A virtual machine for testing compilation/recompilation protocols in multiple inheritance

Julien Pagès

LIRMM – Université de Montpellier

julien.pages@lirmm.fr

Abstract. We present a virtual machine with dynamic loading for a full multiple inheritance language in static typing: Nit. The virtual machine has been designed to be a platform for testing various protocols of compilation/recompilation. Indeed, the runtime efficiency of virtual machines are based on dynamic optimizations, and these protocols, which are generally hidden, are the key factor to their efficiency.

Keywords: Object-oriented programming, virtual machine, dynamic loading, multiple-inheritance, perfect hashing, inlining, re-compilations, devirtualization, preexistence

1 Introduction

Virtual machines are now a spread kind of execution system since languages like Lisp, Smalltalk, Self, Java and C♯. These systems work in dynamic loading, in other words they are under the *Open World Assumption*. This means that classes are discovered at runtime, and their current set offers only a partial and temporary knowledge to the runtime system.

Because of dynamic loading, virtual machines are harder than static compilers to optimize. Nevertheless, these systems are sometimes very fast: the key factor to their performances relies in their dynamic optimization systems. Optimizations such as devirtualization [16] and inlining [8] are the two major optimizations for an object-oriented language and they can be performed in an aggressive way. But, these optimizations could lead to incorrect executions, and thus must be removed. For this reason, we will call the system that take decisions to optimize or de-optimize a *protocol of compilation/recompilation*.

The problems of multiple inheritance. To our knowledge, there is no virtual machine for a full multiple inheritance language in static typing. Our main objective is to develop a virtual machine for a language with these characteristics. We also want this virtual machine to be based on a just-in-time (JIT) compiler, and we have to specify for it a protocol of compilation/recompilation.

In this paper we will describe the first step in this project: basing us on an existing interpreter to develop a virtual machine and simulate the behavior of a JIT compiler.

Multiple inheritance has always be a difficult feature to implement but it has obvious advantages for programmers by increasing reusability and so, factorization of code. The specifications of the widely-used Java and C♯ languages offer only a limited form of multiple inheritance, with the so-called interfaces which are in multiple subtyping.

Indeed, in single-inheritance, a read/write access to an attribute is as efficient as a field access in C. As attribute accesses are very frequent in object-oriented programs, it is thus essential to keep this efficiency as long as possible. This is the reason why Java interfaces cannot declare any attribute. An attribute access is also a short sequence of code and then, it is mostly inlined which makes it difficult to de-optimize.

In the runtime systems of multiple-subtyping languages, the implementation of classes is as efficient as it can be. However, the implementation of interfaces has always been suspected to not be as efficient. The title of [2] is a good example of this suspicion. Another good example of that is the implementation of the subtyping test in Hotspot Java virtual machine [5]. The subtyping test against a class is very efficient since it is derived from Cohen's technique [6] which is in constant time. In contrast, a subtyping test against an interface is implemented by a linear search in an array. In practice, the performances are rather good thanks to cache mechanisms for the array's search. This implementation was designed to optimize the common cases (tests against classes) but not the less common (tests against interfaces).

Since implementing efficiently Java interfaces seems to be a challenge, it would seem that implementing multiple inheritance is unreachable, at least if one wants to obtain the efficiency of single inheritance, or even multiple subtyping. Dynamic loading makes the set of classes grow during execution. This makes it impossible to use global efficient techniques like coloring to implement multiple inheritance.

In our virtual machine we use an implementation technique based on perfect hashing proposed in [9,12]. This implementation is compatible with dynamic loading, it also has the key feature that it is compatible with that of single inheritance (eg classes in Java). Therefore, the sequences of code that are efficient with single inheritance can be reused for most classes, which are used only, or mainly, in single inheritance, and less efficient sequences of code, using perfect hashing, can be reserved to a few classes. We will detail, hereafter, how the decision is taken. We will couple these implementations with an abstract compilation protocol for a virtual machine whose principle has been proposed in [10].

A virtual machine for a full multiple inheritance language. In this paper, we will present a virtual machine currently in development for a full multiple-inheritance object-oriented language in static typing. This virtual machine is an execution system for the Nit language [20]. Nit is a research language (formerly known as PRM [19]). We have two main objectives for this project:

1. Develop a virtual machine for a full multiple inheritance language (Nit)
2. Study optimization protocols in virtual machines

The first objective can be seen as a proof of concept and as a requirement to achieve the second one. The goal is also to test in a realistic way perfect hashing. Our second objective is to develop and compare several optimization protocols in virtual machines. These systems are mandatory for good performances but are often poorly described in scientific literature or not described at all. However, some references exist, like [3] which describes optimizations in Jikes RVM (a research Java virtual machine).

