Implementing Statically Typed Object-Oriented Programming Languages

ROLAND DUCOURNAU
LIRMM – CNRS et Université Montpellier II, France

Object-oriented programming languages represent an original implementation issue due to the mechanism known as *late binding*, aka *message sending*. The underlying principle is that the address of the actually called procedure is not statically determined, at compile-time, but depends on the dynamic type of a distinguished parameter known as the *receiver*. In statically typed languages, the point is that the receiver's dynamic type may be a subtype of its static type. A similar issue arises with attributes, because their position in the object layout may depends on the object's dynamic type. Furthermore, subtyping introduces another original feature, i.e. subtype checks. All three mechanisms need specific implementations, data structures and algorithms. In statically typed languages, late binding is generally implemented with tables, called *virtual function tables* in C++ jargon. These tables reduce method calls to function calls, through a small fixed number of extra indirections. It follows that object-oriented programming yields some overhead, as compared to usual procedural languages.

The different techniques and their resulting overhead depend on several parameters. Firstly, inheritance and subtyping may be single or multiple and a mixing is even possible, as in Java, which presents single inheritance for classes and multiple subtyping for interfaces. Multiple inheritance is a well known complication. Secondly, the production of executable programs may involve various schemes, from global compilation frameworks, where the whole program is known at compile time, to separate compilation and dynamic loading, where each program unit—usually a class in an object-oriented context—is compiled and loaded independently of any usage. Global compilation is well known to facilitate optimization.

In this paper, we review the various implementation schemes available in the context of static typing and in the three cases of single inheritance, multiple inheritance, and single inheritance but with multiple subtyping, e.g. Java. The survey focuses on separate compilation and dynamic loading, as it is the most commonly used framework and the most demanding. However, many works have been recently undertaken in the global compilation framework, mostly for dynamically typed languages but also applied to the Eiffel language in the Smart Eiffel compiler. Hence, we examine global techniques and how they can improve implementation efficiency. Finally, a mixed framework is considered, where separate compilation is followed by a global step, similar to linking, which uses global techniques, as well for implementation, with coloring, as for optimization, with type analysis. An application to dynamic loading is sketched.

Categories and Subject Descriptors: D.3.2 [**Programming languages**]: Language classifications—*object-oriented languages*; C++; Java; Eiffel; Theta; D.3.3 [**Programming languages**]: Language constructs and features—*classes and objects*; *inheritance*; D.3.4 [**Programming languages**]: Processors—*compilers*; *linkers*; *loaders*; E.2 [**Data**]: Data Storage Representation—*object representation*

General Terms: Languages, Measurement, Performance

Author's address: R. Ducournau, LIRMM, 161, rue Ada – 34392 Montpellier Cedex 5, France

Additional Key Words and Phrases: casting, coloring, contravariance, covariance, downcast, dynamic loading, genericity, late binding, linking, message sending, method dispatch, multiple inheritance, object-oriented languages, single inheritance, static typing, type analysis, separate compilation, virtual function tables

1. INTRODUCTION

The main characteristic of object-oriented programming is expressed by the metaphor of message sending. Instead of applying a procedure or a function to an argument, a message is sent to an object—called the *receiver*, whereas the procedure or function is called a *method*—and the program behaviour, i.e. the code which will be executed, is determined by the receiver itself *at runtime*. In class-based languages, all proper instances of the same class share the same behaviour, hence message sending is interpreted according to the receiver's dynamic type. From an implementation standpoint, it follows that the *static* procedural call of procedural languages must be replaced by some *dynamic* call—control flow jumps to an address extracted from the receiver itself. This is called *late binding*. In separate compilation of statically typed languages, late binding is generally implemented with tables called *virtual function tables* in C++ jargon. Method calls are then reduced to function calls, through a small fixed number of extra indirections. On the other hand, an object—e.g. the receiver—is laid out as an attribute table, with a header pointing at the class table and some added information, e.g. for garbage collection. The cost of most implementations depends on inheritance: with single inheritance, the overhead is small, but multiple inheritance may increase it.

This paper describes the various schemes commonly used for implementing object-oriented specific mechanisms, together with some alternatives, and evaluates and compares them. The scope of this survey is restricted to: i) class-based object-oriented languages, i.e. prototype-based languages like Self [Ungar and Smith 1987; Agesen et al. 1995] will not be considered at all; ii) static typing, i.e. Smalltalk [Goldberg and Robson 1983] and Clos [Steele 1990] will not be considered, apart from a fast comparison; iii) separate compilation, i.e. other compilation frameworks will be considered but not fully surveyed.

Therefore, target languages are mostly C++ [Stroustrup 1998], C# [Microsoft 2001], Java [] and Eiffel [Meyer 1992; 1997]—other languages exist, but these four must be used by more than 90% of object-oriented programmers, at least when only static typing is involved. However, apart from these commonly used languages, new emerging ones might attract a large audience—e.g. Scala [Odersky et al. 2008].

1.1 Object-oriented mechanisms

This survey focuses on the core of object-oriented (OO) programming, the few features which require specific implementation:

—object layout together with read and write accesses to attributes,

—method invocation and *late binding* in its most common form of single dispatch, where the selection is based on one specific parameter, i.e. the receiver, which is

bound to a reserved formal parameter called `self`[1];

—dynamic type checking which is a basis of constructs like *downcast* or `typecase`: though most considered languages are presumed to be type safe, all offer such constructs, which are needed for covariant overriding or for filling the lack of genericity in JAVA (up to 1.4),

—instance creation and initialization, through special methods called *constructor* in C++ and JAVA jargon.

A number of secondary mechanisms must also be considered. They are more or less explicit in the language specifications, but they are all necessary. Some of them are not as trivial as they may seem:

—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`, may depend on its static type; variable assignments, parameter passing and equality tests are instances of this problem;

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

—call to `super`, a way for the overriding method to call the overridden one, is a special case of *method combination*;

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

—`null` value, for uninitialized variables and attributes, may lead to omnipresent, hence costly, tests, so looking for alternatives is worthwhile;

—parameterized classes are a common feature—called *templates* in C++ and newly introduced in JAVA 1.5—that we shall examine in the *bounded genericity* framework;

—most languages specify method invocation in the simple case of single dispatch, where the selection depends only on the receiver's dynamic type, while multiple dispatch—as used in CLOS [Steele 1990] and theorized by Castagna [1997]—is an interesting extension that must be considered, even though this is not yet a standard feature.

Finally, here we shall exclude all non OO features, i.e. all features whose specifications are based only on static types, hence whose behaviour is determined at compile-time:

—non `virtual` methods in C++, or `static` variables and functions in C++ and JAVA, are specified and implemented in the same way as in non OO languages—the only impact of object-orientation is the fact that classes are name spaces;

[1] `Self` is the reserved word used in SMALLTALK: it corresponds to `this` in C++ and JAVA and to `current` in EIFFEL [Meyer 1992; 1997]. Here we follow the SMALLTALK usage, which seems closer to the initial metaphor. `Self` can be considered as a reserved formal parameter of the method, and its static type—even in dynamically typed languages!—is the class within which the method is defined.

—*static overloading* (à la C++ or JAVA) involves using the same name for different methods defined in the same classes and distinguished by parameter types—ambiguities are statically solved by a selection which must not be confused with late binding, in a way equivalent to global renaming [Meyer 2001];

—specific features rule the access rights to some entity for the other ones—besides their various names like *protection*, *visibility* or *export*, they actually are only static access rights to existing implementation. Anyway, an exception will be made for SMALLTALK *encapsulation*, which reserves all accesses to attributes for `self`—this will be shown to be of interest for implementation.

Primitive type issues will also be mostly evaded, since static typing avoids coding the type in the value, as in dynamically typed languages. However the question of polymorphic use of a primitive value—e.g. binding a primitive value to a variable statically typed by the *universal type*, common super-type of primitive types and object classes—deserves some examination. Such a universal type—`any` in EIFFEL, `object` in C# and JAVA 1.5—does not exist in C++ or in JAVA up to version 1.4. We also consider 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 might be the object itself, as in C++ or EIFFEL with the keyword `expanded`: indeed, objects as immediate values forbid polymorphism for attributes and are incompatible with the notion of *object identity*[2]. This simplifying assumption has no consequence on implementation, as long as attributes have fixed size—on the contrary, varying size attribute values would pose problems which are not counterbalanced by the small gain achieved by avoiding one indirection.

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

This paper aims at reviewing implementation techniques, whether they are actually 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 actual implementations are not described in the literature, either because they are not assumed to be novel, or for confidentiality reasons, because they are. In the latter case, they are often protected by US patents. Conversely, many techniques are theoretically described in the literature, without any known implementation. Language specifications might help to determine whether a technique is adapted to some language, but complete specifications are currently not always implemented, or specifications can even be ambiguous. Thus, the schemes that we describe are more likely than real, but principles should not be too far from reality. Moreover, some techniques described here are likely original, but this was not the paper's primary goal.

[2] *Paradoxically, direct object manipulation is not supported under OO programming* [Lippman 1996, p. 28]. Of course, this is not paradoxical: the essence of object-orientation is that the object manipulates it`self`.

1.2 Notations and conventions

Uppercase letters denote classes or types, according to the context. The class specialization relationship is denoted by $\prec_{\mathrm{d}}$: $B \prec_{\mathrm{d}} A$ means that B is a direct subclass of A. One assumes that $\prec_{\mathrm{d}}$ has no transitive edges and its transitive (resp. and reflexive) closure is denoted by $\prec$ (resp. $\preceq$)—the latter is a partial order. The terms *superclass* and *subclass* will be understood as relative to $\prec$, unless they are explicitly described as *direct*, i.e. relative to $\prec_{\mathrm{d}}$. Subtyping is noted $<:$, which is a preorder. We consider that classes are types and that class specialization is subtyping (i.e. $\preceq$ is a subset of $<:$)—even though type theory distinguishes both relationships, this is a common simplification in most languages. τ_s and τ_d, where $\tau_d <: \tau_s$, respectively denote 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. Finally, a *root* is a class (resp. type) without superclass (resp. supertype).

Regarding the properties, i.e. methods and attributes[3], we adopt the following terminology consistent with an implicit metamodel. 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. Moreover, inheritance and late binding require differentiating what we call a *generic property*, which has several *definitions* in different classes, from the definitions themselves. As methods are commonly overridden, this distinction is important for them. On the contrary, multiple definitions of the same attribute are uncommon, so the confusion is harmless for attributes. This notion of *generic property* is missing in most languages: the term was coined in the model of CLOS *generic functions*. In the literature on method dispatch, the SMALLTALK term of *method selector* is often used, whereas Zibin and Gil [2002] use *method family*. In dynamic typing, a generic property can be identified with its name, as in the SMALLTALK method selector. With static typing, a generic property must be identified with both its name and the class introducing it. Moreover, some languages as C++, JAVA and C#, provide *static overloading*, so the parameter types must be added to the name in order to identify the property. An alternative would be to assume that all ambiguities caused by static overloading have been solved by renaming. Finally, this notion of generic property is mandatory to understand multiple inheritance—this is however far beyond the scope of this paper and the reader is referred to [Ducournau and Privat 2008].

However, implementation is not concerned by the actual definitions of methods, or by the inheritance problem itself, especially when multiple—i.e. which method definition is inherited by some dynamic type? The only point is the existence of the corresponding generic property and an efficient way of calling the appropriate definition, whatever it is. Therefore, for the sake of simplicity, we shall use the terms *attribute* and *method* to denote generic properties and, in some rare cases, we shall qualify them with *definition* to denote the occurrence of the generic property in the

[3]The object-oriented terminology is far from unified: C++ uses *member*, *data member* and *function member* in place of, respectively, *property*, *attribute* and *method*; EIFFEL *features* take the place of *properties*; *attributes* are often named *instance variables* (SMALLTALK, JAVA)—which is explained by syntactic reasons—or *slots* (CLOS).

declarations of a given class.

1.3 Production line of executable programs

Implementation techniques are closely related to the way executable programs are produced. We shall distinguish three main kinds of runtime production:

—*separate compilation* and *dynamic loading* (linking) is a common paradigm with JAVA and .NET platforms;

—*separate compilation* and *global linking*: this may be the common naive view, e.g. for C++ programmers, even though the language and most operating systems allow more dynamic linking;

—*global compilation*, hence including linking, is less common in production languages: EIFFEL is our only example, e.g. in the GNU compiler SMART EIFFEL[4] [Zendra et al. 1997; Collin et al. 1997].

This ordering is such that all techniques available for an approach are also available for the following ones. Conversely, all functionalities that can be implemented in an approach cannot always be implemented in the following ones.

Global compilation vs. global linking. One may distinguish several tasks in compiling a class. The first one is common to all programming languages—it involves generating the code for all procedures or functions, i.e. methods. The second one consists of computing what we call here an *external schema* of the class, which must be used by all subclasses and *client classes*[5], in order to know which properties exist or are accessible in the considered class—this is a kind of interface, but it is not restricted to public properties and it includes method definitions, i.e. not the code but the existence of a definition. A third tasks exists in non OO languages, but it becomes quite important with object-orientation: it computes the data structures associated to classes, together with object layouts, by fixing the size of all tables and the offset of all entities.

Task two may be done by hand, as in C++ (`.h` header files), but it is better when it is done by the compiler as in JAVA: note that the schema must be extractable as well from source code as from compiled code. Task three may be done from the external schema, i.e. the source code is not needed. The point is to determine whether task one, which is the essential task of compilation, does insert the very offset values in the generated code. The obvious alternative is to use symbols as offsets, leaving the substitution of values to symbols to a further task, for instance linking, for which symbol resolution is a usual capability. The advantage of doing that at link-time is that many recompilations might be avoided, e.g. modifying the code of a method in a class should not require recompilation of subclasses and client classes. Augmenting the schema of a class by introducing new attributes or methods should not require any subclass recompilation: however, subclass schemata should be checked in order to look for possible name conflicts.

Hence, if one accepts to give up dynamic loading, an alternative to global compilation is to delay, at a global linking, the computation of all offsets and data

[4]Formerly known as SMALL EIFFEL.

[5]Given a class A, a class B is a *client* of A if A, or a subclass of A, is used as a type annotation in the code of B. An alternative terminology is to say that B *imports* A.

structures. Note also that almost all implementations require a minimal global step, for instance for generating class identifiers by global numbering.

Separate compilation and dynamic loading. Separate compilation, and dynamic loading even more, require that the corresponding tasks are incremental, or at least that they could be globally recomputed at each load. This is not always possible and this point must be carefully examined.

Our main objective is separate compilation, as it is common in most of the considered languages. Dynamic loading is used to an increasing extent in modern platforms—it happens that all common implementation techniques, e.g. C++, are compatible with it. However, JAVA and .NET platforms are based on a runtime system—inherited from LISP and SMALLTALK—composed of a *virtual machine* coupled with an *interpreter* and a so-called *JIT compiler*. Such an architecture has no effect on our survey, as implementations are the same as with more classic runtime systems. Specific optimizations are mostly at a different level (see for instance [Arnold et al. 2005]).

Actually, global linking is underused. Despite this, we shall also consider more global techniques, especially when appliable at link-time, at least for making some comparisons.

1.4 Evaluating efficiency

There are two basic criteria for efficiency, i.e. time and space. Time efficiency can be judged on average but the ideal thing is constant-time mechanisms. Space efficiency is evaluated by the amount of memory needed for runtime programs. Space and time efficiencies usually vary in opposite directions—a single criterion is impossible and a compromise is always needed. Nevertheless, increasing the space occupation will not always improve time efficiency, as it will also increase cache misses.

Eventually, run-time efficiency is the main goal but compile-time efficiency must not be forgotten—attention should be payed to NP-hard optimizations.

1.4.1 *Space efficiency.* Three kinds of memory resources must be considered. The dynamic part consists of the objects themselves, implemented as an attribute table, with one or more pointers to class tables: *garbage collection* must be envisaged for this part. The static part consists of the data structures associated with 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.

Static memory analyses have been done in the framework of global techniques which need to compact large tables [Driesen et al. 1995; Ducournau 1997], but this has not been done for separate compilation and static typing. As for dynamic memory, commonly used techniques may have a large overhead. First, some languages may implement some mechanism through a constant dynamic overhead, e.g. synchronization in JAVA [Bacon et al. 2002]. Second, dynamic space may be sacrificed for time efficiency, by putting part of static tables in dynamic tables, so dynamic overhead may need special optimizations [Eckel and Gil 2000]. We base our evaluation of space efficiency on statistics computed on benchmarks commonly used in the literature, together with a simple worst case analysis. Those benchmarks are reduced to an abstract description of class hierarchies, without the code of methods, i.e. mostly independent of actual langages. A more accurate analysis of memory

usage, cache and heap optimizations is beyond the scope of this survey.

1.4.2 *Time efficiency and processor architecture.* With classic processors, a measure of the time efficiency of a mechanism is the number of machine instructions it requires. Modern processors make this measure obsolete since they have a pipe-line 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 longer one cycle. Moreover, composing two mechanisms may follow a law of maximum rather than of sum. The instruction number is only a space measure.

Implementing method calls with direct accesses into method tables has long been considered as optimal—the effective overhead vs. static function calls seemed unavoidable. However, branching prediction of modern processors appears to have better performance with the technique, known as *inline cache* or *type prediction*, which involves comparing the receiver's actual type with an expected type, whose method 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 might be considered out of date. Nevertheless, two arguments oppose this thesis. Branching prediction favors type prediction as the prediction is restricted to conditional branching: but it could be extended, in future processors, to indirect branching, putting both techniques on an equal step [Driesen 2001]. Moreover, type prediction is not well adapted to separate compilation.

We present an evaluation of time efficiency based on a model of modern processors and on an intuitive pseudo-code, both borrowed from [Driesen and Hölzle 1995; 1996; Driesen et al. 1995; Driesen 1999; 2001]. 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. Note that both values assume that the data is cached, i.e. a cache miss would add up to 100 cycles. This is a sensible assumption as objects are likely present in cache for several accesses. Regarding method tables, their duration in cache should be longer, as they may be shared by several instances. More details may be found in K. Driesen's works.

Obviously, this model does not give an exact evaluation but only a rough idea, more accurate than the number of instructions, of the cost of each mechanism. An objective evaluation should measure the execution time of benchmarks programs, each of them compiled according to the different implementation techniques. Whereas such benchmarks programs are common in functional programming, they do not exist for OO languages: this is mostly due to the diversity of the languages and of their class libraries. An important work of evaluation and comparison has been made in the framework of a single language, but mostly for dynamically typed languages as Smalltalk, Self and Cecil. On the whole, there is no comparison between C++, Java and Eiffel implementations based on execution of actual programs. Some work has been done in order to compare various implementation techniques by measuring execution time of articial programs, automatically generated from an abstract language: [Privat and Ducournau 2005] is a first step towards this goal, but the results are too new for drawing conclusions in this survey.

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

1.5 Structure of the paper

One may classify statically typed languages by their inheritance and subtyping relationships, according to whether they are single or multiple. Moreover, when inheritance is multiple, it can be arborescent, i.e. the superclasses of every class form a tree [Krogdahl 1985]: Sakkinen [1989; 1992] calls this kind of inheritance *independent multiple inheritance*, in contrast with full multiple inheritance, called *fork-join inheritance*. Each of the four resulting categories uses specific techniques (Table I). The next two sections present the standard implementation principles in two extreme cases: single subtyping (SST), then multiple inheritance (MI). Only the core of object-oriented programming is discussed here and the various notions are precisely introduced on the way. In section 4, we examine some alternatives for multiple inheritance, including C++ "non virtual" implementation (NVI) which is only sound with arborescent inheritance (AI). Whereas the previous sections consider the case of separate compilation, section 5 proposes a short survey of the techniques available in more global frameworks. Coloring, type analysis and link-time generation of dispatch code are examined as they seem to be good candidates for use at link-time in a separate compilation framework. The next section describes the median case of single inheritance (SI) but multiple subtyping (MST), illustrated by Java and .Net languages. Some applications to multiple inheritance and the special case of *mixins* are examined. Section 7 presents some complementary and not yet considered mechanisms: genericity, type variant overriding, class attributes, multiple dispatch, polymorphic primitive types, and so on. The article ends with a conclusion and some prospects. Appendix presents space statistics on common benchmarks and the pseudo-code for all the presented techniques.

2. SINGLE INHERITANCE AND SUBTYPING (SST)

With single inheritance and subtyping, types may be identified with classes and each class has at most one superclass.

2.1 Principle

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 (Reference). *Given an object, a reference—method parameter or returned value, local variable, attribute—on this object is invariant w.r.t. the reference's static type.*

Invariant 2.2 (Position). *Each attribute or method p has a unique offset, noted δ_p, unambiguous and invariant by inheritance.*

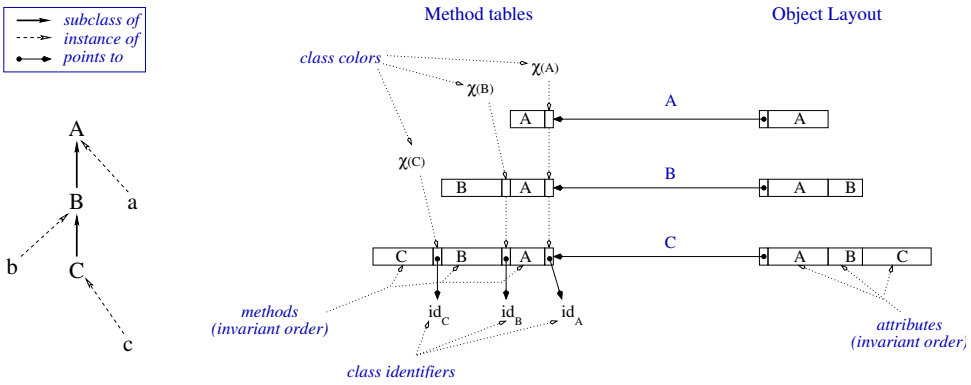


Fig. 1. A small class hierarchy with 3 classes A, B and C with their respective instances, a, b and c. Method tables—including Cohen's display—(left) and object layout (right) in single subtyping. This diagram respects the following conventions. In method tables (left), A (resp. B, C) represents the set of addresses of methods introduced by A (resp. B, C). Method tables are drawn from right to left to reduce edge crossing. In object layouts (right), the same convention applies to attributes but the tables are drawn from left to right. Solid lines are pointers.

Conversely, given a static type, the position of an attribute (resp. method) in the object layout (resp. method table) is invariant w.r.t. the receiver's dynamic type. This is the basis for constant-time access. 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 represents the ideal *type erasure* of type theory—see also Section 7.4.

Omitting parameter passing which is done in the same way as in non OO languages, method calls are then compiled into a sequence of three instructions:

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

where `#tableOffset` is a constant, usually 0, and attribute accesses are as immediate as for a record field:

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

Note that attribute invariance is due to the fact that attributes always have fixed size values: when values have variable size, the attribute is a pointer at the effective value.

Computing the tables, i.e. method and attribute offsets δ_p, is a special algorithmic case of coloring heuristics (see Section 5.3). Single inheritance and static typing, which both avoid *introduction overloading*[6], ensure a sound and optimal result—no

[6]Here *introduction overloading* means that two properties with the same name might be *introduced* in two unrelated (by specialization) classes (hence without being *defined* in any common superclass): with static typing (and of course without multiple inheritance) the two occurrences are no more related than with *static overloading*—i.e. any method call in the code may be statically and unambiguously assigned to one of the two methods—but this is not the case with dynamic typing, e.g. in SMALLTALK.

offset conflict can occur between inherited properties. Therefore, if p is the last allocated method (resp. attribute), at offset δ_p, the offset for the next introduced method (resp. attribute) q will be $\delta_q = \delta_p + 1$, without any need for checking that this new offset is really free.

Furthermore, it should be noted that each access to an attribute or a method of an object (`object` in pseudo-code examples) must be preceded by comparing `object` with the `null` value, as local variables and attributes may be left uninitialized. Assuming that the failure case, which must signal an exception, may be shared, then this adds one instruction and cycle per access (see Sections 2.4 and 7.7).

Although pure SST languages are not common, this implementation is the basis of most implementations, in both JAVA without interfaces (see Section 6) and C++ when restricted to single and "non virtual" inheritance (see Section 4.1).

2.2 Instance creation

Finally, instance creation amounts: to i) allocating a memory area according to the number of attributes, ii) assigning the method table address at `tableOffset`, iii) calling a method for initializing attributes (improperly called a *constructor* in JAVA and C++).

The i) and ii) stages are usually static, as the instanciated class occurs as a constant in the code. However, when instanciating a formal type in a parameterized type (see Section 7.4), or in some cases of covariant overriding (see Section 7.1), the instanciated class is not known at compile time—therefore a special method is needed. Regarding the initializing method, this is—or at least it should be—a standard method, ruled by late binding[7]. The question of uninitialized attributes may be dealt with by generating, at compile time, some assignments to null values (see Section 7.7).

2.3 Casting

2.3.1 *Principle.* The word *cast* is commonly used in reference to various mechanisms close to type coercion, from a *source* type to a *target* type. Among its 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 cast* because it requires no particular syntax. It simply involves a polymorphic assignment (or parameter passing) $x := y$, when the static type of x (resp. y) is X (resp. Y) and Y is a proper subtype of X ($Y <: X$). Such a mechanism should have no name as it is conceptually vacuous—this is pure inclusion polymorphism—but its implementation may be non-trivial.

Downcast assumes that an entity of a given static type $\tau_s = X$ is actually an instance of another type Y, i.e. $\tau_d <: Y$ and, usually, $Y <: X$. This is a type unsafe assumption, which cannot (without loss of generality) be statically proven—it thus requires a dynamic type check which may fail if it turns out that the assumption was false. *Downcast* is done by special syntactic constructs as `dynamic_cast` in C++, parenthesized syntax (a C syntax which must not be used in C++!) or `instanceof`

[7]Contrary to C++, where all calls in a constructor are static, as if `this` were not polymorphic, which is not justified.

in JAVA, `typecase` in THETA [Liskov et al. 1995] or *assignment attempts* in EIFFEL [Meyer 1992; 1997]. *Downcast* uses may be justified by the fact that covariant models are implemented in type-safe, i.e. contravariant or invariant, languages (see Section 7.1). They are also common in JAVA because of its lack of genericity (up to version 1.5). Downcast failures may be treated in several ways, either by signaling an exception (JAVA, C++ for references only), returning a null value (EIFFEL, C++ for pointers only), or in a boolean way (`typecase`). Exception `catch` clauses often involves downcasts.

Besides these two opposite directions, casting may also be *static* or *dynamic*[8], according to whether the target type is statically known or not. Explicit syntactic constructs are always static casts as the target type is a constant of the construct. Reflection provides means for dynamic casts, such as the `isInstance` method in JAVA. Furthermore, we shall see that some mechanisms may need dynamic casts, which means that the target type has to be reachable from the considered object. Finally, downcasts may be *inlined*, or realized through an explicit function call.

2.3.2 *Casting in single subtyping.* As references to objects do not depend on static types, *upcast* has no more reality in the implementation, which is trivial, than as a concept. Due to reference invariance, *downcast* amounts to dynamic type checking. There are two classic simple ways to implement type checking with SST. Both are inlined.

Class coloring and Cohen's display. The first technique consists of assigning an offset to each class in the method tables of its subclasses—the corresponding table entry 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 offset δ_C (C *color*), 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 2.2 as methods and attributes.

A point must be carefully examined. In theory, $tab_{\tau_d}[\delta_C]$ is sound only if δ_C does not run out of the bounds of tab_{τ_d}. If one assumes that class offsets are positive integers, a comparison of δ_C with the length of tab_{τ_d} should be needed, together with memory access to the length itself—this would hinder efficiency. Fortunately, there is a simple way to avoid this test, by ensuring that, in some dedicated memory area, the value id_C occurs always at offset δ_C of some *tab*. Consider that method tables contain only method addresses and class identifiers and that they have contiguous allocations in a dedicated memory area, which therefore contains only addresses and identifiers. As adresses are even numbers (due to word alignment), coding class identifiers with odd numbers would avoid any confusion between the two types. Finally, the dedicated memory area must be padded with some even number, to a length corresponding to the maximum *tab* size[9]. Nevertheless, method tables might

[8]These terms are unrelated with the C++ keywords `static_cast` and `dynamic_cast` which are both *static* (see Section 4.1).

[9]In the framework of dynamic loading, this maximum size is not known—an upper bound must be used, which is a parameter of the runtime platform. Moreover, the method table area does not need to be unique—when it is full, a new one may be allocated, and the maximum size may be adjusted.

contain more data than adresses and identifiers, e.g. offsets for accessor simulation (see Section 4.4)—but they are also even numbers. Therefore, either a more complex coding or an indirection might be required. Anyway, if class identifiers are gathered within specific tables, distinct from method tables and allocated in the same contiguous way, this extra indirection will have the same cost as access to length, but the test itself will be saved. A more common way to save on this extra test, frequently proposed (e.g. [Click and Rose 2002]), is to use fixed size tab_{τ_d}, at the expense of a large constant overhead[10].

Space efficient variants exist, based on a compact, variable length, coding of class identifiers, which is no longer absolute but relative to classes with the same color (i.e. depth). However, this is to the detriment of i) separate compilation, or at least dynamic linking, ii) time efficiency and code size, as colors are no longer aligned to word or half word.

This first technique is simple and works well in separate compilation and dynamic loading, i.e. it is time efficient but not space optimal. This is again a special case of coloring (see Section 5.3), described by Cohen [1991] as an adaptation of the 'display' originally proposed by [Dijkstra 1960]. It has been widely reused by different authors (e.g. [Queinnec 1998; Alpern et al. 2001]).

Double numbering. The second technique is time efficient and, in practice, space optimal but it has a twofold drawback, i.e. it does not generalize to multiple inheritance, at least in constant time, and it is not incremental, thus is less compatible with dynamic loading. It is a double class numbering, noted n_1 and n_2: n_1 is a preorder 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. When used in a dynamic loading framework, this technique requires a complete computation each time a class is loaded. As the complexity is linear in the number of classes, this is not a problem. Actually, the main point is that $n_1(C)$ and $n_2(C)$ would need memory access, i.e. it would substitute a dynamic cast to a static one.

In the optimal case, both techniques have the same time efficiency ($2L+2$ cycles). The latter has a better memory cost in the static tables (in the 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; Gil and Zibin 2005], but no other technique has such a power of simplicity.

