Towards a full multiple-inheritance virtual machine

Roland Ducournau
LIRMM – Université Montpellier 2, France
ducournau@lirmm.fr

Floréal Morandat
LIRMM – Université Montpellier 2, France
morandat@lirmm.fr

ABSTRACT

Late binding and subtyping create run-time overhead for object-oriented languages, especially in the context of both *multiple inheritance* and *dynamic loading*, for instance for JAVA interfaces. It is, however, generally agreed that the efficiency of JAVA and .NET systems comes from the fact that, in these languages, classes are in single inheritance. In this paper, we present the abstract architecture of a virtual machine for unrestricted multiple-inheritance, which should provide the same runtime efficiency as JAVA and .NET.

Categories and Subject Descriptors

D.3.2 [**Programming languages**]: Language classifications—*object-oriented languages,* C#, JAVA, SCALA, PRM; D.3.3 [**Programming languages**]: Language Constructs and Features—*classes and objects, inheritance*; D.3.4 [**Programming languages**]: Processors—*compilers, run-time environments*; E.2 [**Data**]: Data Storage Representations—*object representations*

General Terms

Experimentation, Languages, Measurement, Performance

Keywords

adaptive compiler, closed-world assumption, dynamic loading, late binding, method tables, multiple inheritance, open-world assumption, perfect hashing, subtype test, virtual machine

1. INTRODUCTION

Multiple inheritance is generally considered as a cause of multiple difficulties from the standpoints of both semantics and runtime efficiency. Multiple subtyping, i.e. JAVA-like interfaces, was mostly proposed as a response to these difficulties. In this paper, we do not address the semantic aspect, which is discussed in [Ducournau and Privat, 2008], and we focus on the efficiency aspect.

MASPEGHI/ICOOOLPS '10 Maribor, Slovenia

Multiple inheritance impacts on runtime efficiency through three mechanisms that are typical of object-oriented programming, namely method invocation, attribute access and subtype testing. All three mechanisms depend on the dynamic type of the receiver. Moreover, in a dynamic-loading setting which entails the *open world assumption*, multiple inheritance yields compile-time uncertainty about the position of the accessed method, attribute or supertype.

In this paper, we present the abstract architecture of a virtual machine designed for multiple inheritance which should provide the same runtime efficiency and scalability as JAVA and .NET platforms. This proposal is assessed by simulating class loadings on a class hierarchy based on a real program, the PRM compiler [Ducournau et al., 2009]. As an example, the proposed approach could be substituted to the current JAVA/.NET implementation of the SCALA language [Odersky et al., 2008], and the language specifications could thus drop the class/trait distinction which seems to be inessential.

The structure of the paper is as follows. Section 2 presents the point of object-oriented implementation, and describes the two techniques that underly our proposal, namely single-subtyping implementation and perfect class hashing [Ducournau, 2008]. The next section describes the proposed virtual-machine architecture which is an adaptation to dynamic loading of the double compilation proposed for attribute access in [Myers, 1995]. Section 4 then presents a simulation on the class hierarchy of the PRM compiler. Conclusion and prospects end the paper.

2. OBJECT-ORIENTED IMPLEMENTATION

Implementation techniques are concerned with object representation, that is the object layout and the associated data structures that support method invocation, attribute access and subtype testing.

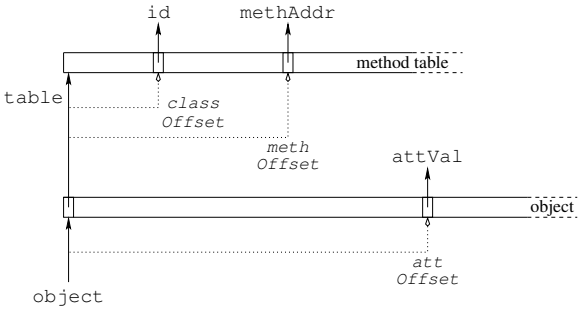
2.1 Single-subtyping implementation

In separate compilation of statically typed languages, late binding is generally implemented with method tables, which reduce method invocations to calls to pointers to functions through a small fixed number (usually 2) of extra indirections. An object is laid out as an attribute table, with a pointer at the method table. With single inheritance and single subtyping, when classes are the only types, the class hierarchy is a tree and the tables implementing a class are straightforward extensions of those of its single direct superclass (Figure 1). The resulting implementation respects two essential *invariants*: (i) a *reference* to an object does not depend on the static type of the reference; (ii) the *position*

```
// attribute access
load [object + #attOffset], attVal

// method invocation
load [object + #tableOffset], table
load [table + #methOffset], methAddr
call methAddr

// subtype test
load [object + #tableOffset], table
load [table + #offsetC], idC
comp idC, #targetId
bne #fail
// succeed
```



Code sequences for the 3 basic mechanisms and the corresponding diagram of object layout and method table. Pointers and pointed values are in Roman type with solid lines, and offsets are italicized with dotted lines.

