the median case of single inheritance (SI) but multiple subtyping (MST), illustrated by Java and DotNet languages. Some applications to multiple inheritance and the special case of *mixins* are examined. Section 6 presents some complementary mechanisms not treated up to there: genericity, type variant overriding, class attributes. Whereas the previous sections consider the case of separate compilation, section 7 proposes a short survey of the techniques used in global compilation and dynamic typing. Two global techniques, coloring and type analysis, are sounded as they seem good candidates for use at link-time in a separate compilation framework. The article ends with a conclusion and some perspectives. Appendix presents space statistics on common benchmarks and the pseudo-code for all the presented techniques.

2. SINGLE INHERITANCE AND SUBTYPING (SST)

2.1 Principle

With single inheritance and subtyping, types may be identified with classes. Each class has at most one superclass. Thus the subclass tables are simply obtained by adding newly introduced methods and attributes at the end of the superclass tables (Figure 1). Two invariants characterize this implementation:

Invariant 2.1.1. *A reference—method parameter, local variable, attribute or returned value—on an object is invariant w.r.t. the reference's static type.*

Invariant 2.1.2. *Each attribute or method p has a unique offset, noted δ_p, unambiguous and invariant by inheritance and w.r.t. the receiver's static type.*

Thus, standard SST implementation is characterized by an absolute invariance w.r.t. static types. This enhances the basic semantics of object orientation, which states that dynamic type is the object's essence and that static types are pure contingency. Moreover, it realizes the ideal type erasure of type theory.

Method calls are then compiled by a sequence of three instructions:

```
load [object + #tableOffset], table
load [table + #selectorOffset], method                    2L + B
call method
```

and attribute accesses are as immediate as for a record field:

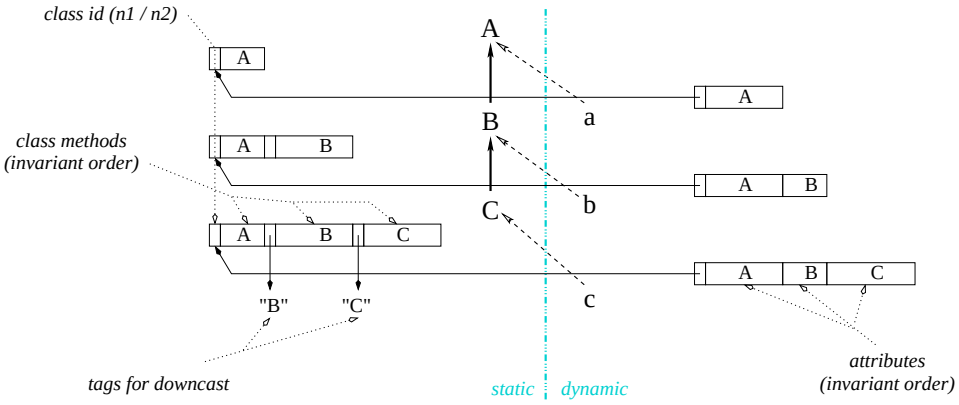


Fig. 1. Object layout and method tables in single subtyping: 3 classes A, B et C with their respective instances, a, b and c.

```
load [object + #attributeOffset], attribute                    L
```

Notice that attribute invariance is due to the fact that attributes have always fixed size values: when an attribute type has variable size values, the attribute is a pointer to those values.

Computing the tables, i.e. method and attribute offsets, is a special algorithmic case of the coloring heuristics (see section 7.4). Single inheritance and static typing, which avoids *definition overloading*[3], ensure a sound and optimal result: no offset conflict can occur between inherited properties and there is no need to check that the next free offset is really free.

Though pure SST languages are not common, this implementation is the basis of most implementation, as well JAVA without interfaces (see section 5) as C++ when restricted to single and "non virtual" inheritance (see section 4.1).

2.2 Casting

2.2.1 *Principle.* The word *cast* is commonly used to term various mechanisms close to type coercion, from a source type to a target type. Among the various interpretations, two are of interest for object-orientation, as they concern the relationship between static and dynamic types: source and target types are then related by subtyping.

Upcast is often called *implicit* because it needs no particular syntax. It simply consists in a polymorphic assignment (or parameter passing) $x := y$, when the static types of x and y are respectively X and Y and when Y is a proper subtype of X ($Y <: X$). Such a mechanism should have no name as it is conceptually empty—this is pure inclusion polymorphism—but its implementation may be non trivial.

Downcast amounts to assume that an entity of a given static type $\tau_s = X$ is actually an instance of a subtype Y, i.e. $\tau_d <: Y$ and, usually, $Y <: X$. This is a type

[3] We name here *definition overloading* the fact that a property could be defined in two unrelated (by specialization) classes, without being defined in any common superclass: with static typing the two occurrences are no more related than with *static overloading* but this is no more the case with dynamic typing, e.g. in SMALLTALK.

unsafe assumption, which cannot be statically checked (without loss of generality): it thus needs a dynamic type check which may signal an exception if it turns out that the assumption was false. *Downcast* is realized by special syntactic constructs as `dynamic_cast` in C++, parenthesized syntax in JAVA (not to use in C++), `typecase` in THETA [Liskov et al. 1995] or *assignment attempts* in EIFFEL [Meyer 1992; 1997]. *Downcast* uses are often justified by the fact that covariant models are implemented in type safe, i.e. contravariant or invariant, languages (see section 6.1). They are also common in JAVA because of its lack of genericity.

Besides those two opposite directions, casting may also be *static* or *dynamic*[4], according to whether the target type is statically known or not. Explicit syntactic constructs are always static casts: the target type is a constant of the construct. However, we will see that some mechanisms may need dynamic casts, which impose that the target type is reachable from the considered object.

2.2.2 *Casting in single subtyping.* As references to objects don't depend on static types, *upcast* has no more reality in the implementation than as a concept.

Downcast. Due to reference invariance, this amounts to dynamic type checking. There are two classic ways to implement type checking with SST. The first one consists in assigning an offset to each class in the method tables of its subclasses: the corresponding entry of the tables must contain the class identifier. An object is an instance of a class C iff the object's method table, noted tab_{τ_d}, contains at the offset δ_C the identifier id_C:

$$\tau_d \preceq C \Leftrightarrow tab_{\tau_d}[\delta_C] = id_C \tag{1}$$

Class offsets are ruled by the same Invariant 2.1.2 as methods and attributes. This first technique is simple and works in separate compilation: it is time efficient, but not space optimal. This is again a special case of coloring (see section 7.4), described by Cohen [1991] and reused by Queinnec [1997].

The second technique is optimal but has a twofold drawback: it does not generalize to multiple inheritance, at least in constant time, and it is not incremental, thus incompatible with dynamic loading. It is a double class numbering, noted n_1 and n_2: n_1 is a pre-order depth-first numbering of the inheritance tree and n_2 is defined by $n_2(C) = \max_{D \preceq C}(n_1(D))$. Then:

$$\tau_d \prec C \Leftrightarrow n_1(C) < n_1(\tau_d) \leq n_2(C) \tag{2}$$

This technique, due to Schubert et al. [1983], is often called *Schubert's numbering* or *relative numbering*. Only two short integers are needed and the first one (n_1) can serve as class identifier. For the test (2), $n_1(\tau_d)$ is dynamic, whereas $n_1(C)$ and $n_2(C)$ are static when the cast is static: they may be compiled as constants. Both techniques have the same time efficiency ($2L + 2$ cycles). The latter has a better memory cost in the static tables (worst-case linear instead of quadratic) but the former has a more compact code (4 instructions instead of 6). Type checking remains an active topic of research, even in single subtyping [Raynaud and Thierry 2001; Zibin and Gil 2001], but no other technique has such a power of simplicity.

[4] Those terms are unrelated with the C++ keywords `static_cast` and `dynamic_cast` which are both *static* (see section 4.1).

2.3 Evaluation

Time efficiency is optimal as everything is done with at most a single indirection in a table. Even downcasts are constant-time. Dynamic space efficiency is also optimal: object layout is akin to record layout, with the only overhead of a single pointer to class method table. Method tables depend only on object dynamic types. As a whole, they occupy a place equal to the number of valid class-method pairs, which is the optimal compactness of the class-method tables used by constant-time techniques in dynamic typing, multiple inheritance and global compilation (see section 7). Let M_C denote the number of methods known (defined or inherited) by a class C: then the method tables have $\Sigma_C M_C$ entries.

This optimal implementation of SST is the reference which serves to measure the overhead of multiple inheritance or subtyping, for both time and space efficiency.

3. MULTIPLE INHERITANCE (MI)

Multiple inheritance complicates the implementation to a considerable extent, as [Ellis and Stroustrup 1990, chapter 10] demonstrates it for C++. C++ is all the more complicated because it offers some unsound features which are aimed to reduce MI overhead. The keyword `virtual`, when used to annotate superclasses, is the way to obtain a sound MI semantics whereas what we will term *non virtual multiple inheritance* (NVI), when the keyword is not used, offers a cheaper implementation but degraded semantics, sound only for *arborescent inheritance* (AI) (see section 4.1). Therefore, in this section, we consider that, in C++, we would have to use `virtual` to annotate every superclass. Those introductory precautions are not necessary for a langage of sound constitution as Eiffel [Meyer 1992; 1997].

3.1 Principle

In separate compilation and MI, there is no way to assign a minimal and invariant offset to each method and attribute (Invariant 2.1.2) without causing future conflicts. Let B and C two unrelated classes occupying the same offsets: then, it will always be possible to define a common subclass to those two classes, say D (Figure 2). Two attributes (or methods) respectively of B and C, both present in D, will conflict for the same offset. Giving up Invariant 2.1.2 leads to do the same for Invariant 2.1.1, if one wants that method calls take only one indirection in a table. We will see further how to keep reference invariance, by giving up constant-time mechanisms (see section 5.4) or separate compilation (see section 7.4).

3.1.1 *Object layout and method tables.* Offset invariance is thus relaxed, in two different ways for attributes and methods.

Invariant 3.1.1. *Each attribute has an offset, noted δ_a, unambiguous and invariant, in the context only of the static type which* introduces *the attribute.*

Invariant 3.1.2. *Each method has an offset, noted $\delta_m^{\tau_s}$, unambiguous and invariant in the context of every static type which* knows *the method.*

The object consists of subobjects, one for each superclass of its class (i.e. for each static type supertype of its dynamic type) and each subobject is equipped with its own method tables (Figure 2). Reference invariance is then replaced by:

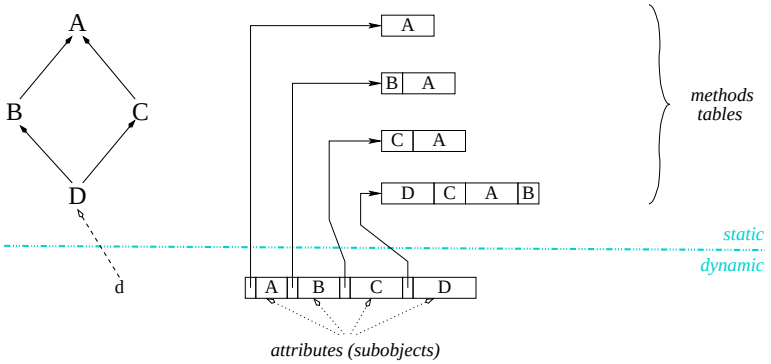


Fig. 2. Object layout and method tables with multiple inheritance

INVARIANT 3.1.3. *Any entity whose static type is T is bound to the subobject corresponding to T. Subobjects of different static types are distinct.*

A subobject consists only of the attributes introduced by the corresponding static type, whereas the method table contains all the methods known by it, with values (addresses) corresponding to methods inherited by the dynamic type. Thus, two proper instances of different classes will not share any method table of their common superclasses: thoses tables are isomorphic but don't contain the same addresses (Table II). For a given static type, method offsets don't matter, but it is reasonable to group methods by introduction classes, as in the figures: this may offer some local invariance (in case of SI) but this organization has no effect on efficiency.

3.1.2 *Method call.* Invariant 3.1.3 imposes to recompute the value of `self` for each method call where the receiver's static type τ_s is different from the class W, superclass of τ_d, which defined the selected method. One needs to know the position of subobject W, w.r.t. subobject τ_d, noted[5] $\Delta_{\tau_s,W}$ (Figure 3). Thus method tables have double entries for each method, an address and a shift (Figure 6 on top). On the whole, method calls are compiled by the sequence[6]:

```
load [object + #tableOffset], table
load [table + #deltaOffset], delta
load [table + #selectorOffset], method                    2L + B + 1
add object, delta, object
call method
```

Instead of putting shifts in the tables, an alternative consists in defining a small intermediate function, called *thunk*[7] by Ellis and Stroustrup [1990] or *trampoline* by Myers [1995], which shifts the receiver before calling the method:

```
add object, #delta, object
jump #method
```

[5] The notation $\Delta_{T,U}$, as all the following notations Δ, implies a given dynamic type $\tau_d \preceq T, U$: expliciting it would make the notation heavy.
[6] Instructions added w.r.t. SST are italicized (see page 5).
[7] According to Lippman [1996], *thunk* is *Knuth* spelled backwards.

Table II. Method tables for Figure 2 example, w.r.t. static and dynamic types. For a same static type (column), the tables are isomorphic but differ by their contents (method addresses), whereas, for a same dynamic type (row), isomorphic pieces contain the same addresses but different shifts.

type static → / ↓ dynamic	A	B	C	D
A	A	—	—	—
B	A	A B	—	—
C	A	—	A C	—
D	A	A B	A C	A B C D

The call sequence is then the same as with SST. Thunks may be shared when the address and the shift are the same: they may be considered as different entry points to the same procedure and the thunk with the null shift is the method itself. Thus, one saves on one access in the table for an extra direct branching. The *thunk* could also be inlined in the method table instead of being pointed by it: one could save on L cycles, but one would lose the null cost of the null shift and the ability to share thunks with the same shifts.

3.2 Casting

Subobjects make casting real: [Rossie et al. 1996] defines it as a subobject change. Notice first two basic properties of Δ:

$$\forall \tau_d, T, U, V : \qquad \tau_d \preceq T, U, V \Rightarrow \Delta_{T,V} = \Delta_{T,U} + \Delta_{U,V} \tag{3}$$

$$\forall \tau_d, T : \qquad \tau_d \preceq T \Rightarrow \Delta_{T,T} = 0 \tag{4}$$

3.2.1 *Cast to dynamic type.* Because of the reference variance w.r.t. static types, equality tests between two references need to equal first their types. Without loss of generality, both references must be reduced to their dynamic type: each method table must contain a shift Δ_{τ_s,τ_d}, noted $\Delta^{\tau_s}_{\Downarrow}$. When the two types are in a subtyping relation, one upcast will be enough.

3.2.2 *Upcast.* Changing from the current subobject, with type τ_s, to a statically known supertype T, needs a shift $\Delta_{\tau_s,T}$, which depends on τ_d. An extra table, noted $\Delta^{\uparrow}_{\tau_s}$, is needed in every method table: T offset in $\Delta^{\uparrow}_{\tau_s}$ is invariant w.r.t. dynamic type and statically known:

Invariant 3.2.1. *Each class has an unambiguous and invariant offset in the static context of each of its subclasses.*

This offset is noted $i_{\tau_s}(T)$ and it is, unlike Δs, independant of τ_d: then $\Delta_{\tau_s,T} = \Delta^{\uparrow}_{\tau_s}[i_{\tau_s}(T)]$, shortened in $\Delta^{\uparrow}_{\tau_s}(T)$. Instead of being a proper table, $\Delta^{\uparrow}_{\tau_s}$ can lie within method table. Indeed, upcasts can be handled as if every target class introduces a method for upcast towards itself: but it is more efficient to put a shift instead of an address in the method table entry.

3.2.3 *Accesses to attributes.* The table $\Delta^{\uparrow}_{\tau_s}$ is also used for accessing attributes when they are introduced in a superclass of τ_s. Let $\delta(p, U)$ be the position of an attribut p w.r.t. the subobject of type U, and δ_p the offset of p in the type T_p ($\tau_s \prec T_p$) that introduces it (Invariant 3.1.1). Then:

$$\delta(p, \tau_s) = \Delta_{\tau_s,T_p} + \delta(p, T_p) = \Delta^{\uparrow}_{\tau_s}(T_p) + \delta_p \tag{5}$$

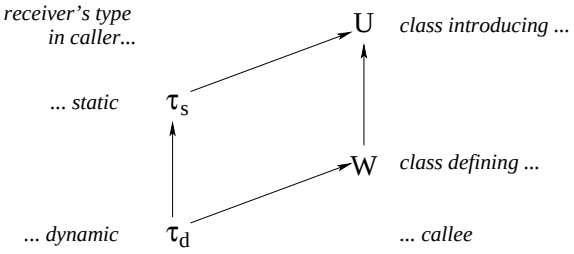


Fig. 3. Receiver's types in a method call: a shift $\Delta_{\tau_s,W}$ is needed

When an attribute is not introduced in the receiver's static type ($T_p \neq \tau_s$), there is an appreciable overhead in comparison with SST:

```
load [object + #tableOffset], table
load [table + #castOffset], delta
add object, delta, object                                    3L + 1
load [object + #attributeOffset], attribute
```

In worst case, assignment $a.x := b.y$ needs three upcasts, on a, b and between y and x types. Some parallelism is likely but the sequence is 5 times longer than in SST, as well in cycle number ($5L+3$ vs. $L+1$) as in instruction number (11 vs. 2).

3.2.4 *Downcast.* Shifting from static type τ_s to a static subtype T needs both a type check and the value of $\Delta_{\tau_s,T}$. Contrary to SST, a direct access is difficult, at least in separate compilation (see section 7.4) and some sequential search may be better. Each class τ_d has an association structure, e.g. a hashtable, which maps all supertypes T of τ_d to $\Delta_{\tau_d,T}$: this structure, noted $\Delta^{\Uparrow}$, can be referenced by each method table, not only by τ_d. A downcast from τ_s to T looks up for T in the table. If T is not found, a type error is signaled. Otherwise, $\Delta_{\tau_d,T}$ is returned and:

$$\Delta_{\tau_s,T} = \Delta_{\tau_s,\tau_d} + \Delta_{\tau_d,T} = \Delta^{\tau_s}_{\Downarrow} + \Delta^{\Uparrow}(T) \tag{6}$$

It must be noticed that both tables $\Delta^{\uparrow}_{\tau_s}$ and $\Delta^{\Uparrow}$, have the same contents, with different structures and uses: in the former, τ_s is statically known, whereas in the latter, τ_d is not (Table III). Their contents is also the same as the shifts included in the method tables or *thunks*. One can get advantage from static types, by associating to each subobject a new table $\Delta^{\tau_s}_{\downarrow}$, the restriction of $\Delta^{\Uparrow}$ to the classes between τ_s and τ_d: $\Delta^{\tau_s}_{\downarrow} = \Delta^{\Uparrow}/[\tau_s, \tau_d] + \Delta^{\tau_s}_{\Downarrow}$. They avoid a two step downcast:

$$\Delta_{\tau_s,T} = \Delta^{\tau_s}_{\downarrow}(T) \tag{7}$$

but the temporal advantage is small and at the cost of a memory overhead. With $\Delta^{\Uparrow}$, downcasts may be generalized to *side casts* (also called cross casts), where the target type T is a supertype of τ_d, but not always a subtype of τ_s (Figure 4, right).