2.4 Basic optimizations

On the basis of this simple implementation, two classic optimizations of programming languages may improve the resulting code, even in separate compilation.

[10]Statistics in Appendix show that the maximum of superclass number may be 5 times greater than its average (Table XX).

Intra-procedural data flow analysis may: i) avoid useless `null` checks (see Section 7.7), ii) detect *monomorphic* cases, when τ_d is known at compile-time—method calls are then static function calls, without table accesses, and downcasts are statically solved. For instance, consider the following code:

```
1   x : A          // x = null
    ....
2   x := new B     // B ≺ A
    ....
3   x.foo()
4   x.bar()
```

According to the control flow of this program, the call to `foo` might be monomorphic, hence replaced by a static call to the method `foo` in B. If it is not—e.g. because of the code between lines 2 and 3—the `null` check might at least be avoided. Finally, if line 2 is removed, the `null` check is useless in the call to `bar`, even if it is needed for the previous call, as x has already been proven different from `null`.

Languages may offer specific syntactic features that allow the programmer to express that a class cannot be specialized, or that a method cannot be redefined, for instance, keywords `final` in JAVA and `frozen` in EIFFEL. To the detriment of reusability, these keywords allow easy detection of some monomorphic calls in an intra-procedural way.

Inlining is another common optimization of procedural languages—it involves copying the code of the callee in the caller, for instance when the callee is either small or not often called. With OO languages, inlining can only apply to static calls, e.g. to monomorphic calls, and with separate compilation, it can only apply to methods whose source code is known, hence defined in the current code unit. Alternatively, the code to inline must be included in the *external schema* of the unit, like in C++ in the `.h` file.

Despite their restricted usage, both optimizations may have a significant effect, as the situation in the previous example is quite frequent.

2.5 Evaluation and first methodological conclusions

Time efficiency is optimal as everything is done with at most a single indirection in a table. Apart from attribute initialization, instance creation is time-constant. Even downcasts are time-constant. 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 space 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 5.2). Let M_C denote the number of methods known (defined or inherited) by a class C, then the method tables have $\sum_C M_C$ entries.

This optimal implementation of SST is the reference that serves to measure the overhead of multiple inheritance or multiple subtyping, for both time and space efficiency. Besides this first evaluation, several conclusions can been drawn from the SST implementation that can serve as conceptual and methodological guide in our study.

Abstract classes. Another point must be considered, namely the cost of abstract classes. Designing a class hierarchy goes through successive specialization (adding a new subclass) and generalization steps. Generalization consists of splitting an existing class C into two related classes C_1 and C_2, such that $C_2 \prec C_1$, and $\forall D, C \prec D \Rightarrow C_1 \prec D$ and $D \prec C \Rightarrow D \prec C_2$. Generalization does not change the behaviour of existing programs, as C_2 takes the place of C, i.e. it only opens the door to further specialization of C_1. Generalization often yields abstract classes, i.e. classes without proper instances, which are used only for sharing code. These abstract classes entail no overhead at all in this implementation. More generally, when splitting a class, one does not add any dynamic overhead, only a small static space overhead if one does not know that the generalized class (C_1) is abstract. This point is a measure of the fact that reusability may be increased without penalizing current programs.

Production of runtimes. Regarding production of runtimes, this implementation also has all qualities, i.e. it is fully compatible with dynamic loading. Class recompilation is needed only when the schemata of superclasses or imported classes are modified. Offset computation may be done at compile-time or delayed at load or link-time and left to a quite general symbol resolution mechanism (see Section 1.3). Only class identifiers must be computed at load or link time and not at compile time, but it can be done by the compiled code itself and does not require a specific linker or loader.

Mechanism equivalence. The three mechanisms that we consider—namely method invocation, attribute access and subtype testing—would seem to be equivalent, as they reduce to each other. Obviously, method tables are object layout *at the meta-level*. So, apart from memory-allocation considerations, they are equivalent. Moreover, an attribute can be read and written through dedicated *accessor* methods—hence, attribute access can always reduce to method invocation (see Section 4.4).

An interesting analogy between subtype tests and method calls can also be drawn from Cohen's display. Suppose that each class C introduces a method `amIaC?` which returns `yes`. In dynamic typing, calling `amIaC?` on an unknown receiver x is exactly equivalent to testing if x is an instance of C—in the opposite case, an exception will be signaled. In static typing, the analogy is less direct, since a call to `amIaC?` is only legal on a receiver statically typed by C, or a subtype of C—this is type safe but quite tautological. However, subtype testing is inherently type unsafe and one must understand `amIaC?` as a pseudo-method, which is actually not invoked but whose presence is checked. The test fails when this pseudo-method is not found, i.e. when something else is found at its expected position. This informal analogy is important—it implies that one can derive a subtype testing implementation from almost any method call implementation. We actually know a single counter-example, when the implementation depends on the static type of the receiver, as in subobject-based implementations (Section 3).

The converse does not hold as well, if one wants to closely mimic implementations, but Queinnec [1998] proposes a general method dispatch technique which only relies on subtype testing, irrespective of the precise technique.

Prefix condition. The SST implementation satisfies what we call the *prefix condition*. If $B \prec A$, then the implementation of A forms a *prefix* in the implementation

of B. As a corollary, attributes and methods are *grouped* according to their introduction class. No other implementation satisfies it for all pairs of $\prec$-related classes, but we will see that this condition can be a good opportunity for optimizations when two classes satisfy it.

Space linearity. In the SST implementation, the total size of the tables is roughly linear in the size of the inheritance graph, i.e. linear in the number of pairs (x, y) such that x is a subtype (subclass) of y ($x \preceq y$). Cohen's display uses exactly one entry per such pair and the total size is linear if one assumes that methods and attributes are uniformly introduced in classes. Moreover, the size occupied by a class is also linear in the number of its superclasses. More generally, linearity in the number of classes is actually not possible since efficient implementation requires some compilation of inheritance, i.e. some superclass data must be copied in the tables for subclasses. Therefore, usual implementations are, in the worst case (i.e. deep rather than broad class hierarchies), quadratic in the number of classes, but linear in the size of the inheritance relationship. In contrast, C++ is, in the worst case, cubic in the number of classes—i.e. all known implementations are cubic and it is likely an inescapable consequence of the language specifications—because the total number of method tables needed for all classes in the hierarchy is exactly the size of the inheritance relationship $\preceq$ (see Section 3). The inability to do better than linear-space is likely a consequence of the constant-time requirement. As a counter-example, Muthukrishnan and Muller [1996] propose an implementation of method invocation with $\mathcal{O}(N + M)$ table size, but $\mathcal{O}(\log \log N)$ invocation time, where N is the number of classes and M is the number of method definitions.

3. MULTIPLE INHERITANCE (MI)

Multiple inheritance complicates implementation to a considerable extent, as [Ellis and Stroustrup 1990, chapter 10] and [Lippman 1996] demonstrate it for C++. C++ is all the more complicated because it offers some unsound features which aim at reducing MI overhead. The keyword `virtual`, when used to annotate superclasses, is the way to obtain sound MI semantics in the general case—called by Sakkinen [1989; 1992] *fork-join inheritance*—whereas what we shall term *non-virtual multiple inheritance* (NVI), when the keyword is not used, offers a cheaper implementation but degraded semantics, which is 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. These introductory precautions are not necessary for an OO language of sounder constitution like Eiffel [Meyer 1992; 1997].

3.1 Principle

With both separate compilation and MI, there is no way to assign a minimal and invariant offset to each method and attribute (Invariant 2.2) without causing future conflicts. When B and C are 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 2.2 leads to doing the same for Invariant 2.1, if one wants method calls to take only one indirection in a table.

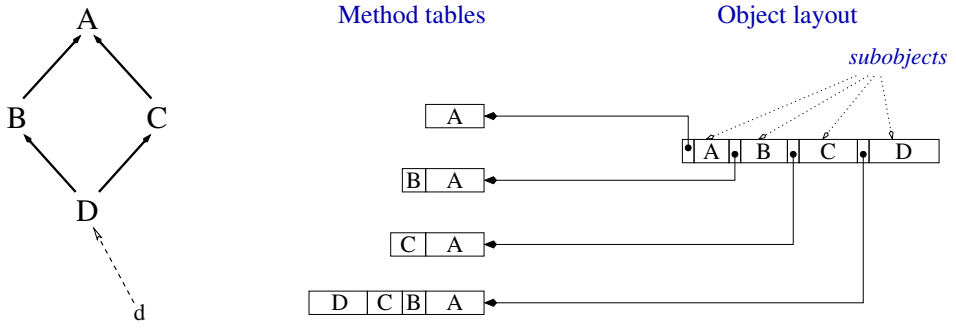


Fig. 2. Object layout and method tables with multiple inheritance: an instance d of the class D is depicted—the diagram follows the same convention as in Figure 1

We shall see further how to keep reference invariance, by giving up constant-time mechanisms (see Section 6.4) or separate compilation (see Section 5.3).

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

Invariant 3.1 (Private attribute position). *Each attribute a has an offset, noted δ_a, unambiguous and invariant w.r.t. the dynamic type, which determines the attribute position only in the context of the static type which* introduces *the attribute.*

Invariant 3.2 (Type-dependent method position). *Given a method m and a static type τ_s which* knows *the method, an offset, noted $\delta_m^{\tau_s}$, unambiguous and invariant w.r.t. the dynamic type determines the method position.*

In contrast with SST implementation, method (resp. attribute) offsets are no longer absolute, but relative to the current static type (resp. the type which introduces the attribute). The object layout consists of subobjects, one for each superclass of its class (i.e. for each static type that is a supertype of its dynamic type) and each subobject is equipped with its own method table (Figure 2). Reference invariance is then replaced by:

Invariant 3.3 (Type-dependent reference). *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 its 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—these tables are isomorphic but do not contain the same addresses (Table II). For a given static type, method offsets do not matter, but it is reasonable to group methods by introduction classes, as in the figures, i.e. this may offer some local invariance (in case of SI) but this organization has no effect on efficiency.

Note the dissymmetry between attributes and methods that appears in this implementation, in contradiction with our previous conclusions (see Section 2.5).

Table II. Method tables for Figure 2 classes, according to static and dynamic types. In each table, the class names (A, B, C, D) stand for the methods introduced by the class. Though method ordering for a given static type is arbitrary, it is convenient to gather them per introduction class, with an invariant ordering. Hence, for the same static type (column), the tables are isomorphic, i.e. same method ordering, but differ by their contents (method addresses), whereas, for the same dynamic type (row), isomorphic pieces contain the same addresses but different shifts. Moreover, in this example as well as in Figure 2, all method tables—apart from the two last ones, since AC is not a prefix of ABCD—verify the *prefix condition* which is needed for *empty subobject optimization* (see Section 3.3). The last line corresponds to Figure 2 tables.

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

Whereas both kinds of property are conceptually analogous, the implementation distinguishes *introduced* attributes and *known* methods. The explanation is that methods are *shared read-only attributes*, whose values are method addresses. Being read-only, they can be copied without changing the program semantics[11]. Hence, copying methods is an optimization which cannot be applied to attributes, which are not read-only. Section 4.2 describes a more symmetrical, but less efficient, implementation. This dissymmetry is however consistent with the dissymmetry between attributes and methods in the C++ language specifications, especially with multiple inheritance (see for instance [Ducournau and Privat 2008]).

3.1.2 *Method call and* `self` *adjustment.* Invariant 3.3 requires recomputing of 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_s, noted[12] $\Delta_{\tau_s,W}$ (Figure 3). We call this relative position between subobjects a *shift*. Thus method tables have double entries for each method, an address and a *shift* (Figure 7 on top). On the whole, method calls are compiled into the sequence[13]:

```
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 is to define a small intermediate piece of code, called *thunk*[14] by Ellis and Stroustrup [1990] or *trampoline* by Myers [1995], which shifts the receiver before calling the method:

[11] This argument might be applied to hypothetical languages where methods are not read-only—in such a language, it is likely that method assignment would not have the same efficiency requirements as attribute assignment. Therefore, copying methods would remain an acceptable solution.
[12] The notation $\Delta_{T,U}$, as all the following notations Δ, implies a given dynamic type $\tau_d \preceq T, U$—specifying it would make the notation heavy.
[13] Instructions added w.r.t. SST are italicized (see page 10).
[14] According to Lippman [1996], *thunk* would be *Knuth* spelled backwards. However, most people agree that the term originates in Algol `call-by-name` parameter passing:

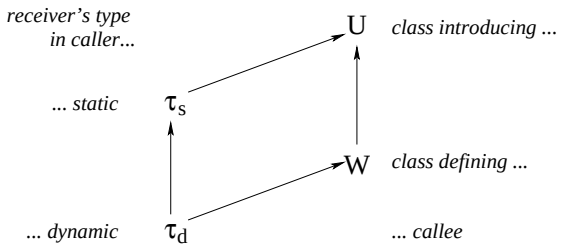


Fig. 3. Receiver types in a method call—a shift $\Delta_{\tau_s,W}$ is needed

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

The advantage is that `#delta` (i.e. $\Delta_{\tau_s,W}$) is now a constant. Then, the call sequence is the same as with SST. Thus, one access is saved in the table for an extra direct branching. Thunks may be shared when the address and the shift are the same, and the thunk with the null shift is the method itself. The *thunk* could also be inlined in the method table instead of being pointed by it—L cycles could be saved, but the null cost of the null shift would be lost. Note that the thunk cannot be generated when compiling class W, as the shift depends on τ_d, not only on τ_s. Generating the thunk as a method entry point, when compiling W, would require an explicit upcast, i.e. a shift with table access. Therefore, the thunk may be understood as a method redefinition which only calls `super` (see Section 7.6).

3.2 Casting

Subobjects make casting real, and Rossie et al. [1996] define it as a subobject change. First note 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.* As references are relative to static types, an equality test between two references first requires that their types are equal. 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.

Historical note: There are a couple of onomatopoeic myths circulating about the origin of this term. The most common is that it is the sound made by data hitting the stack; another holds that the sound is that of the data hitting an accumulator. Yet another suggests that it is the sound of the expression being unfrozen at argument-evaluation time. In fact, according to the inventors, it was coined after they realized (in the wee hours after hours of discussion) that the type of an argument in Algol-60 could be figured out in advance with a little compile-time thought, simplifying the evaluation machinery. In other words, it had "already been thought of"; thus it was christened a 'thunk', which is "the past tense of 'think' at two in the morning" (`http://www.retrologic.com/jargon/T/thunk.html`). `http://www.webopedia.com/TERM/T/thunk.html` and `http://en.wikipedia.org/wiki/Thunk` report similar origins.

The precise meaning is slightly different here. Anyway, a thunk may be defined as a small piece of code, or a stub function, used for parameter adjustment.

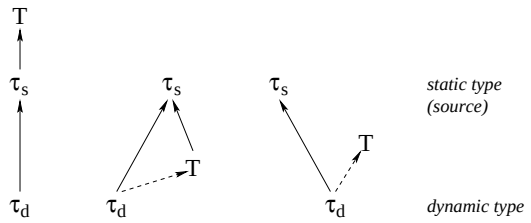


Fig. 4. Upcast (left), downcast (center) and side cast (right), to a target type T—the dashed subtype relation must be checked at run-time before shifting.

3.2.2 *Upcast.* Changing from the current subobject, with type τ_s, to a statically known supertype T, requires 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 3.4 (Type-dependent upcast). *Each class has an unambiguous and invariant offset in the static context of each of its subclasses.*

This offset is noted $i_{\tau_s}(T)$, where $\tau_s <: T$, and it is, unlike Δs, independent 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 be inlined in the method table, either by interweaving method entries and shifts, or with a bidirectional layout. 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. Therefore, the shifts are ruled by the same Invariant 3.2 as methods. This is complementary to the equivalence sketched in Section 2.5—upcast could be actual methods—but useless in invariant-reference implementations. On the contrary, the equivalence between subtype testing and method invocation does not hold here.

3.2.3 *Accesses to inherited attributes.* Table $\Delta^{\uparrow}_{\tau_s}$ is also used for accesses to 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). Then:

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

When an attribute is not introduced in the receiver's static type ($T_p \neq \tau_s$), there is 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 the worst case, assignment $a.x := b.y$ needs three upcasts, for accesses to a and b, and between y and x types. Some parallelism is likely but the sequence is 5-fold longer than in SST, w.r.t. cycle count ($5L + 3$ vs. $L + 1$) and 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, direct access is difficult, at

least in separate compilation (see Section 5.3) and some sequential search may be unavoidable. Each class τ_d has an association structure 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 for T in the table. If T is not found, a failure occurs. Otherwise, $\Delta_{\tau_d,T}$ is returned and:

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

Note that both tables $\Delta^{\updownarrow}_{\tau_s}$ 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 are also the same as the shifts included in the method tables or *thunks*. It might be advantageous to use static types, by associating a new table $\Delta^{\tau_s}_{\downarrow}$ to each subobject, i.e. the restriction of $\Delta^{\Uparrow}$ to the classes between τ_s and τ_d: $\Delta^{\tau_s}_{\downarrow} = \Delta^{\Uparrow}/[\tau_s, \tau_d] + \Delta^{\tau_s}_{\Downarrow}$. This would avoid a two step downcast:

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

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

Hashtables. A first way to implement $\Delta^{\Uparrow}$ and $\Delta^{\tau_s}_{\downarrow}$ is hashtables. This type checking implementation may be costly if it means calling some function `dcast(object, id`$_T$`)` where `object` is the considered object and id_T the target's identifier. Of course, this function call could be inlined but access to hashtable is not such a small function. In practice, C++ compilers (e.g. `gcc`) do not inline this call[15]. Appendix B.2 presents one of the simplest hashtable implementations, a variation on *linear probing* [Knuth 1973; Vitter and Flajolet 1990], with a 15-instructions code with $3L+3+k(L+3)$ cycles, where $1 \leq k \leq n$ and n is the number of superclasses, including τ_d. As such hashtables are varying length, they can be inlined in the method table at negative offsets only.

Note that these hashtables have a strong particularity, i.e. they are *constant hashtables*—once they are computed, at link or load time, there is no longer any need for insertion or deletion. Therefore, for each class C, knowing the identifiers of all its superclasses, it is possible to optimize the hashtable in order to minimize i) the average number of probes, either in positive or negative cases, ii) the table size. A generic formulation would be to define the hashvalue of id_D, where $C \preceq D$, as the result of a function $h(id_D, H_C) = h_C(id_D)$. Simple examples of h are remainder of integer division, or bit-wise `and`. The optimization problem is to minimize H_C, for all C, for minimizing access time and table size. In the ideal case, H_C may be defined as the least integer such that all tests need only one probe, i.e. h_C is injective for all C [Ducournau 2008]. This is known as the *perfect hashing* problem [Sprugnoli 1977; Mehlhorn and Tsakalidis 1990]. It is worth noting that this use of perfect hashing is the only purely incremental technique that we know for multiple inheritance.

[15]Experiments with the same compiler present a cost almost linear in the number of superclasses, which is quite surprising, as if no hashtable were used [Privat and Ducournau 2005].

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}	$[\Delta_{BA},0]$	$((B,0)(D,\Delta_{BD}))$	
C	Δ_{CD}	$[\Delta_{CA},0]$	$((C,0)(D,\Delta_{CD}))$	
D	0	$[\Delta_{DA},\Delta_{DB},\Delta_{DC},0]$	$((D,0))$	$((A,\Delta_{DA})(B,\Delta_{DB})$ $(C,\Delta_{DC})(D,0))$

Direct access. Alternatively, a truly constant-time mechanism avoiding a function call would require coding the specialization partial order—this is a difficult problem in MI. Moreover, it would not work since the problem is no longer boolean, i.e. shifts are needed. A technically simple solution allowing 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 a failure. Such a matrix requires $2N^2$ bytes, i.e. $2N$ bytes per class, which is a reasonable cost when there are $N = 100$ classes, but it is not when $N >> 1000$ (see Appendix A). Class identifiers id_C must be computed globally for all classes C. A convenient injective numbering of N classes, is to associate with each class a number in $1..N$, in such a way that $id_C > id_D$ when $C \prec D$, i.e. numbering is a *linear extension* of $\prec$. This is the natural chrnonological numbering in the class load ordering. This allows to replace the square matrix by a triangular one, i.e. by one vector per class, using $vect_U[id_T]$ instead of $mat[id_T, id_U]$. The cost is reduced to an average of N bytes per class, which is the worst-case cost of the coloring scheme as well in SST (see Section 2.3.2) as in MI (see Section 5.3), but the average cost of coloring is far lower. Note that this looks like a non optimal solution to *perfect hashing*, when $H_C = id_C$ and h is the remainder of integer division[16]. However, an important difference is that, with direct access, there is only one entry per class, while there are two for hashing. Therefore, in SMI, perfect hashtables are an acceptable solution only if H_C is, in average, significantly lesser than $id_C/2$.

3.3 Empty subobject optimization (ESO)

On the basis of this uniform implementation, a simple optimization reduces space overhead. Indeed, an exception to Invariant 3.3 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 within the subobject of one of its direct superclasses, say E. Some cases are to distinguish.

In the first case (Figure 5-a), E is the only direct superclass of F and F introduces no method, i.e. E and F have the same set of methods. The F subobject may be merged into the E subobject as, without merging, the contents of both method tables, i.e. method addresses, would be the same. Here, merging works because it is invariant w.r.t. dynamic type, i.e. 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

[16] A similar unacceptable solution exists when h is bit-wise `and`, i.e. H_C is then defined as the bit-wise `or` of all $id_D, C \preceq D$.

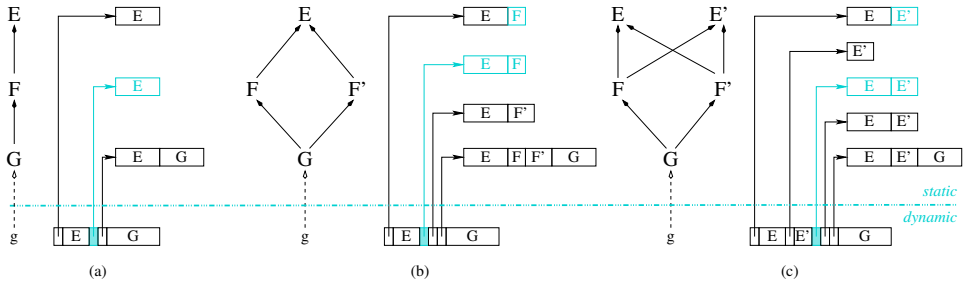


Fig. 5. Three cases of empty subobjects : (a) for all $G \prec F$, E and F subobjects may be merged, (b) for such a G, merging of either F or F' is possible, not both, (c) same thing, however F might be merged in E, and F' in E' if the prefix condition were satisfied for both. Grey parts show the result of merging F into E.

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 $\tau_d \preceq \tau_s \preceq F \prec_{\mathrm{d}} E$. The code generated for all $\tau_s \preceq F$ takes into account the merging of E and F: e.g. access to 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 (Figure 5-b), or more than one direct superclass (Figure 5-c)—the latter condition implies the former, as superclasses yield upcast pseudo-methods—but the method ordering of E is a *prefix* of the method ordering 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 only if E is not already merged within another F', in the same $\tau_d \preceq F'$. Once again, merging works in this restricted case because the aforementioned *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 not a property of E or F, but only of some $\tau_d \preceq F$. In this case, merging allows sharing method tables, not saving on shifts in the code or in upcast tables. The code generated for all $\tau_s \preceq F$ cannot consider that E and F are merged ($i_{\tau_s}(E) \neq i_{\tau_s}(F)$), but the data structures for some $\tau_d \preceq F$ may do this merging. In particular, access to an attribute introduced in E, on a receiver typed by F, needs a cast, but the shift will be null ($\Delta_{E,F} = 0$).

The case where the E subobject is itself empty must be carefully examined—it roughly amounts to merging F within the superclass within which E is merged. Finally, F may be a *root class*, i.e. it must then be merged into subclasses G. This is always statically possible when G is not empty and has no other superclasses. Other cases should be also carefully examined.

The prefix condition is the basis of SST Invariant 2.2. In MI, it is always partly satisfiable—indeed, the method ordering of a class, which has until now been arbitrary, can always be made as an extension of the method ordering 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 this rather exceptional case, only one merging will be possible. When F has more than one superclass (Figure 5-c), a way to improve

merging chance involves the choice of superclass (E or E') for which the prefix condition holds. The choice might be either at random, or depending on F, e.g. based on a hash value of F's name. This choice would take place at compile-time—it might be replaced by an optimized assignment at link time. Anyway, neither F nor G can take a static advantage of merging.

There is some evidence that empty subobjects are quite frequent and that this simple optimization is essential (see Appendix A). An allusion in [Lippman 1996][17] suggests that it is used in actual compilers. However, experiments prove that it is not implemented in all compilers, for instance, G++ 2.8 and SUN 5.3 C++ compilers do not use it. This optimization may be already known—however, if it is the case, it has not been widely noticed, as most benchmarks in the literature do not include any information on attribute number [Gil and Sweeney 1999; Eckel and Gil 2000; Sweeney and Burke 2003].

3.4 Evaluation

Multiple inheritance overhead is rather large. On the whole, the main drawback of this implementation is that the overhead is the same when MI is not used—separate compilation is unable to foresee that a given class will be always specialized in SI.

Dynamic memory. In each object, the overhead is equal to the number of indirect superclasses of the object class. With ESO, it might be restricted to superclasses which introduce at least one attribute.

Static memory. The total number of method tables is no longer the class number N but the size if the inheritance relationship $\preceq$. Thus it is quadratic in N in the worst-case, whereas the total size is no longer quadratic, but cubic: $\sum_C(\sum_{C\preceq D} M_D)$. This cubic worst-case space is a property of all known implementations and it is likely an inescapable consequence of the language specifications—static pointer adjustments require cubic-size Δ tables. However, the formula should be corrected to take into account the empty subobject optimization: $\sum_D$ is restricted for D's which are not merged with some E, with $C \preceq D \prec_{\mathrm{d}} E$. Precise statistics confirm both the overhead of this implementation and the benefits of empty subobjects. In a context of mainly "non virtual" inheritance, [Driesen 2001; Driesen and Hölzle 1995] report a ratio larger than 3 on the table sizes, vs. the size of tables in SST ($\sum_C M_C$). Statistics given in the appendix show that the ratio may exceed 6 with standard implementation, but that it is reduced to 4 with ESO. When taking the shifts into account in the tables or in the thunks, the ratio climbs again to 6. *Thunks* seem roughly equivalent to putting shifts in method tables, i.e. there are less *thunks* than table entries, they need two extra 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. Instance creation is no longer time-constant but linear in the number of superclasses, since one must initialize the pointers of all subobjects on

[17] *My recommendation is not to declare nonstatic members within a virtual base class. Doing that goes a long way in taming the complexity* (p. 139). *In general, the most efficient use of a virtual base class is that of abstract virtual base class with no associated data members* (p. 101).

the corresponding method table. A shift is needed each time an assignment or a parameter passing is not with constant static types. This imposes extra access to the method table. The real impact on method calls is more questionable, i.e. shifting might be done within the processor's latencies or in parallel. Experiments by [Driesen 2001] seem to give a small advantage to thunks. But this conclusion is based on benchmarks mainly involving classic C++ programs making heavy usage of "non virtual" inheritance—in this particular case, shifts are mostly null and the thunk is the method itself (see Section 4.1). Although no conclusion about plain MI—i.e. pure 'virtual' inheritance—can be drawn, it seems that shifts have a sensible overhead. This is not surprising since, even though the cycle numbers are almost equal, this is at the expense of some parallelism, and it is likely that shifts take the place of some other code sequence. Furthermore, regarding downcast, some actual implementations seem quite unefficient, with an explicit function call and a cost apparently linear in the number of superclasses. Obviously this could be improved and a short pseudoconstant-time access should be obtained. Constant-time downcast remains a possibility—we have sketched a solution using perfect hashing. Large-scale experiments (see [Ducournau 2008] and Appendix A.4) proved its feasibility, with average table size standing at a middle point between pseudoconstant-time and direct access tables.

Monomorphic cases. Intra-procedural analysis allows the same optimizations as in SST. However, besides static calls and `null` tests, monomorphic cases may also optimize upcasts as all shifts are known at compile-time, when τ_d is. Therefore, there is almost no MI overhead in the monomorphic cases. In the case where the call is static but the receiver is still polymorphic—e.g. when the callee has been declared `final`—this advantage vanishes, as an upcast may be required.

Runtime production. Like the SST implementation, this MI implementation is fully compatible with separate compilation and dynamic loading. All offsets might be computed at link or load time, even though this is not the case in actual C++ compilers. Moreover, this does not require a specific linker or loader, since the compiler itself can generate the code for this link-time generation. Java and .Net platforms actually could be specified in plain MI with this implementation. Of course, the addition of differents overheads, induced by MI and virtual machine, would likely weaken the resulting efficiency.

Abstract class overhead. In contrast with SST implementation, splitting a class will cause some dynamic overhead—one more subobject increases the object size together with the need for pointer adjustment. This is a severe counter-argument to the central OO philosophy of reusability. ESO would partly counter-balance this overhead, as splitting a class increases ESO probability.

4. ALTERNATIVES FOR MULTIPLE INHERITANCE

Although C++ is the only example of the previous implementation, we shall call it *standard* MI (SMI) implementation as it appears as the unique table-based implementation compatible with: i) sound MI semantics, ii) separate compilation and dynamic loading. The complexity of this "standard" implementation explains part of the reluctance to use multiple inheritance as well as some unsound C++ features

[Cargill 1991; Waldo 1991; Sakkinen 1992]. Various alternatives have been sought: the most radical, proposed by the SMART EIFFEL compiler that we shall examine later (see Section 5.4.7), avoids method tables at the expense of global compilation [Zendra et al. 1997]. In this section, we examine some small variations around the standard implementation. They consist of 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 (Section 4.4).

4.1 C++ non-virtual inheritance (NVI)

As previously noted, the main drawback of standard MI implementation is that its overhead does not depend on effective use of MI. C++ "non virtual" inheritance is an answer to this problem. It is syntactically expressed by not annotating superclasses with the keyword `virtual`. The implementation is more efficient but it is sound only for *arborescent inheritance* (AI), i.e. when the superclasses of any class form a tree, what Sakkinen [1989; 1992] calls *independent multiple inheritance*. This implementation was proposed by Krogdahl [1985] in this restricted case only, but it is used in C++ in an unrestricted way.

4.1.1 *Pure NVI.* NVI is better described by its implementation than by its semantics [Ellis and Stroustrup 1990; Lippman 1996]. 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 introduced by the class are added at the end and the method table of the class extends the method table of one of its direct superclasses—the choice of this direct superclass is a matter of optimization and it defines a new relation $\prec_{\mathrm{e}}$, a subset of $\prec_{\mathrm{d}}$. Thus, the main difference is that there is no proper subobject for the class itself and all $\Delta_{T,U}$ are invariant w.r.t. τ_d. Therefore, upcasts and attribute accesses are truly static in the sense that they do not need table accesses. However, the code generated for method calls is the same as with standard MI implementation. The shifts on the receiver are still required but they are quite often null—thunks make most of them disappear in practice. Moreover, *thunk elimination* is now possible because the shift on the receiver is independent of τ_d, i.e. the overriding method defined in class W expects a value of `self` typed by the class U introducing the method and two static shifts are made in the caller (from τ_s to U) and in the callee (from U to W) [Lippman 1996, p. 138]. Alternatively, general support for multiple entry points avoids explicit thunks—an entry point is generated for each V such that $W \preceq V \preceq U$ (Figure 3).

The layout is a direct generalization of SST and, in case of a class hierarchy which only involves SI, it is exactly the same as with SST, i.e. only one subobject per object and without any shift. In the general case of arborescent inheritance, the number of subobjects or method tables is equal to the number of *root* superclasses.

4.1.2 *Arborescent vs. repeated inheritance.* The flaw of non-virtual inheritance occurs when inheritance is no longer arborescent, i.e. when the inheritance graph contains undirected cycles, as the diamond in Figure 2. It involves what we call here

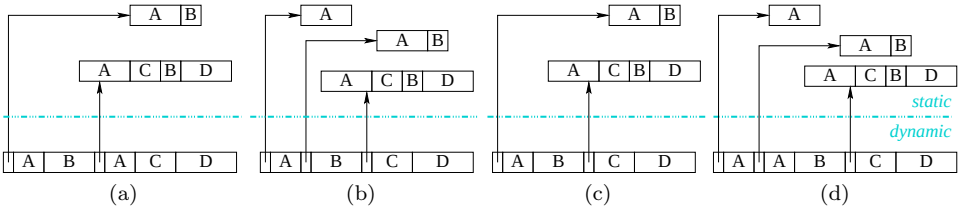


Fig. 6. Non-virtual inheritance: (a) pure NVI and repeated inheritance; (b) mixed NVI when $D \prec_{\text{nv}} B, C \prec_{\text{v}} A$; (c) in ideal and (d) actual implementations when $B \prec_{\text{nv}} A$ and $C \prec_{\text{v}} A$.

repeated inheritance[18], in the sense that some subobject is repeated in the object layout—in Figure 6-a, this is the case for A which is present in both subobjects of B and C. In the worst case of repeated inheritance, the number of tables and the number of repetitions become exponential in the number of superclasses, as it represents the number of paths from the considered class to its root superclasses. Consider a chain of n diamonds made of classes A_i, B_i, C_i and D_i, akin to Figure 2, such that $D_i = A_{i+1}$: A_1 will be repeated 2^n times in A_n layout.

4.1.3 *Mixing "virtual" and non-virtual inheritance.* This is the usual programming style in C++. Inheritance edges, i.e. $\prec_{\text{d}}$, are partitioned in two sets, $\prec_{\text{v}}$ and $\prec_{\text{nv}}$, according to whether the keyword `virtual` is used or not. The layout of a class C is made of two parts. The non-virtual part NV_C is computed as for pure NVI, but the concatenation on direct superclasses is restricted to the $\prec_{\text{nv}}$ relationship (thus $\prec_{\text{e}}$ is a subset of $\prec_{\text{nv}}$):

$$NV_C = attr_C + \sum_{C \prec_{\text{nv}} D} NV_D \tag{8}$$

where $attr_C$ denotes attributes introduced in class C and sum denotes subobject concatenation. When there is no such D, NV_C consists only of $attr_C$ plus a pointer at the method table, as in SST implementation. When such a D exists, $attr_C$ is concatenated to the corresponding subobject.

The virtual part V_C is a set of inherited non-virtual parts. On the whole, the object layout consists of *subobjects groups*—each group is made of subobjects in a fixed relative position, whereas groups have arbitrary relative positions. However, there are at least two ways of specifying the virtual part, since, with the diamond situation in Figure 2, mixing may be ambiguous, e.g. when class A is repeated in some direct superclass, and virtual in some other cases.

Ideal implementation. In an ideal specification, NVI would be used only for efficiency. Therefore, all partitions of $\prec_{\text{d}}$ are not sound, i.e. $\prec_{\text{nv}}$ should yield arborescent inheritance in order to avoid repeated inheritance. Moreover, when a class is inherited through virtual and non-virtual edges, only one implementation should be used (Fig. 6-c). Hence, the virtual part V_C is made of the non-virtual parts of all superclasses, either direct or indirect, which are not reachable through $\prec_{\text{nv}}$ paths, either from C (i.e. which are not already included in NV_C) or from superclasses of

[18]Meyer [1997] uses the term with a different meaning, i.e. denoting class hierarchies with cycles.

V_C (i.e. which are not already included in V_C):

$$V_C = min_{\prec^*_{\mathrm{nv}}}\{NV_D \mid C \prec D \texttt{ and } \mathrm{C} \not\prec^*_{\mathrm{nv}} D\} \tag{9}$$

where $\prec^*_{\mathrm{nv}}$ is the transitive closure of $\prec_{\mathrm{nv}}$.

Actual C++ *implementation.* C++ actually differs from this ideal implementation—although virtuality is an annotation of inheritance edges, the C++ object model makes it an annotation of classes. Thus, as far as one can extrapolate example-based specifications [Koenig 1998, p. 165], the virtual part V_C is the union of all virtual parts of superclasses, plus the set of non-virtual parts of direct superclasses inherited through $\prec_{\mathrm{v}}$:

$$V_C = \{NV_D \mid C \prec_{\mathrm{v}} D\} \cup \bigcup_{C \prec_{\mathrm{d}} D} V_D \tag{10}$$

Therefore, a superclass inherited twice, through virtual and non-virtual edges, is repeated (Fig. 6-d) and both links must be virtual in order to avoid repeated inheritance (Fig. 6-b).

4.1.4 *Casting.* In case of arborescent inheritance and pure NVI, the relative positions of two subobjects whose types are related by subtyping do not depend on the dynamic type. Thus upcast is unambiguous and, when required, the shift is static without access to the table. Downcast can also be made in a way that is very close to the SST coloring scheme, as $\Delta_{\tau_s,T}$ is independent of τ_d. Each subobject only has to implement the coloring scheme reduced to the types corresponding to the subobject. 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] prohibits some cases. However, according to the specifications of `dynamic_cast` [Koenig 1998], the effective implementation seems to use both $\Delta^{\tau_s}_{\downarrow}$ (in case of repeated inheritance) and $\Delta^{\Uparrow}$ (for sidecast). If the target is unambiguous in one of the two tables, the cast succeeds, i.e. no static prohibition is necessary.

4.1.5 *Evaluation.* Non-virtual inheritance has the great advantage of presenting overhead only when one uses MI (provided that implementation uses thunks). But repeated inheritance is a major semantic drawback. Sound semantics are possible by mixing virtual and non-virtual inheritance, but it is a matter of either hand-made optimization, or global analysis, as separate compilation cannot predict that B and C will not have a common subclass and where the `virtual` keyword must be used. Only a subsequent diagnosis, when defining D, is possible, or an automatic devirtualization analysis on the whole program (see Section 5.4.2). From a programmer's standpoint, mixing virtual and non-virtual inheritance may be quite complicated and this even worsens with inheritance protections. It should 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 single root,

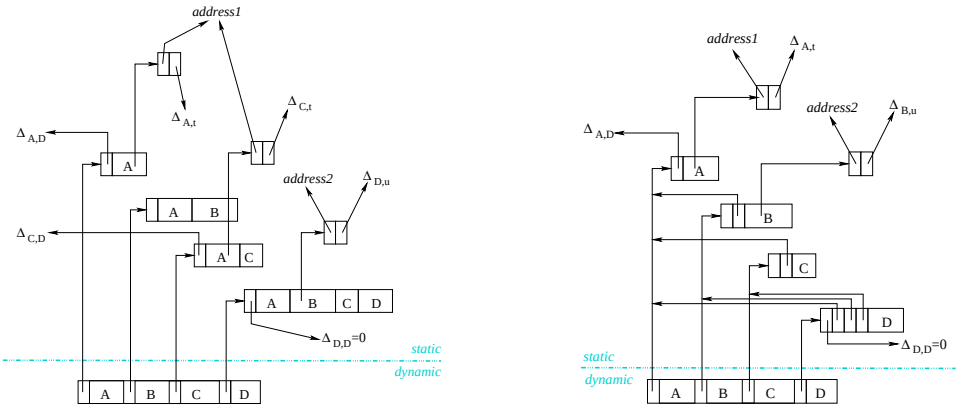


Fig. 7. Standard and compact method tables

`Object`, the class of all objects, AI reduces to SI.

Besides its semantic flaw, pure NVI is not less costly, in all generality, than standard MI implementation—NVI complexity is indeed exponential in the worst case (see Appendix A). Furthermore, empty subobject optimization applies only to classes without non-virtual direct superclasses.

4.1.6 *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 MST rather than plain MI (see also note 17, page 24). The main criticism may be that understanding the language requires understanding the implementation, e.g. by reading [Lippman 1996]. Specifications are not always clear and it is likely that, in complicated cases, actual compilers have different behaviour. NVI gave rise to some attempts to base its semantics on subobject models [Rossie et al. 1996; Snyder 1991a].

4.2 Compact method tables

Another drawback of standard MI implementation is its static space overhead—in the worst-case, table size is cubic instead of quadratic in SST. Two obstacles prohibit sharing of the tables of different static types for the same dynamic type—shifts and variant method offsets. A solution may be to restore the symmetry between attributes and methods. The first point can be reduced by two step shifting, as for attributes (5) and downcast (6). The second point is solved by applying to methods Invariant 3.1, which rules attributes. Therefore, the method table associated with 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, so 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 7, 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
load [table2 + #deltaOffset], delta2                    3L + B + 1
add object, delta1, object
load [table2 + #selectorOffset], method
add object, delta2, object
call method
```