Figure 1: Single-subtyping implementation.

of attributes and methods in the table does not depend on the dynamic type of the object. Therefore, all accesses to objects are straightforward. This accounts for method invocation and attribute access under the OWA. The efficacy of this implementation is due to both static typing and single inheritance. Otherwise, the same kind of complication may occur when the same property name is at different places in unrelated classes.

The technique proposed by Cohen [1991] for subtype testing also works under the OWA. It involves assigning a unique ID to each class, together with an invariant position in the method table, in such a way that an object x is an instance of the class C if and only if the method table of x contains the class ID of C, at a position uniquely determined by C. Readers are referred to [Ducournau, 2008] for implementation details that avoid bound checks and indirections.

In this implementation, the total table size is roughly linear in the cardinality of the specialization relationship, 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 table 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.

There are almost no SST languages, but this implementation is that of Java and .Net for class representation and class-typed invocations.

2.2 Perfect hashing

In [Ducournau, 2008], we proposed a new technique based on perfect hashing for subtype testing in a dynamic loading setting. The problem can be formalized as follows. Let $(X, \preceq)$ be a partial order that represents a class hierarchy, namely X is a set of classes and $\preceq$ the specialization relationship that supports inheritance. The subtype test amounts to checking at run-time that a class c is a superclass of a class d, i.e. $d \preceq c$. Usually d is the dynamic type of some object and the programmer or compiler wants to check that this object is actually an instance of c. Classes are loaded at run-time in some total order that must be a *linear extension* (aka *topological sorting*) of $(X, \preceq)$—that is, when $d \prec c$, c must be loaded before d.

The *perfect hashing* principle is as follows. When a class c is loaded, a unique identifier id_c is associated with it and the set $I_c = \{id_d \mid c \preceq d\}$ of the identifiers of all its superclasses is known. If needed, yet unloaded superclasses are recursively loaded. Hence, $c \preceq d$ iff $id_d \in I_c$. This set I_c is immutable, hence it can be hashed with some *perfect hashing function* h_c, i.e. a hashing function that is injective on I_c [Czech et al., 1997]. The previous condition becomes $c \preceq d$ iff $ht_c[h_c(id_d)] = id_d$, whereby ht_c denotes the hashtable of c. The technique is incremental since all hashtables are immutable and the computation of ht_c depends only on I_c. The perfect hashing functions h_c are such that $h_c(x) = hash(x, H_c)$, whereby the hashtable size H_c is defined as the least integer such that h_c is injective on I_c. Two *hash* functions were considered, namely *modulus* and bit-wise `and`[1]. A recent study led us to consider that the latter must be preferred [Ducournau and Morandat, 2010].

In a static typing setting, the technique can also be applied to method invocation and we did propose, in the aforementioned article, an application to Java interfaces. For this, the hashtable associates, with each implemented interface, the offset of the group of methods that are introduced by the interface. Of course, this easily generalizes to method invocation in full multiple inheritance. Figure 2 recalls the precise implementation in this context. The method table is bidirectional. Positive offsets involve the method table itself, organized as with single inheritance, whereby methods are grouped by introduction classes and these groups are arbitrarily ordered. Negative offsets consist of the hashtable, which contains, for each superclass, the offset of the group of methods it introduces. The object header points at its method table by the `table` pointer. `#hashingOffset` is the position of the hash parameter (`h`) and `#htOffset` is the beginning of the hashtable. At a position `hv` in the hashtable, a two-fold entry is depicted that contains both the superclass ID, that must be compared to the target class ID, and the offset `iOffset` of the group of methods introduced by the superclass that introduces the considered method. The table contains, at the position `#methodOffset` determined by the considered method in the method group, the address of the function that must be invoked.