This type checking implementation is costly. Coding the specialization partial order is difficult in MI and it would not work since the problem is no more boolean: shifts are needed. A simple solution allowing a direct access is a $N \times N$ matrix, where N is the class number: $mat[id_T, id_U]$ contains $\Delta_{T,U}$ if $U <: T$, and otherwise a distinguished value meaning that there is a type error. Such a matrix needs $2N^2$ bytes, i.e. $2N$ bytes per class: this is a reasonable cost when there are $N = 100$

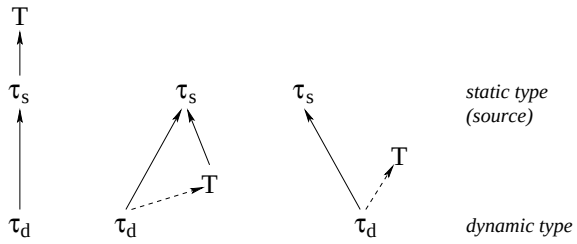


Fig. 4. Upcast (left), downcast (center) and side cast (right), to a target type T: the dashed subtype relation must be checked at run-time before shifting.

classes, but it is not when $N >> 1000$ (see appendix A). Class identifiers id_C must be computed globally for all classes C: an adequat numbering allows to replace the matrix by a triangular one, i.e. by one vector per class, using $vect_U[id_T]$ instead of $mat[id_T, id_U]$. This reduces the cost to an average of N bytes per class, which is the worst-case cost of the coloring scheme as well in SST (see section 2.2.2) as in MI (see section 7.4) but the average cost of coloring is far lesser.

3.3 Empty subobjects optimization (ESO)

On the basis of this uniform implementation, a simple optimization reduces the space overhead. Indeed, an exception to Invariant 3.1.3 is possible, when a subobject is empty, i.e. when the corresponding class, say F, introduces no attribute. It is then possible to merge the F subobject with the subobject of one of its direct superclasses, say E. Two cases are to distinguish.

In the first case, E is the only direct superclass of F and F introduces no method: the set of methods of E and F are the same. The F subobject may be merged into the E subobject as, without merging, the contents of both method tables, i.e. method addresses, would have been the same. Here, merging works because it is invariant w.r.t. dynamic type: E and F subobjects are merged in all subclasses of F. Multiple inheritance problems are avoided because F has no more methods than E: if another subclass F' of E, is in the same case, the subobjects E, F and F' can be merged in any subclass of both F and F'. In some way, this merging is a property of F: method tables are shared and shifts between F and E are avoided, as $\Delta_{F,E} = 0$ and $i_{\tau_s}(E) = i_{\tau_s}(F)$, for all $\tau_d \preceq \tau_s \preceq F \prec_d E$. The code generated for all $\tau_s \preceq F$ takes into account the merging of E and F: e.g. accessing an attribute introduced in E on a receiver typed by F will need no cast.

In the second case, F has either more methods than E, or more than one direct superclass—the latter condition implies the former, as superclasses imply upcast methods—, but the method order of E is a prefix of the method order of F, i.e. the offsets of E methods are the same in F. Therefore, the E and F subobjects may be merged in the implementation of $\tau_d \preceq F$, but at the only condition that E is not already merged with another F', in the same $\tau_d \preceq F'$. Once again, merging works in this restricted case because the prefix condition is also invariant w.r.t. dynamic type: however, E and F will not be merged in all subclasses of F. This means that merging is neither a property of E, nor of F, but only of some $\tau_d \preceq F$. In that case, merging allows to share method tables, not to save on shifts, neither in the code nor in upcast tables. The code generated for all $\tau_s \preceq F$ cannot consider

Table III. Cast tables for the class (τ_d) D of Figure 2: $\Delta_{\Downarrow}$ is a scalar, $\Delta^{\uparrow}$ a vector, whereas $\Delta_{\downarrow}$ and $\Delta^{\Uparrow}$ are hashtables.

τ_s	$\Delta_{\Downarrow}$	$\Delta^{\uparrow}$	$\Delta_{\downarrow}$	$\Delta^{\Uparrow}$
A	Δ_{AD}	$[0]$	$[(A,0)(B,\Delta_{AB})$ $(C,\Delta_{AC})(D,\Delta_{AD})]$	
B	Δ_{BD}	$[\Delta_{BA},0]$	$[(B,0)(D,\Delta_{DB})]$	
C	Δ_{CD}	$[\Delta_{CA},0]$	$[(C,0)(D,\Delta_{CD})]$	
D	0	$[\Delta_{DA},\Delta_{DB},\Delta_{DC},0]$	$[(D,0)]$	$[(A,\Delta_{DA})(B,\Delta_{DB})$ $(C,\Delta_{DC})(D,0)]$

that E and F are merged, but the data structures for some $\tau_d \preceq F$ may do that merging. In particular, accessing an attribute introduced in E, on a receiver typed by F, needs a cast, but the shift will be likely null.

It is always possible to construct the method order of any class, which is up to now arbitrary, as an extension of the method order of at least one of its superclasses. Therefore, almost all empty subobjects can be merged: the only exception will be when two different subobjects are merged into the same subobject, in the second case of merging. In that rather exceptional case, only one merging will be possible.

There is some evidence that empty subobjects are quite common and that this optimization is essential (see appendix A). However, there is also some evidence that this simple optimization, or at least its effect, has not been noticed, as most of benchmarks in the literature do not include any information on attribute number [Sweeney and Burke 1998; Gil and Sweeney 1999; Eckel and Gil 2000].

3.4 Evaluation

Multiple inheritance overhead is rather large.

Dynamic memory. In each object, the overhead is equal to the number of indirect superclasses of the object class, which introduce at least one attribute.

Static memory. The number of method tables is no more linear in the class number N, but worst-case quadratic whereas total size is no more quadratic, but cubic: $\Sigma_C(\Sigma_{C \preceq D} M_D)$. However, the formula must be corrected to take into account the empty subobjects optimization: Σ_D is restricted for those D which are not merged with some E, with $C \preceq D \prec_d E$. Precise statistics confirm both the overhead of this implementation and the benefits of empty subobjects. In a context of mainly "non virtual" inheritance, [Driesen 1999; Driesen and Hölzle 1995] report a ratio larger than 3 on the table sizes, w.r.t. the size of tables in SST ($\Sigma_C M_C$). Statistics in appendix show that the ratio may exceed 6 with standard implementation, but that it is reduced to 4 with ESO. When taking into account the shifts, in the tables or in the thunks, the ratio climbs again to 6. *Thunks* seem roughly equivalent to putting shifts in method tables: there are less *thunks* than tables entries, they need two words instead of one and the code sequence is two instructions shorter. Statistics show that they are more costly for large hierarchies (see appendix A).

Time efficiency. A shift is needed each time an assignment or a parameter passing is not with constant static types. It imposes an extra access to method table. The real impact on method calls is more questionable: shifting could be done within processor's latencies or in parallel. Experiments by [Driesen 1999] seem to give a small advantage to thunks. But this conclusion is based on benchmarks mainly

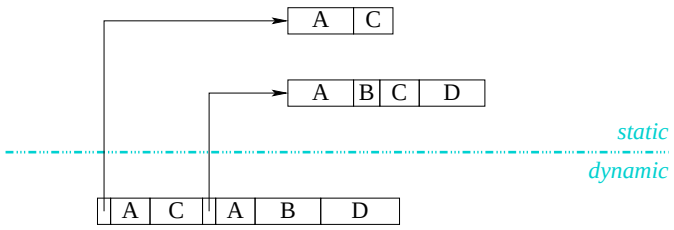


Fig. 5. Object layout and method tables in "non virtual" inheritance

constituted of classic C++ programs making an heavy usage of "non virtual" inheritance: in this particular case, shifts are mostly null and the thunk is the method itself. So, although it does not allow to conclude about plain MI, it seems that shifts have a sensible overhead. This is not surprising since, even if the cycle numbers are almost equal, this is at the price of some parallelism, and it is likely that shifts take the place of some other code sequence. Furthermore, constant-time downcast needs very large tables.

The main drawback of this implementation is that the overhead is the same when one does not use MI: separate compilation is unable to foresee that a given class will be always specialized in SI.

4. ALTERNATIVES FOR MULTIPLE INHERITANCE

The complexity of the previous implementation explains part of the reluctance to multiple inheritance as well as some unsound C++ features [Cargill 1991; Waldo 1991; Sakkinen 1992]. Various alternatives have been searched for: the most radical, proposed by the Small Eiffel compiler that we will examine later (see section 7.3.6), avoids method tables at the price of a global compilation [Zendra et al. 1997]. In this section, we examine some small variations around the standard implementation. They consist either in searching a MI implementation without overhead when used only in SI (section 4.1), or various compromises between time and space efficiencies (sections 4.2 and 4.3). Eventually, attribute implementation can be reduced to methods (see section 4.4).

4.1 C++ non virtual inheritance (NVI)

As previously noticed, the main drawback of standard MI implementation is that its overhead does not depend on an effective use of MI. C++ "non virtual" inheritance is an answer to this problem. It is expressed by not annotating superclasses with the keyword `virtual`. This gives a more efficient implementation but it is sound only when the superclasses of any class form a tree, i.e. for what we call arborescent inheritance (AI). Mixing "virtual" and non virtual inheritance may be quite complicated and it worsens with inheritance protections. Thus we will only describe pure non virtual inheritance.

NVI is better described by its implementation than by its semantics [Ellis and Stroustrup 1990]. A class without superclass is implemented as in SST. When a class has one or more direct superclasses, the instance layout consists of the concatenation of the instance layouts of all direct superclasses: the attributes and methods introduced by the class are added to one of the subobjects. Extending the

Table IV. Method tables, with sharing or non virtual inheritance: a class means a reference to the table associated to it as a static type, on the same line.

type static → ↓ dynamic	A	B	C	D
A	[A]	—	—	—
B	B	[A | B]	—	—
C	C	—	[A | C]	—
D	C/D	D	[A | C]	[A | B | C | D]

last subobject allows to make $\Delta_{T,U}$ invariant w.r.t. τ_d[8]. Thus, the main difference is that there is no proper subobject for the class itself. However, the generated code is the same as with standard MI implementation. The only difference is that the need for shifts is less frequent for upcasts and attributes: only when source and target, or τ_s and T_p, don't share the same subobject. As for method calls, the shifts are always needed but they are more often null: thunks make most of them disappear in practice. In case of SI, the layout is exactly the same as with SST, with only one subobject and there is no shift at all with thunks. In the general case of arborescent inheritance, the number of subobjects or method tables is equal to the number of superclasses which are roots, i.e. without superclass.

4.1.1 *Arborescent vs. repeated inheritance.* The flaw of non virtual inheritance occurs when inheritance is no more arborescent, i.e. when the inheritance graph contains undirected cycles, as in Figure 2. It consists in what we call *repeated inheritance*, in the sense that some subobject is repeated in the object layout: in Figure 5, this is the case for A which is present in both subobjects of B and C. In case of repeated inheritance, the table number of NVI implementation becomes exponential in the number of superclasses, as it is the number of paths from the considered class to its root superclasses.

4.1.2 *Casting.* In case of arborescent inheritance, and at the condition that the current class data structures extend the structures of the last subobject in the concatenation, the relative positions of two subobjects whose types are related by subtyping don't depend on the dynamic type. Thus upcast is unambiguous and, when it is needed, the shift is static, without accessing the table. Downcast can also be made in a very close way to the SST coloring scheme, as $\Delta_{\tau_s,T}$ is independant of τ_d. One needs only that each subobject implements the coloring scheme reduced to the types corresponding to the subobjects. Only side casts need the MI technique.

On the other hand, repeated inheritance makes casting ambiguous. For upcasts, the ambiguity is on the target and intermediate upcasts may be needed to remove it. As for downcasts, the ambiguity is on the source and checks must be repeated as many times as the source is. Side and dynamic casts are not always possible.

Mixing virtual and non virtual inheritance makes the problem even more complicated. [Ellis and Stroustrup 1990, section 10.6c] forbids some cases. However, according to the specifications of `dynamic_cast` [Koenig 1998], the effective implementation seems to use both $\Delta_{\downarrow}^{\tau_s}$ (in case of repeated inheritance) and $\Delta^{\Uparrow}$ (for

[8] The point is not explicitly noticed, but it seems to be a common assumption [Ellis and Stroustrup 1990; Sweeney and Burke 1998].

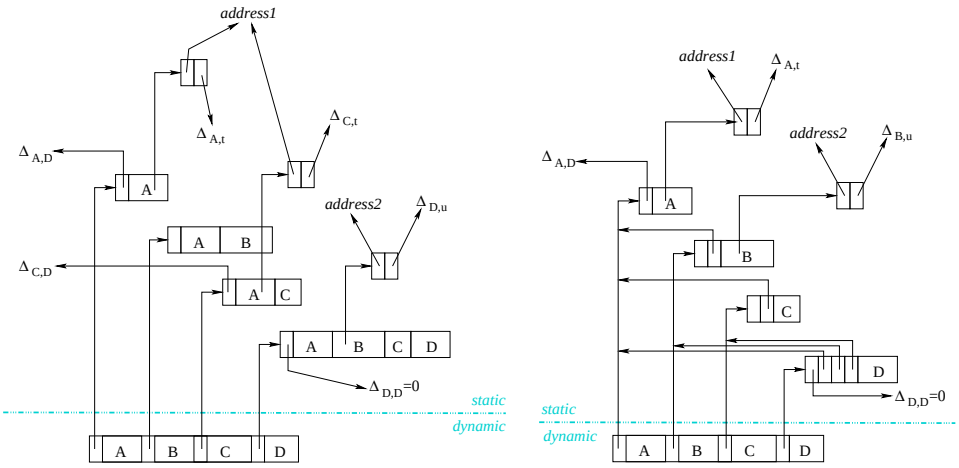


Fig. 6. Standard and compact method tables

sidecast). If the target is unambiguous in one of the two tables, the cast succeeds: no static prohibition is necessary.

4.1.3 *Evaluation.* Non virtual inheritance has the great advantage of presenting an overhead only when one uses MI (as long as thunks are used). But repeated inheritance is a major semantic drawback. A sound semantics is possible by mixing virtual and non virtual inheritance, but it is a matter of either hand made optimization, or global analysis. Separate compilation cannot predict that B and C will not have a common subclass and which one must use the `virtual` keyword. Only a subsequent diagnosis, when defining D, is possible, or an automatic devirtualization analysis on the whole program (see section 7.3.2). It could be possible to restrict the use of NVI to AI: defining D would be forbidden. But it would be a limitation to reusability. Moreover, when the class hierarchy has a root, `Object`, the class of all objects, AI reduces to SI.

Besides its semantic flaw, NVI is not less costly than standard MI implementation, in all generality: NVI complexity is indeed exponential in the worst-case (see appendix A). Furthermore, empty subobjects optimization does not apply.

4.1.4 *Criticizing* C++. NVI comes from Simula [Krogdahl 1985]. Multiple inheritance in C++ has been commented and criticized [Cargill 1991; Waldo 1991; Sakkinen 1992], often for opposite reasons. [Cargill 1991] criticizes "virtual" inheritance and it is significant that Waldo's answer [1991] is based on an example which needs rather MST than plain MI. The main criticism may be that understanding the language goes through understanding the implementation. NVI gave rise to some attempts to base its semantics on models of subobjects [Rossie et al. 1996; Snyder 1991].

4.2 Compact method tables

Another drawback of standard MI implementation is its static space overhead: table size is cubic instead of quadratic in SST. Two obstacles forbid to share the tables

of different static types for the same dynamic type: shifts and variant method offsets. The first point can be reduced by a two step shifting, as for attributes (5) and downcast (6). The second point is solved by applying to methods Invariant 3.1.1, which rules attributes. Therefore, the method table associated to a static type contains only the methods introduced by the type, not inherited methods. A method call must begin with an upcast towards the type which introduces the method, as with attributes (5), at least when $T_p \neq \tau_s$. The cost of this technique is exactly the sum of the costs of upcasts and method calls: thus, the call sequence has 8 instructions and $4L + B + 2$ cycles. Instead of getting the second table from the cast object, one can use pointers between the different method tables (Figure 6, right). This allows some parallelism and reduces the cost to $3L + B + 1$ cycles:

```
load [object + #tableOffset], table1
load [table1 + #table2Offset], table2
load [table1 + #castOffset], delta1                    3L + B + 1
load [table2 + #deltaOffset], delta2
add object, delta1, object
load [table2 + #selectorOffset], method
add object, delta2, object
call method
```

In that case, the two step shift may indifferently go through either τ_d (6) or T_p (5). A minor overhead is that upcast structures are implemented twice, as shifts between subobjects and as pointers to method tables.

Evaluation. This technique reduces static space overhead to the only cost of doubling the upcast tables size, together with 3 extra instructions for each method call such that $\tau_s \neq T_p$. The objective of this compact implementation is to reduce table size, i.e. static memory, but it may be hindered by this sensible augmentation of the code. Only the first case of empty subobject optimization works: merging is possible when both the subobject and its method table are empty. Moreover, time efficiency is reduced: L extra cycles underestimates the effective extra overhead. Anyway, it was worth-while to convince oneself that standard implementation offers, in some way, the best compromise between time and static memory. Various other compact schemes exist, but no one is better than this simple one.

4.3 Less indirections

Contrarily to previous variants which try to reduce the static memory cost, some optimizations aim to reduce time overhead, at the price of dynamic memory. The best way to do it is to move the upcast tables $\Delta^{\uparrow}_{\tau_s}$ from the method tables to the objects themselves. In a second step, shifts common to all instances may be replaced by pointers to subobjects, proper to each instance. In the C++ jargon [Ellis and Stroustrup 1990], they are called *virtual base pointers* (VBPTR) and, according to [Sweeney and Burke 1998], this technique is used in several C++ compiler. As a matter of fact, upcasts are very frequent, as well for accessing attributes as for polymorphic assignments. They are viewed by standard implementation as methods, which impose a method table access (see section 3.2.2, page 10). VBPTRs amount to view upcasts as attributes, but they are implemented in each subobject:

```
load [object + #castOffset], object                    2L
```

```
load [object + #attributeOffset], attribute
```

There are two kinds of VBPTRs: e-VBPTRs are *essential* because they reference direct superclasses, whereas i-VBPTRs are *inessential* and can be replaced by some indirections using e-VBPTRs. Implementing only e-VBPTRs in the objects would not improve much on $\Delta^{\uparrow}$. It is thus consistent to lie all VBPTRs in the objects.

Evaluation and optimizations. As method tables, $\Delta^{\uparrow}_{\tau_s}$ tables are, in worst case, quadratic in number and cubic in size. In each instance, the number of VBPTRs is quadratic in the number of superclasses, and the worst case occurs with SI: a n class chain induces $n(n-1)/2$ VBPTRs. In constrast, assuming a uniform distribution of attributes introduced in each class, the number of attributes may be considered as linear in the number of superclasses: thus, VBPTRs may easily occupy more space than attributes (see appendix A)! Furthermore, VBPTRs are compatible only with the first case of empty subobjects merging: indeed, in the second case, the F subobject would not be empty since it must contain a VBPTR to the E subobject (see section 3.3).

A similar optimization will consist in replacing $\Delta_{\Downarrow}$ by pointers in object layout [Sweeney and Burke 1998]. The overhead is smaller, as it is equal to subobject number, but the gain is also smaller as shifts to the dynamic type are less frequent.