In this 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 just to the cost of doubling the upcast table 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 marked 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, i.e. L extra cycles underestimate the effective extra overhead. Anyway, it is worthwhile to convince oneself that standard implementation offers, in some way, the best compromise between time and static memory. Actually, even in this compact variant, the worst-case size remains cubic, because upcasts require an encoding of the transitive closure of the inheritance relationship—encoding only the transitive reduction would entail non-constant-time upcasts. Various other compact schemes exist, but none of them are better than this simple one.

4.3 Less indirections with VBPTRs

Contrary to the previous variant which tries to reduce the static memory cost, some optimizations aim at reducing time overhead, at the price of dynamic memory. The best way to do this 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 (Figure 8, left). In C++ jargon [Ellis and Stroustrup 1990], they are called *virtual base pointers* (VBPTR) and, according to [Sweeney and Burke 2003], this technique is used in several C++ compilers. As a matter of fact, upcasts are very frequent, for access to attributes and polymorphic assignment. They are viewed by standard implementation as methods, which imposes a method table access (see Section 3.2.2, page 20). VBPTRs amount to viewing upcasts as read-only attributes, but they are implemented in each subobject. The general case of attribute access reduces to:

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

A similar optimization will consist of replacing $\Delta_{\Downarrow}$ by a pointer to τ_d subobject in the object layout [Sweeney and Burke 2003].

Evaluation in standard implementation. As method tables, $\Delta^{\uparrow}_{\tau_s}$ tables are, in the worst case, quadratic in number and cubic in size. Therefore, with standard

MI implementation, 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. Of course, splitting a class C adds overhead in all its D subclasses, which is linear in the number of classes between C and D. 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). Instance creation is no longer linear in time but quadratic in the number of superclasses, since one must initialize VBPTRs. Furthermore, VBPTRs are compatible only with the first case of empty subobject 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). As for the pointer at τ_d, the overhead is smaller, as it is equal to the subobject number, but the gain is also smaller as shifts to the dynamic type are less frequent. On the whole, the large dynamic overhead of VBPTRs is not counterbalanced by a small gain in time.

A by-product of VBPTRs is the ability to implement instance classification, a monotonous and type-safe 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 does not need to be connected—specializing an object's class amounts to adding subobjects and updating pointers at method tables.

Evaluation in NVI and optimizations. When mixing virtual and non-virtual inheritance, the overhead is far smaller: this explains why some actual compilers use VBPTRs. The reason of this smaller overhead is twofold. Firstly, with NVI, a class can reuse the VBPTRs of its direct non-virtual superclasses, which is not the case with $\Delta^{\uparrow}_{\tau_s}$ tables since they are relative to τ_s. Secondly, one may distinguish two kinds of VBPTRs: e-VBPTRs are *essential* because they reference superclasses reachable only through $\prec_{\mathrm{v}}$, whereas i-VBPTRs are *inessential* and can be replaced by a static shift w.r.t. an e-VBPTR. Implementing only e-VBPTRs adds one extra shift to upcast, but no overhead for attribute access as the shift may be added to the offset at compile-time. Furthermore, pointers to τ_d are required only in root (w.r.t. $\prec_{\mathrm{nv}}$) subobjects. Instance classification is no longer possible as subobjects may have fixed relative positions.

Sweeney and Burke [2003] propose a general framework for analyzing the space overhead of various object layouts, according to the distribution 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 aimed at reducing VBPTR overhead. Some of them are global and compare badly with techniques used in global compilation, i.e. coloring (see Section 5.3) or tree-based dispatch (see Section 5.2.2), which do not need shifts. Other optimizations are usable in separate compilation—they use bidirectional tables (see Section 6.1). On the whole, with an optimal balance of virtual and non-virtual inheritance, the dynamic overhead of VBPTRs remains high, but it is of the same order of magnitude as the number of subobjects (see appendix A).

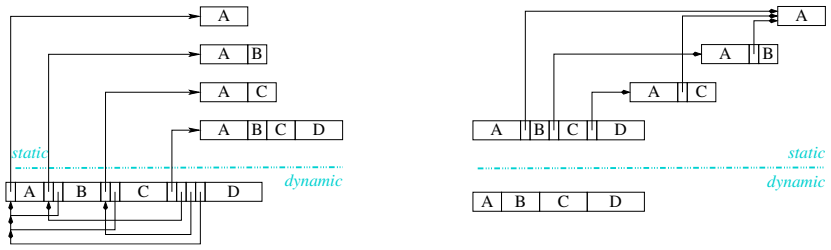


Fig. 8. Implementation variants with VBPTRs (left) and method table flow (right).

4.4 Attribute accessors and accessor simulation

As aforementioned, attributes and methods are very similar entities and some languages behave similarly towards attributes and methods. In CLOS [Steele 1990], accesses to attributes are made through special read and write *accessors* methods (*generic functions* in CLOS jargon). In SATHER [Szyperski et al. 1994], an attribute contributes to the class type by its accessor signatures. In EIFFEL [Meyer 1992; 1997], a method without argument may be overridden as an attribute. Attribute implementation can be encapsulated in accessor methods—when an attribute position changes in a subclass, one needs only to override its accessors. In that way, attribute offsets do not matter.

Two variants must be considered: either accessors are actually defined as methods or they are simulated, in which case the 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 20). In a second step, attribute offsets can be factorized if attributes are grouped by subobjects, with Invariant 3.1. 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 $\tau_s \neq T_p$.

There is somewhere a catch in simulating accessors, 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 in the frequent case of access to `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 in SI). The appropriate version is chosen at link-time.

However, true accessors may be needed for some language features as redefining a method without parameter as an attribute (EIFFEL) or implementing the full specifications of the CLOS keyword `:allocation` (see Section 7.5).

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 do not matter, direct access to method or subobject offsets is needed. Thus accessors are not an alternative in themselves but we shall see different ways of incorporating accessor simulation in a general implementation framework, i.e. preserving object reference variance (see Section 6.1.3) or recovering SST invariance, with a global computation (see Section 5.3) or a flow of method tables (see Section 4.5). Attribute accessors can be used with any method implementation.

4.5 Method table flow

A radical alternative to standard implementation involves moving the links between object layout and method tables from data structures to program data flow. Any object value is then twofold—the object itself and its method table. One can think of it as though the compiler replaces any variable by two variables, one for the object and the other for the method table. Of course only polymorphic variables are concerned: monomorphic variables, typed by a primitive type, do not need tables. The object layout consists only of the fields for attributes, almost doubled because of the tables but without any pointer to method tables or VBPTRs. Therefore, accesses to attributes should be made by simulating accessors. A secondary consequence is that method parameters and returned values are almost doubled.

As for method tables, they have the same structure as in standard implementation, with VBPTRs in the method tables of the same τ_d which are all linked with each other. Invariant 2.1 is respected for the object part of the value, whereas Invariant 3.3 rules the method table part. All the pointer adjustments needed by standard implementation, for casting or for the receiver, are done on the method table in such a way that the method table part of a value is always the table corresponding to its static type (Figure 8, right).

In this approach, any variable `x` is twofold: `x` itself which references an object and the method table `tablex`. In the polymorphic assignment `x:=y`, the appropriate table is obtained from `tabley`:

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

Method call to `x` follows:

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

Evaluation. Maintaining a systematic flow of method tables in parallel to data flow seems at first glance to be a time efficient solution, just at the cost of almost doubling memory space, both dynamic and static, in the code or in the data structures. However, the actual overhead might be less than this pessimistic view. Many data are monomorphic and this technique would have no influence on them. Dynamic memory overhead will be partially counterbalanced by the gain of all pointers, w.r.t. to method tables or VBPTRs. As a matter of fact, the proposed implementation is as time efficient as standard implementation with VBPTRs, whereas VBPTRs and pointers to method tables often more than double dynamic memory (see appendix A).

Increasing the code size with extra parameters is counterbalanced by the fact that there is no longer a need for an access to the object to get its method table—moreover, an optimized compiler will remove all accesses to tables which are not used.

On the whole, while the approach is probably not an efficient alternative, it should be considered for some very specific niches. Access to interface-typed entities in a MST framework is an example (see Section 6). This may also be a good technique for implementing universal types, i.e. common superclasses of primitive types and standard classes, which avoids using *boxing* as in JAVA (see Section 7.2). However, the main drawback will involve collections, i.e. arrays, since polymorphic arrays

will double (see Section 7.4). Debugging the compiler would also be a mess. This technique seems to be very close to the way method invocation is implemented in the EMERALD language, where method tables are generated at load-time in a lazy way [Jul 2008].

4.6 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 of avoiding this drawback to the detriment of semantics. A general solution could be that languages allow to express the fact that a class should not be specialized in MI or, more generally, with non-arborescent inheritance—this would be to 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 with accesses to `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 time overhead and may be unable to effectively gain on static memory, whereas VBPTRs can be envisaged only in a context of massive NVI. However, as VBPTRs are used in some actual C++ compilers (Microsoft and SUN compilers, but not GNU), caching method tables could be considered as a valuable alternative.

5. GLOBAL TECHNIQUES FOR MULTIPLE INHERITANCE

Previous sections considered only separate compilation and, mostly, dynamic loading. Separate compilation is a good answer to the modularity requirements of software engineering—it provides 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. Moreover, separate compilation provides the best framework for reusability which strongly suggests the *open world assumption* (OWA).

On the contrary, global compilation supposes the *closed world assumption* (CWA). However, it also has advantages, as it allows many optimizations. Moreover, global linking may be envisaged as a tradeoff between global compilation and dynamic loading. Here we present some alternatives based on a complete knowledge of programs, at least of class schemata, whether this knowledge takes place at compile or link time.

5.1 Advantages of global techniques

5.1.1 *Hierarchy closure.* The first advantage of global techniques is that the class hierarchy is closed—no extra class can be added after the compilation, unless there is a new complete compilation of the whole hierarchy. It is then possible to know whether a class is specialized in single or multiple inheritance, whether two unrelated classes have a common subclass or not, and so on. Moreover, the external schema of each class is known. Hence, conversely, one knows where methods are defined—methods with a single definition may then be treated as static calls (monomorphic calls). In dynamic typing, this is known as the *unique name* heuristics [Calder and Grunwald 1994]—despite its simplicity, its effect is not small

as this applies to almost 45% of methods in SMALLTALK. In static typing, the term of *unique definition* would be better. Therefore, *class hierarchy analysis*[19] is a promising approach for optimizing programs whenever the whole hierarchy is available [Dean et al. 1995].

With static typing. Hierarchy closure brings even more information in a static typing setting: 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 do not need it [Eckel and Gil 2000] (see Section 4.1). Alternatively, one can decide about primary and secondary superclasses (see Section 6.1.4).

5.1.2 *Knowledge of method code.* When the compilation itself is global, 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 imply an underlying general implementation technique.

5.2 Global compilation in dynamic typing

In a dynamically typed language like SMALLTALK, the lack of type annotations makes separate compilation quite unefficient. So many techniques have been worked out in the framework of dynamic typing, e.g. in the SELF and CECIL languages—of course they all apply to static typing as well (the converse being untrue).

5.2.1 *Dynamic typing and single inheritance.* Note first that, with dynamic typing, SI is no longer a simplification, due to *introduction overloading* (see note 6, page 10), i.e. when the same method name is introduced in two unrelated classes, Invariant 2.2 does not hold, at least in separate compilation. This also concerns attributes as soon as they are not encapsulated as in SMALLTALK. The JAVA type system may be understood as the minimal type system required for statically typing SMALLTALK—the interface notion is the answer to introduction overloading. So with dynamic typing, implementation is at least as difficult as with MST, or even with plain MI when attributes are not encapsulated.

5.2.2 *Implementation 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 here under the *type prediction* name, which are based on decision trees. Therefore, one may classify them into *table-based* and *tree-based* techniques. Whereas table-based techniques are mostly applicable both at link-time and compile-time, tree-based techniques are a matter of code generation, i.e. compilation, unless one considers a linker equipped with such capabilities—this would make the fronteer between compiling and linking quite fuzzy, but JIT compilers and dynamic loading already make this fronteer fuzzy.

[19] '*Class hierarchy analysis*' is a common term that denotes any analysis of the class hierarchy. It is however also the label (CHA) of the specific analysis proposed by [Dean et al. 1995].

Compacted large table. Table-based techniques involve 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, $\sum_C M_C$. Two techniques for compacting that table have been proposed: row displacement [Driesen and Hölzle 1995], after a sparse table compression techniqued proposed by Tarjan and Yao [1979], and method coloring [Dixon et al. 1989; Pugh and Weddell 1990; André and Royer 1992; Ducournau 1991; 1997]. The latter technique will be further detailed.

In both cases, the result is a table, each entry of which contains a single class-method pair or is empty. However, the lack of static type checking means that a method may be called on a wrong type receiver, 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 selector and the actual one. Static typing makes this extra test useless. On the whole, there are few empty entries, less than 10% in most benchmarks, with a 50% upper bound for all experiments—compared to standard MI, the total size remains close to $\sum_C M_C$ (see appendix A).

Type prediction and tree-based techniques. Type prediction originates from the idea that a given type may be considered as more likely for some method call site, for instance, type `integer` for the method `+`. So type prediction consists of compiling a method call by a comparison between the expected type and the actual receiver's type—if the test succeeds, a static call is done, otherwise the call must be done using another technique.

There are many variations of 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 this case, the call sequence is a small *decision tree* which compares the receiver's type with all the expected types.

The technique is not a priori sufficient since misprediction or cache misses must be handled, when the receiver's type is not amongst the predicted types. Therefore some underlying technique is needed, for instance a dynamic *lookup* à la Smalltalk. An alternative is to rule out cache miss—the decision tree must then exhaust all possible types. This requires the CWA and, in practice, this is realistic only with static typing as the available static type amounts to an upper bound of all expected types. Type analysis may give more precise information, the *concrete type* (see Section 5.4.3). Finally, when the number of expected types is large—it may exceed 100—it is better to balance the tree. This gives *binary tree dispatch* (BTD) [Zendra et al. 1997].

On the whole, indirect branching (B latency) and memory accesses (L latency) are avoided. The entire call sequence is made of conditional branching: modern processors are equipped with a prediction capability such that a predicted branch has a 1-cycle cost, whereas an unpredicted one has a B-cycle cost. In contrast, the number k of expected types, thus the branch number, may be large and the search

is in $O(\lceil log_2(k) \rceil)$. If one assumes a uniform distribution of type probabilities, one may expect that the average number of well predicted branches will be $log_2(k)/2$. Therefore, the average decision cost would be $log_2(k)(B+1)/2$. However, a uniform distribution is quite unlikely—in practice, the same type repeatedly occurs at the considered call site and the misprediction rate will be far lower than $1/2$. Moreover, one should not compare the two techniques by taking the ratio of cycle numbers, here $(log_2(k)(B+1)/2)/(L+B)$, i.e. one must also add to both the risk of cache miss. In both cases, there is more or less the same risk when branching to the method, but all other branches in the decision tree are local. Furthermore, the load memory access adds some risk. Hence, the exact but unknown ratio must be $(log_2(k)(B+1)/2+x_1)/(L+B+x_1+x_2)$, where x_1 and x_2 are the unknown cache miss risks. All this explains why experiments [Driesen et al. 1995; Zendra et al. 1997] show that tree-based dispatch is very efficient.

On the whole, various detailed implementations should be considered: if sequences, binary trees, switch tables. [Driesen and Zendra 2002] analyzes the efficiency of these variants, according to the call site patterns, i.e. the receiver's type may be constant, random, cyclic, etc.

Subtyping test. In tree-based techniques, the comparison between the effective and expected types may be either an equality test or a subtyping test. With the former, all types must be exhausted—though inequalities allow some grouping—whereas only callees need to be, in the latter. But the subtyping test is expensive in the general case. [Queinnec 1998] handles only SI and uses Cohen's [1991] coloring technique, which is prefered to Schubert's numbering as it is incremental. *Interval containment* [Muthukrishnan and Muller 1996] uses Schubert's numbering. Type prediction and inline cache techniques use equality tests. BTD uses inequality tests.

Mixed techniques. There are many ways to mix table-based and tree-based techniques, by putting either tables in tree leaves [Queinnec 1998] 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 these techniques is to save static memory w.r.t. standard table-based techniques—the size of static data structures may be much smaller than $\sum_C M_C$. However, this is detrimental to time efficiency, i.e. as with type prediction, method call is no longer time-constant. Moreover, unlike pure type prediction, table access is required.

Attributes. The case of attributes is rarely covered in the literature. This may be explained by the fact that they are often encapsulated in the methods, as in SMALLTALK: only accesses to `self` attributes are allowed. As `self` has the uncommon feature of being the only statically typed entity of the language, Invariant 2.2 holds for attributes, in case of SI. Accessor simulation is sufficient for MI but non-encapsulated attributes require plain accessors, because of introduction overloading (see note 6, page 10).

5.3 Coloring heuristics

We now detail the coloring approach as it is quite versatile and it naturally extends the SST implementation to MI.

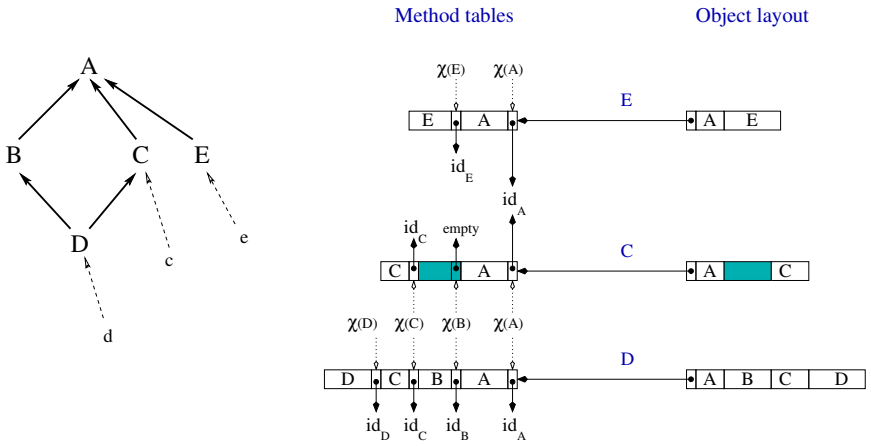


Fig. 9. Unidirectional coloring applied to classes, methods and attributes—the holes (in grey) in the C tables are reserved space for B in the tables of D. A and B classes are presumed to have the same implementation as in Figure 1 and the diagram follows the same convention.

5.3.1 *Principle of coloring.* 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 on large coloring computation time and the technique has been considered uneffective ever since. However, a previous work by Pugh and Weddell [1990] had reported positive results, confirmed later by Ducournau [1991; 1997]. One may define coloring as upholding SST Invariants 2.1 and 2.2 in MI.

Invariant 5.1 (Color). *An attribute (resp. method) has an offset (color) invariant by specialization. Two attributes (resp. methods) with the same color do not belong to the same class.*

As a corollary, two classes which have two different attributes (resp. methods) with the same color cannot be specialized by a common subclass. Here is the reason why coloring is global. An injective numbering of attributes (resp. methods) verifies the invariant, so this is a matter of optimization. The first proposition, by Dixon et al. [1989], André and Royer [1992], was to minimize the color number, which is a NP-hard graph coloring problem [Garey and Johnson 1979]. The first improvement, proposed by Pugh and Weddell [1990] and Ducournau [1991; 1997], consists of minimizing the total size of method (resp. attributes) tables—tables resulting from coloring are then similar to SST tables, except that they may contain *holes*, i.e. empty entries. The second improvement, by Pugh and Weddell [1990], is bidirectional coloring, with positive and negative colors. These new problems were proven to be as difficult as the original problem [Pugh and Weddell 1993; Takhedmit 2003], therefore heuristics are needed and some experiments, by Pugh and Weddell [1990] Ducournau [1997; 2002a; 2006] and Takhedmit [2003], show their efficiency and that improvements are effective. Dixon et al. [1989], André and Royer [1992] first applied coloring to methods, whereas Pugh and Weddell [1990], and Ducournau [1991] applied it to attributes, and Vitek et al. [1997] and Ducournau [2002a] to

Table IV. Coloring bibliography

paper		optimization criterion			colored entities			%
		#colors	size	bidirect.	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

classes, as a generalization of the SST technique of Cohen [1991].