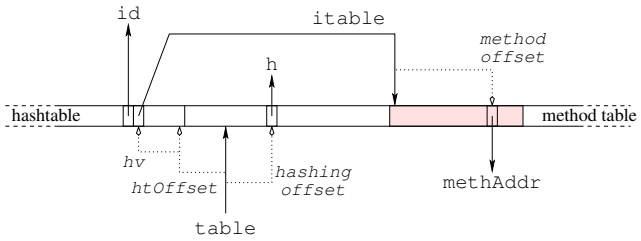
The efficiency of PH is rather good. From the time standpoint, experiments in Prm show that the PH overhead is real but low [Ducournau et al., 2009]. From the memory occupation standpoint, exhaustive random simulations on large-

[1] With `and`, the exact function maps x to $\texttt{and}(x, H_c - 1)$.

```
// preamble
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #classId, h, hv
sub table, hv, htable

// method invocation
load [htable +#htOffset], itable
load [itable +#methOffset], methAddr
call method

// subtype testing
load [htable +#htOffset-fieldLen], id
comp #classId, id
bne #fail
// succeed
```



The preamble is common to both mechanisms. The grey rectangle denotes the group of methods introduced by the considered class.

Figure 2: Perfect hashing

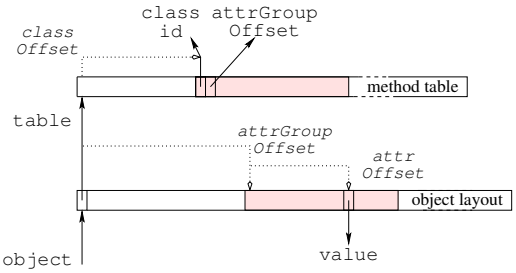
scale hierarchies show that a variant called *perfect class numbering* has a quasi-linear behaviour [Ducournau and Morandat, 2010]. To our knowledge, PH is the only constant-time technique that allows for both multiple inheritance and dynamic loading at reasonable spatial cost and applies to both method invocation and subtype testing.

2.3 Accessor simulation (AS)

An accessor is a method that either reads or writes an attribute. True accessors require a method call for each access, which can be inefficient. However, a class can simulate accessors by replacing the method address in the method table with the attribute offset. This approach is called *field dispatching* by [Zibin and Gil, 2003]. Another improvement is to group attributes together in the method table when they are introduced by the same class. Then one can substitute, for their different offsets, the single relative position of the attribute group, stored in the method table at an invariant position, i.e. at the class offset with Cohen's test (Fig. 3) [Myers, 1995]. With PH, the attribute-group offset is associated with the class ID and method-group offset in the hashtable, yielding 3-fold table entries.

Accessor simulation is a generic approach to attribute access which works with any method invocation technique, and makes it work in a full multiple-inheritance setting; only grouping can be conditioned by static typing, since attributes must be partitioned by the classes which introduce them. Among the various implementation techniques, some apply only to method invocation and subtype testing, e.g. perfect hashing. Hence, these techniques can be used for JAVA interface implementation. Accessor simulation is a way of applying them to full multiple inheritance.

```
// attribute access
load [object + #tableOffset], table
load [table + #classOffset+fieldLen], attrGroupOffset
add object, attrGroupOffset, attgr,
load [attgr + #attrOffset], value
```



The diagram depicts the precise object representation with accessor simulation coupled with Cohen's test, to be compared with Fig. 1. The offset of the group of attributes introduced by a class (`attrGroupOffset`) is associated with its class ID in the method table and the position of an attribute is now determined by an invariant offset (`attrOffset`) w.r.t. this attribute group.

Figure 3: Accessor simulation with Cohen's test

3. VIRTUAL MACHINE SPECIFICATIONS

In our abstract view, a virtual machine is a runtime system which performs the following tasks: (i) it loads code units that generally represent yet unloaded classes; (ii) it computes the object representation for the newly loaded classes; (iii) it compiles or recompiles pieces of code (generally methods); and, finally, (iv) it runs the compiled code which can, cyclically, trigger class loadings. Furthermore, compilation is both lazy and adaptive. Laziness is out of the scope of this paper, but adaptiveness is a key feature. Each piece of code is compiled under a provisory *closed-world assumption* (CWA) which allows for efficient code sequences for the invocation of object-oriented mechanisms. In contrast, we only consider object representations that do not need any recomputation, hence which must be computed under the *open world assumption* (OWA).

Overall, the virtual machine specifications consist of two parts:

- an object representation that supports two kinds of alternative implementations for mechanism invocation: (i) a general background implementation is required to work in any situation and to present very good worst-case efficiency; (ii) one or more optimized implementations represent shortcuts with excellent efficiency, but they do not work everywhere and everytime;
- a protocol for selecting the appropriate implementation and propagating possible recompilations, when the compiler must switch from an optimized implementation to a less optimized one.