Sweeney and Burke [1998] propose a general framework for analyzing the space overhead of various object layout, according to the repartition of data between class tables and object layout. [Gil and Sweeney 1999; Eckel and Gil 2000] present some statistics on the VBPTR cost in some large benchmarks and they propose optimizations aiming to reduce VBPTR overhead. Some of them are global and compare badly with techniques used in global compilation, whether coloring(see section 7.4) or Small Eiffel (see section 7.3.6), which do not need shifts. Other optimizations are usable in separate compilation: they use bidirectional tables (see section 5.1). The dynamic overhead of VBPTRs remains important.

A by-product of VBPTRs is the ability to implement instance classification, a monotonous special case of instance migration (e.g. `change-class` function in Clos) when the target class is a subclass of the source class. Indeed, with VBPTRs, the object layout needs not to be connected: specializing an object's class amounts to add subobjects and change pointers to method tables.

4.4 Attribute accessors

Some languages behave towards attributes and towards methods in a very close way. In Clos [Steele 1990], attributes are accessed through special read and write methods (*generic functions* in the Clos jargon) called *accessors*. In Sather [Szypersky et al. 1994], an attribute contributes to the class type by its accessors signatures. In Eiffel [Meyer 1992; 1997], a method without argument may be overridden as an attribute. Obviously, attribute implementation can be encapsulated in accessor methods: when an attribute position change in a subclass, it needs only to override its accessors. Thus, attribute offsets don't matter.

Two variants must be considered: either accessors are actually defined as methods or they are simulated, in which case method table contains attribute offsets instead of accessor addresses. Simulating accessors avoids a true method call: the access cost is the same as with standard implementation, in the general case ($T_p \neq \tau_s$,

page 10). In a second step, attribute offsets can be factorized if attributes are grouped by subobjects, with Invariant 3.1.1. Method tables contain the position of subobjects w.r.t. the object address. The access code is now exactly the same as in standard implementation, when $\tau_s \neq T_p$.

In some way, simulating accessors is only a catch since it brings back to a subobject implementation. Moreover, shifts are needed in every case, even when $\tau_s = T_p$. In order to save shifts when accessing `self`, Myers [1995] proposes a double compilation of each class: in the first one, `self` attributes are compiled with shifts, whereas in the second one, they are compiled without shifts, assuming that the subobject position will be preserved in all subclasses (e.g. if the class is only specialized with SI). The appropriate version is chosen at link-time.

However, true accessors may be needed for some features as redefining a method without parameter as an attribute (Eiffel) or implementing the full specifications of the Clos keyword `:allocation` (see section 6.3).

Nevertheless, accessors are a solution for attributes only when the question of methods has been solved. If the object layout may be constituted of a table where attribute offsets don't matter, a direct access to methods or attribute offsets is needed. Thus this is not an alternative in itself but we will see different ways to incorporate accessor simulation in a general framework, either preserving object reference variance (see section 5.1.3) or recovering SST invariance, with a global computation (see section 7.4) or a flow of method tables (see section 5.2.2). Attribute accessors can be used with any method implementation.

4.5 Comparisons

The main drawback of standard MI implementation is that its overhead does not depend on the fact that one uses MI, or not. NVI is a way to avoid this drawback, at the detriment of semantics. A general solution could be that languages allow to express the fact that a class should not be specialized with MI or, more generally, with non-arborescent inheritance: it would be at the detriment of reusability.

Accessor simulation coupled with Myers' double compilation is a better alternative since there is no detriment at all, but the benefits are only when accessing `self` attributes: method calls are not concerned, but thunks would cancel the overhead in case of SI. It is then a good solution when attributes are mostly encapsulated, à la Smalltalk. Other alternatives seem hopeless: compact method tables add a small time overhead but may be unable to effectively gain on static memory, due to code size increasing, whereas VBPTRs bring a considerable dynamic overhead, for a small time improvement.

5. SINGLE INHERITANCE AND MULTIPLE SUBTYPING (MST)

Between the two extreme cases, is the middle case where classes are in SI but types are in MST, whilst class specialization remains a special case of subtyping. This is typically the case of Java and some other languages have a very close type policy: Theta [Myers 1995; Day et al. 1995], as well as all languages designed for the Microsoft platform DotNet, C# [Microsoft 2001] or Eiffel# [Simon et al. 2000]. The specification is the one of Java [Gosling et al. 1996; Grand 1997]: the `extents` relation between classes is single, and the `implements` relation between classes and interfaces or the `extents` relation between interfaces is multiple. Only

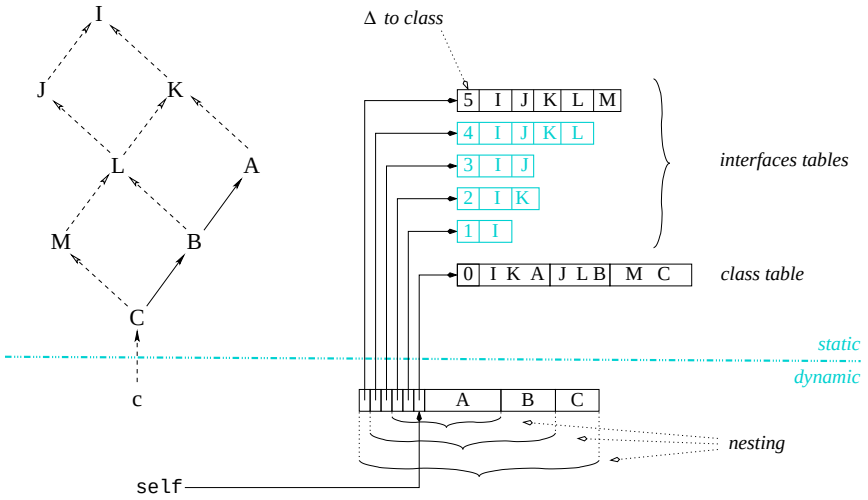


Fig. 7. Single inheritance and multiple subtyping: multiple inheritance variant, with 3 classes A, B, C and 5 interfaces, I, J, K, L, M. In grey, interface tables saved by sharing.

classes may define attributes together with method bodies and may have proper instances: interfaces are *abstract* classes and define only method signatures.

In such a framework, the SST invariants 2.1.1 and 2.1.2 cannot be verified by interfaces: more precisely, a method introduced by some interface will be implemented at different offsets in the different classes which implement the interface. However standard MI would be too complicated, as both invariants could be verified by classes. Hence, two kinds of solution can be designed, according to whether one simplifies MI implementation or one complicates SST one.

5.1 Multiple inheritance variant

Standard MI implementation can be notably simplified in the present case. Of course, all interface subobjects are empty but as all interfaces are likely to introduce some method, only the second case of subobject merging can work (see section 3.3). A more specific presentation is better.

5.1.1 *Principle.* Starting from standard implementation, the first step consists in conciliating different method tables with attribute invariance. Due to attribute invariance, only one method table is needed for the classes. Due to interface specification, interface subobjects are empty. Therefore all pointers to method tables can be grouped in a header and object layout is bidirectional: positive offsets are for the attributes, negative offsets are for interface tables, and offset 0 points to the class table. Each interface table begins with the offset of the pointer to this table, which stands for $\Delta^{\tau_s}_{\Downarrow}$: this value will serve to shift the receiver in a method call, when it is typed by an interface. When the receiver is typed by a class, no shift is needed as any entity typed by a class points to offset 0, whereas entity typed by an interface points to the offset corresponding to the interface.

In a second step, one can order interface tables, in the header, in such a way that the superclass implementation is nested inside the subclass one: the subclass add

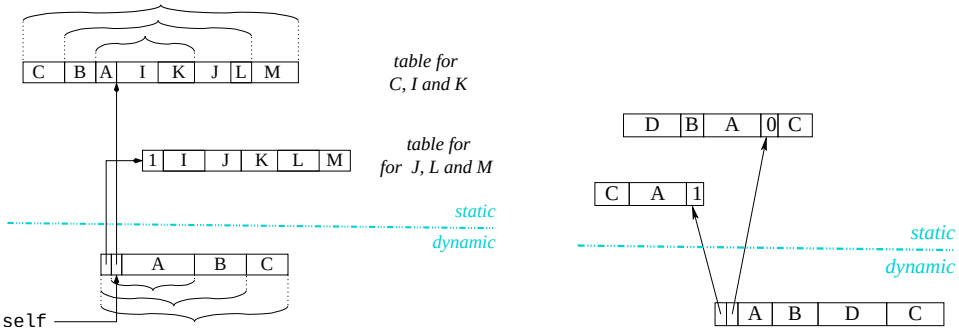


Fig. 8. THETA techniques, for examples of Figure 7 (left) and Figure 2 (right), where B is considered as primary.

new interfaces at negative offsets and new attributes at positive offsets (Figure 7).

INVARIANT 5.1.1. *Superclass implementation is nested inside subclass implementation: interface shifts ($\Delta^{\tau_s}_{\Downarrow}$) are invariant by inheritance, i.e. w.r.t. τ_d.*

The third step consists in factorizing interface tables. Sharing tables needs to take a specific convention on the way method tables are built (see section 3.3): each interface and class orders its super-interfaces in some arbitrary order, e.g. depth-first one, in such a way that, in case of SST, the superclass order is a prefix of the subclass order. Methods are also grouped in tables according to this order, and method offsets are invariant in each group, as with Invariant 3.1.1. Two interfaces may share the same table when the superinterface order is a prefix of the subinterface order. In Figure 7, class table is shared by K and interface table of M is shared by all other interfaces.

Code for accessing attributes is the same as in SST, as well as method calls when the receiver is typed by a class. When the receiver is typed by an interface, the code is the same as in MI, with the difference that `deltaOffset` does not depend on the method.

5.1.2 *Casting.* Several cases must be considered:

(1) from class to class, it is done exactly as with SST, and from interface to interface, as with MI;

(2) from class to interface, for an upcast, shift is static (constant and invariant w.r.t. dynamic type, thus without table access) thanks to nesting; a sidecast will need the table $\Delta^{\Uparrow}$;

(3) from interface to class, the shift is in the table header, but there are two cases: a method call to a receiver typed by an interface does not need any type checking, but downcast needs type check, which is done as in SST.

When a case reduces to MI, $\Delta^{\Uparrow}$, and possibly $\Delta^{\uparrow}$, tables are needed, respectively in the class and in the interface tables.

5.1.3 *Case of* THETA. The language THETA uses a very close technique with some optimizations [Myers 1995]. The basic idea is to extend object layout bidi-

rectionality to method tables[9]. The positive part contains methods declared in interfaces, when negative part contains methods introduced in a class (Figure 8). On the example, the sharing is not better than with the first variant (Figure 7), but it is intuitive that bidirectionality might improve sharing: without bidirectionality, methods introduced by a class forbid any sharing for the subclass interfaces. Myers [1995] proposes also an optimizing algorithm for computing tables.

5.1.4 *Application to multiple inheritance.* Myers [1995] proposes two extensions of his technique to MI. Both are based on the principle that, when a class has more than one direct superclasses, one of them is considered as primary, and the other are secondary. Implementation will respect the nesting principle for the primary superclass only. Obviously, the best choice of the primary superclasses is a matter of global optimization (see section 7).

Variant with method copy. The first variant is based on the code copy of all methods and attributes defined in secondary superclasses: code sharing is done only with direct and indirect primary superclasses (see section 7.3.1). The technique is sound only for language with a strict attribute encapsulation, à la SMALLTALK: if this is not the case, accessors are needed (see section 4.4). Once attributes and methods have been copied, secondary superclasses can be treated as if they were interfaces (Figure 8).

The main advantage is that MI costs only when it is used and it is always as efficient as standard inheritance, at least from a dynamic point of view: the static overhead of copying needs some assessment. But copying method code is not compatible with separate compilation. According to the author, experiments show a sensible improvement w.r.t. C++: on the example, the result is as good as with NVI (Figure 5).

Variant with accessor simulation. Instead of copying attributes, they are accessed by simulating accessors, and shifts to subobjects are added in the method tables (see section 4.4). The double compilation saves many shifts but thunks are now needed since methods may be defined in secondary superclasses. As a whole, this is certainly a good alternative to standard implementation, as efficient as NVI but without its flaws.

5.2 Single subtyping variant

The other approach consists in starting from SST implementation and extending it to interfaces.

5.2.1 *Principle.* Reference Invariant 2.1.1 is then conserved in any case, whereas Invariant 2.1.2 is always restricted to class-typed entities. Object layout and class method tables are the same as with SST, but some data must be added in method tables to deal with the case of interface-typed entities. For each interface implemented by a class, a data structure is needed to find methods and many techniques can be envisaged (Figure 9):

[9] Bidirectionality is also found in [Eckel and Gil 2000], where a positive or negative direction is arbitrarily assigned to classes without superclasses: specializing two classes with opposite direction save VBPTR (see section 4.3). Bidirectionality seems to originate in [Pugh and Weddel 1990].

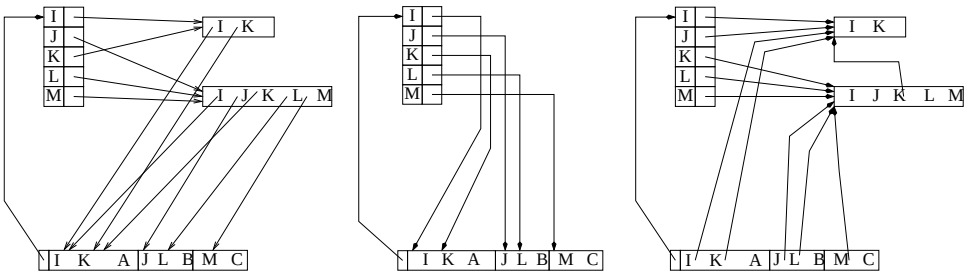


Fig. 9. Multiple subtyping and single inheritance: single subtyping variants.

(1) offset conversion tables between interface and class,
(2) first class interface method tables,
(3) if methods are grouped by class and interface, with an invariant order in each as with Invariant 3.1.1, shifts to each group are sufficient,
(4) shifts of (3) can be replaced by pointers to method groups.

But it does not seem possible to have a constant-time direct access to those data structures for a given interface. Thus an association structure such as a hashtable, like $\Delta^{\Uparrow}$, is needed in the class method table.

Starting for this general framework, some optimizations are possible. Time optimizations will consist in caching interface tables as long as the reference to the object does not change: so one access to the hashtable could serve for several method calls (see section 5.2.2). Space optimizations are based on sharing data, between interfaces and even between classes (inheritance), for variants 1 and 3. Variant 2 allows sharing only for the same class. Variants 1 and 2 have the advantage that they deal with all the methods of the interface as a whole, whereas variants 3 and 4 consider only the methods introduced by the interface. Caching will be more efficient with the former.

As for casting, it is realized mostly as in SST. Due to reference invariance, only downcasts and dynamic type check must be considered. When the target type is a class, any SST technique applies. When the target is an interface, one will use the interface association structure in a boolean way, or any other MI boolean technique.

5.2.2 *Constant-time variant, with inline table cache.* Most dynamic accesses to interface tables may be saved by inline caching and static accesses from the current statically typed values. As a matter of fact, in a method call, when one binds a value to an interface-typed parameter (which is not the receiver), the question is that the callee has no direct access to the interface methods, whereas the caller may have such a direct access since it knows the static type in the callee (unless type overriding, section 6.1) and since typing rules requires that the type in the caller is a subtype of the type in the callee. So the idea is to pass not only a reference to the object, but also the interface table corresponding to its target type. For an assignment of local variables, it is even simpler since caller and callee are the same.

In this approach, any interface-typed entity `x` is twofold: `x` itself which references an object and the interface structure `tablex` needed to send method to it, e.g. a

method table in variant 2. Moreover, each method table, as well interface tables as class tables, contains pointers on super-interface tables, in the same way as $\Delta^{\uparrow}$ (Figure 9, right). When the return type of a method is an interface, two values are returned. In the assignment `x:=y`, there are two cases likened to upcasts in MI. If `y` is typed by an interface subtype of `x`'s type, one gets the appropriate table from `tabley`:

```
load [tabley + #interfaceOffset], tablex
move y, x
```

if `y` is typed by a class, one must first get the class table of `y`:

```
load [y + #tableOffset], tabley
```

Method call to `x` follows:

```
load [tablex + #selectorOffset], method
call method
```

The case of interface-typed attributes, which are assumed to be rather rare, is more difficult: an extra pointer on interface tables in the object layout, would bring a dynamic space overhead. Thus, it will be better to search for the table when reading the attribute or at the first need: it would be the only non constant-time accesses.

Evaluation. Maintaining a systematic interface table flow in parallel of interface-typed data flow seems at first glance an efficient solution. Increasing the code size with extra parameters is balanced by the fact that there is no more need to access the object to get its method table when needed: moreover, an optimized compiler will remove all accesses to tables which are not used.

Application to multiple inheritance. Inline table cache and flow could work in plain MI: it should be used for all entities, whatever their type is. The object layout could be a mixing of SST (reference invariance and unique method table) with MI (attributes grouped by subobjects and accessed by simulating accessors, section 4.4). The omnipresent shifts of standard implementation would be replaced by an omnipresent table flow, incuring no specific overhead w.r.t. standard implementation. However, the case of attributes could be prohibitive: doubling them would bring an almost 100 % dynamic space overhead, whereas a systematic table access at each read access would bring an important time overhead. Statistics (see appendix A) show that standard MI implementation may incur a 100 % overhead in object layout. However, this overhead does not concern collections (e.g. arrays) whereas table cache would be very costly for them. Thus, this technique cannot be used as the main implementation technique. Nevertheless, it could be envisaged as a secondary technique, for instance for the case of universally-typed entities, whose value may be an instance of a class or a primitive type. It would be an alternative to the *wrappers* of Java.

5.3 Application to Java.

According to its specifications [Gosling et al. 1996; Grand 1997], Java could adopt either MI variant or SST variant. However, the language implementation seems to be constrained by the specifications of its run-time environment, the *Java Virtual Machine* (Jvm) [Meyer and Downing 1997].

5.3.1 *Interface typed receiver.* In the JVM, a specific operation, `invokeinterface`, addresses the case of method call to an interface typed receiver. [Alpern et al. 2001] present a state of the art of existing implementations, which mostly use dynamic typing techniques (see section 7.2). As the problem of searching a method on an interface-typed receiver can be reduced to the problem of searching an interface table, various techniques come back to the approach of a large class-method table, possibly transformed in a smaller class-interface table. This is the case of virtual machines Cacao [Krall and Grafl 1997] and Sable [Gagnon and Hendren 2001]. Other techniques as inline cache, possibly polymorphic [Hölzle et al. 1991], are quoted. In Jalapeño, [Alpern et al. 2001] associates to each class a fixed-sized hashtable containing either a method address, or a decision tree indexed on interface identifiers.

5.3.2 *Subtyping checks.* Dynamic class loading forbids the simple approach of Schubert's numbering. However, the [Cohen 1991] coloring variant is incremental and it is used in Jalapeño [Alpern et al. 2001]. When the target type is an interface, this technique doesn't work and a boolean class-interface matrix associates to each class the interfaces implemented by the class. Some virtual machine implementations seem to have adopted a hashtable technique as well for subtyping test as for `invokeinterface` [Meyer and Downing 1997, chapitre 9].