Invariant 5.2 (Class Color). *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 Weddell [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, a class coloring experiment on larger MI programs 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).

Regarding dynamic memory, the overhead may be significant—one should minimize the total dynamic memory, which needs a profiling of class instanciation. A conservative solution would be to simulate accessors (see Section 4.4) instead of coloring attributes—the offset of each subobject would be colored in the same way as methods. Dynamic memory overhead disappears, to the detriment of a constant time overhead for attribute accesses—it may be reduced by Myers' [1995] double compilation. [Zibin and Gil 2003b] proposes a bi-dimensional improvement, which increases the rate of attributes which do not require indirection.

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 vs. SST only concerns static tables and accesses to some attributes, but this overhead remains far from all other MI implementations overhead (Sections 3 and 4). Ducournau [2006] surveys the approach. Note that row displacement [Driesen and Hölzle 1995] would give similar efficiency for method tables and subobject offsets—but it obviously does not apply directly to attributes. Application to subtype testing has not been yet considered, but would be straightforward, according to the equivalence of all mechanisms.

5.3.2 *Link-time coloring.* Like all global layout techniques, coloring only requires the *external schema* of classes and it might be computed at link-time. Furthermore, the choice between attribute coloring and accessor simulation might also be done at link-time, as the code for the former is exactly the same as the efficient version of the code for the latter. Therefore, the linker may choose the best implementation, according to a user-defined overhead threshold. Although Pugh and Weddell [1990] already noted this possibility, the approach does not seem to have been tested until recently, in the Prm language [Privat and Ducournau 2005;

Privat et al. 2006; Privat and Morandat 2008].

5.3.3 *N-directional coloring.* A generalization to *n-directional perfect*—i.e. without holes—*coloring* has been proposed by [Pugh and Weddell 1993], the independantly rediscovered by [Zibin and Gil 2003a]. Unidirectional (resp. bidirectional) coloring corresponds to $n = 1$ (resp. $n = 2$). Bidirectional colori,ng is a significant improvement. However, when $n > 2$, the approach leads to several ($\lceil n/2 \rceil$) independant subobjects, which must be handled either with pointer adjustments, or with accessor simulation. As the computation is global, if one discards the first solution, it remains an optimization of accessor simulation and Myers's double compilation where the number of efficient accesses is improved.

5.4 Global optimizations

Separate compilation allows only a few optimizations, with limited applicability (see Section 2.4). Global compilation makes these optimizations more powerfull and allows many others.

5.4.1 *Method copy.* When the source code of superclass methods is known at compile-time, it is possible to copy the code of inherited and not overridden methods in a class. The main advantage of this technique, termed *customization* by [Chambers and Ungar 1989], is that `self` becomes monomorphic—all method calls to `self`, which are quite numerous, can be compiled into a static call, without any 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 class hierarchy analysis—may be removed from method tables. Finally, method copy gives more precise type information, in case of type overriding, either safe as return type, or unsafe as parameter or attribute types. EIFFEL's *anchored types* and *virtual types* [Torgersen 1998] 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 to the detriment of static space—method code is duplicated with a factor linear in the class number. Thus this technique cannot be envisaged without an association with optimizations that drastically reduce the size of the generated code. Obviously, *dead code* must be ruled out.

A similar but more general technique is method *specialization*, which consists in compiling different versions of a method definition according to the concrete type of the receiver: `self` is no longer monomorphic, but its concrete type may be determined in such a way that inner calls are static [Dean et al. 1995]. Due to its selectivity, specialization avoids a too large increasing of compilation time or code size.

5.4.2 *Automatic devirtualization.* It is common opinion that "repeated inheritance is an abomination" [Zibin and Gil 2003a, note 2] and that the C++ keyword `virtual` is justified only by efficiency (see Section 4.1). A static global analysis can determine which classes must be implemented as virtual, and which classes do not

[Gil and Sweeney 1999; Eckel and Gil 2000]. More precisely, it can determine how to share subobjects. In the diamond example, it is possible to merge the subobjects D and B, and on the other hand, the subobjects A and C, thus saving 3 method tables in Table II. Devirtualization[20] involves partitioning $\prec_{\mathrm{d}}$ into two relationships, $\prec_{\mathrm{v}}$ and $\prec_{\mathrm{nv}}$, in such a way that the latter induces AI. A simple devirtualization scheme involves marking as virtual all classes which have two direct subclasses with a common subclass, i.e. all A such that $\exists B, C, D : C \neq B, D \prec B \prec_{\mathrm{d}} A$ and $D \prec C \prec_{\mathrm{d}} A$. In a second step, one determines virtual edges ($\prec_{\mathrm{v}}$), 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. Of course, the choice of the only non-virtual edge is a matter of optimization.

5.4.3 *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 always compute an approximate (upper bound) of exact concrete types, they may be more or less precise and costly. [Grove and Chambers 2001] presents a survey of the 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]. Finally, type analysis open the way to *code specialization*, a generalization of *customization* at the method level [Dean et al. 1995] or at the class hierarchy level [Tip and Sweeney 2000].

5.4.4 *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 with a type analysis highlights classes which are never instanciated, methods which are never called and attributes which are never read, 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, for applications where class instanciation results from an interaction with a user, a DBMS, another program or a network, especially for languages equipped with a *meta-object protocol* like Java (package `java.lang.reflect`). In this context, all application classes are potentially alive, as well as all their methods which can be activated in the same interactive way.

5.4.5 *Inlining.* This is a common optimization of procedural languages: it involves copying the code of a callee in the caller, for instance when the call is

[20] As the `virtual` keyword has two roles in C++, one must expect the word *devirtualization* to be used in two different meanings—we consider here `virtual` inheritance. Regarding `virtual` functions, devirtualization amounts to the detection of monomorphic calls.

monomorphic and the method is small or not often called. In all cases, this is a final optimization of monomorphic calls and, in global compilation, it can also be applied to type prediction. Attribute inlining must also be considered, which consists of replacing an object address by the object itself in the layout of another object, as with the EIFFEL `expanded` keyword. It will save a `load` instruction but it is only sound under two conditions—type analysis must prove that the attribute is monomorphic and alias analysis must prove that the same object will not be inlined in two different objects.

5.4.6 *Profiling.* Profiling involves gathering statistics on program executions in order to optimize some particular points, including memory management. In OOP, the technique can be used at various levels. At the call site level, *type feedback* [Agesen and Hölzle 1995] is an improvement of *type prediction*, where expected types are ordered according to their measured frequency. The technique was already used in CEYX, ten years before [Hullot 1985]. At the object layout level, profiling can measure the instance number of each class in order to optimize the total number of holes in attribute coloring. *Object splitting* [Chilimbi et al. 1999] has been proposed to improve heap locality and reduce cache and page misses. It consists of splitting the object layout in two subobjects: the first one contains frequently used attributes and points to the second one which contains rarely used attributes.

5.4.7 SMART EIFFEL. The GNU EIFFEL compiler is typical of the use of these global techniques in the framework of a statically typed language. It is based on a double a priori, i.e. global compilation without method tables. In the object layout, the pointer at the method table is replaced by the class identifier[21]. The compiler 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 balanced binary decision tree, based on equality tests on type identifiers—the same technique is used for polymorphic accesses to attributes, when the offset varies 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 these techniques in detail, along with experimental results which show a clear improvement w.r.t. existing EIFFEL compilers.

Note that the approach is probably not adapted to separate compilation and global linking as: i) implementation is tree-based, ii) most optimizations need to

[21] Of course, method table addresses might serve as class identifiers, but short integers are better. A simple class numbering allows, for instance, more efficient dense switch tables [Driesen and Zendra 2002].

know the code of the whole program, iii) all techniques are closely connected, with each one making the others more efficient. Separate compilation would impose to leave both customization and inlining: it would be interesting to experiment this to see at what point it would make the implementation inefficient.

5.5 Link-time optimizations

Almost all global optimizations might be applied at link-time. Whatever the optimization, the approach amounts in generating different versions of the same source sequence, or in postponing the corresponding code generation to link-time.

5.5.1 *Double or conditional compilation.* In the first approach, separate compilation generates two or more versions of the code for, either a class, a method or a single statement. The best version is chosen at link-time, on the basis of the *external schema* of all classes.

The double compilation technique was first proposed by Myers [1995]: it aims at optimizing accesses to attributes with accessor simulation (see Section 4.4) and it implies a small global analysis to determine whether a class is all the time 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. This is only interesting for optimizing accesses to `self` and to other entities typed by the current class. The main point with double compilation is its coarse grain.

A finer grain could be obtained with conditional compilation, which would consist in generating code with several versions of some sequences, e.g. for monomorphic, oligomorphic or megamorphic[22] call sites. The code would be tagged, in an HTML-like way, each tag expressing the condition required for choosing among the different versions. To our knowledge, conditional compilation remains for now a speculative idea—it requires mastery of the compiler back-end and it appeared to be somewhat incompatible with compilation towards C code, contrary to the expectations of [Privat and Ducournau 2004].

5.5.2 *Link-time generation of dispatch code.* In all implementation techniques, the dispatch code for method calls is a quite simple piece of code which might be generated at link-time, provided that linkers are equipped with such capabilities. In that approach, followed by SmartEiffel and Prm [Privat and Ducournau 2005], a method call reduces to a simple static function call. This function generated at link-time consists of the small required sequence of code—specific to a given implementation technique and to the considered call site—which ends by jumping to the callee address. Therefore, these pieces of code are exactly like *thunks*—instead of calling a method, one calls a thunk which jumps to the method and the resulting overhead is only of one jump. The only difference is that the static and dynamic calls are inverted.

The present approach might have the same kind of benefits as thunks, namely the fact that in optimal cases, the thunk may be the method itself. Thunks take

[22] *Polymorphism* declines into more accurate neologisms: monomorphism (one type), oligomorphism (a few types) and megamorphism (many types). To be precise, we are not concerned here by the number of types in the concrete type but by the number of method definitions which must be discriminated by the dispatch.

their advantage from the fact that there are many null shifts (see Appendix A.4). In the present case, the gain is only possible for monomorphic calls.

To gain from this in a global linking setting, one must assume that some global optimization allows to detect a large number of monomorphic calls. The *unique name* heuristics—and its statically typed variant—gives quite good results, as benchmarks show that, in average, more than 50 % methods have only one definition (see Appendix A.4). One makes easily better, with *class hierarchy analysis* (CHA) [Dean et al. 1995], when considering all call sites such that the method is not overriden in the subclasses of the receiver's static type. All this requires only that the *external schemata* of all classes are available at link-time. In that simplest case, the dispatch function name must only include, in some simple mangling syntax, the name of the method—possibly with the types of parameters in case of static overloading—and the receiver's static type. All the call sites with the same method and static type share the same function.

Separate type analysis. More sophisticated optimizations are possible, especially when using more accurate type analysis. But accurate link-time type analysis needs another framework. Type analysis requires source code, but it may be 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 and Ducournau [2004; 2005] propose to join these two phases by an *internal schema*, produced by intra-class analysis during separate compilation and which stands for an abstract of the class code, specifying the flow of types in the methods. Internal schemata are closely related to the notion of *template* proposed by [Agesen 1996]—roughly speaking, the internal schema of a class is the set of templates of all method definitions of the class. At link-time, inter-class analysis uses these internal schemata to construct the call graph and determine concrete types for all expressions in internal schemata.

Such an approach would allow to detect dead code and to rule it out from the executable. However, local optimizations like inlining are not directly possible but an optimized version could use the result of the previous global analysis in a separate compilation. A similar approach were proposed in a functional language framework [Boucher 1999; 2000].

Finally, with link-time generation of all call sequences, the approach also allows static monomorphic calls, decision trees for oligomorphic sites and coloring for megamorphic ones.

Access to attributes and subtyping check. Generating code at link-time for other mechanisms than method call is also possible. However, accesses to attributes are usually inlined and this would be a considerable overhead to call a function for this. This is surprisingly the Smart Eiffel choice.

Regarding subtyping checks, the issue is more to see what could be globally optimized. The best optimization expected from type analysis would be that the test is useless, because it will either always succeed or always fail—it is the equivalent of monomorphic cases. In a middle case, the optimization would involve choosing between different techniques, e.g. coloring or BTD, according to the receiver's polymorphism. However, these optimizations suppose that a function is called anyway,

and it would be likely more efficient to uniformly implement subtype test by inlined class coloring instead of a call to a possibly empty function.

The Prm *compiler-linker.* The Prm compiler has been designed as a testbed for experimenting various implementation techniques in different compilation schemes, from pure separate to pure global compilation, with some middle terms as global linking, global compilation with precompiled libraries, etc. The linker implements coloring and BTD, with various type analyses like CHA, RTA, XTA and CFA-0. The compiler architecture benefits from the original notions of modules and class refinement that are key features of the Prm language [Ducournau et al. 2007]—it makes it easy to replace a module implementing some mechanism by a given technique by another module that implements the same mechanism with an alternative technique. The Polyglot compiler relies on a similar modular architecture [Nystrom et al. 2003; Nystrom et al. 2004; Nystrom et al. 2006]. In Prm, modules are code units that are compiled into C files that are linked together, firstly by a dedicated linker that generates all the required thunks, then by the common linker. This compilation scheme presents however a current limitation—dead code inside living modules cannot be eliminated.

5.5.3 *Counter-examples.* Using a global technique at link-time, after a separate compilation, must be carefully evaluated. Automatic devirtualization is a good counter-example. It may reduce the number of subobjects, thus the need for pointer adjustment, but there is no way to know, at compile-time, whether a shift will be needed or not. A double compilation of a class, or even of a method, is not the solution: each possible upcast should have two versions. 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 in the size of the tables, either static or dynamic—in case of SST, there will be a SST layout (augmented with upcast tables) but with a MI code. However, upcast tables also can be saved if offsets $i_{\tau_s}(C)$ are computed at link-time. Therefore, link-time devirtualization could be only envisaged as an optimization for VBPTRs implementation (see Section 4.3), but it is likely that it will badly compare with link-time coloring. 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 establish that some second kind merging is possible in all subclasses, but it would have no effect on the code or the layout.

Actually, standard MI might gain from this optimization of monomorphic calls but the standard MI overhead would remain, and it is not worth the trouble doing this optimization while keeping the subobject-based implementation.

5.5.4 *Load-time global optimizations.* Applying global optimizations to plain MI is a great challenge, especially at load-time. There are however several ways to

Dynamic loading and adaptive compilers. At first glance, the present approach seems impossible in the framework of dynamic loading and JIT compilers, since global information is always missing. Nevertheless, an optimization is possible

in adaptive compilers [Arnold et al. 2005]. When loading a class, each call site in the loaded code may be recognized as either already polymorphic, or possibly monomorphic, on the basis of a simple class hierarchy analysis, restricted to already loaded classes[23]. The call is compiled, in the former case, as usual, and, in the latter case, into a static call to a dispatch function, which jumps to the method address. It does not seem appropriate to use more accurate type analyses, as they are too costly for a frequently used runtime mechanism.

At the time of a further class loading, the previous call site may become polymorphic and the dispath function must be updated. There are two ways to do that. i) Replacing the method address by the address of another piece of code has the disadvantage to add one jump. ii) Dispatch function may have enough space to receive a longer code sequence: for instance, if usual call sequences are 3 instructions long, this would not be a considerable overhead to reserve 3 words for each dispatch function. In any case, dispatch functions must be allocated in the data area, as they must be writable. In the case of interface-typed receiver (see Section 6), the optimization applies also, but one can distinguish one more level, between the static call and the general case, i.e. the interface may be implemented by only one class (and its subclasses). Therefore, the call may be compiled as a class-typed receiver call.

This is a very appealing optimization for JIT compilers, which may be new—e.g. [Arnold et al. 2005] does not mention any similar technique. There remains an issue, i.e. the detection of transition from monomorphism to polymorphism, for all call sites. This must be done each time a method is overridden, for all classes between (in the $\prec$ sense) the class of the overridden method and the class of the overriding one. The overhead at load-time does not seem important, but experiments should valid this assumption.

Incremental coloring. On the other hand, global techniques such as coloring are intrinsically not incremental, unless in SST. In an incremental setting, when a class has more than one superclass, it introduces conflicts between the colors of the superclasses, which are quite difficult to solve and propagate. Enlarging method tables or allocating new method tables would not be possible: therefore the implementation must rely on an extra indirection, the color table being pointed by the method table. In that way, it is possible to reallocate color tables when they must be enlarged, without any need for updating old instances. So, incremental coloring is possible but quite intricated. Palacz and Vitek [2003] propose to use it in a restricted form, namely in MST, for subtyping tests when the target is an interface.

The aforementioned equivalence of the three mechanisms (Section 2.5) makes it possible to enlarge the approach to method invocation and attribute access [Ducournau 2006]. First, methods and attributes are grouped by introduction classes in the method tables and object layouts. The relative positions of these groups does not matter. Then, the color table is made of 3-fold entries—the class color which

[23]One assumes that, when a class code is loaded: i) all its superclasses have been previously loaded, ii) the external schema of all imported classes has been already loaded. If this is not yet the case, recursive load is always possible.

serves for subtyping test, the attribute group offset for simulating accessor, and the method group offset for method invocation. This implements method invocation and subtype test with only an extra indirection—this is an acceptable overhead. Access to attributes involves accessor simulation with an extra indirection—this is a less acceptable overhead compared to direct access. However, the main issue remains the load-time recomputation, since the coloring problem is NP-hard and heuristics are at least cubic.

6. SINGLE INHERITANCE AND MULTIPLE SUBTYPING (MST)

Between the two extreme cases of SST and MI, is the middle case where classes are in SI but types are in MST, whilst class specialization remains a special case of subtyping. JAVA is a typical example [Gosling et al. 1996; Grand 1997], whereby the `extends` relation between classes is single, and the `implements` relation between classes and interfaces and the `extends` relation between interfaces are multiple. Only classes may define attributes together with method bodies and may have proper instances—interfaces are *abstract* classes and only define method signatures. Many 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 .NET, C# [Microsoft 2001] or EIFFEL# [Simon et al. 2000]. Furthermore, the absence of multiple subtyping was viewed as a deficiency of the ADA 95 revision, and this feature was incorporated in the next version [Taft et al. 2006].

This special middle case deserves examination only in the framework of separate compilation and dynamic loading, since global techniques, even at link-time, proved their efficiency on plain MI. Once again, this is the case of JAVA and .NET platforms.

In such a framework, the SST invariants 2.1 and 2.2 cannot hold for interfaces—more precisely, a method introduced by some interface will be located at different offsets in the different classes which implement the interface. However, standard MI would be too complicated, as both invariants could hold for classes. Hence, two kinds of solution can be designed, according to whether one simplifies MI implementation or complicates SST implementation.

6.1 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.

6.1.1 *Principle.* Starting from standard implementation, the first step consists of conciliating different method tables with attribute invariance. Due to attribute invariance, there is only one subobject for the class and all its superclasses and one method table for this subobject. 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 $\Delta^{\tau_s}_{\Downarrow}$, and this value will serve to shift the receiver in a method call, when it is typed by an interface. No shift is needed for a class-typed receiver, as it points to offset 0, whereas and

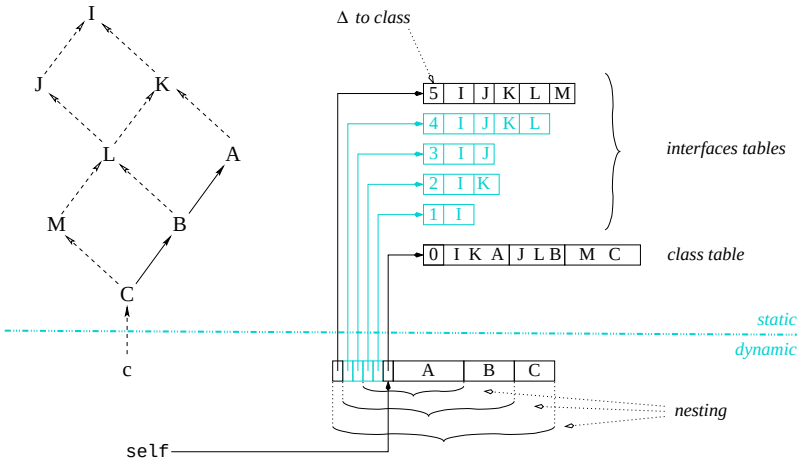


Fig. 10. Single inheritance and multiple subtyping: multiple inheritance variant, with 3 classes A, B, C and 5 interfaces, I, J, K, L, M. In grey, interface tables saved by sharing.

interface-typed entity 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 adds new interfaces at negative offsets and new attributes at positive offsets (Figure 10).

INVARIANT 6.1. *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 involves factorizing interface tables. Sharing tables requires a specific convention on the way method tables are built, with the same kind of *prefix condition* as aforementioned (see Section 2.5 and 3.3)—each interface or class orders its super-interfaces in some arbitrary order, e.g. depth-first, 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. Two interfaces may share the same table when the superinterface order is a prefix of the subinterface order. In Figure 10, class table is shared by K and interface table of M is shared by all other interfaces.

The code for attribute access 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 (page 3.1.2), with the difference that `#deltaOffset` does not depend on the method.

6.1.2 *Casting.* Several cases must be considered:

(1) from class to class, this 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,

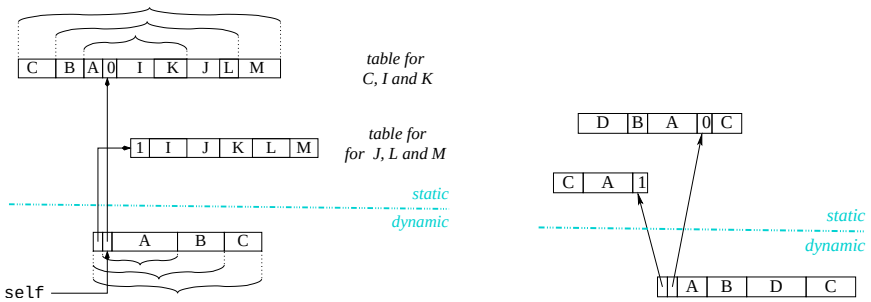


Fig. 11. THETA techniques, for examples of Figure 10 (left) and Figure 2 (right, where B is considered as primary).

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.

6.1.3 *Case of* THETA. The THETA language uses a very close technique with some optimizations [Myers 1995]. The basic idea is to extend object layout bidirectionality to method tables[24]. The positive part contains methods introduced in interfaces, when the negative part contains methods introduced in a class (Figure 11). In the example, sharing is not better than with the first variant (Figure 10), but it is intuitive that bidirectionality might improve sharing: without bidirectionality, methods introduced by a class prohibit any sharing for the subclass interfaces. Myers [1995] also proposes an optimizing algorithm for computing tables.

6.1.4 *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 superclass, one of them is considered as primary, and the others 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 5).

Variant with method copy. The first variant is based on the code copy of all methods and attributes defined in secondary superclasses, i.e. code sharing is done only with direct and indirect primary superclasses (see Section 5.4.1). The technique is sound only for language with a strict attribute encapsulation, à la SMALLTALK, but if this is not the case, accessors are required (see Section 4.4). Once attributes and methods have been copied, secondary superclasses can be treated as if they were interfaces (Figure 11).

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

[24]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 saves on VBPTRs (see Section 4.3). In [Zibin and Gil 2003b], bidirectionality is used for improving accessor simulation (see Section 4.4). Bidirectionality seems to originate in [Pugh and Weddell 1990] (see Section 5.3).

overhead of copying needs some assessment. But copying method code is not compatible with separate compilation. According to the author, experiments show a marked improvement vs. C++—in the example, the result is as good as with NVI (Figure 6).

Variant with accessor simulation. Instead of copying attributes, accessors are simulated, 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. On the whole, this is certainly a good alternative to standard implementation, as efficient as NVI but without its flaws.

Variant with static method. In Scala, each mixin is compiled as a Java class whose all methods are static and a class which implements a mixin defines, for each mixin method, a method which calls the static one.

6.2 Single subtyping variant

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

6.2.1 *Principle.* Reference Invariant 2.1 is then conserved in any case, whereas Invariant 2.2 is still 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 the corresponding methods and for downcast. In any case, instead of associating an address to each method introduced by an interface, it is more efficient to associate some data to each interface implemented by the class, together with grouping methods introduced by the same interface. Several techniques can be envisaged (Figure 12):

(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, shifts to each group are sufficient,
(4) shifts of (3) can be replaced by pointers to method groups.

Direct access. Constant-time direct access to these data structures for a given interface would imply a non conflicting numbering of interfaces: in a global linking framework, coloring would be a solution. Dynamic loading requires an injective numbering which may yield large but mostly empty tables, as interfaces may be quite numerous (up to 1000 in the largest benchmarks, see Appendix A). Of course, the tables may be restricted to their non empty part, at the cost of one more table access. This leads to the code in Appendix B.5, which has a $4L + B + 1$ cycle numbers.

Hashtables. If these tables are considered as too large, the solution will consist in an association structure such as a hashtable, like $\Delta^{\Uparrow}$, within the class method table (see Section 3.2.4 and Appendix B.2). *Perfect hashing* [Ducournau 2008] should be envisaged—it offers a constant-time access, with a cost of $4L + B + 3$ cycles, and

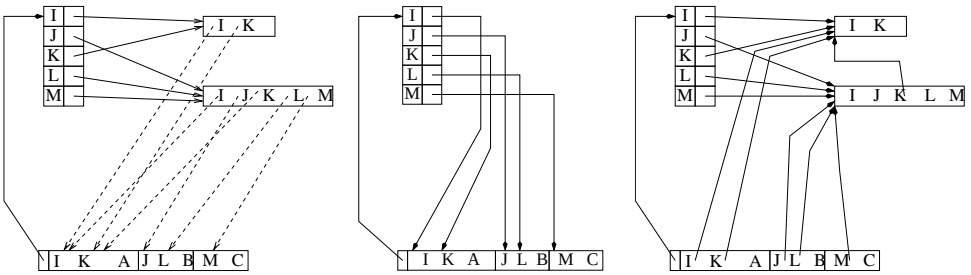


Fig. 12. Multiple subtyping and single inheritance: method tables in single subtyping variants, for example in Fig. 10. Dashed lines stand for offsets.

middle size tables, between direct access and pseudoconstant-time (see Appendix A.4).

In both cases, this varying length association structure can be inlined in the method table at negative offsets. Starting with this general framework, some optimizations are possible. Time optimizations consist of caching interface tables as long as the reference to the object does not change—so one access to the hashtable might serve for several method calls[25] (see Section 6.2.2). 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. The advantage of variants 1 and 2 is that they deal with all methods of the interface as a whole, whereas variants 3 and 4 consider only methods introduced by the interface. Caching will be more efficient with the former.

Downcast. As for casting, it is done mostly as in SST. Due to reference invariance, only downcast and dynamic type check must be considered. When the target type is a class, any SST technique applies, but an incremental one will be preferred. When the target is an interface, one will use the interface association structure in a boolean way, or any other MI boolean technique. However, there is a small difference here: a failure may occur with downcast, i.e. one must be able to check for failure, whereas method calls are safe. Hence, in the direct access variant, the interface color must be compared to the array bounds.

6.2.2 *Constant-time variant, with method table flow.* Implementation of interface-typed entities may provide a good opportunity for method table flow (see Section 4.5). Indeed, the space overhead of this technique would concern only interfaces, which are presumed to be less used than classes for typing program entities. Furthermore, here it would replace a nonconstant-time technique, so the gain might counter-balance the overhead.

[25]Caching method tables between several method calls is an improvement only when it is faster to get the table from the stack than from the object: hence this is useless with SST implementation, and it would be only valuable for interface-typed receivers, especially with hashtables.

6.3 Actual JAVA implementation

According to its specifications [Gosling et al. 1996; Grand 1997], JAVA could adopt either the MI variant or the 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]. The same is also true for C# and CLR.

6.3.1 *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] presents a state of the art of existing implementations, which mostly use dynamic typing techniques (see Section 5.2). 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 like inline cache, possibly polymorphic [Hölzle et al. 1991], are quoted. In Jalapeño, [Alpern et al. 2001] associates to each class a fixed-size hashtable containing a method address or a decision tree indexed on interface identifiers.

6.3.2 *Subtyping checks.* Dynamic class loading prohibits the simple approach of Schubert's numbering. However, the Cohen's [1991] coloring variant is incremental and it is used in Jalapeño [Alpern et al. 2001]. When the target type is an interface, this technique does not work and a boolean class-interface matrix associates to each class the interfaces implemented by the class. [Click and Rose 2002] proposes another variant of Cohen's technique, where the test is accelerated with a cache when the target type is an interface. Some virtual machine implementations seem to have adopted a hashtable technique for subtyping test and for `invokeinterface` [Meyer and Downing 1997, chapter 9]. *Perfect hashing* might be an optimal solution for both [Ducournau 2008].

6.3.3 *Other* JAVA *features.* A problem with 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 be 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 do not use synchronization[26]. Implementing them in the 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 do not need it—locks are then defined as an extra attribute in the former only. As offset invariance cannot be ensured, one must assume that accesses to that lock attribute are encapsulated within accessors, either effective or simulated. More-

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

over, as instances of non-synchronized classes may also be synchronized, a global data structure, called a *lock nursery*, is dedicated to it, and a default accessor is defined in `Object`. Statistics show that most synchronized objects are instances of synchronized classes, so the gain in locks for non-synchronized classes more than counterbalances the overhead of the lock nursery.

6.4 Application to multiple inheritance: *mixins* and *traits*

The literature on *mixins* is rather large [Stefik and Bobrow 1986; Bracha and Cook 1990; Ancona et al. 2000; Ernst 2002], and the term's usages vary from formal definitions to quite unformal ones. Mixins involve 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 to avoid MI or to discipline its usage. Our definition is as follows. A *mixin* is an abstract class, in the Smalltalk sense, i.e. a class without proper instances, but, contrary to Java interfaces, a *mixin* may define attributes and method bodies. Mixins aim at defining functionalities which are presumed to be orthogonal to classes and which can be mixed without them without conflict risk. We shall not discuss here this hopeless objective but only propose a specification of specialization in a world where classes and mixins can be mixed. (1) classes are in SI: a class cannot specialize more than one class. (2) a mixin can specialize at most one class, but no mixins, i.e. mixins must be thought of as unrelated functionalities, but some of them may apply only to instances of some precise class. (3) a class may specialize as many mixins as one wants. (4) when a class F specializes a mixin which specializes (directly or not) another class E, this must not lead to MI—E must be a direct or indirect superclass of F. This means that removing mixins from the transitive closure of specialization must yield SI. *Traits* [Ducasse et al. 2005] are essentially a variation on mixins with the limitation that a trait cannot define attributes but can be combined to other traits by well-defined operators. The Scala programming language implements mixins—they are called traits—on top of the Jvm, in a static typing framework [Odersky et al. 2008].