In this paper we will present the current state of work of this project:

1. A functional virtual machine for Nit implemented from an AST interpreter with perfect hashing
2. How we plan to study protocols of compilation/recompilation in the future

We plan to implement several protocols to compare them in a form like [11] which presents a similar study but for for comparing object implementations. This project is a continuation of a first paper [10] where only abstract interpretation was made. Developing a more realistic system with actual interpretation, was the second step.

2 Structure of the Nit virtual machine

To avoid writing a virtual machine from scratch, we chose to implement the Nit virtual machine directly in Nit by basing us on existing Nit compilers. Some solutions to factorize development of virtual machines exist, we can cite VMKit [15] which is a framework for virtual machines. If offers a JIT compiler through LLVM and also a garbage collector with MMtK. The framework was designed to allow users to choose the object model and memory layout and it proposes a limited skeleton of implementation. But the project is now officially retired, and anyway, it would be difficult to use it with Nit. Indeed, the framework is in C++ and if we want to avoid rewrite the entire model of Nit in C++ we would have to make cohabit Nit and C++. This solution is possible but a lot of bindings would be necessary and many layouts of languages would be here. An alternative is to develop only in C++ by rewriting the model of Nit and a parser.

Others systems like *Truffle* and *Graal* [22] offer a compatibility with the JVM. In our case, this solution is not feasible because we would not have control on the virtual machine side. Plus using the JVM means using the Java bytecode and its limitations: lack of multiple inheritance, the erased implementation of genericity etc.

A virtual machine from an AST interpreter. Originally, Nit has different execution engines: a global compiler, a separate compiler and a naive interpreter which is implemented from an abstract syntax tree (AST) decorated by the model. All theses compilers are written in Nit with some parts of C for the global and separate compilers. Nit is compiled into C by its compilers.

The interpreter in written exclusively in Nit an its structure is simple and easy to modify. This interpreter is used to rapidly add functionalities to the language and so, it is not optimized at all. We chose to based our virtual machine on this naive interpreter and modify it step by step into a virtual machine with dynamic loading. Since Nit virtual machine is written in Nit and executes Nit programs this is a meta-circular virtual machine.

In terms of software engineering we have many advantages with this approach:

1. Reducing the cost of development: parsing and semantic analysis of Nit programs are made by the Nit tools in a first time (while we do not have an intermediate representation of Nit like a bytecode)
2. Allowing an incremental design: what is not yet implemented on the virtual machine side is always functional with interpreter mechanisms. So, our virtual machine all the programs executed by the interpreter, partly with its own implementation. The part of mechanisms implemented on the virtual machine side is increased incrementally with new developments.
3. Writing some parts of the virtual machine in C to have a thin control on memory and using a high-level language to write the more general code. This is feasible thanks to the *Foreign Function Interface* (*FFI*) of Nit. This system allows to write some parts of a Nit program in a few languages (C, C++, Python, Java...) and we have used it to write some parts of the virtual machine in C. This mostly concerns virtual function tables and implementation of object mechanisms. Currently, our virtual machine does not implement the FFI and then, it is not bootstrapable.

Adding virtual machine characteristics. In virtual machines, the loading of a class is triggered just before the first instantiation of this class. Loading a class means loading in a first time all of its superclasses. In the context of our Nit virtual machine, dynamic loading means constructing runtime structures and the model of the class at its first access.

A second standard characteristic of virtual machine is the JIT compilation. For now, our system does not contain a JIT compiler and the code of methods is still interpreted. However, we treat the code by computing SSA and numbering local variables. These transformations are made at the first access to a method not already treated. While we do not have a JIT compiler, this is a way of simulating a JIT compilation.

2.1 Implementation of object mechanisms

One of the key factor for performances of an object-oriented language is to efficiently implement the three object mechanisms: method dispatch, attribute access and subtyping-test. We will call a *callsite* a piece of code which contains one of these mechanisms. For a method dispatch, a site could also be a *monomorphic* one: only one method is called by this site. Previous works have shown that a majority of method dispatches are monomorphic.