3.1 Object representation

The general idea is to use perfect hashing as the underlying object representation, in such a way that all mechanisms could be invoked through PH. PH is thus coupled with accessor simulation for attribute access. However, the single-subtyping code can be used when the current situation verifies the position invariant, i.e. when the receiver's

static type has the same position in all its subclasses.

With PH, attributes and methods are grouped together in object layouts and method tables according to their introduction class. When a class is loaded, its attribute and method groups are determined according to its superclasses and ordered according to an algorithm which will be detailed hereafter. The resulting order is immutable, i.e. it will not be changed by possible further recompilations. Perfect class hashing is also computed and the complete method table is then allocated and filled. The method-table structure is thus immutable, and only the entries corresponding to method addresses can be further changed.

In contrast, computing the order associated with a newly loaded class, say C, may assign to some superclass of C, say B, a position that differs from the single previous position of B. This triggers some recompilations in B, when the SST code that was used in B for some invocations is no longer sound.

In the following, a `self`-*invocation* is a method invocation or an attribute access either on `self`, or on a receiver which is typed by the current class (or a subtype of the current class, but the considered method or attribute must have been introduced by the current class). Only `self`-invocations can be optimized by this adaptive approach.

When a method definition is compiled, all mechanism invocations are handled in the following ways:

1. subtype testing is implemented with PH;
2. all methods introduced by the hierarchy root (i.e. `Any` in EIFFEL or PRM, or `Object` in JAVA or C#) are invoked through the SST implementation; the root has indeed an immutable position;
3. all `self`-invocations of methods or attributes are invoked through the SST implementation when the considered method or attribute group has been implemented, so far, at the same position;
4. PH is used for all other methods and attributes.

A particular method definition can use both SST for method invocation and PH for attribute access, or conversely.

Recompiling a class is required when the positions of its attribute or method groups have been changed in some subclass. It requires to change case-3 invocation sites into case-4. Only the methods that contain still optimized `self`-invocations are concerned, and a method can be recompiled at most twice, since the recompilation must be done only the first time the group is moved. However, the approach does not distinguish between `self`-invocations of methods/attributes introduced by the current class (which have actually been moved) and `self`-invocations of methods/attributes introduced in superclasses (which may have not been moved).

3.2 Protocols and algorithms

Superclass ordering. The efficiency of the approach depends on the superclass ordering, which can be understood as a *linearization*, in the sense of *multiple-inheritance linearizations* [Ducournau and Privat, 2008]. Whereas multiple-inheritance linearizations must be linear extensions, the *prefix condition* is here essential. Given a class c, with the set $supc(c)$ of its direct superclasses, then there must be some $c' \in supc(c)$ such that the order associated with c' is a prefix of the order associated with c. Therefore, the superclasses of c' (including c') will not be moved and do not require any recompilation. In contrast, the positions of superclasses of c that are not superclasses of c' are irrelevant.

In practice, c' is selected in $supc(c)$ as the class which maximizes the number of `self`-invocations that are presumed to be at a constant position, i.e. the number $f(c') = \sum_{c' \preceq d} w_d * k_d$, whereby k_d is the number of `self`-invocations in the method definitions of d, and $w_d = 1$ if d still has a single position, 0 otherwise.

There is no need for the superclass order to be the same for methods and attributes, since the hashtable is the only common point between the two kinds. These heuristics can also be improved by taking classes that introduce no attributes or methods into account.

Recompilation protocol. Once a class d has been loaded, it maintains the following meta-information about its state:

- a_d and m_d that represent the numbers of `self`-invocations of attributes and methods in the method definitions of d, and stand for k_d in the above definition of $f(c')$;
- ap_d and mp_d that represent, respectively, the single position of the attribute or method group, or a distinguished value for multiple positions;
- A_d, M_d and AM_d that represent sets of methods defined in d and candidates to possible recompilations; methods in A_d (resp. M_d) use the SST implementation for at least one attribute access (resp. method invocation), and methods in AM_d use SST for both.

When loading a class c, a direct superclass c' is first selected as previously described. Each superclass d of c that is not superclass of c' is then candidate to recompilation, and the methods in A_d (resp. M_d) and AM_d are recompiled if the attribute (resp. method) group has been moved. After recompilation, the A_d, M_d or AM_d sets of recompiled methods are removed. If recompilation concerns only method invocation (resp. attribute access), the AM_d set is then added to A_d (resp. M_d). The algorithm for deciding which methods must be recompiled is thus straightforward. As in all adaptive compilers, the recompilation itself can be eager or lazy.