The mixin or trait 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].

6.4.1 *Multiple inheritance variant.* There is a striking resemblance between *mixins* and the application of MST techniques to MI proposed by Myers [1995] (see Section 6.1.4): 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*, i.e. not compile them (see Section 7.4). 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 2001]. 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: homogeneous implementation of genericity described in Section 7.4 does not allow to

make $M\langle A\rangle$ a subtype of A.

[Myers 1995] second variant also applies—accesses to `self` attributes are then optimized in the class methods and no double compilation is needed as classes are in SI. However, contrary to SI and MST which allows a more efficient implementation than plain MI, it does not seem that mixins provide such an improvement. More generally, the fact that a class C is abstract, i.e. without proper instances, does not bring any significant optimization, i.e. 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 2.2 for classes. On the whole, MI overhead is only within mixins. In some way, the mixin approach can be likened to NVI—both improve efficiency to the detriment of semantics or reusability.

6.4.2 *Single subtyping variant.* The Section 6.2.1 scheme (Figure 12) 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 does not specialize another mixin, one can group together attributes and methods of the mixin in the object layout and in method tables. A mixin method only needs to know the two shifts corresponding to attributes and methods, for access to `self`. As `self` is immutable, these 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, their accesses are the same as for a class.

Definitions of mixins are rather fuzzy. The examples given by [Bracha and Cook 1990] make them orthogonal to interfaces—they do not aim at introducing a new type but at disciplining the use of `call-next-method` (see Section 7.6). The implementation proposed here is based on our precise specification which forbids specialization between mixins, so a single shift is enough. If mixins could specialize mixins, one shift per specialized mixin would be necessary and this implementation would no longer be justified.

6.5 Evaluation

Two main variants must be considered: à la Java or à la Theta. Both variants reduce to SST implementation when no interface is used. 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 are time-constant, as with MI, with simplified shifts. In the Java variant, the shifts are avoided but either method calls are no longer time-constant, 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 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 might induce a very substantial dynamic space overhead. Moreover, the type annotations also can be automatically computed—in such cases, the interface-typed entities might be numerous and the constant-time variant may be better. A more precise comparison between these

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 5). As for mixins, they are a restriction to MI and reusability which may not be counterbalanced by efficiency increases.

7. COMPLEMENTS

Most languages complete the core of object-oriented programming that we have examined with a set of features which may need specific implementation.

7.1 Type variant overriding

It is well known that type safety requires that the parameter type in the overriding method must be a supertype of that in the overridden method, whereas the overriding return type must be a subtype of the overridden one. The return type is said to be covariant, and the parameter type contravariant. This is the so called *contravariance rule*. As for attributes, they must be invariant. Actually, strictly contravariant parameters are not interesting, since the models that one wants to implement are mostly covariant. See for example [Cook 1989b; Weber 1992; Castagna 1995] for a critique of covariance and [Meyer 1997; Shang 1996; Ducournau 2002b] for a defence. Combined with the difficulties of implementation that we shall see, this explains why languages like C++ and JAVA prohibit 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, i.e. either upcasts or downcasts.

7.1.1 *Single subtyping.* With SST, object references are invariant w.r.t. static type, thus safe overriding needs no specific implementation and unsafe overriding amounts to dynamic type checks.

Covariant parameters. This 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 6.1.3), 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 definition $m_C(t_1, ..., t_k)$ in a class C, overriding some method definitions $m_{C_i}(t_1^i, ..., t_k^i)$ in superclasses C_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, but write accesses must 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 then needs a *dynamic* downcast, since the attribute type depends on τ_d. Thus the type identifier must be stored in the method table. On the whole, one gets the following code (with Schubert's numbering):

```
load [object + #tableOffset], table1
load [val + #tableOffset], table2
load [table1 + #attributen1Offset], n1target
load [table2 + #n1Offset], n1source
load [table1 + #attributen2Offset], n2target                    3L + 3
comp n1target, n1source
blt #fail
comp n2target, 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 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. $4L + B + 3$). In contrast, the code sequence for dynamic downcasts is longer.

EIFFEL *anchored types and virtual types.* They provide 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 access to 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.

7.1.2 *Multiple inheritance.* Even type-safe overriding is now costly because of the shifts between subobjects. C++ allows only return type overriding[27]—this may not only be because of type safety and static overloading.

Covariant return type. In multiple inheritance, the 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 is now a question: what should the static type of the returned value be? Two alternative invariants can be envisaged.

[27]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. Actually, 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]. Lippman [1996] mentions it only in a NVI case. Nevertheless, it is implemented in some compilers (Intel or SUN) not in all (`g++` 2 for instance).

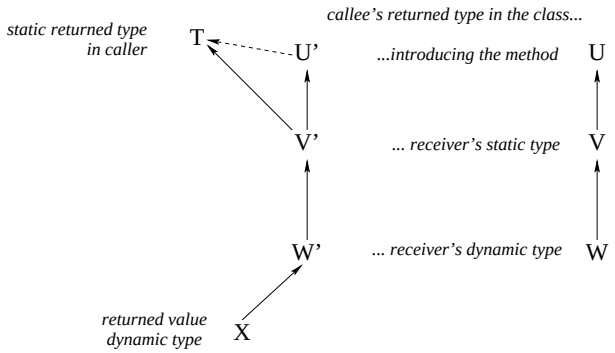


Fig. 13. Covariant return type: when $U' \neq W'$, upcast from W' to T is dynamic whereas casting from U' to V is static but no longer upwards.

INVARIANT 7.1. *The static type of the return value is the return type in the class which introduces the method (U' in Figure 13).*

INVARIANT 7.2. *The static type of the return value is the return type of the callee (W' in Figure 13).*

The former has the advantage of saving global invariance, i.e. 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 $U' \neq V'$.

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

```
add object, #delta, object                    // cast from W to V
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 advantage is that there is no longer a downcast, and that it has a cost only when needed, i.e. for pairs (V, W) such that $V' \neq W'$. The main drawback is that the method call in the thunk is no longer a 'tail' thunk, i.e. it is a `call`, not a `jump`. However, the second part of the thunk may be generated when compiling W, as V is a superclass and the cast explicit—only the first shift on the receiver depends on τ_d. Therefore, a solution to this double call may be to generate one function per superclass V, when compiling W, with usual thunks for jumping to these functions. This is not too expensive if the method is small or there are only a few V for which such a thunk must be generated. When this is not the case, functions with multiple entry and exit points may avoid the double call [Lippman 1996, p. 137]. Of course, NVI will simplify this, as both shifts are static.

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 $\Delta^{\uparrow}_{\tau_s}$ of the parameter value.

With a stricly contravariant overriding, casting is always upwards and safe, but

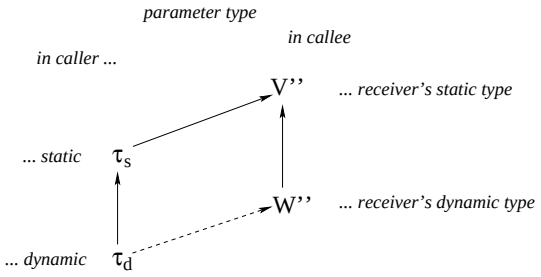


Fig. 14. Covariant parameters: a sidecast from τ_s to W'' is needed.

it is no longer 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 14 and 6 types are involved:

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

By construction, 3 subtyping relationships are posed:

$$\tau_d <: \tau_s \qquad \text{polymorphism} \tag{11}$$
$$\tau_s <: V'' \qquad \text{compile-time type checking} \tag{12}$$
$$W'' <: V'' \qquad \text{covariant overriding} \tag{13}$$

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:

$$\Delta_{\tau_s,W''} = \Delta_{\tau_s,V''} + \Delta_{V'',W''} = \Delta^{\uparrow}_{\tau_s}(V'') + \Delta^{V''}_{\downarrow}(W'') \tag{14}$$
$$\Delta_{\tau_s,W''} = \Delta_{\tau_s,\tau_d} + \Delta_{\tau_d,W''} = \Delta^{\tau_s}_{\Downarrow} + \Delta^{\Uparrow}(W'') \tag{15}$$

In both cases, a first static shift is done in the caller. A type check and a dynamic shift are then done, either in a thunk (14), or in the callee (15). The former approach has the advantage of imposing a casting only when needed. Note 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 7.1 or 7.2 to rule the attribute value. A read access then needs exactly the same casting in the caller as for a method call. A write access first needs a type check, which is done with a dynamic downcast—the target type must be stored in the method table, as in SST. With Invariant 7.1, an extra upcast is 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.

Thunk 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 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. Anyway, 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 14),

—the method call itself,

—a possible upcast on the return type (Figure 13).

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 of assigning different offsets to different signatures may be an improvement.

7.2 Polymorphic use of primitive types

The object layout described here is common to all objects, instances of usual classes, but excepts primitive values, instances of primitive types, e.g. numbers and boolean. However, primitive types are usually integrated in the class hierarchy, in an ad hoc way. With static typing, there is no need to encode the primitive type in the values themselves, as it must be done in dynamic typing, e.g. in LISP or SMALLTALK [Ingalls et al. 1997]. There is no problem regarding method invocation, when the receiver's static type is primitive—such methods are final and cannot be overridden, so a static call works. So the only issue with primitive types regards their polymorphic usage, when the langage has either abstract primitive types (e.g. `number`) or some *universal types* which are supertypes of both primitive types and usual classes. This is the case in EIFFEL with `Any`, and in C# and JAVA 1.5, with `Object`. Such a universal type does not exist in C++. Therefore, the points are: i) upcasts from primitive types to universal types, ii) converse downcasts, iii) equality tests.

7.2.1 *Boxing and unboxing.* As the method code of universal types requires a common layout, a primitive value must be *boxed* into a data structure pointing at, or containing, the value and equipped with a method table corresponding to the primitive type. Conversely, besides the dynamic type checking, downcasts amount to *unboxing* the value.

Boxing and unboxing are costly and would be burdensome for the programmer[28]: of course, garbage collection is required. Moreover, data flow analysis might spare some boxing/unboxing. However, the main issue is arrays and other collections—we examine this point in the more general framework of genericity (see Section 7.4).

[28]In JAVA, up to version 1.5, boxing and unboxing must be done by hand, by using *boxed types*, e.g. `integer` in place of `int`.

Equality test. Testing the equality of two universally typed entities is also important, since programmers are presumed to be interested by the equality of boxed values, not by the equality of boxes. Therefore, $x = y$ either if x and y are the same addresses, or if x and y have the same method tables (i.e. same type), the same values in their first field, and if the common method table contains some certificate ensuring that this is a primitive type.

7.2.2 *Method table flow.* An alternative to *boxing* would be method table flow (see Section 4.5). The main drawback would be that the overhead would apply also to non-primitive values. However, this drawback does not hold when one considers abstract super-types of primitive types, e.g `number`. The equality test is simpler as it amounts to parallel testing of both values and tables.

7.3 Multiple dispatch

With multiple dispatch, method selection depends on the dynamic type of all parameters. It may be viewed as a weak form of the *pattern-matching* of many functional languages, e.g. HASKELL [Peyton Jones 2003]. CLOS [Steele 1990] introduced it in the form of *generic functions* which are usual LISP functions dispatched between all the *methods* of the considered function. The point is that generic functions are no longer included in classes, contrary to methods in usual OOP—they are orthogonal to classes. In this dynamically typed form, multiple dispatch may be understood as single dispatch in *product types*, i.e. in the cartesian product of type sets. In this framework, dynamic lookup strategies [Chen et al. 1994] and cache-based techniques [Kiczales and Rodriguez 1990] have been examined. Authors' arguments for these nonconstant-time approaches are that all experiments show that multiple dispatch is rarely used, i.e. 95% of generic functions require a dispatch on a single parameter, and 95% of the remainder dispatch only on two parameters. [Dujardin et al. 1998] propose an implementation, based on a generalization of the coloring principle, with several indirections, that is time-linear in the number of parameters. [Holst et al. 1998; Zibin and Gil 2002] propose alternatives. However, all these approaches are in dynamic typing.

Generic functions are not adapted to the modular view of classes for which methods are encapsulated in classes. Some proposals have been made which are both modular and statically typed [Mugridge et al. 1991; Boyland and Castagna 1996; Castagna 1997]. We do not take their possible differences into account. Usual methods defined in a class are replaced by *multi-methods*, which involve several *branches* which differ from each other by their parameter types[29]. Hence, dispatch is two step: i) selection of a multi-method, based on the receiver's dynamic type; ii) selection of the branch in the selected multi-method, based on the dynamic type of all remaining parameters. The point is that typing rules statically ensure that,

[29] As usual in OOP, terminology is not fixed: [Castagna 1997] uses the term *overloaded functions*—which must not be confused with *static overloading*!—to name what we call here *multi-methods*, whereas [Mugridge et al. 1991] uses *multi-method* in place of *generic function*, with the same meaning as in CLOS ancestors (COMMON LOOPS, NEW FLAVORS). In our terminology (see page 5), *method* stands for a *generic property*—SMALLTALK *method selector* and CLOS *generic function* are analogues—for which several definitions exist in different classes. The role of dispatch is to select one definition among them. A *multi-method* is a method definition composed of several branches.

for all actual dynamic types, a single more specific branch exists [Castagna 1997]. Implementation of step i) is as usual. The selected code—i.e. the multi-method—must dispatch between different branches according to the dynamic types, in a way very similar to a `typecase` construct. In a separate compilation framework, no data structure can be associated with parameter types, so some kind of decision tree seems the only solution. If this tree is generated at compile time, it must be based on subtype tests, whereas global link-time generation may use equality tests.

An optimization of this implementation of multi-methods may be based on static overloading, in a way similar to type variant overriding (see Section 7.1). Each branch signature has its own entry in the method tables—static selection dispatches between the different signatures, whereas each multi-method contains the dynamic selection required for the given signature. On the whole, multi-methods can be easily implemented in such a way that an overhead occurs only when multiple dispatch is actually needed at run time.

7.4 Genericity

Instanciations of parameterized classes are usual classes: therefore, they are implemented as such, with one or several method tables, a type identifier and the object layout is made of an attribute table plus some pointers to method tables. However, compiling parameterized classes raises the first specific issue: how are accesses to entities typed by the formal parameters? There is a simple way to elude it, i.e. no compilation at all, as C++ does with *templates*. In this case, each instanciation of a parameterized class amounts to generating a new non-parameterized class, where actual types have been substituted to formal types. All arguments in favour of separate compilation recommend compiling parameterized classes, e.g. see the critique of C++ templates by Lippman [1996]. 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 *heterogeneous* implementation of C++ templates.

However, on the contrary, heterogeneous implementations have some advantages—they produce much more efficient code when they are instantiated by primitive types. Moreover, they allow the programmer to consider the formal type as an usual type in a straightforward way. This is for instance easy to implement mixins as *parametrized heir classes* [VanHilst and Notkin 1996; Smaragdakis and Batory 2002].

7.4.1 *Bounded genericity and type erasure.* A homogeneous approach requires some typing information regarding the formal parameters. We shall mostly consider here *bounded* (aka *constrained*) 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 a good way, maybe the best, to allow static type checking.

When bounded genericity is specified in a type-safe way, it is possible to implement it with *type erasure*, i.e. the formal type T is replaced by the bound B. This is the case in Java 1.5 for instance. Above all, type erasure is a matter of static type checking—when substituting B to T, the code is proven correct for all possible instanciations. Type unsafe cases, e.g. Java array subtyping, should be covered by dynamic type checking, i.e. downcasts. Once the bound B has been substituted for

the formal parameter T, access to methods or attributes on entities typed by T, now B, is straightforward, regardless of underlying implementation. The only issue is the casts required between the actual parameter, say $C <: B$, and the bound B. When calling a method of A which takes a parameter of type T, an upcast from C to B is required. Conversely, when a method of A returns a type T, a downcast from B to C is also required—in constrast with usual downcasts, this one is safe.

Reference invariant implementations. Invariant 2.1 makes type erasure quite easy to implement. References do not depend on the static type B or C, so upcasts from C to B for parameter passing, and downcasts from B to C for returned values, all remain implicit. The point is that static type checking ensures that these downcasts are safe.

Subobject-based implementations. When Invariant 2.1 does not hold, type erasure is much more costly—indeed, casts are no longer implicit as they require a shift. At method invocation, upcast on the parameter type, from C to B must be statically done by the caller—it is better than doing it in the callee, at the extra cost of table access since the source type C would not be static. Downcasts on the returned value are more costly, i.e. even though this downcast is safe, the pointer adjustment must be done. Actually, subobject-based implementation is all the contrary of type erasure, since the whole implementation depends on sattic types and all values are tagged. So, this is not surprising that C++ implementation is heterogeneous.

7.4.2 *Specialization and subtyping of parameterized classes.* Two kinds of specialization must be considered: specialization either of the parameterized class itself, or of the parameter. In the former case, when $A'\langle T <: B'\rangle \prec A\langle T <: B\rangle$[30], then, following the definition, $A'\langle C\rangle \prec A\langle C\rangle$ (Figure 15-a/b/c). All that applies to class specialization applies to this kind of specialization.

Regarding the latter case, when $D <: C <: B$, a consequence of the contravariance rule is that the subtyping $A\langle D\rangle <: A\langle C\rangle$ is generally unsafe—it is type-safe only when the formal type is never used in contravariant position [Cook 1989b; Weber 1992]. Despite this, some languages like Eiffel allow such unsafe subtyping—the unsafe cases must be handled in the same way as for unsafe covariant overriding (see Section 7.1). However the implementation may impose some restriction. For instance, subobject based implementation makes this subtyping difficult, both in heterogeneous and homogeneous approaches, i.e. entities typed by C and D would not point at the same subobject, so substitution is not possible. In fact, the problem is exactly the same as for variant overriding of attribute and return types (Invariants 7.1 or 7.2). For a type-safe language, this is a minor limitation as this specialization is rarely safe [Day et al. 1995].

By combining the two kinds of specialization (Figure 15-c), or when considering more than one parameter (Figure 15-d), genericity may implicitly give rise to MST, even in a strict SST context. However, this situation would not affect Invariant 2.2, if it holds, since the latter case of specialization does not allow definition of new properties, so no offset conflict may follow. Moreover, downcast from $A\langle C\rangle$ to $A'\langle D\rangle$ might be considered—this would impose two type checking, first on the

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

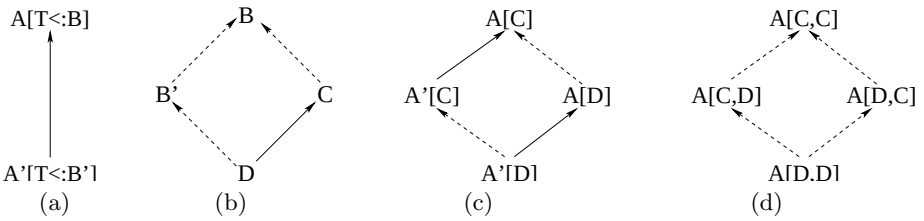


Fig. 15. Specializing parameterized classes

parameterized class then on the parameter.

Another point involves casting either from or to the formal parameter T. Upcasts from T to B or a supertype of B are sensible and safe. Other casts are less likely as they are all unsafe and as they imply that, in the definition of A, there would be some type, say Z, presumably subtype of B, such that either $T <: Z <: B$ or $Z <: T <: B$ should be distinguished cases. As this seems odd, these casts will not be examined.

Finally, given a parameterized class $A\langle T <: B\rangle$, a *joker* (aka *wildcard*) type parameter may be used—noted $A\langle ?\rangle$, it is a supertype of any instanciation $A\langle C\rangle$, such that $C <: B$. Moreover, the *joker* may have a lower and upper bound: $A\langle B_2 <: ? <: B_1\rangle$, with $B_2 <: B_1 <: B$. Of course, not everything may be done with such a joker type—one may just get some returned value of type B_1 (the default upper bound is B), but one can only pass a value of type B_2 as a parameter typed by T, provided that the lower bound is explicit (no default) [Garcia et al. 2003; Bracha 2004; Torgersen et al. 2004].

7.4.3 *Instanciation of the formal type.* The present title has the double meaning of i) binding T to some type $C <: B$, when instanciating $A\langle T <: B\rangle$ into $A\langle C\rangle$, ii) creating a new instance of T (i.e. of C according to the first meaning), through some '`new` T' statement. Both meanings are related through the following point: how far does an instance of $A\langle C\rangle$ remember (i.e. encode) the fact that it results from an instanciation of T by C?

Creating instances of the formal type. Heterogeneous implementation of parameterized classes makes it easy to create instances of the formal type: '`new` T' statements will be translated into '`new` C': the only point is that, for all classes C instanciating T, i) C must not be abstract, ii) C must have a constructor (or initializing method) with the same signature—in Eiffel, one must add the name—as it is used in the `new` statement. Homogeneous implementation makes it harder. Strict type erasure would translate '`new` T' into '`new` B', which is undesirable. This is the reason why Java 1.5 does not allow such an instanciation. A less strict erasure would permit to get the type C instanciating T through the instance of $A\langle C\rangle$, which must memorize C somewhere in its structure—hence, the formal type T. should become an attribute in the final instance layout.

Type identifiers, downcast and recursive parameterized types. Given the need for instanciating the formal type and the aforementioned two kinds of specialization, type identifiers are an issue for homogeneous implementation. Indeed, when either $A\langle D\rangle <: A\langle C\rangle$ or $A\langle D\rangle <: A\langle ?\rangle$ is considered, or more simply when $A\langle T <: B\rangle \prec$

A'', a downcast to the target type $A\langle D\rangle$ should be able to find the D identifier somewhere in the structure of the considered instance. Therefore, the type identifier of an instanciation $A\langle C\rangle$, should be constituted by the identifiers of A and C. A simple solution is to add an entry in the method table of A for the type identifiers of each formal type introduced in A and all superclasses. As a consequence, regarding homogeneous implementations, one must distinguish between the code and the method table—both may be shared by all instanciations, at the price of forgetting the parameter type at run-time. This is a strict type erasure. A less strict erasure keeps different method tables for different instanciations—they are copies from each other, apart from the type information.

When the type instanciating T is itself a parameterized type, the type identifiers of the parameters' parameters must also be present. However, as the bound may be an unparameterized supertype of some parameterized type, the number of type identifiers is not statically known. Thus, the correct scheme would be to reference the method tables of the formal types, not their identifiers.

Moreover, recursive parameterized types could thus be implemented, for instance when $A\langle T <: B\rangle <: B$: the code of such a class may contain recursive instanciation such that $A\langle A\langle T\rangle\rangle$. Implementing the complete data structures needed for such classes cannot be statically done at compile- or link-time. Therefore, if one wants to avoid copying method tables at run-time, at each instanciation, the object layout of such classes should be made of one pointer to the statically computed method table, plus one pointer to the type identifier tree, dynamically computed from the `self` type identifier tree. Of course, the compiler must detect such recursive cases, which may be indirect, e.g. $A\langle T <: B\rangle <: B'$ and $A'\langle T <: B'\rangle <: B$. Obviously, in such recursive cases, heterogeneous implementation leads to infinite loops.

Finally, either T constructors are added as $A\langle T\rangle$ methods in its method table, or they are found in the parameter's own method table.

7.4.4 *Alternative to type erasure.* Type erasure gives an optimal implementation when SST invariants hold, i.e. either in SST or in MI with coloring, unless the parameter is instantiated by a primitive type (see below). However, in other implementations of MI or MST, type erasure implies either downcasts, or method invocation on interface-typed receivers. Therefore, it could be worthwhile to examine alternatives.

Multiple subtyping. When the bound B is an interface, offset invariance is no longer ensured. In the SST variant, access to an entity typed by an interface is not as efficient as access to an entity typed by a class.

A solution might be to distinguish instanciations, depending on whether the formal type is instanciated by a class or an interface. In the former case, there is no way to be more efficient than type erasure. In the latter case, an alternative consists of 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 16). A shift to method groups of each supertype may replace conversion. However, pointers at method tables are not possible, because of polymorphism: if the $A\langle C\rangle$ table contains a C method address, an entity typed by C could not be valued by a subclass $D \prec C$. The method call is then:

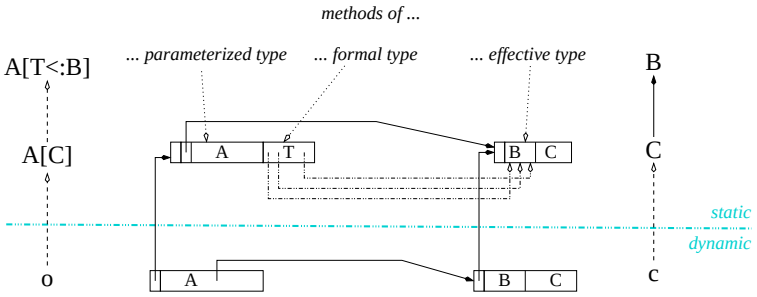


Fig. 16. Parameterized classes: method table with multiple subtyping

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

This technique must also be applied for instanciating the formal type. Of course, this is less efficient than method invocation in SST implementation, but it may be better than a call to an interface-typed receiver in the SST variant of MST.

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 actually passed as parameters and assigned to the conversion table. More generally, the presented technique amounts to considering 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. So, in both cases, an alternative to type erasure would be a double compilation, according to whether the actual type is a class or an interface, which means 2^k compilations when there are k interface-bound parameters.

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 others [Agesen et al. 1997; Solorzano and Alagić 1998] before genericity was finally integrated in Java 1.5. None of these extensions takes the approach presented here because it is likely not compatible with Jvm—compatibility would require method calls to formal types with an explicit call to `self`, whereas we only simulate this call with an offset. Java 1.5 implementation is based on type erasure. As a strict type erasure is done, instances of a parameterized class—i.e. instances of a class instance of the parameterized class—do not record the type which has instanciated the formal type, i.e. information is lost.

Multiple inheritance. Downcasts on returned value are the main issue of type erasure in the standard implementation of multiple inheritance. With type erasure and subobject implementation, all entities typed by T reference B subobjects. An alternative would be that they reference T subobjects. Method calls may be done as in MST. On the other hand, attribute offsets are relative to the type which

introduces them. Access to an attribute p 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 20). However, 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 $\Delta_T^{\uparrow}$ 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 also need to add an entry containing the value of $i_T(B)$ in the method table of A—one may generalize by adding an entry for each supertype of B.

In both cases, a direct upcast to a supertype of B may be replaced by a two step upcast, via B, at the extra cost of one more table access.

7.4.5 *Instanciation with primitive types.* A homogeneous approach may be problematic when the bound is a universal type, which may be instanciated by both usual classes and primitive types. In this case, a homogeneous approach would indeed impose the use of *boxing* (see Section 7.2). A heterogeneous approach would be better, above all in case of arrays and collections which need a particular efficiency. Hence, compiling a bounded parameterized type $A\langle T <: B\rangle$ where the bound B is a universal type, would amount to producing an homogeneous code shared by all non-primitive classes—i.e. the instanciation $A\langle B\rangle$—plus a specific code for each primitive type. Of course, the unsafe subtyping $A\langle C\rangle <: A\langle B\rangle$ would be problematic.

An alternative consists of only one implementation for all primitive types (provided their values have the same size), where the method table for the formal type is a field in the parameterized class layout—for the same reason as with type identifier trees, two pointers in the object layout will avoid copying method tables at runtime. This is a usual way of implementing genericity when there is no subtyping, as in Clu [Atkinson et al. 1978; Myers and Liskov 1994].

A midterm between both alternatives involves an homogeneous code shared by all primitive instanciation, coupled with a method table per instantiation, which would include a copy (or a pointer at) the actual type method table.

A last alternative replaces *boxing* by *tagging*. As addresses and offsets are all multiple of 4, the two low-weight bits can be reserved for encoding three primitive types, like in dynamic typing. So, the generated code uses type prediction for these three types. Of course, reserving two bits for tagging reduces the range of the concerned types. As the `integer` type is the most commonly used, 30-bit integers would be a severe restriction. However, it could be coupled with bignums and rationals to offer an efficient unlimited precision arithmetic.

7.4.6 *Conclusion on genericity.* Homogeneous implementations of genericity are possible without overhead vs. heterogeneous ones, provided that the invariants of SST hold in the underlying implementation. As soon as these invariants do not hold,

there is a substantial overhead—interface-typed receivers with SST, or downcats with MI. Although, one may criticize the lack of bounded genericity in C++ templates, a strict separate compilation with an homogeneous implementation would actually be less efficient than current heterogeneous implementation.

Of course, all approaches can be combined:

—*tagging* for integers, coupled with bignums and rationals;

—homogeneous code with heterogeneous method tables, including that of the actual types, for all other primitive types;

—alternatively, each primitive type has its own heterogeneous implementation, but it could be limited by the parameter combinatorics;

—homogeneous code with heterogeneous method tables for all non-primitive types.

Match-bound polymorphism, i.e. replacing subtyping by *matching* for the bound [Bruce et al. 1997], is a generalization of bounded genericity whose effect on implementation should be examined.

7.5 Shared class attributes

The term *class attribute* 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 reflective model, à la SMALLTALK or CLOS—this notion has no direct equivalent in the languages we consider here, except in the JAVA reflective library—anyway, it does not require any specific implementation;

—the fact that an ordinary attribute might 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 literal meaning of the keyword `:allocation :class` in CLOS; to avoid ambiguity we shall call them *shared attributes*.

The last two notions differ by their semantics—we may 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`. From an implementation standpoint, 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 like `3.5-side-quadrilateral`...

Shared attributes will be implemented in the method table, in the same way as the methods themselves, which may be considered as *read-only shared attributes*. As several classes, with different tables, may share the same attribute, an extra indirection by a *wrapper* (aka a box or a handler) is needed when the attribute is mutable. The wrapper is useless when the attribute is read-only. Multiple inheritance does not complicate the case.

Curiously enough, this sound, useful and efficient mechanism is not proposed by commonly used languages: C++ and JAVA offer only `static` variables. In EIF-

FEL, constant features amount to read-only[31] class attributes: `once` features are a variant where the constant is initialized by the first invocation, subsequent calls return the same value. When `once` features are also `frozen`, they are 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 of implementations described here, regardless of whether they are in single or multiple inheritance. Plain accessors (see Section 4.4) is a way to implement the full functionality of `:allocation` in a general way, with a dynamic space saving, but to the detriment of time efficiency. An alternative would be to implement shared attributes in the instances, with a wrapper to ensure value sharing, thus to the detriment of time. On the whole, the complete specification of the keyword should be reserved to attributes explicitly declared at introduction.