In a multiple-inheritance object-oriented program, all method dispatches are not in a multiple-inheritance context. Some parts of the class hierarchy can be only in single inheritance while others are in multiple inheritance. The role of the optimization system will be to switch between several implementations depending of the context. The parts of the hierarchy in single inheritance will be implemented with the most efficient techniques and we try to avoid to use the most expensive implementations.

Perfect hashing is a hashing technique without any collision, and its application to object implementation was proposed through perfect class hashing [9,12]. The implementation consists of assigning a unique identifier to each class. This operation is perform during the loading of the class. For each class, the set of its superclasses is known and each superclass has a unique identifier too. A class has a perfect hashing mask and a hashing function (we use the bit-wise `and`). This mask ensures that for each superclass a class has, the hashing function will never create a collision in the perfect hashtable.

Loading a class means performing the following operations to construct the runtime structures:

1. Assign an identifier (a small integer) to the class
2. Compute the optimal mask for the class and build its perfect hashtable
3. Construct the virtual tables to store the methods of the class. Since our system is based on an AST interpreter, the virtual table contains pointer to nodes of the AST instead of binary code
4. Prepare the layout of future instances of this class (basically, the structure of the attribute table)

The memory layout is built to support multiple implementation in only one structure. Each property (method or attribute) has an absolute offset in case of single-inheritance implementation, and a relative offset in case of multiple inheritance implementation. To implement subtyping test in single-inheritance, we apply Cohen's display technique which was first describe in [6] but we inline the display inside the method table [12]. Once a virtual function table of a class is built, its structure will never change. Of course, methods can be recompiled and their corresponding entry in method table could be changed. The virtual table is split into blocks. Each block is associated to a superclass and contains all the methods introduced by it, plus its identifier, and the offset of the block of attributes introduced by it. The perfect hashing technique implies a few requirements in structure of the classes:

– Methods need to be grouped together according to their introducing class
– Attributes also need to be grouped with the same criteria
– A virtual table needs to be allocated in a connected memory area because we use pointers and offsets to navigate into it

2.2 Multiple inheritance and dynamic loading

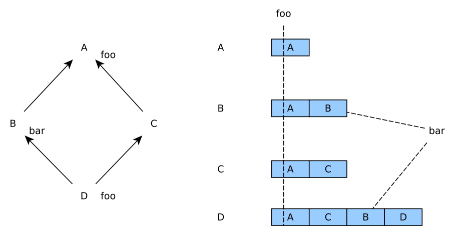

Fig. 1. Multiple inheritance and dynamic loading

Figure 1 presents an example of our multiple inheritance implementation. Let 4 classes `A`, `B`, `C` and `D`. `D` has two direct superclasses. We suppose that `A`, `B` and `C` are loaded. All callsites to the method `foo` can be implemented safely with a static call because `foo` is always defined in `A`. The callsites to `bar` are also implemented with a static call. When the class `D` is loaded, we need to perform several operations to maintain a correct execution:

1. Recompile callsites to `foo` because they are not longer static since `D` redefines `foo`
2. Recompile callsites to `bar` because the block of methods of `B` has been moved in `D`. These callsites must then be implemented with a perfect hashing implementation.

Our object layout is compatible with four implementations, they can be seen as a hierarchy: perfect hashing (multiple inheritance callsites), single-subtyping (single inheritance callsites), static (monomorphic callsites) and inlined (monomorphic callsites and short methods).

3 Protocols of compilation/recompilation

The goal of our future work will be to highlight detailed protocols which are efficient but also to indicate optimizations that give good results together.

For now, our system is just optimized in terms of object mechanisms. We implemented several techniques for each object mechanism, and depending of the context we try to select the most efficient technique. Since a virtual machine is in dynamic loading, the set of classes grows during execution and an optimization can became obsolete. In this case, we have to remove this optimization to maintain a correct execution of the program. Our virtual machine was designed to study the systems that take the decision of optimize a part of program and remove this optimization when needed: protocols of compilation/recompilation.

We see a compilation/recompilation protocol as a toolbox, which contains several tools, coupled with an algorithm which decides which tool must be used in a given situation. Several tools are available, they can be split in three categories: collecting informations, optimizing and repairing. The protocol is composed of several techniques from the three categories.