3.3 Example

Consider a class hierarchy made of a diamond with 4 classes A (the diamond top), B, C and D (the diamond bottom) that are loaded in this order. When A, B and C are successively loaded, the hierarchy is a tree, hence each class has a single position and all `self`-invocations can use the SST implementation. When loading D, multiple inheritance appears and a direct superclass of D must be selected in $\{B, C\}$ by computing $f(B)$ and $f(C)$. Suppose that $f(B) > f(C)$. According to the prefix condition, the superclass order of D must be (A, B, C, D), and the method definitions in C that contain `self`-invocations must be recompiled.

4. EVALUATION

The benchmark used in these tests is an abstract description of the class hierarchy of the PRM compiler used in the testbed presented in [Ducournau et al., 2009].

(a) Method numbers

introduced			defined			inherited		
total	avg	max	total	avg	max	total	avg	max
2671	4.9	126	4407	8.1	126	43803	80.1	236

(b) Attribute numbers

introduced			inherited		
total	avg	max	total	avg	max
623	1.1	29	2588	4.7	31

(c) Invocation numbers

A-S	A-O	A-*	M-R	M-S	M-ST	M-O	M-*
4487	26	4513	1599	2778	252	12747	17376

(d) Number of methods according to their *self*-invocations

A	*AM*	*M*	other	total
1640	659	769	1339	4407

(a) Method numbers present total, per-class average and maximum, and distinguish between whether methods are introduced, defined or inherited in the class.
(b) The same applies to attributes, apart from the differences between introduction and definition.
(c) Invocation numbers retain the distinction between the different kinds made in the benchmark, i.e. root/self-invocations.
(d) The *A*, *M* and *AM* sets represent methods that contain self-invocations on attributes, methods or both.

Table 1: Compile-Time statistics

Benchmark. This description follows the same principle as benchmarks commonly used in the object-oriented implementation community, especially in our previous experiments [Ducournau, 2008, Ducournau and Morandat, 2010], but is more specific as it includes invocation counts. Each class is described by the following elements: (i) class name, (ii) superclass names, (iii) names of introduced attributes, (iv) names of defined methods followed by the number of invocations in the method code. Invocation numbers distinguish between method (M) and attribute (A) invocation. Method invocations are distinguished according to whether the method is introduced by the hierarchy root (R), the receiver is `self` (S) or typed by the current class (ST), or otherwise (O). The same is applied to attribute invocations, except they distinguish only between S and O cases. In our benchmark, there is indeed no A-ST example, and there are generally no A-R example in any language. M-R represent cases that are always optimized, while A-S, M-S and M-ST represent `self`-invocations that can yield recompilations, and A-O and M-O are always unoptimized. The cardinality of the sets *A*, *M* and *AM* is thus computed for each class.

Compile-Time statistics. Table 1 presents various statistics about (i) the number of attributes and methods, (ii) the number of invocation sites per invocation kind (A or M; R, S, ST or O), and (iii) the total sizes of the *A*, *M* and *AM* sets, i.e. the number of method definitions according to the kinds of invocations they contain.

Random load-time statistics. Whereas these first statistics are static, i.e. they could be done at compile-time on the whole hierarchy (under the CWA), the next statistics are dynamic and they should be done at load-time. Therefore,

(a) Numbers of class and method recompilations

class rcp			meth rcp			rcp load		
59	60.8	64	229	239.8	247	28	33.1	37

(b) Variable-position `self`-invocations

variable A-S			total	variable M-S			total
77	77.5	78	4487	261	268.8	278	3030

Each random datum is depicted by its minimum, average and maximum values over all tested class-loading orders.

Table 2: Load-Time statistics

class loading has been simulated, and class-loading orders are generated at random as in [Ducournau and Morandat, 2010]. For each class loading, the number of recompiled classes, recompiled methods, variable-position methods and attributes, and variable-position invocation sites are computed, and Table 2 depicts the statistics on these numbers (minimum, average and maximum values).