7.6 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 performed by the keyword `super`—a pseudo-variable as `self`—which consists of 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[32]. In SI, the superclass method is unambiguously determined as well as the method offset, i.e. everything is invariant w.r.t. dynamic types, and the super call is static.

In multiple inheritance. `Super` does not work since *the* superclass method is not uniquely determined, in the general case. However, there are 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 the method table may be needed for the shift, not for the address. The mechanism is quite versatile, but it has a major drawback, likened 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, i.e. it consists of calling the next method in the linearization order of the superclasses of the receiver's dynamic type [Ducournau et al. 1994; Ducournau and Privat 2008]. The main advantage is that

[31] Binding between the name and the value is immutable, not the value itself.
[32] In JAVA, the static type of `super` is the superclass of the current class.

the repeated inheritance problem is avoided. 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 upsets modularity to some extent [Snyder 1991b] 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 types of the same dynamic type (Table II), `call-next-method` entries will contain different addresses. These 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 induce cost only when and where it is used. This implementation works because standard MI implementation allows a reference to the next method which is parameterized by both static and dynamic types. With reference invariant, this simple implementation must be slightly modified and the extra offset represent an implied new method, say `cnm-foo-A`, whereby the method `foo` defined in class `A` uses `call-next-method`. This is the implementation of this feature in SCALA.

7.7 `Null` pointers

Any variable or attribute typed by a class must be explicitly initialized before any sound access. An intraprocedural data flow analysis may detect uninitialized local variables, but this is more difficult for attributes, which may be initialized some time after instance creation. Thus, an initialization with a `null` value is the only way to avoid random memory states and all accesses to a possibly `null` entity must check this entity. The simplest and most common solution involves a single distinguished value `null`: an equality test is inserted at each doubtful access and an exception is raised in case of failure, i.e. when the value is `null`. These checks may be restricted to accesses which are not proven to be non-`null`. Moreover, the test may be made in parallel. Anyway this adds a uniform overhead to all accesses.

Therefore, it is worthwhile to examine alternatives which avoid all implicit equality tests, while permitting explicit ones. They are of two kinds. The first one depends on the 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 raise an interrupt.

A software based solution consists of a distinguished instance `null` per class—all its attributes are initialized to `null` and its method table points at methods which signal an exception. This instance 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` instances may be shared. In standard MI, a unique `null` object is even possible: it will consists of the unique instance of the class $\bot$, subclass of all classes—e.g. the `AllRef` type in the SCALA language [Odersky et al. 2008]—and will be constituted by one subobject per class. Of course, these shared `null` objects must be computed at link-time and they are not easily compatible with dynamic loading (see Section 5). In SST, a unique `null` object might be 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.

7.8 Garbage collection

Automatic memory handling is an argument for reliability: this is well understood by designers of LISP, SMALLTALK, EIFFEL and JAVA. There are many *garbage collection* (GC) techniques. All need some information concerning individual objects, i.e. length of memory area, boolean status of the object and possibly a pointer on a new generation. This information may be contained in the object layout itself, in method tables or implemented in separate tables. The requirements for most techniques may be summarize by the *mark and sweep* technique, which involves two stages—*marking* the living objects reachable from memory roots, then *sweeping* the heap for collecting free memory areas. *Copying* is a complement or an alternative to sweeping, which allows to compact the heap. Many other techniques exist, but they have more or less the same requirements for object implementation.

[Wilson 1992; Jones and Lins 1996] are surveys of garbage collection techniques.

7.8.1 *Conservative vs. exact GC.* The main task of GC is to determine living objects, i.e. objects reachable from memory roots (static variables, registers and stack). With static typing, there is no way to dynamically distinguish an immediate value from an address. Therefore, given a living object, GC must determine in the object layout the pointers at other living objects. This may be done in an exact way, either by defining a specific method for this purpose [Colnet et al. 1998], or by describing the object layout by a static bitmap where each bit is 0 or 1 according to whether the corresponding field is a pointer or not. Both are automatically generated by the compiler.

Extracting living objects from the stack is also possible in an exact way—the stack must be typed by bitmaps which describe, in the same way as for classes, which words are pointers. This is the case in JAVA and JVM—each method allocates one block in the stack, and this stack block is described by a static map which is pushed in the stack when the block is allocated. This approach is possible for all type-safe languages, but it forces the compiler to be fully integrated with GC, and both points do not hold in C++. When both extractions are exact, GC is said to be *exact* (or *non-conservative*).

When these techniques are not possible, GC must conservatively assume that each field in the object layout or in the stack is a potential pointer at a living object. The GC is then called *conservative*, or *semi-conservative* when extraction from the object layout is exact [Boehm 1993]. This is slower, and it may require additional data structures in order to ensure that the potential pointer does point at an object. In all generality, checking that a memory word is a pointer at an object may involve first checking that this is a valid address, secondly that the first field is a pointer at a method table, which is easier since method tables are static. However, this is a necessary condition but it is not sufficient. A common technique uses 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 only one bit to be reserved per object, but it increases memory fragmentation.

7.8.2 *Mark.* A few marking bits are needed for different status information during the GC process—they are partly used for marking living objects and belong to each instance. They may be implemented in the aforementionned global mapping of the heap. When they are implemented in the object layout, these marking bits may occupy a whole field, but this is a considerable overhead. *Bit-stealing* is a general way to avoid memory overhead, by implementing them as low or high weight 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 may use either the pointer at the method table, or the first attribute, for instance when it is a pointer. In both cases, there is uniform overhead, with two more instructions and cycles for removing these extra bits, at each access to the field. [Bacon et al. 2002] present some experimental statistics on this approach in JAVA, where the pointer at the method table (i.e. `tableOffset`) is used. However, the overhead might be null if the value of these bits is unchanged between two accesses to the field—only 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 not at their default value. Hence, the overhead depends on whether the GC is a concurrent background task, or a locking task. Using the first attribute field requires reserving it for an attribute which does not 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.

7.8.3 *Sweep.* This stage involves sweeping the heap, from memory block to memory block, by decreasing or increasing addresses, with each block being either a free block, a dead object or a living object.

A first point is that all information required by GC must be reachable from the beginning of the object area—if this information is embodied in the method table, the method table must be at offset 0 (`tableOffset` in pseudo-code examples) or, more generally, at a constant position from the object beginning. Therefore when sweeping to the next block, GC must determine: i) whether the block is free or not, ii) if it is not free, whether the object is alive or not, iii) the address of the following block. Question i) may be answered by encoding free blocks in such a way that it differs from objects—e.g. odd numbers at `tableOffset` position. Question ii) depends on marking bit, and question iii) needs to encode the length, on one hand in free blocks, on the other hand, either in object layout or in method tables. If one assumes that all proper instances of a class have the same size, it may be stored in method tables. However, a special treatment is required for some variable size values, such as strings and arrays—in this case, a distinguished value (e.g. 0) in the method table indicates to GC that the actual size is a field in the object layout.

7.8.4 *GC according to various implementations.* Standard SST implementation is well adapted to GC, as all references to objects point at the beginning of object layout. However, Invariant 2.1 may hold without `tableOffset` being at a constant offset w.r.t. object beginning. A typical counter-example is *bidirectional coloring*—and all bidirectional layouts—where attribute offsets may be negative as well as

positive[33]. In that case, a solution may be to duplicate the pointer at the method table at the beginning of all objects for which negative offsets are used [Desnos 2004]. The overhead occurs only for some classes in multiple inheritance and it is lower overall than the gain yielded by bidirectionality. Note that [Gagnon and Hendren 2001] proposes a bidirectional layout especially designed for optimizing garbage collection, by assigning negative offsets to references and positive offsets to immediate values. In a copying garbage collection, this incurs no overhead as all fields must be scanned before being copied and an extra pointer is not required. This is however not applicable to bidirectional coloring, since the key point is that negative offsets are dedicated to references.

When Invariant 2.1 does not hold, the marking stage is complicated since object references may point inside memory areas, at a variable distance from the beginning of the object layout. Downcasts to the dynamic type (using $\Delta_{\Downarrow}^{\tau_s}$) are required, as this is the only way to reach the whole object, and subobjects may be ordered in such a way that the first one corresponds to τ_d. In any case, an overhead must be expected. When copying an object into a new generation, subobjects just add the overhead of computing the new subobject address.

VBPTRs (see Section 4.3) avoid the complication at the marking stage since the object layout is made of explicitly linked subobjects—however, in this case, GC complexity is a function of subobject number not of the object number. Moreover, it may be safer to copy an object as a whole, rather than subobject by subobject.

8. CONCLUSION

One may draw different conclusions from this survey, according to whether one stresses langage expressivity, namely multiple vs. single inheritance, or runtime system flexibility, namely dynamic loading vs. global compilation or linking.

On the one hand, separate compilation of SST is simple and as efficient as possible—indirect method calls are a true overhead which could only be reduced with global techniques or by increasing the processors' capabilities for indirect branching prediction [Driesen 2001]. But SST expressivity is far from what programmers expect—actually, as far as we know, there is no commonly used SST language. 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. Another drawback, less known but explicit both in its cubic worst-case and through benchmark measurement (see Appendix A), is its bad scalability. Therefore, it is not surprising that recent efforts have focused on SI+MST languages, as JAVA or C#—this is a sound middle point between the two extremes, especially if compared to other tradeoffs such as C++ NVI or mixins.

Regarding runtime systems, if dynamic loading is a key requirement, one must choose between standard MI implementation, with its large overhead, and SI+MST languages, which may be more efficient than plain MI, when using a JIT compiler not when bytecode is interpreted by a virtual machine. If the key requirement is separate compilation, and if dynamic loading is not required, global linking with link-time generation and a mixing of coloring and tree-based dispatch, is surely the

[33]Of course, if all offsets are negative, as in AIX (see Section 7.7), sweeping must go backwards.

most efficient. The coloring approach combined with type analysis is an appealing idea on paper: the run-time produced using these techniques would be even more efficient than with SST implementation, due to dead code detection. However the performance of the link-time global step must be established in practice, on actual programs—at the moment, it remains at a prototype stage in the PRM compiler. Finally, global compilation is necessarily the most efficient framework: it is however unclear whether the gain vs. global linking is significant or not. This will need further experiments.

Nevertheless, global techniques have significant drawbacks. Being global, they are incompatible with library sharing between different applications, which is a common optimization of operating systems: hence, the best MI implementation allowing sharing seems to be standard MI implementation with empty subobject optimization. Moreover, besides being global, most of those techniques are not incremental and could not meet the specifications of abstract machines as JVM or CLR. An open issue is an efficient incremental extension of coloring: by efficient, we mean better than standard MI. A dual open issue would be to implement an efficient virtual machine based on plain MI specifications.

JAVA-like languages represent another issue. An efficient implementation of interfaces is as difficult as for plain multiple inheritance and programming usage can imply an intensive use of interfaces—hence, the efficiency must be as high for interfaces as for classes and its scalability must be assessed. A common technique could be used for both method invocation and subtype testing—actually, we did not find a single paper which considers both mechanisms as a whole. A constant-time technique should be preferred—actually, to our knowledge, perfect hashing is the only reference-invariant technique that is also time-constant. These conclusions are irrespective of all of the optimizations that might be provided by adaptive compilers—an efficient basic implementation is required for the cases where no specific optimization applies.

Besides those large scale conclusions, this paper stressed several points where an improvement is possible in the present state of affairs. There is some evidence that standard MI implementation might be improved with empty subobject optimization, when it is not incompatible with VBPTRs, for instance, in GNU and SUN C++ compilers. Link-time optimizations would be an improvement for all languages, but it would require a new compiler-linker architecture, except in the JIT compilers which are already a special case of this new architecture (not so new hence, since JIT compilers are as old as LISP and SMALLTALK systems)—if load-time generation of dispatch code is not already part of all JIT compilers, this would certainly be a decisive improvement. Some experiments seem to prove that standard MI implementations, i.e. C++ compilers, might also gain from more efficient downcast implementations. *Perfect hashing* should be a good compromise between direct access and pseudoconstant-time tables. The implementation of *parametrized classes* also require some improvement, with a tradeoff between homogeneous and heterogeneous implementations.

Regarding language specifications, commonly used statically typed object-oriented languages—i.e. C++, EIFFEL, JAVA and C#—are quite perfectible. Valuable suggestions include shared class attributes instead of `static` variables, method combi-

nation, or `self` encapsulation, besides `private` and `protected` keywords. Implementing these features would not increase the overhead of object-orientation. This is not the case of type variant overriding which add some overhead, especially with subobject-based MI implementations. When some language feature is considered 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. Multiple dispatch, in the theoretical typing framework of Castagna [1997] would be also a valuable extension—with a `typecase`-like implementation, the extra cost just applies to multi-methods, i.e. methods with several branches. Efficiency would depend only on subtyping test which can be quite good, even in the framework of subobject-based MI implementation, in separate compilation and dynamic linking.

Finally, in object orientation, modularity has been historically confused with the notion of class, but the class is a conceptual unit, not always a program unit of the size appropriate for compilation. A higher level notion such as *modules* has long been advocated [Szyperski 1992; Bracha and Lindstrom 1992]. Java *packages* are not a good answer as they are mainly designed as name spaces, but current discussions on the notion of *sealing* may be, however, a forward step towards true modules [Biberstein et al. 2001]. In any case, compiling such modules may raise specific issues, although module design and compilation scheme may be well adapted to each other: see for instance [Ducournau et al. 2007].

Hence, the perspective is manifold—improvement of current implementations, application of global techniques at link-time, modular compilation, inherently incremental techniques and MI virtual machines.

ACKNOWLEDGMENTS

This work has been partially supported by grants from Région Languedoc-Roussillon (034750). The author thanks Yoav Zibin for reading of a preliminary version and for valuable comments.

REFERENCES

Agesen, O. 1996. Concrete type inference: Delivering object-oriented applications. Ph.D. thesis, Stanford University.

Agesen, O., Bak, L., Chambers, C., Chang, B., Hølzle, U., Maloney, J., Smith, R., Ungar, D., and Wolczko, M. 1995. *The SELF 4.0 Programmer's Reference Manual*. Sun Microsystems, Inc., and Stanford University.

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

Agesen, O. and Hölzle, U. 1995. Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages. In *Proc. OOPSLA'95*. ACM Press, 91–107.

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], 108–124.

Alpern, B., Cocchi, A., and Grove, D. 2001. Dynamic type checking in Jalapeño. In *Proc. USENIX JVM'01*.

America, P., Ed. 1991. *Proceedings of the 5th European Conference on Object-Oriented Programming, ECOOP'91*. LNCS 512. Springer.

Ancona, D., Zucca, E., and Drossopoulou, S. 2000. Overloading and inheritance in Java. In 2th Workshop on Formal Techniques for Java Programs.

André, P. and Royer, J.-C. 1992. Optimizing method search with lookup caches and incremental coloring. In *Proc. OOPSLA'92*. SIGPLAN Notices, 27(10). ACM Press, Vancouver, 110–126.

Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. 2005. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE 93,* 2 (Feb.), 449–466.

Atkinson, R., Liskov, B., and Scheifler, R. 1978. Aspects of implementing CLU. In *Proc. ACM 1978 Annual Conference*. ACM Press.

Bacon, D., Fink, S., and Grove, D. 2002. Space- and time-efficient implementation of the Java object model. In *Proc. ECOOP'2002*, B. Magnusson, Ed. LNCS 2374. Springer.

Bacon, D. and Sweeney, P. 1996. Fast static analysis of C++ virtual function calls. See OOPSLA [1996], 324–341.

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

Biberstein, M., Gil, J., and Porat, S. 2001. Sealing, encapsulation and mutability. See Knudsen [2001], 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.

Boucher, D. 1999. Analyse et optimisations globales de modules compilés séparément. Ph.D. thesis, Université de Montréal.

Boucher, D. 2000. GOld: a link-time optimizer for Scheme. In *Proc. Workshop on Scheme and Functional Programming. Rice Technical Report 00-368*, M. Felleisen, Ed. 1–12.

Boyland, J. and Castagna, G. 1996. Type-safe compilation of covariant specialization: a practical case. See Cointe [1996], 3–25.

Bracha, G. 2004. Generics in the java programming language. `http://sun.org`.

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. IEEE Int. Conf. Comp. Lang.* 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, 104–127.

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

Cardelli, L., Ed. 2003. *Proceedings of the 17th European Conference on Object-Oriented Programming, ECOOP'2003*. LNCS 2743. Springer.

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 Trans. Program. Lang. Syst. 17,* 3, 431–437.

Castagna, G. 1997. *Object-oriented programming: a unified foundation*. Birkhaüser.

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

Chen, W., Turau, V., and Klas, W. 1994. Efficient dynamic look-up strategy for multi-methods. In *Proc. ECOOP'94*, M. Tokoro and R. Pareschi, Eds. LNCS 821.

Chilimbi, T. M., Davidson, B., and Larus, J. R. 1999. Cache-conscious structure definition. In *Proc. ACM PLDI'99*. ACM Press.

Click, C. and Rose, J. 2002. Fast subtype checking in the Hotspot JVM. In *Proc. ACM-ISCOPE conference on Java Grande (JGI'02)*. 96–107.

Cohen, N. 1991. Type-extension type tests can be performed in constant time. *ACM Trans. Program. Lang. Syst. 13,* 4, 626–629.

Cointe, P., Ed. 1996. *Proceedings of the 10th European Conference on Object-Oriented Programming, ECOOP'96*. LNCS 1098. Springer.

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

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

Cook, S., Ed. 1989a. *Proceedings of the 3rd European Conference on Object-Oriented Programming, ECOOP'89*. Cambridge University Press.

Cook, W. R. 1989b. A proposal for making Eiffel type-safe. See Cook [1989a], 57–70.

Day, M., Gruber, R., Liskov, B., and Myers, A. 1995. Subtypes vs. where clauses. constraining parametric polymorphism. See OOPSLA [1995], 156–168.

Dean, J., Chambers, C., and Grove, D. 1995. Selective specialization for object-oriented languages. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'95)*. 93–102.

Dean, J., Grove, D., and Chambers, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. See Olthoff [1995], 77–101.

Desnos, N. 2004. Un garbage collector pour la coloration bi-directionnelle. M.S. thesis, Université Montpellier 2.

Dijkstra, E. W. 1960. Recursive programming. *Numer. Math. 2*, 312–318.

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, 211–214.

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

Driesen, K. 2001. *Efficient Polymorphic Calls*. Kluwer Academic Publisher.

Driesen, K. and Hölzle, U. 1995. Minimizing row displacement dispatch tables. See OOPSLA [1995], 141–155.

Driesen, K. and Hölzle, U. 1996. The direct cost of virtual function calls in C++. See OOPSLA [1996], 306–323.

Driesen, K., Hölzle, U., and Vitek, J. 1995. Message dispatch on pipelined processors. See Olthoff [1995], 253–282.

Driesen, K. and Zendra, O. 2002. Stress-testing control structures for dynamic dispatch in Java. In *Proc. Java Virtual Machine Research and Technology Symp., JVM'02*. Usenix, 105–118.

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. 2005. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst. 28,* 2, 331–388.

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. Lavoisier, 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, 3–12.

Ducournau, R. 2006. Coloring, a versatile technique for implementing object-oriented languages. Tech. Rep. 06-001, LIRMM, Université Montpellier 2. (subm. to ACM Trans. Program. Lang. Syst., rev. Aug. 2008).

Ducournau, R. 2008. Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.*, 1–51. (to appear).

Ducournau, R., Habib, M., Huchard, M., and Mugnier, M.-L. 1994. Proposal for a monotonic multiple inheritance linearization. See OOPSLA [1994], 164–175.

Ducournau, R., Morandat, F., and Privat, J. 2007. Modules and class refinement: a metamodeling approach to object-oriented languages. Tech. Rep. 07-021, LIRMM, Université Montpellier 2.

Ducournau, R. and Privat, J. 2008. A metamodeling approach to multiple inheritance. Tech. Rep. 08-xxx, LIRMM, Université Montpellier 2. (submitted).

Dujardin, E., Amiel, E., and Simon, E. 1998. Fast algorithms for compressed multimethod dispatch table generation. *ACM Trans. Program. Lang. Syst. 20,* 1, 116–165.

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, US.

Ernst, E. 2001. Family polymorphism. See Knudsen [2001], 303–326.

Ernst, E. 2002. Safe dynamic multiple inheritance. *Nord. J. Comput 9,* 1, 191–208.

Gagnon, E. M. and Hendren, L. 2001. SableVM: A research framework for the efficient execution of Java bytecode. In *Proc. USENIX JVM'01*. 27–40.

Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., and Willcock, J. 2003. A comparative study of language support for generic programming. In *Proc. OOPSLA'03*, R. Crocker and G. L. S. Jr., Eds. SIGPLAN Notices, 38(10). ACM Press, 115–134.

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, 601–634.

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

Gil, J. and Zibin, Y. 2005. Efficient subtyping tests with PQ-encoding. *ACM Trans. Program. Lang. Syst. 27,* 5, 819–856.

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.

Hochbaum, D. S., Ed. 1997. *Approximation algorithms for NP-hard problems.* PWS Publishing Company.

Holst, W., Szafron, D., Leontiev, Y., and Pang, C. 1998. Multi-method dispatch using single-receiver projections. Technical Report TR-98-03, University of Alberta, Edmonton, ca.

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

Hullot, J.-M. 1985. ceyx version 15. Technical Report 44, 45 et 46, I.N.R.I.A.

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

Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. 1997. Back to the future: The story of Squeak - a usable Smalltalk written in itself. See OOPSLA [1997], 318–326.

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

Jul, E. 2008. Precomputing method lookup. In *Workshop ICOOOLPS at ECOOP'08.*

Kiczales, G. and Rodriguez, L. 1990. Efficient method dispatch in PCL. In *Proc. ACM Conf. on Lisp and Functional Programming.* 99–105.

Knudsen, J. L., Ed. 2001. *Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP'2001.* LNCS 2072. Springer.

KNUTH, D. E. 1973. *The art of computer programming, Sorting and Searching*. Vol. 3. Addison-Wesley.

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.

KROGDAHL, S. 1985. Multiple inheritance in Simula-like languages. *BIT 25,* 2, 318–326.

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

LIPPMAN, S. 1996. *Inside the C++ Object Model*. 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.

LORENZ, M. AND KIDD, J. 1994. *Object-Oriented Software Metrics*. Prentice-Hall, Englewood Cliffs (NJ), USA.

MEHLHORN, K. AND TSAKALIDIS, A. 1990. Data structures. See Van Leeuwen [1990], Chapter 6, 301–341.

MEYER, B. 1992. *Eiffel: The Language*. Prentice-Hall.

MEYER, B. 1997. *Object-Oriented Software Construction*, second ed. Prentice-Hall.

MEYER, B. 2001. Overloading vs. object technology. *J. Obj. Orient. Program. 14,* 5, 3–7.

MEYER, J. AND DOWNING, T. 1997. *JAVA Virtual Machine*. O'Reilly.

MICROSOFT. 2001. C# Language specifications, v0.28. Technical report, Microsoft Corporation.

MUGRIDGE, W. B., HAMER, J., AND HOSKING, J. G. 1991. Multi-methods in a statically-typed programming language. See America [1991], 307–324.

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. See OOPSLA [1995], 124–139.

MYERS, A. AND LISKOV, B. 1994. Efficient implementation of parameterized types despite subtyping. Thor note 9, LCS, MIT.

NYSTROM, N., CHONG, S., AND MYERS, A. C. 2004. Scalable extensibility via nested inheritance. In *Proc. OOPSLA'04*, J. M. Vlissides and D. C. Schmidt, Eds. SIGPLAN Notices, 39(10). ACM Press, 99–115.

NYSTROM, N., CLARKSON, M. R., AND MYERS, A. C. 2003. Polyglot: An extensible compiler framework for Java. In *Proc. 12th Int. Conf. on Compiler Construction (CC'03)*. LNCS 2622. Springer, 138–152.

NYSTROM, N., QI, X., AND MYERS, A. C. 2006. $\mathcal{JE}$: Nested intersection for scalable software composition. In *Proc. OOPSLA'06*, P. L. Tarr and W. R. Cook, Eds. SIGPLAN Notices, 41(10). ACM Press, 21–35.

ODERSKY, M., SPOON, L., AND VENNERS, B. 2008. *Programming in Scala, A comprehensive step-by-step guide*. Artima.

ODERSKY, M. AND WADLER, P. 1997. Pizza into Java: Translating theory into practice. In *Proc. POPL'97*. ACM Press, 146–159.

OLTHOFF, W., Ed. 1995. *Proceedings of the 9th European Conference on Object-Oriented Programming, ECOOP'95*. LNCS 952. Springer.

OOPSLA 1994. *Proceedings of the 9th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'94*. SIGPLAN Notices, 29(10). ACM Press.

OOPSLA 1995. *Proceedings of the 10th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'95*. SIGPLAN Notices, 30(10). ACM Press.

OOPSLA 1996. *Proceedings of the 11th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'96*. SIGPLAN Notices, 31(10). ACM Press.

OOPSLA 1997. *Proceedings of the 12th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'97*. SIGPLAN Notices, 32(10). ACM Press.

OOPSLA 1998. *Proceedings of the 13th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'98*. SIGPLAN Notices, 33(10). ACM Press.

OOPSLA 1999. *Proceedings of the 14th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'99*. SIGPLAN Notices, 34(10). ACM Press.

OOPSLA 2001. *Proceedings of the 16th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'01*. SIGPLAN Notices, 36(10). ACM Press.

Palacz, K. and Vitek, J. 2003. Java subtype tests in real-time. See Cardelli [2003], 378–404.

Peyton Jones, S. 2003. *Haskell 98 Language and Libraries*. Cambridge University Press.

Privat, J. and Ducournau, R. 2004. Intégration d'optimisations globales en compilation séparée des langages à objets. In *Actes LMO'2004* in *L'Objet vol. 10*, J. Euzenat and B. Carré, Eds. Lavoisier, 61–74.

Privat, J. and Ducournau, R. 2005. Link-time static analysis for efficient separate compilation of object-oriented languages. In *ACM Workshop on Prog. Anal. Soft. Tools Engin. (PASTE'05)*. 20–27.

Privat, J. and Morandat, F. 2008. Coloring for shared object-oriented libraries. In *Workshop ICOOOLPS at ECOOP'08*.

Privat, J., Morandat, F., and Ducournau, R. 2006. Efficient separate compilation of object-oriented languages. In *Workshop ICOOOLPS at ECOOP'06*.

Pugh, W. and Weddell, G. 1990. Two-directional record layout for multiple inheritance. In *Proc. PLDI'90*. ACM SIGPLAN Notices, 25(6). 85–91.

Pugh, W. and Weddell, G. 1993. On object layout for multiple inheritance. Tech. Rep. CS-93-22, University of Waterloo.

Queinnec, C. 1998. Fast and compact dispatching for dynamic object-oriented languages. *Information Processing Letters 64,* 6, 315–321.

Raynaud, O. and Thierry, E. 2001. A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. See Knudsen [2001], 165–180.

Rossie, J. G., Friedman, D. P., and Wand, M. 1996. Modeling subobject-based inheritance. See Cointe [1996].

Sakkinen, M. 1989. Disciplined inheritance. See Cook [1989a], 39–58.

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 relationship. *Computer 16*, 53–60.

Shang, D. L. 1996. Are cows animals?

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.

Smaragdakis, Y. and Batory, D. 2002. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Soft. Eng. Meth. 11,* 2, 215–255.

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

Snyder, J. 1991b. Inheritance in object-oriented programming. In *Inheritance Hierarchies in Knowledge Representation and Programming Langages*, M. Lenzerini, D. Nardi, and M. Simi, Eds. John Wiley & Sons, Chapter 10, 153–171.

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

Sprugnoli, R. 1977. Perfect hashing functions: a single probe retrieving method for static sets. *Comm. ACM 20,* 11, 841–850.

Steele, G. 1990. *Common Lisp, the Language*, Second ed. Digital Press.

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

Stroustrup, B. 1998. *The C++ programming Language, 3^e ed.* Addison-Wesley.

Sweeney, P. F. and Burke, M. G. 2003. Quantifying and evaluating the space overhead for alternative C++ memory layouts. *Softw., Pract. Exper. 33,* 7, 595–636.

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

Szyperski, 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.

Taft, S. T., Duff, R. A., Brukardt, R. L., Ploedereder, E., and Leroy, P., Eds. 2006. *Ada 2005 Reference Manual: Language and Standard Libraries*. LNCS 4348. Springer.

Takhedmit, P. 2003. Coloration de classes et de propriétés : étude algorithmique et heuristique. M.S. thesis, Université Montpellier 2.

Tarjan, R. E. and Yao, A. C. C. 1979. Storing a sparse table. *Comm. ACM 22,* 11, 606–611.

Tip, F. and Sweeney, P. F. 2000. Class hierarchy specialization. *Acta Informatica 36,* 12, 927–982.

Torgersen, M. 1998. Virtual types are statically safe. In *Proc. of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL 5)*. San Diego, CA.

Torgersen, M., Hansen, C. P., Ernst, E., von der Ahé, P., Bracha, G., and Gafter, N. 2004. Adding wildcards to the java programming language. In *Proceedings of SAC'04*.

Ungar, D. and Smith, R. 1987. SELF: The power of simplicity. In *Proceedings of OOPSLA'87*, N. Meyrowitz, Ed. special issue of ACM SIGPLAN Notices, 22(12). 227–242. Également publié dans *Lisp and Symbolic Computation*, 4(3), Kluwer Academic Publishers, pages 187–205, 1991.

Van Leeuwen, J., Ed. 1990. *Algorithms and Complexity*. Handbook of Theoretical Computer Science, vol. 1. Elsevier, Amsterdam.

VanHilst, M. and Notkin, D. 1996. Using role components to implement collaboration-based designs. See OOPSLA [1996], 359–369.

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.

Vitter, J. S. and Flajolet, P. 1990. Average-case analysis of algorithms and data structures. See Van Leeuwen [1990], Chapter 9, 431–524.

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. See Knudsen [2001], 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.

Zendra, O., Colnet, D., and Collin, S. 1997. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. See OOPSLA [1997], 125–141.