3.1 The protocol toolbox

Collecting informations. In virtual machines we just have a partial knowledge on source program, so a common technique to gather informations is *profiling*. It could be implemented by placing counters into the code of methods, and then having the number of calls. Another option is to use static analysis techniques. Instead of observing the running program we can statically analyzed it and base optimizations on this knowledge. For example, a simple class-hierarchy analysis [7] can be used. More complex static analysis (like control-flow analysis) would be difficult to implement in a dynamic system because of their cost.

Optimization techniques. In a virtual machine for an object-oriented language, the two majors optimizations are devirtualization and inlining [8]. Devirtualization deletes the cost of a method dispatch by doing a static call instead. Once a method is devirtualized, and if it is a short method, an inlining is possible. Inlining avoids the cost of a function call (passing arguments...) by putting the code of the called method inside the caller one. Many classic optimizations on low-level code (constant propagation, loop unrolling...) in literature could be applied in a virtual machine and are often performed on intermediate representations.

Repairing techniques. Dynamic loading makes hard to predict recompilations because when the system optimizes the code, it does not know which classes will be loaded later. Future loadings can provoke recompilations by invalidating previous optimizations. Three categories of repairing techniques exist:

- *Guards* is a technique which just involves avoiding any repairing, by guarding the optimized version and executing the non-optimized one when the guard fails. [4] proposes a light alternative to classic guards implementations with the *thin guards*.
- *On-stack replacement* [18,14] is a repairing technique which could be applied at any moment. This technique can modify the method inside the stack by patching its return address. During this operation the execution needs to be stopped.
- *Code patching* [16] avoids the complete recompilation of a method by just patching the current compiled code, and for example by replacing a static call by a virtual call.

Overall, all three techniques work well, but it would be better to avoid all of them. For our protocols study, we are also working on an extension of the *preexistence* property. This property was introduced in [8]. Originally, it is linked to the receiver of a call by ensuring that no class loading will be performed that could make the call incorrect.

We propose in [13] to extend this property to a simple inter-procedural static analysis that considers type preexistences and not only values. Basically, preexistence asserts that a given call site will remain correct, even if a new class is loaded while the method including the all site is active. Therefore, preexistence guarantees that no emergency repair is needed for this call site. This extension of preexistence will be based on the result of SSA-computation which gives us dependences between variables and expressions in right parts of assignments and allows us a more accurate analysis.

Recompilations. Recompilations do not only occur when some piece of code became incorrect. Indeed there are two others reasons to recompile a method. The first one is to optimize: when new informations on the code are available, the method can be recompiled in a more efficient way. The second reason is to avoid complicated recompilations during execution of the method. This can be seen as a preventive recompilation, this is how on-stack replacement or code patching could be avoided.

3.2 Future implementations of protocols

As described before, the Nit virtual machine is based on an AST interpreter, so some techniques will be difficult to implement (code patching for example) and will need to be simulated. We will implement recompilations by rewriting AST nodes to change the implementations instead of recompiling binary code like in a standard JIT compiler. We will also simulate the behavior of a JIT compiler by treating the code of method at their first access and choosing the implementation of each callsites before executing them. We also want to test a protocol with a different behavior of dynamic loading: when a method is compiled, directly load all classes that are instantiated in this method. This policy could be helpful to avoid repairing techniques.

Since our virtual machine is based on an interpreter, the efficiency of a protocol is not so easy to measure, we expect that the interpreter is too poorly optimized and too noisy to give realistic time results. So, instead of counting time, we will count for each tested protocol:

- Percentage of callsites of each categories: perfect hashing, single-subtyping, static, inlined
- Number of recompilations
- Number of preexisting sites for testing the preexistence extension

4 Related work

[17] and [23] present techniques to optimize an AST interpreter. They have good performances for AST interpreters with a simpler virtual machine than a

one with a JIT compiler. They performances came from AST nodes that are rewriting during execution in order to specialize the executed code. A rewritten node can be a more specific implementation. For the R virtual machine, nodes are specialized depending of data types in expressions.

A few Java virtual machines have been made for experiments and are meta-circulars. For example Jikes RVM [1] is a research Java virtual machine with a lot of work done on optimizations. They have described their protocol of optimizations, they use profiling and on-stack replacement. Their JVM has several level of optimizations represented by several compilers. When the profiling system detects a often-called method they increase the level of optimizations by recompiling the code with a more optimizing compiler. Maxine [21] is also a research Java virtual machine written in Java. They only made compilation of the code without interpretation.