5.3.3 *Other* JAVA *features.* A problem of SI is the implementation of general system-defined functionalities which require some implementation in the object layout. A typical example is *synchronization*, which needs *locks* in the objects. As the instances of any class may be synchronized, a class *Synchronized-Object* would have been incompatible with SI, and an interface would fail to add some implementation in the objects: so the functionality is introduced in `Object`. Implementing locks with attributes would be a simple solution, but this attribute should be defined in `Object`: the overhead would be for all objects, even when they don't use synchronization[10]. Implementing them in object header would be the same as an attribute defined in `Object`. [Bacon et al. 2002] proposes an approach based on statically distinguishing classes which need synchronization from classes which need not: locks are then defined as an extra attribute in the former only. In order to maintain the offset invariance required by the synchronization code, this attribute is implemented at a negative offset, unlike other attributes. However, as instances of non-synchronized classes may be synchronized as well, a global data structure, called a *lock nursery*, is dedicated for it. Statistics show that most synchronized objects are instance of synchronized classes, so the gain of locks for non-synchronized classes more than counterbalances the overhead of the lock nursery.

5.4 Application to multiple inheritance: *mixins*

Literature on *mixins* is rather large [Stefik and Bobrow 1986; Bracha and Cook 1990; Ancona et al. 2000; Ernst 2002], but they are still missing a precise definition. They consist in a variation on the notion of *abstract classes*, in a less abstract way than the interface notion of JAVA. They are often presented as a way, either to avoid

[10] According to [Bacon et al. 2002], the average attribute number in a class is small, around 4 or 5: the overhead would be 20 %.

MI, or to discipline its usage. Our definition is the following. A *mixin* is an abstract class, in the SMALLTALK sense, i.e. a class without proper instances, but, contrarily to JAVA's interfaces, a *mixin* may have attributes and method bodies. Mixins are aimed at defining functionalities which are presumed orthogonal to classes and which can be mixed to them without conflict risk. We will not discuss here this hopeless objective but only propose a specification of specialization in a world where classes and mixins can be mixed. First, classes are in SI: a class cannot specialize more than one class. Second, a mixin can specialize at most one class, but no mixins: mixins must be thought of as unrelated functionalities, but some of them may apply only to instances of some precise class. Third, a class may specialize as many mixins as one wants. Four, when a class specializes a mixin which specializes (directly or not) another class, this must not lead to MI: the class specialized by the mixin must be a direct or indirect superclass of the class. It means that removing mixins from the transitive closure of specialization must lead to SI.

The mixin approach is thus very close to SI and MST of JAVA. The difference is that mixins may have code and attributes, and that their specialization is quite restricted. To take up a distinction from linguistics, a class is *categorematic*, whereas a *mixin* is *syncategorematic*: it cannot stand alone by itself [Lalande 1926].

5.4.1 *Multiple inheritance variant.* There is a striking resemblance between *mixins* and the application of MST techniques to MI proposed by Myers [1995] (see section 5.1.4): classes are primary superclasses, whereas mixins are secondary superclasses. The approach by method copy applies without any adaptation, even in separate compilation: it suffices to do with *mixins* what C++ does with *templates*, not compile them (see section 6.2). This connection between mixins and templates is not fortuitous: mixins are often presented as *parameterized heir classes*, i.e. as classes parameterized by the superclass of the class resulting from their instanciation. As a matter of fact, C++ templates allow such usage [Ernst 2002]. Let A be a class, M a mixin: then, defining a subclass B of both A and M is the same as defining B as a subclass of $M\langle A\rangle$ [Bracha and Cook 1990; Ancona et al. 2000]. But there is no way to get an implementation from this analogy: implementation of genericity proposed in section 6.2 does not allow to make $M\langle A\rangle$ a subtype of A.

[Myers 1995] second variant applies also: accesses to `self` attributes are then optimized in the class methods and no double compilation is needed as classes are in SI. However, contrarily to SI and MST which allows a more efficient implementation than plain MI, it does not seem that mixins bring such an improvement. More generally, the fact that a class C is abstract, i.e. without proper instances, does not bring any significant optimization: some data structures can be saved for the case where $C = \tau_d$, since τ_d is never an abstract class. In the case of mixins, a basic point is that concrete classes are in SI, thus allowing Invariant 2.1.2 for classes. Thus, MI overhead is only within mixins. In some way, the mixin approach can be likened to NVI: both improve efficiency at the detriment of semantics or reusability.

5.4.2 *Single subtyping variant.* The section 5.2.1 schema (Figure 9) can be extended to separate compilation of mixins in the following way. The only differences with interfaces are attributes and method bodies defined in the mixins. As a mixin doesn't specialize another mixin, one can group attributes and methods proper to the mixin, in the object layout and in method tables. A mixin method needs only

to know the two shifts correponding to attributes and methods for accessing `self`. As `self` is immutable, those two shifts can be computed only once, when entering the method. They also may be cached in the same way as interface method tables. As for attributes and methods inherited from a superclass, they are accessed in the same way as for a class.

The definition of mixins is rather fuzzy. The examples given by [Bracha and Cook 1990] make them orthogonal to interfaces: they are not aimed to introduce a new type but to discipline the use of `call-next-method` (see section 6.4.1). The implementation proposed here is based on our precise specification which forbids specialization between mixins: a unique shift is thus enough. If mixins could specialize mixins, one shift per specialized mixin would be necessary and this implementation would be no more justified.

5.5 Evaluation

Two main variants must be considered: à la JAVA or à la THETA. In both, time efficiency is the same as SST as long as class-typed entities are concerned. In the THETA variant, method calls to interface-typed entities is constant-time, as with MI, with simplified shifts. In the JAVA variant, the shifts are avoided but either method calls are no more constant-time, or there is an interface table flow which has to be experimented. Static space overhead is rather small, due to sharing, and dynamic space overhead is null in JAVA and small in THETA.

Both implementations have reasonable and close costs as long as interfaces are not intensively used as it is probably the case for hand-made programs. However, when interfaces are numerous, for example when they are automatically computed [Huchard and Leblanc 2000], the THETA variant could bring a too large dynamic space overhead. Moreover, the type annotations also can be automatically computed: in such a case, the interface-typed entities might be numerous and the constant-time variant may be better. A more precise comparison between those two main variants, as well as between subvariants, would need experiments.

Among the various applications to plain MI, only the approach of THETA by simulating accessors is convincing: MI overhead is reduced without semantic flaw. The only extra cost is a double compilation of each class and an easy analysis, at link-time, to choose the one to use according to the way the class is specialized. The other approaches are not convincing: method copy is incompatible with separate compilation and badly compares with other global techniques (see section 7). As for mixins, they are a restriction to MI and reusability which may be not counterbalanced by efficiency increasing.

6. COMPLEMENTS

6.1 Type variant overriding

It is well known that type safety imposes that the parameter type in the overriding method must be a supertype of the one in the overridden method, whereas the overriding return type must be a subtype of the overridden one. The return type is said covariant, and the parameter type contravariant. As for attributes, they must be invariant. The fact is that strictly contravariant parameters are not interesting, since the models that one wants to implement are mostly covariant. See

for example [Cook 1989; Weber 1992; Castagna 1995] for a criticize of covariance and [Meyer 1997; Ducournau 2002b] for a defence. Combined with the difficulties of implementation that we will see, this explains why languages as C++ and JAVA forbid parameter type variance, otherwise than as static overloading. However, implementing variant overriding comes into question, either for type safe overriding, or for pure covariant languages like EIFFEL. In all cases, type variant overriding amounts to casting, either upcasts or downcasts.

6.1.1 *In single subtyping.* With SST, object references are invariant w.r.t. static type: thus safe overriding need no specific implementation and unsafe overriding amounts to dynamic type checks.

Covariant parameters. It may be done in the overriding callee, with the drawback that it would be systematically done, even when the static type of the parameter in the caller is a subtype of the parameter type in the callee.

In a different context (see section 5.1.3), Myers [1995] proposes to assign an offset to each method signature, not only to each method name. This looks like static overloading implementation (which is of course incompatible with type overriding) but two different offsets may here reference the same method address: offsets express syntax, whereas addresses in the entries express semantics. The various entries corresponding to the same method with different signatures and offsets point to *thunks* which do the required type checks. For a method $m_C(t_1, ..., t_k)$, overriding some methods $m_{C_i}(t_1^i, ..., t_k^i)$ with different signatures and offsets δ_i, the C method table contains, for each offset δ_i, a pointer to a *thunk* which type checks parameters for all j such that $t_j^i \neq t_j$, before jumping to the m_C address.

Covariant attributes. Read accesses are as usual, at the condition that write accesses always check the assigned value. There are two ways to type check it: either one systematically uses writer methods (see section 4.4) and one reduces the question to covariant parameters, or one type checks in the caller. One needs then a *dynamic* downcast, since the attribute type depends on τ_d. Thus the type identifier must be stored in the method table:

```
load [object + #tableOffset], table1
load [val + #tableOffset], table2
load [table1 + #attributen1Offset], n1cible
load [table2 + #n1Offset], n1source
load [table1 + #attributen2Offset], n2cible                    2L + 4
comp n1cible, n1source
blt #fail
comp n2cible, n1source
bgt #fail
store val, [object + #attributeOffset]
```

Both techniques have drawbacks. The former imposes a method call for each attribute writing, but the type check is done by a thunk, only when needed. The latter avoids a method call but requires an access to the table and a type check in any case, even when there is no overriding. Dynamic downcasts are clearly less costly than method calls plus static downcasts ($2L + 4$ vs. $4L + B + 3$). As against this, the code sequence for dynamic downcasts is longer.

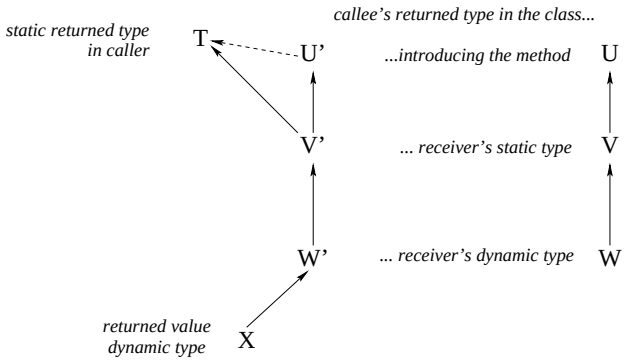


Fig. 10. Covariant return type: when $U' \neq W'$, upcast from W' to T is dynamic whereas casting from U' to V is static but no more upwards.

EIFFEL *anchored types.* They bring two kinds of optimizations. Firstly, all entities with the same anchor may share an entry in the method table. Secondly, some type checks may be avoided, when the compiler can prove that it is safe, typically when two anchored types are anchored to the same receiver: this is the only case where an anchored type may be its own subtype. [Meyer 1997] proposes the so-called *catcall rule* to ensure type safety by forbiding to access covariant properties when the receiver is polymorphic ($\tau_d \neq \tau_s$). This rule does not work in separate compilation. Moreover, it is too restricted since it forbids calls that a global analysis might accept.

6.1.2 *Multiple inheritance.* Even type safe overriding is now costly because of the shifts between subobjects. C++ allows only return type overriding[11]: this may be not only because of type safety and static overloading.

Covariant return type. In multiple inheritance, covariant return type is safe but a shift is needed, which must be done either in the callee or in the caller. However, satisfying Invariant 3.1.1 is now a question: what should be the static type of the returned value? Two alternative invariants can be envisaged.

INVARIANT 6.1.1. *The static type of the return value is the return type in the class which introduces the method (U' in Figure 10).*

INVARIANT 6.1.2. *The static type of the return value is the return type of the callee (W' in Figure 10).*

The former has the advantage of saving a global invariance: the callee does an upcast towards U'. When U' is a subtype of the type T of the entity using the returned value, the caller does a second upcast. Otherwise, a downcast or sidecast is required, but only if the overriding is visible from the caller, i.e. if $U' \neq V'$.

[11] Ellis and Stroustrup [1990, pages 210 *sq.* and 421] say that this is part of the ANSI specification, but that it would complicate method calls. Effectively, it was not compatible with some casting prohibitions (see section 4.1). Covariant return type is however explicit in the current language specifications [Koenig 1998].

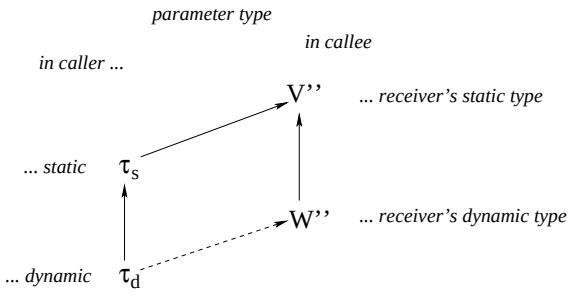


Fig. 11. Covariant parameters: a sidecast from τ_s to W'' is needed.

With the latter invariant, the solution will consist in a thunk, which does the shift $\Delta^{\uparrow}_{W'}(V')$, for each pair (V, W).

```
add object, #delta, object
call #method
load [return + #tableOffset], table
load [table + #castOffset], delta
add return, delta, return                    // cast from W' to V'
```

A second cast from V' to T must then be made in the caller. The main drawback is that the method call in the thunk is no more terminal, i.e. it is a `call`, not a `jump`. The main advantage is that there is no more downcast, and that it does cost only when needed, i.e. for pairs (V, W) such that $V' \neq W'$.

Variant parameters. When the parameter type is invariant, parameter passing needs only a static upcast from its static type in the caller to its static type in the callee: the shift is given by the table $\Delta^{\uparrow}_{\tau_s}$.

With a stricly contravariant overriding, casting is always upwards and safe, but it is no more static, since the target depends on the receiver's type. Extending method tables to parameter shifts would be possible, but very costly in static space: thunks are certainly more adapted and type overriding would not add more thunks.

The case of covariant overriding is described in Figure 11 and 6 types are implied:

—τ_s and τ_d are the static and dynamic types of the parameter in the caller;
—V'' and W'' are the static types of the parameter in the respective methods of the receiver's static and dynamic types.

By construction, 3 subtyping relationships are posed:

$$\tau_d <: \tau_s \qquad \text{polymorphism} \qquad (8)$$
$$\tau_s <: V'' \qquad \text{compile-time type checking} \qquad (9)$$
$$W'' <: V'' \qquad \text{covariant overriding} \qquad (10)$$

The goal is to check that $\tau_d <: W''$ and to shift with $\Delta_{\tau_s, W''}$: casting may be upwards, downwards or sidewards, but it is dynamic. There are two approaches:

$$\Delta_{\tau_s,W''} = \quad \Delta_{\tau_s,V''} + \Delta_{V'',W''} = \quad \Delta^{\uparrow}_{\tau_s}(V'') + \Delta^{V''}_{\downarrow}(W'') \qquad (11)$$
$$\Delta_{\tau_s,W''} = \quad \Delta_{\tau_s,\tau_d} + \Delta_{\tau_d,W''} = \quad \Delta^{\tau_s}_{\Downarrow} + \Delta^{\Uparrow}(W'') \qquad (12)$$

In the two cases, a first static shift is done in the caller. A type check and a dynamic shift are then done, either in a thunk (11), or in the callee (12). The former approach has the advantage of imposing a casting only when needed. Notice that thunks and subobjects exempt from the signature indexing of Myers [1995].

Covariant attributes. This is a mixing of return type and parameter type. One must first choose one of the two invariants 6.1.1 or 6.1.2 to rule the attribute value. A read access needs then exactly the same casting in the caller as for a method call. A write access needs first a type check, done with a dynamic downcast: the target type must be stored in the method table, as in SST. With Invariant 6.1.1, an extra upcast will be needed. Implementing write access with true accessors may be a solution (see section 4.4), but it would be of no use for read access.

Thunks balance. Multiple inheritance overhead notably increases—it roughly doubles—as soon as overriding is not invariant, even in the safe case of return type. However, covariant method overriding may be implemented within the thunks, without increasing their number—one thunk per pair (τ_s, τ_d)—with the advantage that a true overhead occurs only in case of effective overriding between τ_s and τ_d (as in SST with the technique of Myers [1995], but without needing extra offsets). In this particular case, thunks gain credence. On the other hand, for attributes, thunks should be avoided as they impose true accessors and systematic extra method calls. As against this, downcasts are systematic.

On the whole, for a method introduced by a type U, there is exactly one thunk for each pair (τ_s, τ_d) with $\tau_d \preceq \tau_s \preceq U$. Each *thunk* does successively:

—the cast on the receiver (Figure 3), from τ_s to the type W which defines the method inherited by τ_d;

—the possible downcasts on the parameter whose static type in W is a strict subtype of the type in U (Figure 11),

—the method call itself,

—a possible upcast on the return type (Figure 10).

When $\tau_s = W$, the thunk is the method itself, without any shift or cast.

With another MI implementation. With other techniques like NVI or compact tables, type overriding is roughly implemented in the same way. However, as tables may be shared between several static types, the technique of Myers [1995] consisting in assigning different offsets to different signatures may be an improvement.

6.2 Genericity

Compiling parameterized classes rises a specific problem. There is a simple way to elude it: not compile them, as C++ does with *templates*. In this case, each instanciation of a parameterized class amounts to the generation of a new non parameterized class, where actual types have been substituted to formal types. All arguments in favour of separate compilation urge to compile parameterized classes: see, for instance, the criticize by Lippman [1996] of C++ templates for their lack of separate compilation. Moreover, such a compilation allows code sharing between the different future instanciations of the same parameterized class. This is what

Odersky and Wadler [1997] call a *homogeneous* implementation, as opposed to C++ *heterogeneous* implementation.

We will mostly consider here bounded genericity, i.e. parameterized classes $A\langle T <: B\rangle$ where the formal type T is constrained to be a subtype of some bound B: this is the best way to allow static type checking. *Match-bound* polymorphism, i.e. replacing subtyping by *matching* for the bound [Bruce et al. 1997], has no effect on the implementation.

6.2.1 *Instanciation with primitive types.* However, a heterogeneous approach seems better when formal types are instanciated by primitive types, above all in case of collections which need a particular efficiency. In this case, a homogenous approach would indeed impose the use of *wrappers*, as in Java. So, compiling a bounded parameterized type $A\langle T <: B\rangle$ where the bound B is a universal type, will amount to produce an homogeneous code shared by all non primitive classes plus a specific code for each primitive type. Of course, the instanciation $A\langle B\rangle$ needs a third kind of technique, e.g. wrappers, and the generally unsafe subtyping $A\langle C\rangle <: A\langle B\rangle$ is not possible.

6.2.2 *In single subtyping.* Invariant 2.1.2 makes bounded genericity implementation very easy: attribute and method offsets of the formal type are the same as the bound. Notice that genericity may give implicitly rise to MST as soon as one accepts the generally unsafe subtyping between different instances of the same parameterized type (see figure 13-c/d). But this restricted case of MST does not require special implementation.

6.2.3 *In multiple subtyping.* When the bound B is a class, SST implementation still works. On the other hand, when the bound B is an interface, offset invariance is no more ensured. A solution is extending the parameterized class method table by a conversion table converting offsets from formal type to actual types. Method calls need then one extra table access (Figure 12). A shift to method groups of each supertype may replace conversion. However, pointers to method tables is not possible, because of polymorphism. If $A\langle C\rangle$ table contains a C method address, an entity typed by C could not be valued by a subclass $D \prec C$. Method call is then:

```
load [generic + #tableOffset], table1
load [object + #tableOffset], table2
load [table1 + #selectorOffset], offset                    3L + B + 1
add table2, offset, method
load [method], method
call method
```

In Pizza [Odersky and Wadler 1997], an alternative to bounded genericity is to pass methods as parameters of the instanciation. This reduces easily to the previous implementation: method offsets are effectively passed as parameters and assigned to the conversion table.