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.

Zibin, Y. and Gil, J. 2003a. Incremental algorithms for dispatching in dynamically typed languages. In *Proc. POPL'03*. ACM, 126–138.

Zibin, Y. and Gil, J. 2003b. Two-dimensional bi-directional object layout. See Cardelli [2003], 329–350.

Received November 2002; revised July 2005, August 2008

APPENDIX

A. SPACE BENCHMARKS

Some large benchmarks are commonly used in the object-oriented implementation community[34], 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.

Seven techniques have been considered: ideal SST and pure NVI implementations—even though both are not applicable as all benchmarks are in unconstrained MI—, ideal devirtualization scheme of Section 5.4.2 (DVI), standard MI (SMI) and its variant with the empty subobject optimization (ESO), and coloring, as well unidirectional (COL_1) as bidirectional (COL_2). DVI may be understood as the best C++-like implementation[35], with a sound multiple inheritance semantics, for a hand made program, with a complete knowledge of the whole program. On the contrary, ESO is presumed the best subobject based MI implementation in separate compilation. In some cases, the effect of the first kind of empty subobjects (ESO_1) has been isolated: this is the optimization to consider when one wants to combine VBPTRs and ESO.

Regarding SI+MST implementations, no specific measures have been made as those implementations are mainly variants of SST and MI implementations. Therefore, the benchmarks presented hereafter measure mainly the overhead of different MI implementations, compared to SST.

At last, no simulation of tree-based techniques (see Section 5.2.2) have been made as their measurement would require programs, while our benchmarks are libraries.

A.1 Benchmark description, interpretation and correction

Each benchmark is a file of class descriptions, akin to *external schemata*, 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 do not comport informations about attributes. Therefore, we present statistics based on two groups of benchmarks. The first one includes data on attributes: Unidraw is a C++ program mainly in SI, Lov and Geode are Eiffel-like programs making an intensive use of MI and Smart Eiffel is the GNU Eiffel compiler. The second group involves several Java benchmarks (from IBM-SF to HotJava) plus some benchmarks of different languages which are dynamically typed but included here for comparison (Cecil, Self, Vortex3, Dylan, Harlequin).

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. We can only certify that non-object-oriented data has been removed from the Smart Eiffel, PRMcl and Java1.6

[34]Many people contributed to those benchmarks, among which Karel Driesen and Jan Vitek: at the beginning of this work, a current repository was Yoav Zibin's web site, `http://www.cs.technion.ac.il/~zyoav/`.

[35]However, as noticed in section 4.1.3, DVI is not possible in C++ (at least with the same semantics) and actual best C++ implementation would be less efficient (see Table XVIII).

benchmarks[36]. 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 attributes is not clear: the same attribute name in two related classes has been interpreted as overriding (as in EIFFEL), not as static overloading (as in C++ or JAVA): hence, Unidraw may be slightly underestimated. Renaming, which is proper to EIFFEL, has not been considered in the SMART EIFFEL benchmark[37], 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 SMART EIFFEL and PRM compilers which represent programs, not only libraries. Palacz and Vitek [2003] report that their experiments did not exceed one thousans classes loaded from almost 10-thousand class libraries.

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 *introduction overloading* (see Note 6, page 10): two methods or attributes with the same name, introduced in two unrelated classes, are considered as different[38]. Second, all methods are considered, whereas Yoav Zibin's benchmarks remove *degenerate methods*, i.e. methods which have only one definition. However, these degenerate methods might have been removed from the original benchmarks—this might explain the very low number of monomorphic methods in Table XI (see also Note 41, page 87).

Furthermore, some measures are uniquely determinate (class and method numbers, SMI numbers) whereas many other (ESO, DVI, coloring and even NVI) are the result of either an arbitrary choice or an approximate optimization: there is a very little chance that two different experiments, by two different programmers, give the same number. NVI, DVI and ESO depend on the choice of the direct superclass whose layout is extended by the current class: in all three cases, our heuristics choose a class with the greatest number of methods, which tends to minimize table size. NVI and ESO are then almost uniquely determinate. Unless otherwise stated, we use simple greedy heuristics.

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

A.2 Class hierarchies

Definitions. A *class hierarchy* is a partially ordered set of classes, noted $(X, \prec)$. $(X, \prec_{\mathrm{d}})$, the transitive reduction of $(X, \prec)$, is a directed acyclic graph. The *core* of a class hierarchy is the subset $X_c \subseteq X$ of classes which are concerned with MI, i.e.

[36]Thanks to Floréal Morandat and Olivier Zendra.

[37]In EIFFEL, it may also 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`.

[38]In JAVA, the same method may be introduced in two unrelated interfaces (introduction overloading) which are both implemented by a class: we count 2 methods in the class, but this is insignificant as the case is rather rare in our benchmarks. The same thing may occur in EIFFEL when the features are *deferred*.

Table V. Statistics on classes, including empty and virtual classes, and inheritance edges ($\prec_{\mathrm{d}}$), including virtual edges ($\prec_{\mathrm{v}}$) according to DVI scheme

	classes								edges	
name	total	empty		virtual	e+v	double	core	leaves	total	virtual
Java1.6	5075	1298	1535	53	136	239	1422	3825	6363	168
Geode	1318	163	436	78	106	231	989	732	2486	204
Unidraw	614	115	81	3	1	6	25	481	623	4
Lov-obj-ed	436	39	132	21	40	75	271	218	747	61
SmartEiffel	397	33	116	3	9	12	67	311	437	11
PRMcl	479	38	209	16	14	37	133	294	541	28
Total	8319	1686	2509	174	306	600	2907	5861	11197	476
IBM-SF	8793			346		524	4770	6001	12033	458
JDK1.3.1	7401			63		299	1512	5806	8605	227
Orbix	2716			10		31	271	2440	2923	26
Corba	1699			35		44	383	1473	2072	52
Orbacus	1379			25		64	502	954	1812	52
HotJava	736			3		55	217	525	927	33
Self	1802			4		57	154	1134	1889	34
Vortex3	1954			90		209	696	1216	2431	166
Cecil	932			31		76	306	601	1127	50
Dylan	925			8		13	65	806	953	11
Harlequin	666			24		84	278	371	907	59
Total	29003			639		1456	9154	21327	35679	1168

which have two direct superclasses, or have a subclass with two direct superclasses. X_c is the minimal set such that

$$C \prec_{\mathrm{d}} C_1 \ \&\ C \prec_{\mathrm{d}} C_2 \ \&\ C_1 \neq C_2 \Rightarrow C \in X_c$$
$$D \in X_c \ \&\ D \prec C \Rightarrow C \in X_c$$

Coloring, devirtualization and double compilation are based on the *conflict graph* (X_c, E_c), where

$$E_c = \{(C_1, C_2) \mid \exists D \in X_c, D \prec C_1 \ \&\ D \prec C_2 \ \&\ C_1 \not\preceq C_2 \ \&\ C_2 \not\preceq C_1\}.$$

In the coloring approach, coloring heuristics amounts to coloring the conflict graph [Pugh and Weddell 1990]: all other classes extend the core implementation in the same algorithmic way as with SST. *Double* classes X_d are the classes which must be used in the less efficient form in the double compilation approach, i.e. which are not a primary superclass of all their subclasses (see Section 6.1.4): they form a minimal *vertex cover* of the conflict graph. The *minimum vertex cover* problem is NP-hard [Garey and Johnson 1979], but it is polynomially 2-approximable (with a *maximal matching* [Hochbaum 1997]): therefore our numbers are less than twice the optimal and, in practice, very close to it. Any class has atmost one direct superclass in $X \backslash X_d$. In the DVI approach, *virtual classes* are classes in the core which are inherited by some subclass through two disjoint paths:

$$X_v = \{C \in X_c \mid \exists (C_1, C_2) \in E_c, C_1 \prec C \ \&\ C_2 \prec C\}$$

C, C_1, C_2 and D (from E_c definition) form a diamond. *Virtual edges* are a subset $\prec_{\mathrm{v}}$ of $\prec_{\mathrm{d}}$, which can be computed as a minimal subset of $X_d \times X_v$, the set of

Table VI. Statistics (total number, average per class and non-empty class, and maximum) on attributes, either introduced (a_C) or inherited (A_C). The total number of inherited attributes represent the total space occupied by one instance per class.

name	introduced (a_C)				inherited		(A_C)
Java1.6	8201	1.6	3.7	55	27580	5.4	137
Geode	2919	2.2	4.1	182	14392	10.9	217
Unidraw	1574	2.6	3.8	36	5103	8.3	47
Lov-obj-ed	1262	2.9	4.8	74	3554	8.2	105
SmartEiffel	977	2.5	3.9	39	1956	4.9	44
PRMcl	578	1.2	2.5	28	2411	5.0	29
Total	15511	1.9	3.8	182	54996	6.6	217

edges from a *double* class to a *virtual* one[39], such that $(X_c, \prec_{\mathrm{nv}}\}$ is acyclic, in the undirected sense (where $\prec_{\mathrm{nv}}$ stands for $\prec_{\mathrm{d}} \setminus \prec_{\mathrm{v}}$).

Class numbers. Table V 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 5.4.2). The number of classes without attributes is surprisingly high in all benchmarks, between 30 and 45 %. 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. Furthermore, the number of classes without attribute or non-virtual direct superclass (column "e+v") is quite small, as would be the benefit of combining ESO and DVI. The "double" column indicates the number of doubles classes: it is roughly 25 % of the "core" number. Both numbers are a good indication on how much MI is used.

In the Java benchmarks, no specific measure of interfaces has been made. However, $X_d \cup X_v$ is a good approximate of the set of interfaces: a Java class is never virtual (except `Object`, as far as one considers that it is a class) and all classes might be in $X \setminus X_d$. Therefore, the set of interfaces is a solution to $X_d \cup X_v$, but it is likely that our heuristics miss some interfaces and replace them by classes. In Fig. 10, $X_v = \{I, K, L\}$ and X_d might be as well $\{J, L, M\}$ as $\{K, A, B\}$, or some combination of the two.

A.3 Object layout and dynamic space

Attribute numbers. Table VI presents statistics on the number of attributes, as well introduced in a class (a_C) as the total number for the class ($A_C = \sum_{C \preceq D} a_C$). It is interesting to notice that the average number of attributes per class is quite uniform.

Subobject numbers. Attribute number is to compare with the number of subobjects (O_C), i.e. pointers to method tables in the object layout: this gives a good idea of the dynamic overhead of MI implementations (Table VII). The SMI number of subobjects is exactly n_C, the number of superclasses, including the class C itself: hence, it gives a good indication on how much specialization is used. 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 and DVI reduces the

[39] In C++ best devirtualization, virtual edges would be exactly $\prec_{\mathrm{d}} \cap (X \times X_v)$.

Table VII. Attribute number (A_C) and absolute overhead in object layout (O_C), according to the different techniques, i.e. subobject or hole number (average and maximum per class)

name	A_C		NVI		DVI		ESO		ESO_1		SMI		COL_1		COL_2	
Java1.6	5.4	137	1.6	18	3.1	20	2.2	19	4.2	22	5.4	23	1.0	1	1.0	1
Geode	10.9	217	10.5	140	8.3	28	8.5	33	13.4	48	14.0	50	2.9	60	2.4	22
Unidraw	8.3	47	1.0	4	1.4	3	3.1	7	3.5	8	4.0	10	1.0	3	1.4	2
Lov-obj-ed	8.2	105	4.2	19	5.1	16	5.4	17	7.8	20	8.5	24	2.7	30	1.6	4
SmartEiffel	4.9	44	2.1	8	2.1	4	2.4	5	6.1	11	8.6	14	1.0	1	1.0	1
PRMcl	5.0	29	1.5	7	1.8	5	3.0	9	4.4	12	4.6	12	1.2	9	1.3	5
Total	6.6	217	3.1	140	3.8	28	3.5	33	5.9	48	6.9	50	1.4	60	1.3	22
IBM-SF			4.0	39	3.6	14					9.2	30				
JDK1.3.1			1.5	21	1.9	21					4.4	24				
Orbix			1.2	6	1.3	6					2.8	13				
Corba			1.7	19	1.9	11					3.9	18				
Orbacus			2.0	11	2.3	11					4.5	19				
HotJava			2.0	15	2.6	15					5.1	23				
Self			20.2	43	20.8	25					30.9	41				
Vortex3			3.1	65	3.5	11					7.2	30				
Cecil			2.2	30	2.8	9					6.5	23				
Dylan			1.2	6	1.7	4					5.5	13				
Harlequin			2.2	15	2.7	16					6.7	31				
Total			3.6	65	3.7	25					7.7	41				

Table VIII. Relative overhead in object layout, i.e. $(O_C - 1)/(1 + A_C)$ (average and maximum per class)

name	NVI		DVI		ESO		ESO_1		SMI		COL_1		COL_2	
Java1.6	0.10	5.7	0.33	5.7	0.19	6.0	0.50	7.0	0.68	7.3	0.00	0.0	0.00	0.0
Geode	0.79	26.5	0.61	7.5	0.63	6.0	1.04	8.0	1.09	10.0	0.18	19.7	0.13	7.0
Unidraw	0.00	0.3	0.04	0.3	0.23	0.8	0.27	2.0	0.32	2.0	0.00	0.5	0.05	1.0
Lov-obj-ed	0.35	7.5	0.44	3.0	0.48	4.5	0.74	8.5	0.82	10.5	0.21	6.0	0.07	3.0
SmartEiffel	0.19	1.5	0.19	1.5	0.24	1.0	0.86	3.5	1.29	5.0	0.00	0.0	0.00	0.0
PRMcl	0.08	1.2	0.13	1.0	0.32	1.5	0.56	3.0	0.60	3.0	0.03	2.0	0.06	1.3

overhead by 40 %. Of course, DVI is the best subobject technique, as it is near optimal. One should not be surprised by the fact that NVI is worse than ESO and DVI in the case of Geode (in average) and worse than SMI (in worst case): the worst-case complexity of NVI is exponential. Furthermore, empty subobjects bring no optimization to NVI.

Table VII also compares subobject number with hole number in attribute coloring, augmented for taking into account the pointer to method table and the extra head pointer for garbage collection (in the bidirectional case only, see Section 7.8). The latter explains that average overhead for bidirectional coloring (COL_2) is not much better than in the unidirectional case (COL_1). Table VIII presents the relative overhead per object for the same data. On the whole, it appears that bidirectional coloring is far better than the best subobject-based implementations (DVI and ESO). Unidirectional coloring has also a good average overhead, compared to ESO and DVI, but the maximum relative overhead may be greater: this must be due to the fact that heuristics were not designed to minimize this parameter, but only the overall overhead.

Table IX shows the space occupied, either by upcast tables in method tables, or by VBPTRs in object layout, the latter in the DVI case only (see Section 4.3). One

Table IX. Statistics (total, average and max) on upcast table size, and VBPTRs in the DVI case

name	e-VBPTRs	i-VBPTRs	DVI	ESO=ESO$_1$	SMI
Java1.6	18080 4. 54	18367 4. 54	18501 4. 54	39535 8. 77	65816 13. 108
Geode	18510 14. 96	26477 20. 185	28240 21. 185	89547 68. 481	94992 72. 490
Unidraw	244 0. 3	264 0. 7	264 0. 7	3310 5. 28	4711 8. 45
Lov-obj-ed	2768 6. 32	3170 7. 40	3310 8. 41	10111 23. 117	11503 26. 118
SmartEiffel	446 1. 3	2102 5. 21	2102 5. 21	6678 17. 54	13732 35. 89
PRMcl	522 1. 10	692 1. 13	694 1. 13	4081 9. 53	4558 10. 53
Total	40570 5. 96	51072 6. 185	53111 6. 185	153262 18. 481	195312 23. 490
IBM-SF	48083 5. 28	66398 8. 53	68547 8. 54		374677 43. 302
JDK1.3.1	8371 1. 29	8676 1. 39	8840 1. 39		64498 9. 87
Orbix	1230 0. 7	1246 0. 9	1378 1. 10		8678 3. 51
Corba	2448 1. 25	2795 2. 32	3045 2. 34		11846 7. 95
Orbacus	2513 2. 13	2553 2. 13	2824 2. 13		12519 9. 73
HotJava	1434 2. 17	1434 2. 17	1457 2. 17		8729 12. 74
Self	35753 20. 25	35861 20. 50	35866 20. 50		355366 197. 487
Vortex3	10834 6. 39	15156 8. 69	15243 8. 69		53107 27. 273
Cecil	3192 3. 35	4096 4. 54	4133 4. 54		18357 20. 183
Dylan	668 1. 4	1693 2. 13	1693 2. 13		12434 13. 63
Harlequin	1861 3. 31	3332 5. 50	3343 5. 50		14317 21. 176
Total	116387 4. 39	143240 5. 69	146369 5. 69		934528 32. 487

must remember that there is no need for upcast tables, or VBPTRs, with NVI. This demonstrates that VBPTRs have a very large overhead, especially with SMI in actual compilers. ESO has a poor effect: this is due to the fact that empty classes are mostly of the second kind, which does not save on upcast tables. On the contrary, DVI is a great improvement: upcast tables are 3 times smaller in the worst case, and VBPTRs are still fewer. On the whole, in the worst case (Geode), DVI with e-VBPTRs is more than doubling dynamic space w.r.t. SST, which is larger than the expected overhead of attribute coloring. On average, the difference between e-VBPTRs and i-VBPTRs is not significant.

If one accepts doubling (Geode), or almost doubling (Smarteiffel, Lov) dynamic memory, one should envisage *method table flow* (see Section 4.5), which is stricly less than doubling dynamic memory as monomorphic attributes do not need tables. Unfortunately, benchmarks include no information on polymorphic/monomorphic attributes[40]. One must however notice that collections, i.e. arrays, are not taken into account in our statistics: the cost of doubling arrays instead of adding a small constant overhead may be larger than doubling standard objects.

On the whole, MI overhead in the object layout may be less than the one that previous works as [Gil and Sweeney 1999; Eckel and Gil 2000; Sweeney and Burke 2003] 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 do not include any per benchmark statistics on either subobjects or upcast tables.

[40]In the context of an attribute we use the adjective monomorphic (resp. polymorphic) to denote the fact that the attribute type is monomorphic, i.e. a primitive type (resp. polymorphic, i.e. a class). In the context of a method (see further), we will use those adjective to denote the fact that the method has one definition, or more.

Table X. Statistics on method number (total, average and maximum per class)

name	introduced (m_C) (a)			defined ($m_C^{\mathtt{def}}$)			inherited (M_C) (b)			(b/a)
Java1.6	22121	4.4	286	35352	7.0	291	190289	37.5	670	8.6
Geode	8078	6.1	193	14214	10.8	207	305560	231.8	880	37.8
Unidraw	1752	2.9	103	3328	5.4	103	14781	24.1	124	8.4
Lov-obj-ed	3631	8.3	117	5026	11.5	127	37436	85.9	289	10.3
SmartEiffel	4854	12.2	222	7865	19.8	222	53704	135.3	324	11.1
PRMcl	2369	4.9	115	3793	7.9	115	37517	78.3	208	15.8
Total	42805	5.1	286	69578	8.4	291	639287	76.8	880	14.9
IBM-SF	25000	2.8	257	116152	13.2	320	394375	44.9	346	15.8
JDK1.3.1	9567	1.3	149	28683	3.9	150	142445	19.2	243	14.9
Orbix	1135	0.4	64	3704	1.4	78	22637	8.3	109	19.9
Corba	627	0.4	43	3201	1.9	50	13578	8.0	67	21.7
Orbacus	1716	1.2	74	4996	3.6	79	24877	18.0	137	14.5
HotJava	1310	1.8	80	3397	4.6	85	25149	34.2	189	19.2
Self	26267	14.6	233	29415	16.3	233	1040415	577.4	969	39.6
Vortex3	933	0.5	148	2496	1.3	148	305755	156.5	204	327.7
Cecil	2743	2.9	61	4208	4.5	62	73366	78.7	156	26.7
Dylan	814	0.9	64	1784	1.9	64	71308	77.1	139	87.6
Harlequin	416	0.6	62	1016	1.5	67	23167	34.8	129	55.7
Total	70528	2.4	257	199052	6.9	320	2137072	73.7	969	30.3

A.4 Methods and static space

Notations and method numbers. In the following, m_C (resp. $m_C^{\mathtt{def}}$) is the number of methods *introduced* (resp. *defined*) in class C, and M_C is the total number of methods defined in, or inherited by C: $M_C = \sum_{C \preceq D} m_D$.

Table X presents statistics on these three parameters m_C, $m_C^{\mathtt{def}}$ and M_C. One may first notice that, contrarily to attributes, method numbers vary between benchmarks, with a ratio greater than 10 for introduction and definition, and up to 70 for inherited methods[41]. Moreover, from Tables VI and X, one sees that the number of attributes (A_C) and methods (M_C) in a class is far from 2^{15}: thus, an implementation of Δs and offsets with short integers is valid. This explains why, in the following, the numbers for upcast tables and shifts will be divided by 2.

Secondly, as already noticed in the case of SMALLTALK [Ducournau 1997], in most benchmarks $2\sum_C m_C$ is far greater than $\sum_C m_C^{\mathtt{def}}$, which means that most methods have a unique definition[41]. The case of Self is extreme, even though not representative of this survey, as it is a dynamically typed prototype language.

Table XI confirms this first evaluation: the table shows the number of introduced monomorphic and polymorphic methods (see note 40), together with the number of definitions per polymorphic method. The numbers are very impressive: (1) there are more than 60 % monomorphic methods on average[41], and up to 90 % in some benchmarks: therefore, all calls to a large majority of methods could be compiled

[41] The JAVA benchmarks present un exception to these observations. However, the method numbers in Tables X and XI rise a question about JAVA benchmarks. The numbers for Java1.6 are markedly higher than those in other JAVA benchmarks, especially for monomorphic methods—this must come from the fact that degenerate methods have been removed from other JAVA benchmarks. In spite of this removing, there remain many monomorphic methods because degenerate methods only consider the method name, not the introduction class.

Table XI. Number and rate of monomorphic methods; number of polymorphic methods; number of polymorphic method definitions (total, average and maximum per method)

name	mono		poly	defined		
Java1.6	18443	0.83	3678	16909	4.6	433
Geode	7294	0.90	784	6920	8.8	773
Unidraw	1419	0.81	333	1909	5.7	42
Lov-obj-ed	3219	0.89	412	1807	4.4	225
SmartEiffel	4450	0.92	404	3415	8.5	53
PRMcl	1965	0.83	404	1828	4.5	119
Total	36790	0.86	6015	32788	5.5	773
IBM-SF	5344	0.21	19656	110808	5.6	2245
JDK1.3.1	3889	0.41	5678	24794	4.4	548
Orbix	350	0.31	785	3354	4.3	351
Corba	321	0.51	306	2880	9.4	445
Orbacus	357	0.21	1359	4639	3.4	143
HotJava	582	0.44	728	2815	3.9	72
Self	25351	0.97	916	4064	4.4	133
Vortex3	472	0.51	461	2024	4.4	123
Cecil	2217	0.81	526	1991	3.8	374
Dylan	556	0.68	258	1228	4.8	69
Harlequin	275	0.66	141	741	5.3	50
Total	39714	0.56	30814	159338	5.2	2245

as static calls and a simple *class hierarchy analysis*, would be of great improvement if it were applied at link-time; (2) some of the few polymorphic methods may have up to 2245 method definitions: type prediction and tree-based dispatch might be defeated in some very specific situations. In the absence of compilers optimizing monomorphic calls, those numbers are a strong argument in favor of non-OO functions in OO languages, like non-`virtual` functions of C++ or `static` functions in Java—however, in some cases (Unidraw), it is unknown whether such functions are included in the benchmarks or not.

Method tables. According to the different techniques θ in {SST, DVI, NVI, ESO, SMI}, the total size of method tables, noted T_θ, may be computed by the following formulas:

$$T_\theta = \sum_C f_C^\theta, \text{ where } f_C^\theta = \begin{cases} M_C = \sum_{C \preceq D} m_D & \theta = \text{SST} \\ \sum_{C \preceq D} M_D & \theta = \text{SMI} \\ \sum_{C \preceq_{\text{eso}} D} M_D & \theta = \text{ESO} \end{cases} \tag{16}$$

The SMI case can be rewritten as $T_\theta = \sum_D n'_D M_D$, where n'_D is the number of D subclasses, including D. The ESO case amounts to excluding merged classes from the sum on D: $C \preceq_{\text{eso}} D$ means that $C \preceq D$ and there is no D' merged into D, i.e. empty and such that $C \preceq D' \prec_{\text{d}} D$. In the NVI case,

$$f_C^{nvi} = M'_C + \sum_{C \prec_{\text{d}} D} f_D^{nvi} \quad \text{with } M'_C = \begin{cases} M_C - M_D & \text{when } C \prec_{\text{e}} D \\ M_C = m_C & \text{otherwise } (C \text{ is a root}) \end{cases} \tag{17}$$

In the DVI case, the formula is more complicated:

$$f_C^{dvi} = f'_C + \sum_{C \prec_{\text{nnv}} D} f'_D \quad \text{where } f'_C = M'_C + \sum_{C \prec_{\text{nv}} D} f'_D \tag{18}$$

Table XII. Statistics on method table size

	size in K-entries						ratio w.r.t. SST				
name	SST	COL_2	NVI	DVI	ESO	SMI	COL_2	NVI	DVI	ESO	SMI
Java1.6	190	194	240	278	462	693	1.0	1.3	1.5	2.4	3.6
Geode	306	383	1099	911	1241	1905	1.3	3.6	3.0	4.1	6.2
Unidraw	15	15	15	15	33	42	1.0	1.0	1.0	2.2	2.9
Lov-obj-ed	37	49	82	97	123	184	1.3	2.2	2.6	3.3	4.9
SmartEiffel	54	54	90	92	121	295	1.0	1.7	1.7	2.2	5.5
PRMcl	38	40	49	62	102	152	1.1	1.3	1.7	2.7	4.0
Total	639	735	1574	1455	2081	3271	1.1	2.5	2.3	3.3	5.1
IBM-SF	394	553	805	668		2034	1.4	2.0	1.7		5.2
JDK1.3.1	142	147	174	191		536	1.0	1.2	1.3		3.8
Orbix	23	23	27	29		62	1.0	1.2	1.3		2.8
Corba	14	17	23	24		46	1.2	1.7	1.7		3.4
Orbacus	25	25	37	40		94	1.0	1.5	1.6		3.8
HotJava	25	26	30	32		99	1.1	1.2	1.3		3.9
Self	1040	1057	1685	1687		6679	1.0	1.6	1.6		6.4
Vortex3	306	307	940	1042		2184	1.0	3.1	3.4		7.1
Cecil	73	75	148	180		411	1.0	2.0	2.4		5.6
Dylan	71	71	84	118		311	1.0	1.2	1.6		4.4
Harlequin	23	23	49	59		146	1.0	2.1	2.6		6.3
Total	2137	2324	4004	4069		12602	1.1	1.9	1.9		5.9

where $C \prec_{\text{nnv}} D$ iff $C \prec D$ and $\nexists D', C \preceq D' \prec_{\text{nv}} D$.

Table XII 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 between 2 and 4. When 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. As for pure NVI, besides its unsound semantics, its bad worst-case complexity tends to reduce the gap with SMI and ESO.

For assessing the complete static space, one must add to method tables i) upcast tables, ii) receiver adjustment, with thunks or shifts, iii) downcast tables, iv) code sequence for all call sites (Table XIII).

Thunk number. For an exact assessment of MI overhead, one must also 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 θ and SST, $T_\theta - T_{sst}$: there is a thunk per table entry, but for each class-method pair there is exactly one null shift. In contrast, without thunks, one must add shifts in method tables, i.e. $T_\theta/2$ (once again with short integers). On the whole, the total size with thunks (resp. shifts) is $3T_\theta - 2T_{sst} + U_\theta/2$ (resp. $3T_\theta/2 + U_\theta/2$).

The advantage of DVI w.r.t. SMI lies in the null thunks which may be measured by the ratio $(T_{dvi} - T_{sst})/(T_{smi} - T_{sst})$: this ratio is around 40 % in case of intensive use of MI (Geode and Lov): the gain is both in static space and in time efficiency.

Downcast. One should also add downcast tables ($\Delta^{\Uparrow}$): however, from the hashtable scheme proposed in Appendix B.2, one sees that those downcast tables would have about $4n_C$ entries for each class C, where n_C is the number of superclasses of C

Table XIII. Elements of static space, for subobject-based techniques, with thunks or shifts, and for invariant techniques

	thunks	shifts	invariant	Table
method table	T_θ	T_θ	T_θ	XII
upcast table	$U_\theta/2$	$U_\theta/2$	—	IX
receiver adjustment	$2(T_\theta - T_{sst})$	$T_\theta/2$	—	
code size	$3\#call$	$5\#call$	$3\#call$	XIV

Table XIV. Estimate of code size for method calls, and comparison with other elements of static space, in the ESO case (in K-words)

name	classes	methods	$\#call$	code	table	shifts	thunks
Java1.6	5075	35352	141408	424	462	514	543
Geode	1318	14214	56856	171	1241	734	1872
Unidraw	614	3328	13312	40	33	43	36
Lov-obj-ed	436	5026	20104	60	123	102	171
SmartEiffel	397	7865	31460	94	121	123	134
PRMcl	479	3793	15172	46	102	82	130
Total	8319	69578	278312	835	2081	1597	2884

(including C)[42]. In comparison, class coloring uses a little more than n_C entries. The ratio of about 4, is more or less the same than between SMI/ESO and COL for method tables. More precise comparison and perfect hashing are presented hereafter.

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 XIV shows that the 3 instruction sequence code for method call in standard SST has a rather significant global impact on static space. In the following, we assume a number of call sites $\#call = 4\sum_C m_C^{\text{def}}$. Without thunks, the shifts in the code will need two extra instructions per call site, hence a total of $2\#call$. 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, $Z_{shifts} = 2\#call + T_\theta/2$, with the cost of thunks, $Z_{thunks} = 2(T_\theta - T_{sst})$. The total static size will be obtained with $3\#call + T_\theta + Z + U_\theta/2$[43]. In the thunk case, we get a total of $3\#call + 3T_\theta - 2T_{sst} + U_\theta/2$. In the shift case, $5\#call + 3T_\theta/2 + U_\theta/2$. Table XV shows that, even when taking into account the size of the code, which is advantageous for thunks, the thunk space overhead remains greater.