Discussion. The most interesting numbers are the invocation numbers (Table 0(c)). As a consequence of a strict encapsulation discipline, almost all attribute accesses are `self`-invocations, hence can be optimized. In contrast, only 30% of method invocations are or can be optimized. The random simulation (Table 1(b)) shows that only a few percent of `self`-invocations cannot be optimized. Therefore the resulting program would be as efficient as Java or .Net systems for attribute access, since more than 98% of attribute access sites would be optimized. In contrast, only 25% of method call sites would be optimized, and this would looks like a Java program making an heavy use of `invokeinterface`. By extrapolation from the measures presented in [Ducournau et al., 2009], one can conclude that the proposed approach should have a runtime efficiency slightly higher than that of PH coupled with attribute coloring, hence only a few percent slower than the reference coloring implementation.

The load-time recompilation cost is also interesting. At most 37 class loadings yield some recompilation on a total of 547 ('rcp load' column in Table 1(a)), and these recompilations concern at most 64 classes ('class rcp' column) and 247 methods ('meth rcp' column), on 3068 methods containing `self`-invocations. Recompilation should thus represent less than 5% of the compilation cost. This is an upper bound, since compilation would likely be lazy, and recompilation of a method might be triggered before its first compilation. Moreover, classes are not loaded one at a time, and several related classes are often loaded together. It will allow us to improve the used heuristics with more global optimizations which should also reduce the recompilation number.

5. CONCLUSION AND PROSPECTS

In this paper, we propose an abstract architecture of a virtual machine for full multiple-inheritance languages with dynamic loading. This proposal is based on perfect hashing for the object-representation side, and is derived from the double compilation proposed by Myers [1995]. However, Myers's double compilation was only concerned by attribute access and the choice was done at link-time under the CWA, whereas we extended the proposal to method invocation and applied it at load-time. A first simulation on a real program has been done, following the methodology developped

for assessing the PH efficiency. It shows that the resulting efficiency should be close to that of multiple-subtyping languages, from both run-time and load-time standpoints.

This simulation-based approach is a worthwhile alternative to runtime tests. Developping a runtime system, or a compiler, is hard work, hence it is far better to first select suitable techniques by abstract assessment, e.g. simulation. However, simulation is not only a way to spare time and pains. Indeed, with dynamic loading, the class-loading order is crucial, and techniques like PH and recompilations closely depend on it. Therefore, random simulation may well be the only way to evaluate their worst-case efficiency. In contrast, runtime tests often represent only best-case assessments.

Of course, the adaptive compiler of an object-oriented virtual machine must include many other optimizations, but they are independent of the multiple-inheritance feature. For instance, a key optimization amounts to restricting the PH use to the method-invocation sites that actually require it. Monomorphic call sites should thus be identified at compile-time, and provisionnally considered as static calls. This can be done in two ways: (i) the method call can be generated as a static call, and the compiled method is memorized as a possible recompilation for the time when the called method will become polymorphic; (ii) the method call can be generated as a static call to a *thunk* which statically jumps to the called method, and this thunk is memorized as a possible recompilation. This involves a tradeoff between load- and run-time efficiencies. In the former case, the increase in method recompilation number might be high. In the latter case, a thunk with a static jump should be more efficient than an inlined PH sequence, whereas a thunk adds extra overhead when it consists of a PH sequence. Hence, the number of thunks that eventually consist of a PH sequence must be estimated. Our next experimentation will be a simulation of both approaches, in order to provide an abstract assessment of their respective efficiencies.

References

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

Z. J. Czech, G. Havas, and B. S. Majewski. Perfect hashing. *Theor. Comput. Sci.*, 182(1-2):1–143, 1997.

R. Ducournau. Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.*, 30(6):1–56, 2008.

R. Ducournau and F. Morandat. Perfect class hashing and numbering for object-oriented implementation. Research Report LIRMM-10012, Université Montpellier 2, 2010.

R. Ducournau and J. Privat. Metamodeling semantics of multiple inheritance. Research Report LIRMM-08017, Université Montpellier 2, 2008.

R. Ducournau, F. Morandat, and J. Privat. Empirical assessment of object-oriented implementations with multiple inheritance and static typing. In Gary T. Leavens, editor, *Proc. OOPSLA'09*, SIGPLAN Not. 44(10), pages 41–60. ACM, 2009.

A. Myers. Bidirectional object layout for separate compilation. In *Proc. OOPSLA'95*, SIGPLAN Not. 30(10), pages 124–139. ACM, 1995.

Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, A comprehensive step-by-step guide.* Artima, 2008.

Y. Zibin and J. Gil. Two-dimensional bi-directional object layout. In L. Cardelli, editor, *Proc. ECOOP'2003*, LNCS 2743, pages 329–350. Springer, 2003.