More generally, this technique amounts to consider methods of the formal types as methods of the parameterized type: but it is implemented without explicit extra method call. It is then a priori adapted to all the techniques described for MST as long as actual types are classes. When a formal type is instanciated by an interface, i.e. the bound itself is an interface, some restrictions are needed. In the MI variant

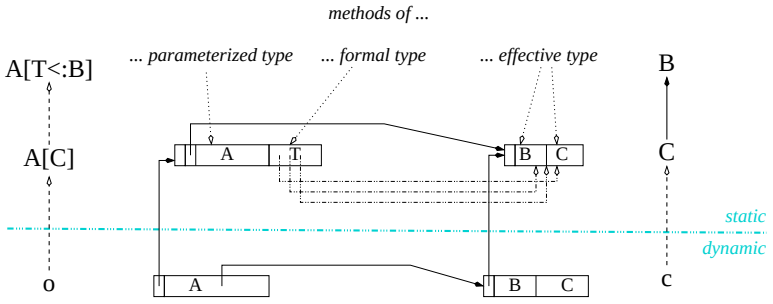


Fig. 12. Parameterized classes: method table with multiple subtyping

(see section 5.1), accessing an interface needs a shift: all method calls will need a shift, even when the type is instanciated by a class. As for the SST variant (see section 5.2.1), it does not allow an instanciation by either a class or an interface, since method calls are realized in a different way, as explicited in the JVM by different operations, `invokevirtual` and `invokeinterface` [Meyer and Downing 1997]. So, in both cases, an efficient solution is a double compilation, according to whether the actual type is a class or an interface: this means 2^k compilations when there are k interface-bound parameters.

With bounded genericity, upcasts from the formal type T to the bound B (or to a bound supertype) must be considered: the problem occurs only with MI variant, when B is an interface. The static solution is to add an entry in the table of A, containing the shift from the actual type to B. Other casting from or to T seems unsound.

Specializing parameterized classes. Two kinds of specialization are to considered (Figure 13). When $A'\langle T <: B'\rangle \prec A\langle T <: B\rangle$[12], then, following the definition, $A'\langle C\rangle \prec A\langle C\rangle$. The proposed implementation is compatible with such a specialization: one only needs to extend the method table of $A\langle T <: B\rangle$ with methods introduced in A'. Casting between A and A' is as usual.

When $D \prec C$, one may accept that $A\langle D\rangle <: A\langle C\rangle$: this is the case in EIFFEL but it is type safe only when T is never used in contravariant position [Cook 1989; Weber 1992]. The unsafe cases must be handled in the same way as for unsafe covariant overriding (see section 6.1). However the implementation imposes some restriction. There is no problem when C and D are classes: then $A\langle C\rangle$ and $A\langle D\rangle$ share the same method table. When C is an interface, the MI variant rises a problem due to the possibly different offsets for C and D. Moreover, if D is a class, SST variant also rises a problem, since method call techniques differ: a low-overhead solution is that calls to interfaces also work for classes. Thus, combining those two kinds of specialization does not lead to MI problems, since only the former add some entries in the method table.

When it is possible, the latter case of specialization gives rise to a downcast problem: casting from $A\langle C\rangle$ to $A\langle D\rangle$ or to $A'\langle D\rangle$ needs that parameterized classes keep some information concerning their formal types. A solution is that A's table

[12] This means that $A'\langle T\rangle \prec A\langle T\rangle$, for all $T <: B'$, with $B' <: B$.

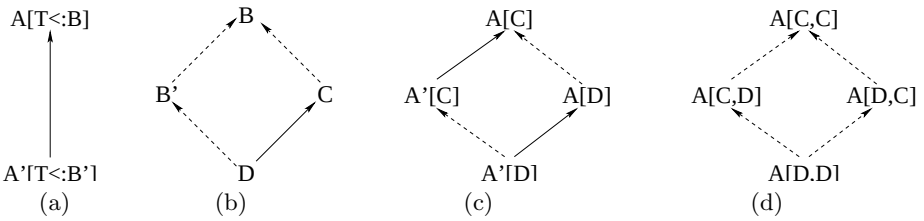


Fig. 13. Specializing parameterized classes

points to the table of the types which instantiate its formal types. Then, downcast from $A\langle C\rangle$ to $A'\langle D\rangle$ imposes two type checking, first on the parameterized classes, then on the parameter.

Genericity in Java. Many generic extensions of Java have been proposed: Pizza [Odersky and Wadler 1997], Generic Java [Bracha et al. 1998], NextGen [Cartwright and Steele 1998] and many other [Agesen et al. 1997; Solorzano and Alagić 1998]. None of them takes the approach presented here because it is not compatible with Jvm: compatibility would imply to realize method calls to formal types with an explicit call to `self`, whereas we only simulate this call with an offset. In practice, *homogeneous* propositions replace the formal type with the bound (type erasure) and add safe downcasts which need no implementation in the SST variant.

6.2.4 *In multiple inheritance.* Multiple inheritance of parameterized classes is possible: so they must be implemented as non parameterized classes (section 3). In each method table, method calls to formal types are the same as in MST. On the other hand, attributes cannot use neither SST technique for attributes, nor MST for methods: they are indeed invariant only w.r.t. the static type which introduces them. An attribute p accessed on the formal type T is necessarily known by the bound B, thus $B \preceq T_p$ and δ_p is known and invariant (see page 10). However, the upcast from T to T_p is not static as T is formal. Thus, for each T_p, an entry of the method table must contain the position of the shift Δ_{T,T_p} in the $\Delta^{\uparrow}_T$ table, i.e. the value of $i_T(T_p)$.

```
load [generic + #tableOffset], table1
load [object + #tableOffset], table2
load [table1 + #castOffset], delta
add table2, delta, table2                                   4L + 2
load [table2], place
add object, place, place
load [place + #attrOffset], attribute
```

Upcasts from T to B need also to add in the method table of A an entry containing the value of $i_T(B)$: one may generalize by adding an entry for each supertype fo B.

However, as in MST with the MI variant, subobject implementation makes the subtyping $A\langle D\rangle <: A\langle C\rangle$ difficult: entities typed by C and D don't point on the same subobject. Thus substitution is not possible. In fact, the problem is exactly the same as for variant overriding of attribute and return types (invariants 6.1.1 or 6.1.2). For a type safe language, this is a minor limitation as this specialization is rarely safe [Day et al. 1995]. For a covariant language as Eiffel, this is more

problematic: systematic casting is needed (see section 6.1) and the method tables of $A\langle D\rangle$ and $A\langle C\rangle$ are no more shared.

6.3 Shared class attributes

The term of *class attributes* is fuzzy and at least three different notions are concerned:

—C++ and JAVA `static` variables are not relevant to object orientation as they cannot be selected, in case of overriding, according to the dynamic type of an object;

—attributes of classes considered as instances of meta-classes, in a reflexive model, à la SMALLTALK or CLOS: this notion has no direct equivalent in the languages that we consider here;

—the fact that an ordinary attribute could be shared by all instances of a class: it may then be allocated in the class data structure instead of in every instance: this is the meaning of the keyword `:allocation :class` in CLOS: to avoid ambiguity we will call them *shared attributes*.

The two last notions differ by their semantics: we can illustrate the former by the set of proper instances of a class, which is a property of the class, not of the instances, and the latter by the side number of `polygons`, which is 4 for all instances of `quadrilateral`, and 3 for all instances of `triangle`. The difference is that a class does not share its instance set with another class, whereas all subclasses of `quadrilateral` share the same side number, except odd ones as `3.5-side-quadrilateral`...

Shared attributes will be implemented in the method table: as several classes, with different tables, may share the same attribute, an extra indirection by a *wrapper* is needed when the attribute is mutable. The wrapper is needless when the attribute is read-only. Multiple inheritance does not complicate the case.

Curiously enough, this sound and efficient mechanism is not proposed by commonly used languages: C++ and JAVA offer only `static` variables. In EIFFEL, constant features amount to read-only[13] class attributes: `once` features are a variant where the constant is itialized by a first call, subsequent calls returning the same value. When they are `frozen`, this is an equivalent of `static` in C++ and JAVA. The keyword `:allocation` of CLOS has actually a more complicated specification than ours, as it may take two values, `:instance` or `:class`, and it may be overridden. This is against all attribute invariants, whether they are in single or multiple inheritance. Plain accessors (see section 4.4) is a way to implement it in a general way, with a dynamic space saving, but at the detriment of time efficiency. An alternative would be to implement shared attributes in the instances, with a wrapper to ensure value sharing, thus at the detriment of time. On the whole, the complete specification of the keyword should be reserved to attributes explicitly declared at introduction.

[13] Notice that this is the binding between the name and the value which is immutable, not the value itself.

6.4 Calling `super`

Almost all languages offer a mechanism for calling the overridden method from the overriding method: this is a sound way to save a kind of behaviour invariance in spite of overriding. In SMALLTALK and JAVA, this is realized by the keyword `super`—a pseudo-variable as `self`—which consists in calling the superclass method on the current receiver, i.e. the `self` value. Although this mechanism is designed for calling the overridden method, `super` syntactically requires the method name: in fact, it allows to call any superclass method[14]. In SI, the superclass method is unambiguously determined as well as the method offset: everything is invariant w.r.t. dynamic types, and the super call is static.

6.4.1 *In multiple inheritance.* `Super` does not work since *the* superclass method is not uniquely determined, in the general case. However, there exists at least three variants of the mechanism: static call (::) in C++, `precursor` in EIFFEL and `call-next-method` in CLOS.

The C++ operator :: is more general than `super` as it allows to call any method as a classic static procedure call: the technique is the same as for a method call, except that an access to method table may be needed for the shift, not for the address. The mechanism is quite versatile, but it has an important drawback, liken to *repeated inheritance* (see section 4.1): in the diamond example of Figure 2, when the method m in class D calls methods B::m and C::m, which both call A::m, then the latter is executed twice when calling D::m.

In EIFFEL, `precursor` differs from `super` in JAVA and SMALLTALK on two points: first, the mechanism applies to MI but only when there is no multiple inheritance conflict, second it applies only on the current method name, which corrects the `super` flaw. As the overridden method is unambiguously determined, implementation is the same as for :: in C++.

CLOS `call-next-method` is more original: it consists in calling the next method in the linearization order of the superclasses of the receiver's dynamic type [Ducournau et al. 1994]. The main advantage is to avoid the repeated inheritance problem. An important drawback is that the next method will depend on the dynamic type, not only on the static type. In the diamond example, the linearization of D (resp. B) may be $\{D, B, C, A\}$ (resp. $\{B, A\}$): so, in B, the next method will be in C (resp. A). This causes some trouble to modularity and it makes implementation more difficult. A direct static call does not work. A simple solution is to assign an extra offset in the method tables, for each method. Whereas primary method entries contain the same address for all static type of the same dynamic type (Figure II), `call-next-method` entries will contain different addresses. Those extra offsets are needed only for methods which explicitly use `call-next-method`, and only in the method tables of the static type where the method is defined: thus this new mechanism would cost only when and where it is used.

6.4.2 *The case of* BETA. The BETA language [Kristensen et al. 1987] differs from all other object-oriented languages by the way method combination is realized. Instead of calling the overridden method from the overriding method as with

[14] In JAVA, the static type of `super` is the superclass of the current class.

`super`, the overridden method calls the overriding method with the keyword `inner`. So there is neither true overriding, nor late binding for method calls which are always static: method tables are only needed for `inner` calls. Nevertheless, all the techniques reviewed here more or less apply.

6.5 `Null` pointers

Any variable or attribute typed by a class must be explictly initialized. An intraprocedural data flow analysis may detect uninitialized local variables, but this is more difficult for attributes, which may be initialized some time after creation. Thus, an initialization with a `null` pointer is a way to avoid random memory state. The simplest solution consists in a distinguished value `null` and in checking at every object access (method call, attribute access, cast) that the receiver is not `null`.

Even when those checks are restricted to accesses which are not proved to be non `null`, this would add a considerable overhead. There are two general kinds of alternatives. The first one depends on hardware and operating system: for instance, in [Alpern et al. 1999] `null` has address 0, and all offsets are negative as, in the AIX system, negative addresses rise an interrupt. Software based solutions consist in a distinguished instance `null` for each class: all its attributes are initialized to `null` and its method table points to methods which signal an exception. This object must be allocated in the code area, assumed to be read only, in order to prevent any assignment. However, read accesses are possible, propagating `null` values, which may make debugging difficult.

Different `null` objects may be shared: in SST, a unique `null` object is even possible if the method table structure allows it (method tables may contain more information than method addresses). This object will be constituted by the largest object layout, with the largest method table. In MI, a unique object is also possible: it will consists in the unique instance of the class $\bot$, subclass of all classes, and will be constituted by one subobject by class. Of course, those shared `null` objects must be computed at link-time (see section 7).

6.6 Garbage collection

Automatic memory handling is an argument for fiability: this has been well understood by designers of Lisp, Smalltalk, Eiffel and Java. There are many *garbage collection* (GC) techniques. All need some information concerning individual objects: length of memory area, boolean status of the object and possibly a pointer on a new generation. Those informations may be contained in the object layout or implemented in separate tables. We will only consider the former case. [Wilson 1992; Jones and Lins 1996] are reviews of garbage collection techniques.

6.6.1 *Single subtyping.* Standard SST implementation is well adapted to GC: all proper instances of a class have the same length, so it may be stored in method tables. Moreover, reference invariant avoids pointing inside memory areas.

A few marking bits are needed for different status informations during the GC process: they are partly used for marking living objects, reachable from memory roots, and are proper to each instance. With static typing, there is no way to dynamically distinguish an immediate value from an address. Thus, an easy way to search for all living objects consists in defining a specific method for this purpose,

which is automatically generated by the compiler [Colnet et al. 1998]. When marking is not implemented in this differentiated way, it must conservatively assume that each field in the object layout is a potential pointer on a living object, which is slower: the GC is then called *conservative* [Boehm 1993]. However, GC must be at least *semi-conservative* since root extraction from registers and stack requires such an assumption. In all generality, checking that a memory word is a pointer to an object may consist in checking firstly that this is a valid address, secondly that the first field is a pointer to a method table, which is easier since method tables are static. This is a necessary condition but it is not sufficient. Now calling the marking method on an address which is not an object may lead to unexpected results: thus, when marking is implemented by a method, root extraction must be exact and it requires a more precise condition. An alternative technique will use a mapping of the heap into a bitstring: each bit represents a word (resp. double-word) allocation, causing a 3 % (resp. 1.5 %) memory overhead. Allocating objects of different sizes or types in different areas is also a technique [Colnet et al. 1998] which allows to reserve only one bit per object but it increases memory fragmentation.

When they are implemented in the object layout, those bits can occupy a field by itself, but this is a considerable overhead. *Bit-stealing* is a general way to avoid the overhead, by implementing them as low or high weigth bits of some field in the object layout. For instance, when the object size is not shared, it needs only two bytes and one extra byte is free for marking bits. In the general case, one can use either the pointer to method table, or the first attribute, for instance when it is a pointer. In both cases, there is a uniform overhead, with two more instructions and cycles for removing those extra bits, at each access to the field. [Bacon et al. 2002] present some experimental statistics on this approach in Java, where pointer to method is used. However, the overhead might be null if the value of those bits is unchanged between two accesses to the field: only the methods implementing the GC, or the access to the first attribute, should be compiled in a specific way to remove the extra bits which are no more at their default value. So the overhead depends on whether the GC is implemented as a concurrent background task, or as a blocking task. Using the first attribute field supposes reserving it for an attribute which don't use 32 bits, for instance when it is a pointer. The overhead will then be supported only by one attribute. However, some classes may have only 32 bits attributes (e.g. integer): GC would then have a dynamic space overhead.

6.6.2 *Multiple inheritance.* When Invariant 2.1.1 is not verified, the marking stage is complicated since object references may point inside memory areas. Marking methods may help to solve the problem but a general alternative exists: each method table may have an entry which contains a shift towards the first subobject in the object layout. Moreover, this entry may be the same as for $\Delta^{\tau_s}_{\Downarrow}$, if one places always the dynamic type subobject first. In any case, an extra overhead must be expected. Notice that VBPTRs (see section 4.3) avoid this complication since the object layout is made of explicitly linked subobjects. In any case, GC complexity is a function of subobject number not of object number.

7. COMPARING WITH GLOBAL COMPILATION

Separate compilation is a good answer to modularity requirements of software engineering: it brings speed of compilation and recompilation together with locality of errors and protects source code from both infringement and hazardous modifications. With separate compilation, the code generated for a program unit, here a class, is correct for all correct future uses. Run-time generation ends with a final step linking all program units.

7.1 Advantages of global compilation

7.1.1 *Hierarchy closure.* The first advantage of global compilation is that the class hierarchy is closed: no extra class can be added after the compilation, unless a new complete compilation of the whole hierarchy. It is then possible to know if a class is specialized in single or multiple inheritance, if two unrelated classes have a common subclass, and so on. Moreover, the schema of each class is known. So, in a dual way, one knows where methods are defined: methods with a unique definition may then be treated as classic procedural calls (monomorphic calls). This is known as the *unique name* heuristics [Calder and Grunwald 1994]: thought its simplicity, its effect is not small as this applies to almost 45 % methods in SMALLTALK.

With static typing. Hierarchy closure brings even more information: a call is monomorphic as soon as the method is not overridden in the subclasses of the static type. Moreover, a hierarchy analysis allows to determine which classes must be compiled as virtual, and which ones don't need it [Eckel and Gil 2000] (see section 4.1). Alternatively, one can decide of primary and secondary superclasses (see section 5.1.4).

7.1.2 *Knowledge of method code.* A second advantage is the knowledge of the code of all methods: when compiling a method, one also knows how the method is used, and when compiling a method call, one also knows the method codes for all possible callees. Many optimizations proceed from this knowledge but they all suppose an underlying general implementation technique.

7.2 Global compilation in dynamic typing

In a dynamically typed language as SMALLTALK, the lack of type annotations makes separate compilation quite unefficient. So, many techniques of global compilation have been worked out in the framework of dynamic typing: of course, they all apply to static typing as well.

7.2.1 *Dynamic typing and single inheritance.* Notice first that, with dynamic typing, SI is no more a simplification, due to *definition overloading* (see note 3, page 6): when the same method name is introduced in two unrelated classes, Invariant 2.1.2 is not verified, at least in separate compilation. This concerns also attributes as soon as they are not encapsulated as in SMALLTALK. JAVA type system may be understood as the minimal type system required for statically typing SMALLTALK: the interface notion is the way to deal with definition overloading. So, with dynamic typing, implementation is at least as difficult as with MST, or even with plain MI when attributes are not encapsulated.

7.2.2 *Compilation techniques.* If one puts aside dynamic search (*lookup*) in the class hierarchy, method call techniques are either with constant-time direct access tables or with a group of techniques summarized under the name of *type prediction*.

Compacted large table. Table-based techniques consist in compacting the large array obtained by a global and injective numbering of all classes and methods. As the class number (resp. method number) may reach and even exceed 1000 (resp. 10000), this table is huge, several M-entries, and too large to be implemented as such. However, the number of valid class-method pairs is far smaller, at most 5 %: it is the total size of method tables in SST, $\Sigma_C M_C$. Two techniques for compacting that table have been proposed: row displacement proposed by Driesen and Hölzle [1995] and method coloring proposed and studied by Dixon et al. [1989], Pugh and Weddel [1990], André and Royer [1992] and Ducournau [1991; 1997]. The latter technique will be detailed further.