As noticed earlier, offsets and shifts can be implemented as short integers, as they are far from 2^{15}. However, treating attribute and method offsets, or class identifiers, as immediate values may depend on the processor. According to Driesen

[42]The n_C statistics is in column SMI of Table VII.

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

Table XV. Final comparison between SST, coloring, DVI and SMI (both with thunks), and between shifts and thunks in the ESO case (in K-words)

	total size						ratio vs. SST				
name	SST	COL_2	DVI	shifts	thunks	SMI	COL_2	DVI	shifts	thunks	SMI
Java1.6	615	631	888	1419	1448	2155	1.0	1.4	2.3	2.4	3.5
Geode	476	563	2307	2191	3328	5323	1.2	4.8	4.6	7.0	11.2
Unidraw	55	56	56	117	110	140	1.0	1.0	2.1	2.0	2.6
Lov-obj-ed	98	112	277	290	359	543	1.1	2.8	3.0	3.7	5.6
SmartEiffel	148	150	264	341	352	879	1.0	1.8	2.3	2.4	5.9
PRMcl	83	86	157	231	279	428	1.0	1.9	2.8	3.4	5.1
Total	1474	1599	3949	4590	5877	9468	1.1	2.7	3.1	4.0	6.4
IBM-SF	1788	1987	2644			6893	1.1	1.5			3.9
JDK1.3.1	487	507	637			1699	1.0	1.3			3.5
Orbix	67	71	87			191	1.1	1.3			2.8
Corba	52	59	84			154	1.1	1.6			3.0
Orbacus	85	88	130			299	1.0	1.5			3.5
HotJava	66	69	88			293	1.0	1.3			4.4
Self	1393	1437	3351			18486	1.0	2.4			13.3
Vortex3	336	344	2552			5997	1.0	7.6			17.9
Cecil	124	128	444			1147	1.0	3.6			9.3
Dylan	93	95	232			817	1.0	2.5			8.8
Harlequin	35	38	146			411	1.1	4.1			11.6
Total	4526	4824	10395			36387	1.1	2.3			8.0

Table XVI. Comparison between static and dynamic sizes—the last ratio represents the number of instances per class that occupy the same space as the static space.

	dynamic space			static space		ratio
	attributes	overhead	total (K)	methods (K)	total (K)	
SST	54996	8319	63	639	1474	23.3
COL_2	54996	10751	66	735	1599	24.3
SMI	54996	57557	113	3271	9468	84.1

et al. [1995], current 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. Of course, the problem may be avoided, provided that the assembly language does the job, which seems to be a current practice (for instance, `gcc` under Linux).

Static vs. dynamic space. When comparing Tables VI, VII, XII and XV, one gains an insight of the relative importance of dynamic and static spaces. Table XVI presents the ratio between static and dynamic spaces, i.e. the number of instances per class that are equivalent to the static space. This ratio is more than 3-fold higher for SMI than for SST/COL. Moreover, the total static overhead of SMI vs. SST/COL represent about 120 instances in the SST implementation. Notwithstanding the fact that those benchmarks are libraries, not actual programs, there is some evidence here that SMI spends more space for programs than data.

Table XVII. Statistics on method table size, when all non-leaf classes are abstract

	size in K-entries						ratio w.r.t. SST				
name	SST	COL_2	NVI	DVI	ESO	SMI	COL_2	NVI	DVI	ESO	SMI
Java1.6	138	140	177	206	346	528	1.0	1.3	1.5	2.5	3.8
Geode	176	212	638	495	697	1089	1.2	3.6	2.8	4.0	6.2
Unidraw	11	11	11	11	25	33	1.0	1.0	1.0	2.2	3.0
Lov-obj-ed	21	27	48	56	75	112	1.3	2.3	2.6	3.6	5.3
SmartEiffel	43	43	72	73	100	240	1.0	1.7	1.7	2.3	5.6
PRMcl	24	25	33	41	69	107	1.0	1.3	1.7	2.9	4.4
Total	413	458	979	881	1311	2110	1.1	2.4	2.1	3.2	5.1
IBM-SF	254	367	561	430		1422	1.4	2.2	1.7		5.6
JDK1.3.1	105	107	129	141		421	1.0	1.2	1.3		4.0
Orbix	18	18	21	22		49	1.0	1.2	1.3		2.8
Corba	10	12	16	17		34	1.2	1.6	1.7		3.5
Orbacus	15	15	24	26		66	1.0	1.6	1.7		4.3
HotJava	19	20	23	24		80	1.0	1.2	1.3		4.2
Self	674	683	1096	1096		4397	1.0	1.6	1.6		6.5
Vortex3	190	191	611	625		1384	1.0	3.2	3.3		7.3
Cecil	48	48	99	115		282	1.0	2.1	2.4		5.9
Dylan	61	61	71	104		273	1.0	1.2	1.7		4.4
Harlequin	13	13	30	34		91	1.0	2.3	2.6		6.9
Total	1407	1536	2682	2632		8497	1.1	1.9	1.9		6.0

Customization. If one assumes a uniform size for all method definitions, an estimate of the overhead for customization is $\sum_C M_C / \sum_C m_C$ (Table X, last column). The ratio runs from less than 7 to more than 37 in the JAVA, C++ and EIFFEL benchmarks. It is even higher in the other benchmarks—up to 327 for Vortex3—but it may be less comparable.

One easily concludes that customization cannot be used alone and systematically. It must be coupled with other optimizations—first of all, dead code elimination—and one might envisage to apply it not on all methods definitions, but only when a significant improvement occurs.

Abstract classes. Many other data are missing, for instance the number of abstract classes, i.e. without proper instances, which would reduce the static space estimate, as in the $\sum_C$ formulas (16) and following, 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: according to Table V, one third of all classes would be abstract. This is an upper bound, twice the rate advocated by [Lorenz and Kidd 1994]. Table XVII presents the same statistics as Table XII, according to this assumption. The effect of abstract classes is important, but the variation of the ratios between the different implementations is not significant.

In a separate compilation framework, abstract classes must be syntactically distinguished if one wants to gain from them, either by being declared *abstract*, as in SMALLTALK, or through the existence of some *deferred feature*, as in EIFFEL.

DVI: ideal vs. C++. As noticed earlier, the devirtualization scheme (DVI) proposed here differs from C++ specification: Table XVIII compares DVI with the best hand-made C++ implementation, with the same sound MI semantics.

Table XVIII. Comparison between DVI (left) and C++ (right) devirtualization schemes

name	virt. edges		subobjects		e-VBPTRs		cast table		table size	
Java1.6	168	450	15934	18048	18080	23515	18501	23998	278	308
Geode	204	417	10894	13113	18510	35506	28240	44733	911	1097
Unidraw	4	9	855	874	244	266	264	296	15	15
Lov-obj-ed	61	107	2211	2438	2768	3364	3310	4106	97	106
SmartEiffel	11	23	843	1241	446	1054	2102	4006	92	126
PRMcl	28	66	846	1207	522	1154	694	1341	62	80
Total	476	1072	31583	36921	40570	64859	53111	78480	1455	1732
IBM-SF	458	2083	31961	50918	48083	172768	68547	183180	668	1087
JDK1.3.1	227	556	14409	16851	8371	12541	8840	13013	191	210
Orbix	26	145	3630	4008	1230	1837	1378	1977	29	32
Corba	52	218	3215	4457	2448	4947	3045	5249	24	32
Orbacus	52	252	3235	3899	2513	3655	2824	3891	40	47
HotJava	33	65	1934	2136	1434	1687	1457	1717	32	33
Self	34	57	37543	38205	35753	36436	35866	36803	1687	1692
Vortex3	166	352	6776	8576	10834	18129	15243	23698	1042	1321
Cecil	50	120	2612	3470	3192	5638	4133	6574	180	233
Dylan	11	28	1587	1846	668	1406	1693	2492	118	137
Harlequin	59	101	1831	2377	1861	3452	3343	5385	59	77
Total	1168	3977	108733	136743	116387	262496	146369	283979	4069	4901

Subtyping test. In Table VII, the SMI subobject number is exactly the number n_C of superclasses of each class C (including itself): this is a good approximate of class coloring, which is the extension of Cohen's technique to MI. Table XIX confirms that good approximation, by presenting the size occupied by Cohen's table in the abstract SST implementation, together with the space occupied by the corresponding class coloring. It appears that the overhead of coloring upon SST is not significant. Once again, class identifiers may be implemented as short integers, as class numbers are far from 2^{15}. Therefore, the average space is between 2 and 6 words per class for all JAVA benchmarks, which must be compared to 11 words reported by [Click and Rose 2002], for a technique which is however incremental, unlike coloring.

Regarding hashtable-based implementation as sketched in Section 3.2.4 and Appendix B.2, efficient pseudo-constant time implementation like *linear probing* requires about $2n_C$ entries—each entry being a pair identifier-shift in subobject-based implementations. Table XIX gives also statistics for *perfect hashing*, i.e. constant-time hashtables with exactly one probe per access. Two hash functions have been tested, using bit-wise and or remainder of integer division: both result in an integer value H_C, which is the number of entries in the hashtable and the parameter of the hash function. With the remainder of integer division, H_C is in average 2-fold the superclass number n_C: it means that perfect hashing is no more costly than classic linear probe. With bit-wise mask, the result is not as good: H_C is between 2 and 6-fold greater than with integer division. However, H_C average remains between 7 and 240-fold smaller than the class number N—in all benchmarks but the most demanding, Geode, H_C is far lesser than $\sqrt{n_C N}$—whereas $N/2$ would be the average size of direct access tables. Furthermore, the overhead w.r.t. pseudo-constant time hashtables is also far lower than method table overhead. Moreover, in both cases, the computation time is not significant, even with a non-optimized algorithm.

On the whole, this is a positive result and perfect hashing should be strongly

Table XIX. Downcast implementations: superclass number (n_C), class coloring and perfect hashing (average and maximum per class) compared to class number (N)

name	SST (n_C)		COL_2		COL_1		hash (`mod`)		hash (`and`)		N
IBM-SF	9.2	30	9.7	30	11.4	30	22.9	128	95.9	8224	8793
JDK1.3.1	4.4	24	4.6	24	5.0	24	8.2	61	51.5	4100	7401
Java1.6	4.4	22	4.7	22	5.0	22	8.3	57	33.8	4097	5074
Orbix	2.8	13	2.8	13	2.9	13	4.4	32	12.3	2050	2716
Corba	3.9	18	3.9	18	4.2	18	6.5	48	19.5	1026	1699
Orbacus	4.5	19	4.7	19	5.2	19	8.1	50	24.0	1026	1379
HotJava	5.1	23	5.5	23	6.8	23	9.8	85	32.8	519	736
JDK.1.0.2	4.6	14	4.7	14	4.8	14	8.4	39	34.2	580	604
Self	30.9	41	31.2	41	31.4	41	66.4	140	181.7	1088	1802
Geode	14.0	50	17.4	50	21.6	50	41.8	267	160.8	1026	1318
Vortex3	7.2	30	7.5	30	8.0	30	15.5	109	79.4	1232	1954
Cecil	6.5	23	6.6	23	7.1	23	12.7	78	53.9	608	932
Dylan	5.5	13	5.5	13	5.6	13	9.0	26	25.5	516	925
Harlequin	6.7	31	7.6	31	9.1	31	13.6	106	39.7	580	666
Lov-obj-ed	8.5	24	11.4	24	12.7	24	19.4	90	53.2	508	436
SmartEiffel	8.6	14	8.6	14	8.7	14	16.3	37	40.0	272	397
Unidraw	4.0	10	4.0	10	4.0	10	6.8	24	17.0	514	614
PRMc	4.5	12	4.5	12	4.8	12	7.5	26	19.5	268	280

envisaged for downcast in SMI, together with access to interface tables in MST. Chosing between the two functions involves a tradeoff between time and space as bit-wise `and` takes between 5 and 20 cycles less than integer division.

A.5 Time efficiency

Assessing time efficiency from those spacial benchmarks requires quite hypothetical assumptions. In all subobject-based implementations, the time overhead is mostly in the shifts, either static or dynamic. Thus the subobjects number is a good indication of null shifts (Table VII), whereas upcast tables size is a good indication of the number of shifts that are statically avoided (Table III).

Method calls. For method calls, one can compare thunks with the technique based on shifts: if one assumes that all table entries are equiprobable, the proportion of null shifts in method calls is exactly the ratio T_{sst}/T_θ. 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. Regarding upcasts, ESO_1 is the only and poor improvement upon SMI.

Attribute accesses. Either with subobject-based MI implementations or with accessor simulation and double compilation, there are the same two kinds of attribute accesses. The direct one, without table access, occurs when $T_p = \tau_s$ in SMI (see Section 3.2.3), or in the efficient version of the double compilation. Otherwise, a shift must be made, with a table access. It would be interesting to compare the rate of efficient accesses in the two techniques.

An exact comparison would need the complete code of all benchmarks, but some assumptions on access distribution allow to compute an approximate. A plausible assumption is a uniform distribution for all `self` attributes in all method definitions, i.e. the number of accesses to an attribute of `self` in a method definitions of class C is kA_C, where k is some unknown coefficient, and a_D is the number of

Table XX. Proportion of efficient attribute accesses, according to two different assumptions of access distribution, uniform on all attributes (left), or two times greater for introduced attributes than for inherited ones.

name	accesses	DVI		double		ESO=ESO$_1$		SMI	
Java1.6	539159	100.0	100.0	91.2	93.7	38.8	55.8	38.5	55.6
Geode	177562	92.6	94.4	88.4	91.2	32.8	49.1	32.0	48.5
Unidraw	27361	100.0	100.0	99.9	99.9	39.9	55.8	35.8	52.7
Lov-obj-ed	67796	96.5	97.7	97.9	98.6	52.2	68.5	51.8	68.2
SmartEiffel	60808	100.0	100.0	99.7	99.8	71.0	83.0	70.8	82.9
PRMcl	22524	92.1	94.9	96.9	98.0	55.3	71.0	54.3	70.4
Total	895210	98.1	98.6	92.2	94.4	41.3	58.3	40.7	57.9

attributes introduced in D. Therefore, the total number of attributes accesses is $\sum_C A_C m_C^{\texttt{def}}$. All accesses are efficient in all invariant implementations (SST and attribute coloring) and in NVI. In the case of SMI, only a_C accesses are efficient in each class C. In the case of double compilation, the number of efficient accesses is given by restricting the sum to non-*double classes*. In the first merging case of ESO, when C is empty, a_C is null and should be replaced by the number of attributes introduced in the direct superclass. In the DVI case, a_C should be replaced by the number of attributes in the non-virtual part of the object, i.e. attributes introduced in classes D such that $C \preceq^*_{\text{nv}} D$, where $\preceq^*_{\text{nv}}$ is the transitive and reflexive closure of $\prec_{\text{nv}}$. On the whole, efficient accesses are given by the following formulas

$$Z_\theta = k \begin{cases} \sum_C A_C m_C^{\texttt{def}} & \text{with } A_C = \sum_{C \preceq D} a_D & \theta = SST, NVI, COL \\ \sum_C a_C m_C^{\texttt{def}} & & \theta = SMI \\ \sum_C A'_C m_C^{\texttt{def}} & \text{with } A'_C = \sum_{C \preceq^*_{\text{nv}} D} a_D & \theta = DVI \\ \sum_{C \notin X_d} A_C m_C^{\texttt{def}} & & \text{double compilation} \end{cases}$$

For a given technique θ, the ratio is given by Z_θ / Z_{sst}. In all cases, the unknown coefficient k vanishes in the ratio.

This uniform access assumption is more consistent with SMALLTALK encapsulation, than with JAVA and C++ `private` keyword. Distinguishing newly introduced attributes is a way to take `private` into account. Hence, a more realistic assumption considers that attributes newly introduced are accessed two times more than inherited ones. With this new assumption, the ratio is given by $(Z_\theta + Z_{smi})/(Z_{sst} + Z_{smi})$ as, in SMI, efficient attributes are exactly private attributes.

Table XX presents the percentage of efficient accesses in the various implementations, according to those two assumptions[44]. NVI is exactly 100 % (not in Table) and DVI is always very close to 100 %. SMI stands between 30 % and 70 %, with the first assumption, between 48 and 82 in the second one. With a uniform distribution, most attributes are inefficient, whereas private attributes improves greatly this rate. Once again, ESO is a poor improvement upon SMI, because only the first kind of empty subobjects provides efficient attribute accesses. Double compilation appears as an interesting middle point: it is almost as good as DVI, and far better than SMI or ESO. Hence, the double compilation technique would be a true improvement.

[44]Due to some cases of introduction overloading, DVI is slightly overestimated.

B. PSEUDO-CODE SURVEY

The processor specifications are mostly the same as processor P95 in Driesen [1995; 1999]: particularly, 2 `load` instructions cannot be executed in parallel, but need a one cycle delay and the maximum number of integer instructions per cycle is 2. Actually, among the following sequences, only one code sequence gains when scheduled with 3 parallel threads. The cycle measure of each example is given as a function of L and B. When the measure is not straightforward, i.e. when some parallelism is involved, a diagram gives a possible schedule.

The following table gives the instruction set of the assembly language [Driesen 2001, p. 193] :

`R1`	a register (any argument without `#`)
`#immediate`	an immediate value (prefix `#`)
`load [R1+#imm], R2`	load the word in memory location `R1+#imm` to register `R2`
`store R1,[R2+#imm]`	store the word in register `R1` into memory location `R2+#imm`
`add R1, R2, R3`	add register `R1` to `R2`. Result is put in `R3`
`and R1, R2, R3`	bit-wise `and` on register `R1` and `R2`. Result is put in `R3`
`call R1`	jump to address in `R1` (can also be immediate), save return address
`comp R1, R2`	compare value in register `R1` with `R2` (`R2` can be immediate)
`bne #imm`	if last compare is not equal, jump to `#imm` (`beq`, `blt`, `bgt` are analogues)
`jump R1`	jump to address in `R1` (can also be immediate)

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. Moreover, `Tableoffset` will be likely 0 in most implementations, unless some specific hardware opportunity, as in AIX (see Section 7.7).

Attribute access:

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

`AttributeOffset` depends on the attribute.

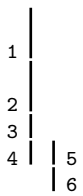
Type check using coloring:

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

The italicized line 2 adds the extra indirection which may be needed when no non-ambiguous coding for class identifers is possible. `TargetColor` and `TargetId` depend on the target class. `Fail` is the address of the code which either signals an exception or executes what must be done: it may be shared by several, or even all, type checks, or proper to each one.

Type check using Schubert's numbering:

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



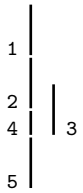
$2L + 3$

`N1Offset` is a constant, whereas `n1` and `n2` depend on the target class.

B.2 Multiple inheritance

Method call:

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



$2L + B + 1$

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

Thunks:

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

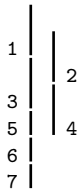
`Delta` and `method` depend on both the method and the receiver's static type. When method table points to the thunk, the code is that of single subtyping, whereas, when the thunk is inlined in the method table, it changes to:

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

$L + B + 1$

Equality test, with unrelated types:

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



$2L + 3$

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

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

$2L + 2$

and `castOffset` depends on both types.

Upcast:

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

$2L + 1$

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

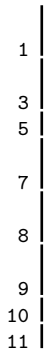
Accesses to attributes

```
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). Italicized lines are required only when $\tau_s \neq T_p$.

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

```
load [objecta + #tableOffset], tablea
load [objectb + #tableOffset], tableb
load [tablea + #castaOffset], deltaa
load [tableb + #castbOffset], deltab
add objecta, deltaa, objecta
add objectb, deltab, objectb
load [objecta + #attributeaOffset], attribute
load [attribute + #tableOffset], table
load [table + #castOffset], delta
add attribute, delta, attribute
store attribute, [objectb + #attributebOffset]
```



5L + 3

Downcast. We present here an implementation of a variant of *linear probing* [Knuth 1973; Vitter and Flajolet 1990]. The hashtable of a class C consists of an integer H_C and an array of a given number, close to H_C, of entry pairs, alternatively class identifiers and Δs. $H_C > n_C$, where n_C is the number of superclasses of C. Class identifiers are integers from 1 to N, where N is the number of classes, in such a way that $id_C > id_D$ when $C \prec D$. The hash value of a class identifier id_D is the bit-wise `and`, $h_C(id_D) = id_D$ `and` H_C, which is prefered to integer division as the latter is a polycycle operation. Classes are inserted in the array in the same way as for linear probing—at the first empty place starting from the hashed value—but the array is considered as infinite instead of circular, as in usual linear probing. Finally, the array is truncated one entry after the last occupied position, if this position is greater than H_C. Therefore, the size is between H_C and $H_C + n_C$. The average number of probes is close to 1 when n_C/H_C is small enough: a good value would be $H_C = 2n_C$. The inlined code might be the following:

```
load [object + #tableOffset], table
load [table + #hashingOffset], h
add table, #htOffset, table
and #id_D, h, h
mul h, 4, h
add table, h, table
loop:
load table, id
comp #id_D, id
beq #succeed
comp #empty, id
beq #fail
add table, #4, table
jump #loop
succeed:
load [table + #2], delta
add object, delta, object
```

1		
2		
4	3	$4L+6$
5		
6		
7		
8		
9		
14		
15		

`HtOffset` is the offset of the hashtable in the method table. `HashingOffset` is the offset of H_C and does not depend on C. The class identifier id_D is hashed then searched in the hashtable, starting from the hashed value, until a value `empty` is found. Line 5 serves to word alignment and might be saved on. The diagram corresponds to a success or failure at the first try. The number of cycles is $3L + 2 + k(L+3)$ (resp. $2L+1+k(L+3)$) when the cast succeeds (resp. fails): k is the number of probes, greater than 1 and lesser than n_C (success) or $n_C + 1$ (failure).

If, for all C, H_C is the least integer such that all n_C superclasses have distinct hash values—this is a *perfect hashing* function [Sprugnoli 1977; Mehlhorn and Tsakalidis 1990]—the table size is exactly $H_C + 1$, with $H_C \leq 2id_C$, and the code may be simplified into the following, with the same cycle number in the failure case:

```
load [object + #tableOffset], table
load [table + #hashingOffset], h
add table, #htOffset, table
and #id_D, h, h
mul h, 4, h
add table, h, table
load table, id
comp #id_D, id
bne #fail
load [table + #2], delta
add object, delta, object
```

1		
2		
4	3	$4L+6$
5		
6		
7		
8		
9		
10		
11		

The too last `load`, which are presumed to be 16-bits, might be replaced by a unique 32-bits `load` followed by some register operations. This remains two times the instruction number and the cycle number of coloring, but it is likely very closed to the optimal for standard MI implementations.

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 + #tableOffset], table1
load [table1 + #table2Offset], table2
load [table1 + #delta1Offset], delta1
load [table2 + #delta2Offset], delta2
add object, delta1, object
load [table2 + #selectorOffset], method
add object, delta2, object
call method
```

$3L + B + 1$

Upcasts with i-VBPTRs:

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

L

or with e-VBPTRs:

```
load [object + #castOffset], object
add object, #delta, object
```

$L + 1$

and attribute access with VBPTRS (either i-VBPTRs or e-VBPTRs):

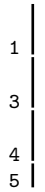
```
load [object + #castOffset], object
load [object + #attributeOffset], attribute
```

$2L$

In both cases, e-VBPTRs are considered in the case of ideal devirtualization: `delta` is the relative position of the target class in a non-virtual part and `attributeOffset` includes this `delta`.

Assignment $a.x := b.y$ shows how much i-VBPTRs improve efficiency:

```
load [objecta + #castaOffset], objecta
load [objectb + #castbOffset], objectb
load [objecta + #attributeaOffset], attribute
load [attribute + #castOffset], attribute
store attribute, [objectb + #attributebOffset]
```



$3L + 1$

Simulating accessors to attributes is exactly as with standard MI:

```
load [object + #tableOffset], table
load [table + #castOffset], place
add object, place, place
load [place + #attributeOffset], attribute
```

$3L + 1$

Finally, inline cache of method tables reduces method calls to:

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

$L + B$

whereas upcasts are exactly as with VBPTRs, but `tableObject` replaces `object`:

```
load [tableObject + #castOffset], tableObject
```

L

and attributes are accessed through accessors.

B.4 Global techniques

B.4.1 *Type prediction.* When n types are predicted, with identifiers t_1, ..., t_n, the code for method call in the case of a `if` sequence will be:

```
load [object + #typeOffset], typeId
comp #t_1, typeId
bne #next_2
call #method_1
jump #next
#next_2
comp #t_2, typeId
call #method_2
...
#next
```

Equality tests may be replaced by inequality when several contiguous types dispatch to the same method. An alternative to this inlined decision tree is a small dispatching procedure, which may be generated as well at compile-time as at link-time:

```
call #call_9257
...
call_9257
load [object + #typeOffset], typeId
comp #t_1, typeId
beq #method_1
comp #t_2, typeId
beq #method_2
...
```

The case of attributes is quite similar...

```
load [object + #typeOffset], typeId
comp #t_1, typeId
bne #next_2
load [object + attributeOffset_1], attribute
jump #next
#next_2
comp #t_2, typeId
load [object + attributeOffset_2], attribute
...
#next
```

... as type checking and downcast:

```
load [object + #typeOffset], typeId
comp #t_1, typeId
beq #ok
comp #t_2, typeId
beq #ok
...
comp #t_n, typeId
bne #fail
#ok
```

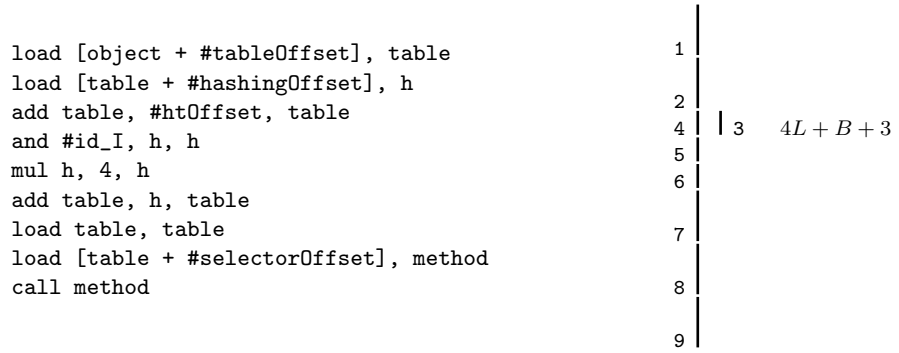
B.4.2 *Coloring.* Coloring does not need a specific pseudo-code: it uses exactly that of single subtyping, except for attributes when the code for simulating accessors is used instead.

B.5 Single inheritance multiple subtyping

Method call in SST variant, with an injective numbering of interfaces, for interface-typed receiver (`invokeinterface`):

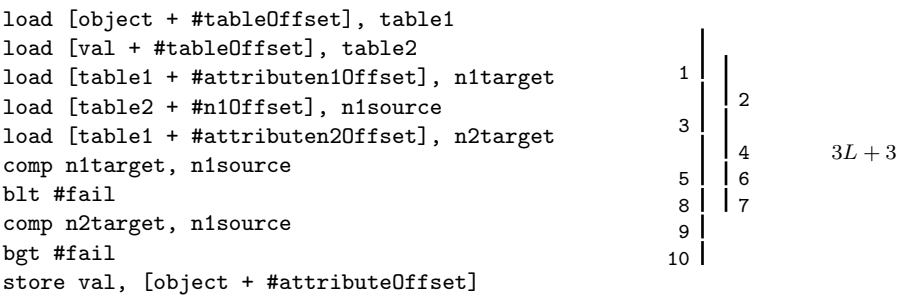
```
load [object + #tableOffset], table
load [table + #interfaceDelta], delta
add table, delta, table                                   4L + B + 1
load [table + #interfaceOffset], table
load [table + #selectorOffset], method
call method
```

The code is the same for variants 2 and 4. `InterfaceDelta` gives the relative position of the table of interfaces, i.e. the position of the first non-empty entry. `InterfaceOffset` is the offset of the pointer to the table of a given interface (i.e. the interface which introduces the called method). When a hashtable is used, a code similar to the one for downcast must be used. Perfect hashing yields the following:



B.6 Complements

B.6.1 *Covariant overriding.* In single subtyping, for attributes, with Shubert's numbering:



The reader is free to combine this sequence with the aforementioned $a.x := b.y$ assignment!

Covariant return type in multiple inheritance, in a thunk, is simpler:

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

but the thunk is here an actual function, which calls the method, instead of jumping, and must return.

B.6.2 *Genericity.* Method call to formal type with offset conversion:

```
load [generic + #tableOffset], table1        1
load [object + #tableOffset], table2         2
load [table1 + #selectorOffset], offset      3
add table2, offset, method                   4
load method, method                          5
call method                                  6
```

$3L + B + 1$

`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        1
load [object + #tableOffset], table2         2
load [table1 + #castOffset], delta           3
add table2, delta, table2                    4
load table2, place                           5
add object, place, place                     6
load [place + #attrOffset], attribute        7
```

$4L + 2$

B.6.3 *Polymorphic primitive type.* The main point is equality $x = y$, when both x and y have a universal type:

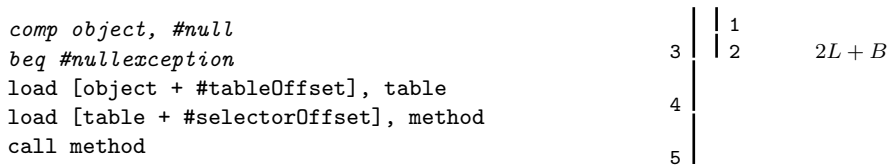
```
comp x, y                                    1
beq #true                                    2
load [x + #tableOffset], tablex              3
load [y + #tableOffset], tabley              4
load [tablex + #stampOffset], stamp          5
comp tablex, tabley                          6
bne #false                                   7
comp stamp, 0                                8
beq #false                                   9
load [x + #valOffset], valx                  10
load [y + #valOffset], valy                  11
comp valx, valy                              12
bne #false                                   13
true:
```

$3L + 6$

`StampOffset` is a fixed position, in all method tables, which indicates whether the type is primitive (0) or not. Of course more parallelism is possible, but this remains

a long inlined sequence regarding such a basic mechanism.

B.6.4 *Null checks.* Including `null` check in SST method call gives the following:



It increases code length, but does not change cycles, as long as a `load` at address `null` does not raise an exception. `Nullexception` is shared by all `null` checks.