In both cases, the result is a table each entry of which either contains a unique class-method pair or is empty. However, the lack of static type checking yields that a method may be called on a receiver of a wrong type, which may amount to either an empty entry or a class-selector pair with a different selector: thus a dynamic type checking is needed but it reduces to a simple equality test between the expected method and the effective one. Static typing makes this extra test useless. On the whole, there are few empty entries, less than 10 % in most of the benchmarks, with a 50 % upper bound for all experiments: compared to standard MI, the total size remains close to $\Sigma_C M_C$ (see appendix A).

Type prediction. Type prediction originates in the idea that a given type may be considered as more likely for some method call: for instance, type `integer` for the method `+`. So, type prediction consists in compiling a method call by a comparison between the expected type and the effective receiver's type: if the test succeeds, a static call is done, otherwise a call to another technique is done. There are many variations on this basic idea. The expected type may be the receiver's type for the previous call, either on the same call site (*inline cache*) or globally (global cache). Prediction may also be *polymorphic* when the receiver's type is tested against several types [Hölzle et al. 1991]: in that case, the call sequence is a small decision tree which compare the receiver's type with all the expected types.

The technique is not a priori sufficient since cache defaults must be handled when the receiver's type is not amongst predicted types. Therefore some underlying technique is needed, for instance a dynamic *lookup* à la SMALLTALK. An alternative is to rule out cache defaults: the decision tree must then exhaust all possible types. This is easier with static typing as the static type is available. A type analysis may give a more precise information, the *concrete type*. At last, it is better to balance the tree, as the number of expected types may exceed 100 [Zendra et al. 1997].

The entire call sequence is made of conditional branching: [Driesen et al. 1995] shows that they are statistically very well predicted by modern processors. Therefore, indirect branching (B latency) and memory accesses (L latency) are avoided. As against this, the number k of expected types, thus the branch number, may be large and the search is in $O(log_2(k))$: but a predicted branch has a 1-cycle cost, whereas an unpredicted one has a B-cycle cost.

Subtyping test. In tree-based techniques, the comparison between the effective type and the expected type may be either an equality test or a subtyping test. With the former, all types must be exhausted, whereas only methods need to be, in the latter. But subtyping test is expensive in the general case. [Queinnec 1997] handles only SI and uses the coloring technique of [Cohen 1991], which is prefered to Schubert's numbering as it is incremental. *Interval containment* [Muthukrishnan and Muller 1996] uses Schubert's numbering. Type prediction techniques use equality tests.

Mixed techniques. There are many ways to mix table-based and tree-based techniques, by putting either tables in tree leaves [Queinnec 1997] or tree roots in table entries [Vitek and Horspool 1994]. *Type slicing* [Zibin and Gil 2002] is a generalization of Schubert's numbering and interval containment to MI. The main effect of those techniques is to save static memory w.r.t. standard table-based techniques: the size of static data structures may be quite smaller than $\Sigma_C M_C$. However, this is detrimental to time efficiency: as with type prediction, method call is no more constant-time. Moreover, unlike pure type prediction, a table access is required.

Attributes. The case of attributes is rarely treated in the literature. It may be explained by the fact that they are often encapsulated in the methods, as in SMALLTALK: so they are accessed only on `self`, which has the uncommon feature to be the only statically typed entity: Invariant 2.1.2 is thus verified for them, in case of SI. Accessor simulation is sufficient for MI but non encapsulated attributes require plain accessors, because of definition overloading.

7.3 Global optimizations

7.3.1 *Method copy.* When the source code of superclass methods is known at compile-time, it is possible to copy in a class the code of inherited and not overridden methods. Method copy has been already seen as an application to MI of the THETA technique (see section 5.1.4). The main advantage of this technique, termed *customisation* by [Chambers and Ungar 1989], is that `self` becomes monomorphic: method calls to `self`, which are quite numerous, can be compiled by a static call, without extra memory access. When the attributes are encapsulated as in SMALLTALK, attribute invariance does not matter and attribute offsets are always static. Otherwise, non `self` attribute accesses must be encapsulated by accessors generated or simulated by the compiler (see section 4.4). Calls to `super` or static calls must be inlined. Moreover, all methods which apply only to `self`—either by following some specific visibility keyword, or as deduced from a global analysis—may be removed from method tables. Finally, method copy gives more precise type informations, in case of type overriding, either safe as return type, or unsafe as parameter or attribute types. Anchored types may be replaced by constant types. However, only accesses to `self` are optimized. An implementation technique is required for the general case. Time efficiency is improved at the detriment of static space: method code is duplicated with a factor linear w.r.t. class number. Thus this technique cannot be envisaged without an association with optimizations reducing drastically the size of the generated code. Obviously, dead code must be ruled out.

7.3.2 *Automatic devirtualization.* It is a common opinion that repeated inheritance is an abomination and that the keyword `virtual` is justified only by efficiency. A static global analysis can determine which classes must be implemented as virtual, and which classes need not [Gil and Sweeney 1999; Eckel and Gil 2000]. More precisely, it can determine how to share subobjects. On the diamond example, it is possible to merge, on one hand, the subobjects D and B, on the other hand, the subobjects A and C, saving 3 method tables in Table II. A simple devirtualization scheme consists in marking as virtual all classes which have two direct subclasses with a common subclass, i.e. all A such that $\exists B, C, D : C \neq B, D \prec B \prec_d A$ and $D \prec C \prec_d A$. In a second step, one identifies virtual edges, in such a way that for each pair of related classes $D \prec A$, there is at most one non virtual edge towards A on all paths from D. After removing virtual edges, there remains only arborescent inheritance. Of course, the choice of the only non virtual edge is a matter of optimization.

7.3.3 *Type analysis.* The main objective of type analysis is to determine the *concrete type* of each program expression, i.e. the set of dynamic types that the expression will take for all possible executions. Without loss of generality, the problem is exponential—even undecidable as an exact answer poses the problem of program termination—but simplifying assumptions make it polynomial [Gil and Itai 1998]. Type analysis may be based on the construction of a call graph, but, with object-oriented languages, the two problems are in circular dependency: a call graph is needed to get precise concrete types, but a precise call graph requires concrete types, at least for receivers. As sound type analyses compute always an approximate (upper bound) of exact concrete types, they may be more or less precise and costly. [Grove and Chambers 2001] makes a review of different techniques. A classic compromise is *Rapid Type Analysis* (RTA) [Bacon and Sweeney 1996]. A secondary goal of type analysis is to type check programs, in case of downcasts or of unsafe type overriding, and it may save some dynamic type checks [Wang and Smith 2001].

7.3.4 *Dead code.* An interesting by-product of type analysis is the ability to distinguish living and dead classes and code. Indeed, the call graph associated to a type analysis makes explicit the classes which are never instanciated and the methods which are never called, in that they are unreachable from the `main` procedure. Type analysis is thus a good way to reduce the code size of applications. However, not all applications will benefit from it. This is not the case, for instance, of applications where class instanciation results from an interaction with a user, a DBMS, another program or a network. In such a context, all application classes are alive, as well as all their methods which can be activated in the same way.

7.3.5 *Inlining.* This is a common optimization of procedural languages: it consists in copying the code of a callee in the caller, for instance when the call is monomorphic and the method is small or not often called. In global compilation, this is a final optimization of monomorphic calls or type prediction.

7.3.6 Small Eiffel. The GNU Eiffel compiler is typical of the use of those global techniques in the framework of a statically typed language. It is based on a double a priori: global compilation without method tables. In the object layout,

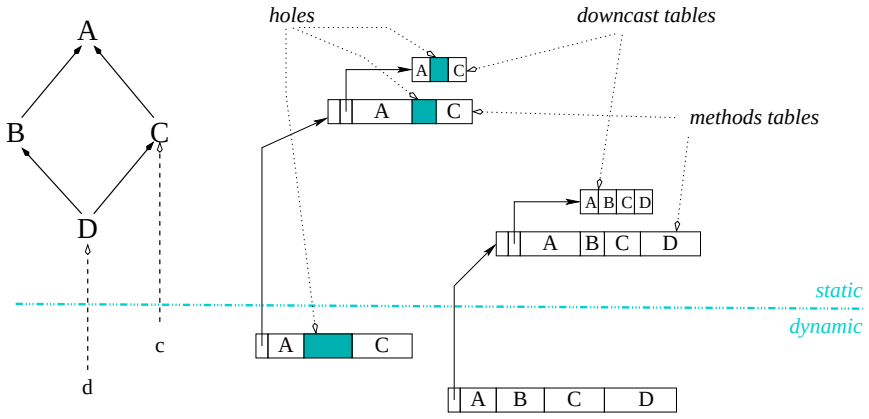


Fig. 14. Unidirectional coloring heuristics applied to classes, methods and attributes.

the pointer to method table is replaced by the class identifier. The compilers uses the following techniques:

(1) method copy which makes `self` monomorphic (customization);
(2) type analysis following the RTA algorithm, which determines concrete types of all expressions;
(3) dead code and classes are then ruled out;
(4) method calls still polymorphic after steps 1 and 2 are implemented by a small decision tree, based on equality tests on type identifiers; the same technique is used for polymorphic accesses to attributes, when the offset vary according to concrete types, as well as for downcasts;
(5) finally, inlining is done in many cases.

Recompilation speed is ensured by producing C code. The compilation of living code from Eiffel to C is systematic, but C files are recompiled only when they have been modified. [Zendra et al. 1997; Collin et al. 1997] describe in detail those techniques and experimental results which show a clear improvement w.r.t. existing Eiffel compilers.

7.4 Coloring heuristics

We detail now the coloring approach as it is quite versatile and it naturally extends to MI the SST implementation. Method coloring has been proposed by Dixon et al. [1989], under the name of selector coloring. One of the first experimentations, by André and Royer [1992], concluded to large coloring computation time and the technique was considered as uneffective ever since. However, a previous work by Pugh and Weddel [1990] had reported positive results, confirmed later by Ducournau [1991; 1997]. One may define coloring as upholding SST invariants 2.1.1 and 2.1.2 in MI.

Invariant 7.4.1. *An attribute (resp. method) has an offset (color) invariant by specialization. Two attributes (resp. methods) with the same color don't belong to the same class.*

Table V. Coloring bibliography

paper		optimization #colors	size	birect.	entities method	attribute	class	% holes
Dixon et al.	[1989]	×			×			
Pugh and Weddel	[1990]			×		×		6
Ducournau	[1991]		×			×		
André and Royer	[1992]	×			×			
Ducournau	[1997]		×		×			6
Vitek et al	[1997]		×				×	
Ducournau	[2002a]			×			×	45

As a corollary, two classes which have two different attributes (resp. methods) with the same color cannot be specialized by a common subclass. An injective numbering of attributes (resp. methods) verifies the invariant: therefore, an optimization is needed. The first proposition, by Dixon et al. [1989], André and Royer [1992], was to minimize the color number, which is an NP-hard graph coloring problem [Garey and Johnson 1979]. A first improvement, proposed by Pugh and Weddel [1990] and Ducournau [1991; 1997], consists in minimizing the total size of method (resp. attributes) tables: the tables resulting from coloring are then similar to SST tables, except that they may contain *holes*, i.e. empty entries. A second improvement, due to Pugh and Weddel [1990], is a bidirectional coloring, with positive and negative colors. The complexity of those new problems is not known but they are likely as difficult as the original problem. In any case, heuristics are needed and some experiments, by Pugh and Weddel [1990] and Ducournau [1997; 2002a], show their efficiency and that improvements are effective. Dixon et al. [1989], André and Royer [1992] applied first coloring to methods, whereas Pugh and Weddel [1990], and Ducournau [1991] applied it to attributes, and Vitek et al. [1997] and Ducournau [2002a] to classes, as a generalization of the SST technique of Cohen [1991]:

INVARIANT 7.4.2. *Each class has an offset (color). Two classes with the same offset have no common subclass.*

The time overhead of multiple inheritance vanishes but holes induce a small space overhead. Pugh and Weddel [1990] report a 6 % hole rate on a FLAVORS program with 563 classes and 2245 attributes. Ducournau [1997] reports the same hole rate on a SMALLTALK distribution with 698 classes and 4518 methods. However, class coloring experiment on large MI programs as Geode or Lov reports a 45 % hole rate, whereas the average hole rate is around 12 % [Ducournau 2002a]. For static tables (methods and classes), this overhead is insignificant compared to the cubic table size of standard implementation (see appendix A).

On the other hand, for dynamic memory, the overhead may be significant: one should minimize the total dynamic memory, which needs a profiling of class instanciation. A conservative solution will be to simulate accessors (see section 4.4) instead of coloring attributes: the offset of each subobject will be colored in the same way as methods. Dynamic memory overhead disappears, at the detriment of a constant time overhead for attribute accesses: it may be reduced by the double compilation of Myers [1995].

On the whole, coloring gives exactly the same implementation as standard SST implementation in case of SST: this corrects the main drawback of standard MI implementation. In case of MI, the overhead w.r.t. SST concerns only static tables

and accesses to some attributes: nevertheless this overhead remains far from all other MI implementations overhead (sections 3 and 4). Notice that row displacement [Driesen and Hölzle 1995] would give similar efficiency for method tables and subobject offsets: however, class coloring does not seem to have any equivalent.

7.5 Global techniques in separate compilation

Separate compilation is not incompatible with global techniques which can be applied at link or load time. Moreover, almost all implementations require a minimal global step, for instance for generating class identifiers by a global numbering.

7.5.1 *Double compilation.* The technique proposed by Myers [1995] implies a small global analysis to determine whether a class is always specialized as a primary one, or not. Double compilation can be based on other criteria, e.g. devirtualization, as long as they are invariant by specialization. It is only interesting for optimizing accesses to `self` and to other entities typed by the current class.

7.5.2 *Link time computation of method tables and object layout.* One can distinguish two different tasks in compilation. On one hand, it generates the method code. On the other hand, it generates the data structures associated to classes, together with object layouts, by fixing the size of all tables and the offset of all entities. Therefore, a general approach is to place the second task, either partly or entirely, at link time: linking will then have to substitute in the code the values of all the offsets computed after the compilation. For instance, this must be done for Schubert's numbering (2) if one wants a truly static cast, i.e. that n_1 and n_2 values are immediate values in the code. Of course, the code generated by separate compilation must be independant of this global step: in case of optimization, the worst case must always be foreseen.

Link time coloring. Coloring is a global technique but, as already noticed by Pugh and Weddel [1990], it could be computed at link time: effectively, coloring requires only the schema of classes, not their code, and the generated code depends only on individual colors associated to the different entities. Those colors are small integers which could be run-time constants computed at link-time. However the approach does not seem to have been experimented.

Link time devirtualization. Any global technique is not interesting at link time, after a separate compilation. Automatic devirtualization is a good counter-example. It may reduce the number of subobjects, thus the need to shift object references, but there is no way to know, at compile-time, whether a shift will be needed or not. Thus, the generated code will be the same as with standard MI implementation, with the only difference that, at run-time, most shifts would be null. Therefore, the only gain will be on the size of static tables: in case of SST, the layout will be the SST one (augmented with upcast tables) but the code will be the MI one. A double compilation—according to whether all superclasses are reachable through non virtual edges—could however improve the code.

The case of empty subobject optimization is quite different as both kinds of merging can be fixed at compile time: no link time processing is needed. At link time, one could only fix that some second kind merging is always possible, but it would have no effect neither on the code nor on the layout.

7.5.3 *Separate type analysis.* Type analysis requires source code, but it may split up into two phases: intra-class analysis and inter-class analysis. This is an object-oriented formulation of classic intra-procedural and inter-procedural analyses. Privat [2002] proposes to joint those two phases by a *template*, produced by an intra-class analysis during a separate compilation and which stands for an abstract of the class code, expliciting the flow of types in the methods. At link-time, the global inter-class analysis will use those templates to construct the call graph and determine concrete types for all expressions in the template. Such an approach would allow to detect dead code and to rule it out from the executable. However, local optimizations as monomorphic calls and inlining are not directly possible but an optimized version could use in separate compilation the result of the previous global analysis.

8. CONCLUSION

On the one hand, separate compilation of SST is simple and as efficient as possible: indirect method calls are a true overhead which could be only reduced with global compilation or an increasing of processors' capabilities for indirect branching prediction [Driesen 1999]. But SST expressivity if far from what programmers expect. On the other hand, separate compilation of plain MI presents a significant overhead w.r.t. SST: the main drawback of the standard implementation is that it is as costly when one does not use MI. Therefore, this is not surprising that recent efforts have focused on SI but MST languages, as Java or C#: this is a sound middle point between the two extremes, especially if compared to C++ NVI or mixins.

There is some evidence that standard MI implementation might be improved, for instance with the empty subobjects optimization, when it is not incompatible with VBPTRs. However, standard implementation is not the only approach to MI. Variants such as the Theta approach correct partially the main drawback of standard implementation but they don't seem to have been effectively used. Small Eiffel proved that quite different approaches are possible but, in that case, definitively incompatible with separate compilation. Global compilation is obviously the way to obtain the best efficiency but its drawbacks are numerous. Therefore, a general way to improve separate compilation is to introduce a touch of global techniques in the run-time production line. This was already the case with Theta. The coloring approach combined with type analysis is an appealing idea on paper: the run-time produced using those techniques would be even more efficient than with SST implementation, due to dead code detection. However the efficiency of the link-time global step must be established in practice: we are currently working towards a prototype.

Nevertheless, most global techniques have a significant drawback besides being global: they are not incremental and could not satisfy the specifications of abstract machines as Jvm or Clr which are mainly dedicated to mobile code.

Commonly used statically typed object-oriented languages—i.e. C++, Eiffel and Java—are quite perfectible: valuable suggestions include shared class attributes instead of `static` variables, method combination, or `self` encapsulation, besides `private` and `protected` keywords. The cost of implementing those features would not increase the overhead of object-orientation. This is not the case of type overriding which add some overhead especially with standard MI. Moreover,

when some language feature is considered as too expensive, the language should provide ways to restrict specialization: some classes might be specialized only in single or arborescent inheritance, some attribute or parameter types might not be overridden, and so on. Furthermore, in object orientation, modularity has been historically confused with the notion of class: however, the class is a conceptual unit, not always the program unit of the size appropriate for compilation. A higher level notion such as *modules* has been advocated for long [Szypersky 1992; Bracha and Lindstrom 1992]. Java *packages* are not a good answer as they are mainly designed as name spaces: however, current discussions on the notion of *sealing* may be a forward step towards true modules [Biberstein et al. 2001]. In any case, modular compilation of object-oriented languages will be a topic of further research. Hence, perspective is threefold: application of global techniques in separate compilation, modular compilation and global techniques compatible with incremental loading.

REFERENCES

Agesen, O., Freund, S., and Mitchell, J. 1997. Adding type parameterization to Java. See OOPSLA [1997], 49–65.

Alpern, B., Attanasio, C., Barton, J., Cocchi, A., Hummel, S., Lieber, D., Ngo, T., Mergen, M., Shepherd, J., and Smith, S. 1999. Implementing Jalapeño in Java. See OOPSLA [1999], 314–324.

Alpern, B., Cocchi, A., Fink, S., and Grove, D. 2001. Efficient implementation of Java interfaces: Invokeinterface considered harmless. See OOPSLA [2001].

Alpern, B., Cocchi, A., and Grove, D. 2001. Dynamic type checking in Jalapeño. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*.

America, P., Ed. 1991. *Proceedings of the Fifth European Conference on Object-Oriented Programming, ECOOP'91*. LNCS 512. Springer-Verlag.

Ancona, D., Lagorio, G., and Zucca, E. 2000. JAM — a smooth extension of Java with mixins. See Bertino [2000], 154–178.

André, P. and Royer, J.-C. 1992. Optimizing method search with lookup caches and incremental coloring. In *Proc. OOPSLA'92*. ACM Press, 110–126.

Bacon, D., Fink, S., and Grove, D. 2002. Space- and time-efficient implementation of the java object model. In *Proc. ECOOP'2002*, J. M. Troya, Ed. LNCS. Springer-Verlag.

Bacon, D. and Sweeney, P. 1996. Fast static analysis of C++ virtual function calls. In *Proc. OOPSLA'96*. SIGPLAN Notices, 31(10). ACM Press, 324–341.

Bertino, E., Ed. 2000. *Proceedings of the 14th European Conference on Object-Oriented Programming, ECOOP'2000*. LNCS 1850. Springer-Verlag.

Biberstein, M., Gil, J., and Porat, S. 2001. Sealing, encapsulation and mutability. In *Proc. ECOOP'2001*, J. L. Knudsen, Ed. LNCS 2072. Springer-Verlag, 28–52.

Boehm, H.-J. 1993. Space-efficient conservative garbage collection. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'93)*. ACM SIGPLAN Notices, 28(6). 197–206.

Bracha, G. and Cook, W. 1990. Mixin-based inheritance. In *Proc. OOPSLA/ECOOP'90*. SIGPLAN Notices, 25(10). ACM Press, 303–311.

Bracha, G. and Lindstrom, G. 1992. Modularity meets inheritance. In *Proc. Int. Conf. on Computer Languages*. IEEE, 282–290.

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. 1998. Making the future safe for the past: Adding genericity to the Java programming language. See OOPSLA [1998], 183–200.

Bruce, K. B., Petersen, L., and Fiech, A. 1997. Subtyping is not a good "match" for object-oriented languages. In *Proc. ECOOP'97*, M. Aksit and S. Matsuoka, Eds. LNCS 1241. Springer-Verlag, 104–127.

Calder, B. and Grunwald, D. 1994. Reducing indirect function call overhead in c++ programs. In *Proc. of ACM Symp. on Principles of Prog. Lang. (POPL'94)*. 397–408.

Cargill, T. A. 1991. Controversy: The case against multiple inheritance in C++. *Computing Systems 4,* 1, 69–82.

Cartwright, R. and Steele, G. 1998. Compatible genericity with run-time types for the Java programming language. See OOPSLA [1998], 201–215.

Castagna, G. 1995. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems 17,* 3, 431–437.

Castagna, G. 1997. *Object-Oriented Programming: A Unified Foundation*. Progress in Theorical Computer Science. Birkhäuser, Berlin.

Chambers, C. and Ungar, D. 1989. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented language. In *Proc. OOPSLA'89*. ACM Press, New Orleans, 146–160.

Cohen, N. 1991. Type-extension type tests can be performed in constant time. *Programming languages and systems 13,* 4, 626–629.

Collin, S., Colnet, D., and Zendra, O. 1997. Type inference for late binding. the SmallEiffel compiler. In *Joint Modular Languages Conference*. LNCS 1204. Springer Verlag, 67–81.

Colnet, D., Coucaud, P., and Zendra, O. 1998. Compiler support to customize the mark and sweep algorithm. In *ACM Sigplan International Symposium on Memory Management (ISMM'98)*. 154–165.

Cook, W. R. 1989. A proposal for making Eiffel type-safe. In *Proc. ECOOP'89*, S. Cook, Ed. Cambridge University Press, 57–70.

Day, M., Gruber, R., Liskov, B., and Myers, A. 1995. Subtypes vs. where clauses. constraining parametric polymorphism. In *Proc. OOPSLA'95*. ACM Press, 156–168.

Dixon, R., McKee, T., Schweitzer, P., and Vaughan, M. 1989. A fast method dispatcher for compiled languages with multiple inheritance. In *Proc. OOPSLA'89*. ACM Press.

Driesen, K. 1999. Software and hardware techniques for efficient polymorphic calls. Ph.D. thesis, University of California, Santa Barbara.

Driesen, K. and Hölzle, U. 1995. Minimizing row displacement dispatch tables. In *Proc. OOPSLA'95*. SIGPLAN Notices, 30(10). ACM Press, 141–155.

Driesen, K., Hölzle, U., and Vitek, J. 1995. Message dispatch on pipelined processors. In *Proc. ECOOP'95*, W. Olthoff, Ed. LNCS 952. Springer-Verlag, 253–282.

Ducournau, R. 1991. *Yet Another Frame-based Object-Oriented Language: YAFOOL Reference Manual*. Sema Group, Montrouge, France.

Ducournau, R. 1997. La compilation de l'envoi de message dans les langages dynamiques. *L'Objet 3,* 3, 241–276.

Ducournau, R. 2002a. La coloration pour l'implémentation des langages à objets à typage statique. In *Actes LMO'2002* in *L'Objet vol. 8*, M. Dao and M. Huchard, Eds. Hermès, 79–98.

Ducournau, R. 2002b. "Real World" as an argument for covariant specialization in programming and modeling. In *Advances in Object-Oriented Information Systems, OOIS'02 Workshops Proc.*, J.-M. Bruel and Z. Bellahsène, Eds. LNCS 2426. Springer-Verlag, 3–12.

Ducournau, R., Habib, M., Huchard, M., and Mugnier, M.-L. 1994. Proposal for a monotonic multiple inheritance linearization. In *Proc. OOPSLA'94*. ACM Press.

Eckel, N. and Gil, J. 2000. Empirical study of object-layout and optimization techniques. See Bertino [2000], 394–421.

Ellis, M. and Stroustrup, B. 1990. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading (MA), USA.

Ernst, E. 2002. Call by declaration. In *Proc. ECOOP'2002 Inheritance Workshop*, A. P. Black, E. Ernst, P. Grogono, and M. Sakkinen, Eds. University of Jyväskylä, 44–50.

Gagnon, E. M. and Hendren, L. 2001. SableVM: A research framework for the efficient execution of Java bytecode. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*.

Garey, M. and Johnson, D. 1979. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco (CA), USA.

Gil, J. and Itai, A. 1998. The complexity of type analysis of object oriented programs. In *Proc. ECOOP'98*. LNCS 1445. Springer-Verlag, 601–634.

Gil, J. and Sweeney, P. 1999. Space and time-efficient memory layout for multiple inheritance. See OOPSLA [1999], 256–275.

Goldberg, A. and Robson, D. 1983. *Smalltalk-80, the Language and its Implementation*. Addison-Wesley, Reading (MA), USA.

Gosling, J., Joy, B., and Steele, G. 1996. *The JAVA Language Specification*. Addison-Wesley.

Grand, M. 1997. *JAVA Language Reference*. O'Reilly.

Grove, D. and Chambers, C. 2001. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst. 23,* 6, 685–746.

Hölzle, U., Chambers, C., and Ungar, D. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. See America [1991], 21–38.

Huchard, M. and Leblanc, H. 2000. Computing interfaces in Java. In *Proc. of IEEE Int. Conf. on Automated Software Engineering (ASE'2000)*. 317–320.

Jones, R. and Lins, R. 1996. *Garbage Collection*. Wiley.

Koenig, A. 1998. Standard – the C++ language. Report ISO/IEC 14882:1998, Information Technology Council (NCTIS). `http://www.nctis.org/cplusplus.htm`.

Krall, A. and Grafl, R. 1997. CACAO - a 64 bits JavaVM just-in-time compiler. *Concurrency: Practice ans Experience 9,* 11, 1017–1030.

Kristensen, B., Lehrmann Madsen, O., Møller-Pedersen, B., and Nygaard, K. 1987. The BETA programming language. In *Research Directions in Object Oriented Programming*, B. Shriver and P. Wegner, Eds. MIT Press, Cambridge (MA), USA, 7–48.

Krogdahl, S. 1985. Multiple inheritance in Simula-like languages. *BIT 35*, 318–326.

Lalande, A. 1926. *Vocabulaire technique et critique de la philosophie*. Presses Universitaires de France, Paris.

Lippman, S. 1996. *Inside the C++ Object Model*. Addison-Wesley, New York.

Liskov, B., Curtis, D., Day, M., Ghemawat, S., Gruber, R., Johnson, P., and Myers, A. C. 1995. THETA reference manual. Technical report, MIT.

Meyer, B. 1992. *Eiffel: The Language*. Prentice Hall Object-Oriented Series. Prentice Hall International, Hemel Hempstead, UK.

Meyer, B. 1997. *Object-Oriented Software Construction*, second ed. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA.

Meyer, B. 2001. Overloading vs. object technology. *Journal of Object-Oriented Programming 14,* 5 (October/November), 3–7.

Meyer, J. and Downing, T. 1997. *JAVA Virtual Machine*. O'Reilly.

Microsoft. 2001. C# Language specifications, v0.28. Tech. rep., Microsoft Corporation.

Muthukrishnan, S. and Muller, M. 1996. Time and space efficient method lookup for object-oriented languages. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*. ACM/SIAM, 42–51.

Myers, A. 1995. Bidirectional object layout for separate compilation. In *Proc. OOPSLA'95*. ACM Press, 124–139.

Odersky, M. and Wadler, P. 1997. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*. ACM Press, New York (NY), USA, 146–159.

OOPSLA 1997. *Proceedings of the Twelfth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'97*. SIGPLAN Notices, 32(10). ACM Press.

OOPSLA 1998. *Proceedings of the Thirteenth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'98*. SIGPLAN Notices, 33(10). ACM Press.

OOPSLA 1999. *Proceedings of the Fourteenth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'99*. SIGPLAN Notices, 34(10). ACM Press.

OOPSLA 2001. *Proceedings of the Sixteenth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'01*. SIGPLAN Notices, 36(10). ACM Press.

Privat, J. 2002. Analyse de types et graphe d'appels en compilation séparée. M.S. thesis, Université Montpellier II.

Pugh, W. and Weddel, G. 1990. Two-directional record layout for multiple inheritance. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'90)*. ACM SIGPLAN Notices, 25(6). 85–91.

Queinnec, C. 1997. Fast and compact dispatching for dynamic object-oriented languages. *Information Processing Letters*.

Raynaud, O. and Thierry, E. 2001. A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. In *Proc. ECOOP'2001*. LNCS 2072. Springer-Verlag, 165–180.

Rossie, J. G., Friedman, D. P., and Wand, M. 1996. Modeling subobject-based inheritance. In *Proc. ECOOP'96*, P. Cointe, Ed. LNCS 1098. Springer-Verlag.

Sakkinen, M. 1992. A critique of the inheritance principles of C++. *Computing Systems 5,* 1, 69–110.

Schubert, L., Papalaskaris, M., and Taugher, J. 1983. Determining type, part, color and time relationships. *Computer 16,* 10, 53–60.

Simon, R., Stapf, E., Mingins, C., and Meyer, B. 2000. Eiffel for e-commerce under .NET. *Journal of Object-Oriented Programming 13,* 5 (October), 42–47.

Snyder, A. 1991. Modeling the C++ object model: An application of an abstract object model. See America [1991], 1–20.

Solorzano, J. H. and Alagić, S. 1998. Parametric polymorphism for Java: A reflective solution. See OOPSLA [1998], 216–225.

Steele, G. 1990. *Common Lisp: The Language, Second Edition*. Digital Press, Bedford (MA), USA.

Stefik, M. and Bobrow, D. 1986. Object-oriented programming: Themes and variations. *The AI Magazine 6,* 4, 40–62.

Sweeney, P. and Burke, M. 1998. A methodology for quantifying and evaluating the space overhead in C++ object models. Tech. Rep. RC21370, IBM T.J. Watson Research Center.

Szypersky, C. 1992. Import is not inheritance. Why we need both: Modules and classes. In *Proc. ECOOP'92*, O. L. Madsen, Ed. LNCS 615. Springer-Verlag, 19–32.

Szypersky, C., Omohundro, S., and Murer, S. 1994. Engineering a programming language: The type and class system of Sather. In *Proc. of First Int. Conference on Programming Languages and System Architectures*. LNCS 782. Springer Verlag.

Vitek, J. and Horspool, R. 1994. Taming message passing: efficient method look-up for dynamically typed languages. In *Proc. ECOOP'94*, M. Tokoro and R. Pareschi, Eds. LNCS 821. 432–449.

Vitek, J., Horspool, R., and Krall, A. 1997. Efficient type inclusion tests. See OOPSLA [1997], 142–157.

Waldo, J. 1991. Controversy: The case for multiple inheritance in C++. *Computing Systems 4,* 2, 157–171.

Wang, T. and Smith, S. 2001. Precise constraint-based type inference for java. In *Proc. ECOOP'2001*. LNCS 2072. Springer-Verlag, 99–117.

Weber, F. 1992. Getting class correctness and system correctness equivalent — how to get covariant right. In *Technology of Object-Oriented Languages and Systems (TOOLS 8)*, R. Ege, M. Singh, and B. Meyer, Eds. 192–213.

Wilson, P. R. 1992. Uniprocessor garbage collection techniques. In *Int. Workshop on Memory Management (IWMM'92)*. LNCS 637. Springer Verlag.

Zendra, O., Colnet, D., and Collin, S. 1997. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proceedings of OOPSLA'97, Atlanta (GA), USA*. special issue of ACM SIGPLAN Notices, 32(10). 125–141.

Zibin, Y. and Gil, J. 2001. Efficient subtyping tests with PQ-encoding. See OOPSLA [2001], 96–107.

Zibin, Y. and Gil, J. 2002. Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. In *Proc. OOPSLA'02*. SIGPLAN Notices, 37(10). ACM Press.

Table VI. Statistics on classes, including empty and virtual classes, and total and virtual inheritance edges

	classes			edges	
name	total	empty	virtual	total	virtual
SmallEiffel	382	27	91	4	416	13
JDK.1.0.2	604	96	153	5	650	20
digitalk3	1357	430	539	0	1356	0
digitalk2	535	156	235	0	534	0
Unidraw	614	115	81	3	623	4
Lov-obj-ed	436	39	132	21	747	68
Geode	1318	163	436	78	2486	251

APPENDIX

A. SPACE BENCHMARKS

Some large benchmarks are commonly used in the object-oriented implementation community[15], e.g. by [Driesen and Hölzle 1995; Vitek et al. 1997; Eckel and Gil 2000; Zibin and Gil 2002]. We present here some statistics computed from these benchmarks according to various implementation techniques.

Four techniques have been considered: ideal SST implementation, even if non applicable in all benchmarks, pure NVI, the simple devirtualization scheme of section 7.3.2 (DVI), MI with the empty subobject optimization (ESO) and standard MI (SMI). DVI may be understood as the best C++-like implementation, with a sound multiple inheritance semantics, for a hand made program, with a complete knowledge of the whole program. On the contrary, ESO is presumably the best standard MI implementation in separate compilation.

A.1 Benchmark description, interpretation and correction

Each benchmark is a file of class descriptions, each of which consists of four items: the class name, the list of its direct superclasses, the two lists of attributes and methods defined in the class. They have been produced and used mostly for assessing techniques for subtyping test and method call, so they often don't comport informations about attributes. Therefore, we restrict hereafter our statistics to the few of them which comport such data. Digitalk is a SMALLTALK distribution[16], whereas Unidraw is a C++ program mainly in SI, Lov and Geode are EIFFEL-like programs making an intensive use of MI.

The contents of the benchmarks is also questionable. One should expect that only pure object-oriented data are included: typically, `static` methods and variables, or non `virtual` methods should be excluded. One will see that this is not the case with the JDK benchmark which includes obviously `static` variables and we can only certify that non object-oriented data has been removed from the SMALL EIFFEL benchmarks. Besides that, name interpretation may be discussed as it is different between the different languages: in C++ and JAVA benchmarks, parameter types have been concatenated to method names in order to deal with static overloading. However, the case of attribute is not clear: the same attribute name in two related

[15] Many people contributed to those benchmarks, among which Karel Driesen and Jan Vitek: a current repository is Yoav Zibin's web site, `http://www.cs.technion.ac.il/~zyoav/`.
[16] As SMALLTALK is dynamically typed, these statistics are not directly applicable.

Table VII. Statistics (total number, mean and max) on attribute number, either introduced or inherited

name	introduced			inherited		
SmallEiffel	1317	3.4	39	3441	9.0	46
JDK.1.0.2	2192	3.6	274	90667	150.1	600
digitalk3	1495	1.1	37	11352	8.4	52
digitalk2	529	1.0	30	3885	7.3	30
Unidraw	1574	2.6	36	5101	8.3	47
Lov-obj-ed	1262	2.9	74	3552	8.1	105
Geode	2919	2.2	182	14355	10.9	217

classes has been interpreted as overriding (as in EIFFEL), not as overloading (as in C++ or JAVA): hence, Unidraw and JDK may be slightly underestimated. Renaming, which is proper to EIFFEL, has not been considered in the SMALL EIFFEL benchmark[17], and no information is available about Lov or Geode.

Moreover, these benchmarks are often libraries, not single applications: it is thus difficult to extrapolate from them the size appropriate for typical applications and to judge the maximal number of classes, methods, attributes, etc. of applications. Anyway, some hundreds of classes seem common in object-oriented programs, e.g. the SMALL EIFFEL compiler [Zendra et al. 1997].

Finally, the statistics may differ from previous ones in several ways. Some are connected to the fact that this paper aims at separate compilation and static typing. First, there is no definition overloading (see note 3, page 6): two methods or attributes with the same name, introduced in two unrelated classes, are considered as different[18]. Second, all methods are considered, whereas Yoav Zibin's benchmarks remove *degenerate methods*, i.e. methods which have only one definition. Furthermore, some measures are uniquely determinate (NVI, SMI) whereas many other (ESO, DVI, coloring) are the result of an optimization problem: heuristics commonly used are greedy and there is a very little chance that two different experiments give the same number.

However, besides all those small flaws, the statistics presented hereafter are a somewhat precise indication of the relative cost of the measured techniques.

A.2 Dynamic space

Table VI presents the number of classes, together with the number of classes without attributes, in the two kinds of merging (see section 3.3), and the number of virtual classes according to the DVI scheme (see section 7.3.2). The number of classes without attributes is surprisingly high in all benchmarks, between 30 and 60 %. It was unexpected and it will have an important effect on the number of subobjects as empty subobjects may be merged into another one. On the contrary, there are very few virtual classes and virtual edges.

Table VII presents statistics on the number of attributes, as well introduced in a class as the total number for the class. JAVA figures obviously deviate from other

[17] In EIFFEL, it may be difficult to distinguish attributes and methods: constant features, i.e. features whose body is a constant, have been treated as methods, except when they are `frozen`.
[18] In JAVA, the same method may be introduced in two unrelated interfaces: thus, the JDK data may be slightly overestimated.

Table VIII. Statistics (total, mean and maximum) on subobject numbers

name	NVI			DVI			ESO			SMI		
SmallEiffel	809	2.1	8	548	1.4	6	1348	3.5	6	2957	7.7	13
JDK.1.0.2	792	1.3	6	698	1.2	4	1927	3.2	8	2802	4.6	14
digitalk3	1357	1.0	1	1357	1.0	1	4089	3.0	8	8673	6.4	14
digitalk2	535	1.0	1	535	1.0	1	1497	2.8	7	3228	6.0	12
Unidraw	634	1.0	4	683	1.1	3	1934	3.1	7	2468	4.0	10
Lov-obj-ed	1846	4.2	19	1490	3.4	19	2352	5.4	17	3707	8.5	24
Geode	13798	10.5	140	6030	4.6	36	11249	8.5	33	18442	14.0	50

Table IX. Statistics on upcast table size (total (u), mean and max)

name	DVI			ESO			SMI		
SmallEiffel	198	1.	9	4595	12.	44	10532	28.	76
JDK.1.0.2	36	0.	3	4992	8.	49	6465	11.	61
digitalk3	0	0.	0	20315	15.	55	26633	20.	91
digitalk2	0	0.	0	7160	13.	45	9198	17.	66
Unidraw	56	0.	1	3310	5.	28	4711	8.	45
Lov-obj-ed	1167	3.	28	10111	23.	117	11503	26.	118
Geode	19285	15.	161	89547	68.	481	94992	72.	490

statistics: the reason is that static variables are taken into account, as well as attributes.

Attribute number is to compare with the number of subobjects, i.e. pointers to method tables in the object layout: this gives a good idea of the dynamic overhead of MI implementations (Table VIII). On the whole, pointers to method tables are more than doubling dynamic space as soon as inheritance is intensively used (Lov and Geode) but ESO (resp. DVI) reduces the overhead by 40 % (resp. 70 %). Of course, DVI is better than any other technique, as it is near optimal. One should not be surprised by the fact that NVI is worse than ESO in the case of Geode: the worst-case complexity of NVI is exponential. Furthermore, empty subobjects bring no optimization to NVI.

Table IX shows the size occupied either by upcast tables, in method tables, or by VBPTRs in object layout (see section 4.3). Besides JAVA because of static variables, this demonstrates that VBPTRs have a large overhead. ESO has a poor effect: this is due to the fact that more than half of empty classes are of the second kind, which does not save on upcast tables. On the contrary, DVI is a great improvement, even if, in the worst case, the total of upcast tables and number of subobjects is almost 2 times greater than the number of attributes, with a 180 % overhead. In the case of ESO, the overhead is 600 %. However, if DVI is not hand made but produced by an automatic devirtualization at link time, the upcast tables should be the same as with SMI or ESO, unless offsets $i_{\tau_s}(C)$ are computed at link time. In that case, an extra entry for the null shifts in each subobject will be sufficient.

On the whole, MI overhead in the object layout may be less than the one that previous works as [Sweeney and Burke 1998; Gil and Sweeney 1999; Eckel and Gil 2000] report, as they consider only the hierarchy structure whereas data on attributes and methods is needed to deal with empty subobjects. As a matter of fact, though most benchmarks are common, it is impossible to compare our statistics with those of Eckel and Gil [2000] because they don't include any per benchmark statistics of either the subobject number or the upcast tables size.

Table X. Statistics on method number

	introduced			defined			inherited		
name	total	mean	max	total	mean	max	total	mean	max
SmallEiffel	3832	10.0	209	6482	17.0	209	37658	98.6	277
JDK.1.0.2	3190	5.3	75	5095	8.4	78	22348	37.0	156
digitalk3	13007	9.6	440	17104	12.6	460	613675	452.2	1065
digitalk2	5536	10.3	271	6858	12.8	272	154616	289.0	677
Unidraw	1751	2.9	103	3327	5.4	103	14167	23.1	123
Lov-obj-ed	3631	8.3	117	5026	11.5	127	36883	84.6	287
Geode	8078	6.1	193	14214	10.8	207	302742	229.7	870

Table XI. Statistics on table size (in K-entries)

name	SST (s)	NVI (nv)	DVI (dv)	ESO (m)	SMI (m')	nv/s	dv/s	m/dv	m'/m	m/s
SmallEiffel	38	61	48	100	188	1.6	1.3	2.1	1.9	2.7
JDK.1.0.2	22	25	24	55	77	1.1	1.1	2.2	1.4	2.4
digitalk3	614	614	614	1739	3154	1.0	1.0	2.8	1.8	2.8
digitalk2	155	155	155	391	729	1.0	1.0	2.5	1.9	2.5
Unidraw	14	14	14	31	40	1.0	1.0	2.2	1.3	2.2
Lov-obj-ed	37	104	80	122	183	2.8	2.2	1.5	1.5	3.3
Geode	303	1348	827	1235	1894	4.5	2.7	1.5	1.5	4.1

A.3 Static space

Table X presents statistics on the number of methods, as well introduced or defined in a class as the total number for the class.

According to the different techniques, the total size of method tables may be computed by the following formulas, where m_C is the number of methods introduced by C:

$$\sum_C f_C, \text{ where } f_C = \begin{cases} M_C = \sum_{C \preceq D} m_D & \text{SST} \\ m_C + \sum_{C \prec_d D} f_D & \text{NVI} \\ m'_C + \sum_{C \prec_{nv} D} f_D + \sum_{C \preceq_v D} f_D & \text{DVI} \\ \sum_{C \preceq D} M_D & \text{SMI} \\ \sum_{C \preceq_{eso} D} M_D & \text{ESO} \end{cases} \tag{13}$$

In the DVI case, m'_C is the number of method of C which are not known by the superclass of D that D extends: $\prec_{nv}$ is the subset of non virtual edges in $\prec_d$, and $C \preceq_v D$ iff $C \preceq D$ and $\nexists D', C \preceq D' \prec_{nv} D$. The SMI case can be rewritten as $\sum_D n_D M_D$, where n_D is the number of D subclasses, including D. The ESO case amounts to exclude merged classes from the sum on D: $C \preceq_{eso} D$ means that $C \preceq D$ and there is no D' merged to D, i.e. empty and such that $C \preceq D' \prec_d D$.

For an exact assessment of MI overhead, one must take into account shifts to receivers, which can be handled by thunks or by 2 extra instructions in the code plus a 50 % increasing of table size (if one uses only short integers). The thunk number is exactly the difference of table sizes between the considered MI technique and SST: there is a thunk per table entry, but for each class-method pair there is exactly one null shift. One must also add, from Table IX, the upcast tables size (u) to SMI, ESO and DVI sizes.

Table XI shows the total size, i.e. number of entries, of method tables in all techniques. The ratio between ESO and ideal SST table sizes is around 3-4. When

Table XII. Estimate of code size for method calls (in K-entries)

name	classes	methods	calls (c)	code ($3c$)	tables (m)	shifts	thunks
SmallEiffel	382	6482	25928	78	100	102	124
JDK.1.0.2	604	5095	20380	61	55	68	64
digitalk3	1357	17104	68416	205	1739	1006	2251
digitalk2	535	6858	27432	82	391	250	472
Unidraw	614	3327	13308	40	31	42	33
Lov-obj-ed	436	5026	20104	60	122	101	170
Geode	1318	14214	56856	171	1235	731	1864

Table XIII. Final comparison between SST, NVI, coloring, and between shifts and thunks in the ESO case (in K-entries)

	total size						ratio vs. SST				
name	SST	NVI	DVI	color	shifts	thunks	NVI	DVI	color	shifts	thunks
SmallEiffel	115	186	146	136	282	304	1.6	1.3	1.2	2.4	2.6
JDK.1.0.2	83	93	90	97	186	182	1.1	1.1	1.2	2.2	2.2
digitalk3	819	819	819	823	2961	4206	1.0	1.0	1.0	3.6	5.1
digitalk2	237	237	237	239	727	949	1.0	1.0	1.0	3.1	4.0
Unidraw	54	54	54	63	114	105	1.0	1.0	1.2	2.1	1.9
Lov-obj-ed	97	297	227	118	288	357	3.1	2.3	1.2	3.0	3.7
Geode	473	3608	2056	639	2181	3313	7.6	4.3	1.3	4.6	7.0

MI is intensively used, the difference between ESO and DVI is significant but less than expected: in fact, ESO improves upon SMI with the same ratio as DVI upon ESO. The advantage of DVI w.r.t. SMI or ESO lies in the null thunks which may be measured by the ratio $(dv-s)/(m-s)$: this ratio is around 50 % in case of intensive use of MI (Geode and Lov). As for pure NVI, besides its unsound semantics, its bad worst-case complexity tends to reduce the gap with SMI and ESO. NVI may be worse than ESO or SMI: in Geode, the maximal number of method tables is far greater with NVI than with SMI (Table IX), and the average method table number and size are greater than with ESO.

A conservative estimate of method coloring consists in adding to SST table size the superclass number (*sp*, i.e. the subobject number for SMI, Table VIII) for simulating accessors, and in majoring the total by 50 %, which is an upper bound for all experiments. In case of SI hierarchies (Digitalk), the 50 % majoration is not needed as coloring is exactly SST.

Code size. Statistics on the number of method call sites in the method code are missing, as well as for any other mechanism: hence, there is no way to measure the static space occupied by the different techniques in the method code. However, previous studies show that the number of call sites may be large enough to make code size significant w.r.t. table size [Driesen et al. 1995; Ducournau 1997]. For instance, a number of 35042 call sites is reported for a SMALLTALK implementation with 774 classes and 8540 method definitions. Assuming an average number of 4 call sites per method definition—which may be quite unlikely in C++ or EIFFEL programs—Table XII shows that the 3 instruction sequence code for method call in standard SST has a rather significant global impact on static space. For assessing the static space overhead or gain of thunks, one must compare the cost of the shifts in the code and in the table, $x = 2c + m/2$, with the cost of thunks, $x = 2(m - s)$. The total static size will be obtained with $3c+m+x+u/2$, which must be compared

Table XIV. Statistics on table size, when all non leaf classes are abstract (in K-entries)

name	SST (s)	NVI (nv)	DVI (dv)	ESO (m)	SMI (m')	nv/s	dv/s	m/dv	m'/m	m/s
SmallEiffel	30	49	33	82	153	1.6	1.1	2.5	1.9	2.7
JDK.1.0.2	17	19	18	43	60	1.2	1.1	2.4	1.4	2.6
digitalk3	411	411	411	1167	2178	1.0	1.0	2.8	1.9	2.8
digitalk2	109	109	109	276	524	1.0	1.0	2.5	1.9	2.5
Unidraw	11	11	11	23	31	1.0	1.0	2.2	1.3	2.2
Lov-obj-ed	21	63	37	75	111	3.0	1.8	2.0	1.5	3.6
Geode	175	790	374	693	1083	4.5	2.1	1.9	1.6	4.0

to $3c + 3s/2 + sp/2$ which is an upper bound for coloring, and to $3c + 3dv - 2s$ for DVI (Table XIII)[19]. Thus, even when taking into account the size of the code, which is advantageous for thunks, the thunk space overhead remains greater.

One must notice, from Tables VII and X, that the number of attributes and methods in a class is far from 2^{15}: thus, an implementation of Δs and offsets with short integers is valid. This explains why sp and u are divided by 2 in the previous formula. However, treating attribute and method offsets, or class identifiers, as immediate values may depend on the processor. According to Driesen et al. [1995], processors may offer from 8 to 13 bits for immediate values: 13 bits are sufficient for treating all that data as immediate values, but 8 bits are not enough for large applications. This is not a problem when offsets are computed at compile-time: the code will then differ according to the offset value, with a small overhead when the value is greater than 128. However, when offsets are computed at link-time, the compiler should be able to predict whether the link-time computation will always produce a 8-bit value: thus, the overhead will be larger.

Abstract classes. Many other data are missing, for instance the number of abstract classes, which would reduce the static space estimate, as in the Σ_C formula (13), C should exclude abstract classes. When the class number is large, it is likely that many classes are abstract: the effect of a precise measure might be as significant as for classes without attributes. A common assumption, which may be far from reality, is that only leaves may have instances. Table XIV presents the same statistics as Table XI, according to this assumption. The effect of abstract classes is important, but one must notice that the variation of the ratios between the different implementations is not significant.

Time efficiency. Assessing time efficiency from those spacial benchmarks requires quite hypothetical assumptions. The time overhead is mostly in the shifts, either static or dynamic. Thus the subobjects number is a good indication of null shifts (Table VIII), whereas upcast tables size is a good indication of the number of shifts that are statically avoided (Table III). For method calls, one can compare thunks with the other technique: if one assumes that all table entries are equiprobable, the proportion of null shifts in method calls is exactly the ratio of SST table size (s) and the size of tables in the considered technique. Therefore, with ESO, thunks save on between 23 % and 40 % of all shifts in method calls, whereas, with DVI from 50 % to 100 % are saved.

[19] One assumes here that code instructions and addresses have the same size: this may be not the case on some 64-bit processors.

B. PSEUDO-CODE SURVEY

The processor specifications are mostly the same as in Driesen [1995; 1999]: particularly, 2 `load` instructions cannot be executed in parallel, but need a one cycle delay.

B.1 Single subtyping

Method call:

```
load [object + #tableOffset], table
load [table + #selectorOffset], method                    2L + B
call method
```

`Tableoffset` is a constant for all types and methods whereas `selectorOffset` depends on the method.

Attribute access:

```
load [object + #attributeOffset], attribute                    L
```

`AttributeOffset` depends on the attribute.

Type check using Schubert's numbering:

```
load [object + #tableOffset], table
load [table + #n1Offset], classid
comp classid, #n1
blt #fail                                                 2L + 2
comp classid, #n2
bgt #fail
// succeed
```

`N1Offset` is a constant, whereas `n1` and `n2` depend on the target class. `Fail` is the address of the code which signals an exception, either shared by several or even all type checks, or proper to each one.

Type check using coloring:

```
load [object + #tableOffset], table
load [table + #targetColor], classId
comp classId, #targetId                                   2L + 2
bne #fail
// succeed
```

`TargetColor` and `TargetId` depend on the target class.

B.2 Multiple inheritance

Method call:

```
load [object + #tableOffset], table
load [table + #deltaOffset], delta
load [table + #selectorOffset], method                  2L + B + 1
add object, delta, object
call method
```

`DeltaOffset` and `selectorOffset` depend on both the method and the receiver's static type.

Thunks:

```
add object, #delta, object
jump #method
```

`Delta` and `method` depend on both the method and the receiver's static type. When method table points to the thunk, the code is that of single subtyping, whereas, when the thunk is inlined in the method table, it changes to:

```
load [object + #tableOffset], table
add table, #selectorOffset, method                    L + 1 + B
call method
```

Equality test, with unrelated types:

```
load [x + #tableOffset], table1
load [y + #tableOffset], table2
load [table1 + #dcastOffset], dx
load [table2 + #dcastOffset], dy                      2L + 3
add dx, x, x
add dy, y, y
comp x, y
```

`Dcastoffset` is a constant. When types are related, this simplifies to:

```
load [x + #tableOffset], table
load [table1 + #castOffset], dx
add dx, x, x                                          2L + 2
comp x, y
```

and `castOffset` depends on both types.

Upcast:

```
load [object + #tableOffset], table
load [table + #castOffset], delta                     2L + 1
add object, delta, target
```

`CastOffset` depends on the static types of both the source and the target.

Accesses to attributes

```
load [object + #tableOffset], table
load [table + #castOffset], delta
add object, delta, object                             3L + 1
load [object + #attributeOffset], attribute
```

`CastOffset` depends on both the receiver's static type (τ_s) and on the class introducing the attribute (T_p) whereas `attributeOffset` is an attribute invariant (δ_p).

Assignment $a.x := b.y$:

```
load [object1 + #tableOffset], table1
load [object2 + #tableOffset], table2
load [table1 + #cast1Offset], delta1
load [table2 + #cast2Offset], delta2
add object1, delta1, object1
add object2, delta2, object2                          5L + 3
load [object1 + #attribute1Offset], attribute
load [attribute + #tableOffset], table
```

```
load [table + #castOffset], delta
add attribute, delta, attribute
store attribute, [object2 + #attribute2Offset]
```

B.3 Alternatives

NVI code for method call is exactly the same as for MI, whereas NVI code for attribute access is the same as for SST: indeed, the required shift is static, so in the SST code, `attributeOffset` is augmented by the value of the shift.

Method call with shared tables:

```
load [object + #table0ffset], table1
load [table1 + #table20ffset], table2
load [table1 + #delta10ffset], delta1
load [table2 + #delta20ffset], delta2
add object, delta1, object                          3L + B + 1
load [table2 + #selector0ffset], method
add object, delta2, object
call method
```

Attribute accesses with VBPTRs:

```
load [object + #castOffset], object                 2L
load [object + #attributeOffset], attribute
```

Simulating accessors to attributes is exactly as with standard MI:

```
load [object + #table0ffset], table
load [table + #castOffset], place                   3L + 1
add object, place, place
load [place + #attributeOffset], attribute
```

B.4 Single inheritance multiple subtyping

Assignment $x := y$ with table caching, when y is interface-typed:

```
load [tableY + #interfaceOffset], tableX            L
move Y, X
```

When y is class-typed, the table should be first accessed by:

```
load [Y + #table0ffset], tableY
load [tableY + #interfaceOffset], tableX            2L
move Y, X
```

Method call to x is then:

```
load [tableX + #selector0ffset], method
call method
```

B.5 Complements

B.5.1 *Covariant overriding.* In single subtyping, for attributes, with Shubert's numbering:

```
load [object + #table0ffset], table1
load [val + #table0ffset], table2
load [table1 + #attributen10ffset], n1cible
```

```
load [table2 + #n1Offset], n1source
load [table1 + #attributen2Offset], n2cible
comp n1cible, n1source                                    2L + 4
blt #fail
comp n2cible, n1source
bgt #fail
store val, [object + #attributeOffset]
```

Covariant return type in multiple inheritance, in a thunk:

```
add object, #delta, object
call #method
load [return + #tableOffset], table
load [table + #castOffset], delta
add return, delta, return                    // casting from u to t
```

B.5.2 *Genericity.* Method call to formal type with offset conversion:

```
load [generic + #tableOffset], table1
load [object + #tableOffset], table2
load [table1 + #selectorOffset], offset                  3L + B + 1
add table2, offset, method
load method, method
call method
```

`Generic` is the current receiver, instance of a parameterized class, and `object` is an object typed by the formal type.

Attributes in multiple inheritance:

```
load [generic + #tableOffset], table1
load [object + #tableOffset], table2
load [table1 + #castOffset], delta
add table2, delta, table2                                  4L + 2
load [table2], place
add object, place, place
load [place + #attrOffset], attribute
```

B.6 Global techniques

B.6.1 *Type prediction.* When n types are predicted, with ordered identifiers t_1, ..., t_n, the code for method call will be a balanced binary tree:

```
load [object + #typeOffset], typeId
comp #t_k #typeId
bgt #sup_k
// case of t_1, ..., t_k
call #method_k
jump #next
#sup_k
// case of t_k+1, ..., t_n
call #method_n
#next
```

When $k = 1$ (resp. $n-1$), there is a static call to `#method_1` (resp. `#method_n`). The dichotomy depends on the choice of k. The ordered list $\{t_1, ..., t_n\}$ may be partioned

into segments of equal method values: the best is to choose k as an extremity of the median segment. When such a k does not exist, i.e. when all values are equal, method call is static. If one assumes a uniform probability p of right prediction for all conditional branches and that `method_k` is different of `method_k+1`, for all k, then the average time of method call will be $L + (1 + p + B(1 - p))log_2(n)$.

The case of attributes is the same.

B.6.2 *Coloring.* Coloring does not need a specific pseudo-code: it uses exactly that of single subtyping, except for attributes when the code for simulating accessors is used instead.