5 Conclusion

This paper has presented a virtual machine currently in development for the Nit language: a full multiple object-oriented language. For now, our system is based on an existing execution system of Nit which is an AST interpreter. Our virtual machine is already faster than the iNitial AST interpreter and it can execute all Nit programs without FFI. The Nit virtual machine will be used to study study protocols of compilation/recompilation, for this we will simulate in a first time the behavior of a just-in-time compiler.

The study of protocols will be made by focusing us on the extension of pre-existence and based our optimizations on it. We will concentrate on optimizing object mechanisms such as method dispatch, attribute access and subtyping test.

In the future, we also want to make our system more realistic by adding an intermediate representation for the code, and so introduced a JIT compiler to our virtual machine. Eventually this intermediate representation could be a support to perform optimizations. We could also use a bytecode representation to pre-treat Nit programs before executing them with the virtual machine. We will probably have to develop our representation since existing bytecode of C♯/.NET and Java have several limitations, respectively many features which are not fully object-oriented and the lack of multiple inheritance, erased genericity.

References

1. Alpern, B., Augart, S., Blackburn, S.M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., et al.: The jikes research virtual machine project: building an open-source research community. IBM Systems Journal 44(2), 399–417 (2005)
2. Alpern, B., Cocchi, A., Fink, S., Grove, D.: Efficient implementation of java interfaces: Invokeinterface considered harmless. In: ACM SIGPLAN Notices. vol. 36, pp. 108–124. ACM (2001)
3. Arnold, M., Fink, S.J., Grove, D., Hind, M., Sweeney, P.F.: A survey of adaptive optimization in virtual machines. Proceedings of the IEEE 93(2), 449–466 (2005)

4. Arnold, M., Ryder, B.G.: Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In: ECOOP'02, pp. 498–524. Springer (2002)
5. Click, C., Rose, J.: Fast subtype checking in the hotspot jvm. In: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande. pp. 96–107. ACM (2002)
6. Cohen, N.H.: Type-extension type test can be performed in constant time. TOPLAS 13(4), 626–629 (1991)
7. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: ECOOP'95. pp. 77–101. Springer (1995)
8. Detlefs, D., Agesen, O.: Inlining of virtual methods. In: ECOOP'99, pp. 258–277. Springer (1999)
9. Ducournau, R.: Perfect hashing as an almost perfect subtype test. ACM Trans. Program. Lang. Syst. 30(6), 33:1–33:56 (Oct 2008)
10. Ducournau, R., Morandat, F.: Towards a full multiple-inheritance virtual machine. Journal of Object Technology 11(3) (2012)
11. Ducournau, R., Morandat, F., Privat, J.: Empirical assessment of object-oriented implementations with multiple inheritance and static typing. In: OOPSLA'09. p. 18 (2009)
12. Ducournau, R., Morandat, F.: Perfect class hashing and numbering for object-oriented implementation. Software: Practice and Experience 41(6), 661–694 (2011)
13. Ducournau, R., Pagès, J., Privat, J., Vidal, C.: Preexistence revisited (2015), submitted to ICOOOLPS'15
14. Fink, S.J., Qian, F.: Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In: International Symposium on Code Generation and Optimization. pp. 241–252. IEEE Computer Society (2003)
15. Geoffray, N., Thomas, G., Lawall, J., Muller, G., Folliot, B.: Vmkit: a substrate for managed runtime environments. In: ACM Sigplan Notices. vol. 45, pp. 51–62. ACM (2010)
16. Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H., Nakatani, T.: A study of devirtualization techniques for a java just-in-time compiler. In: ACM SIGPLAN Notices. vol. 35, pp. 294–310. ACM (2000)
17. Kalibera, T., Maj, P., Morandat, F., Vitek, J.: A fast abstract syntax tree interpreter for r. In: VEE '14. pp. 89–102. ACM (2014)
18. Paleczny, M., Vick, C., Click, C.: The java hotspot tm server compiler. In: Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1. pp. 1–1. USENIX Association (2001)
19. Privat, J.: De l'expressivité à l'efficacité: une approche modulaire des langages à objets: le langage PRM et le compilateur prmc. Ph.D. thesis, Montpellier 2 (2006)
20. Privat, J.: Nit language. http://nitlanguage.org/ (2008)
21. Wimmer, C., Haupt, M., Van De Vanter, M.L., Jordan, M., Daynès, L., Simon, D.: Maxine: An approachable virtual machine for, and in, java. ACM Transactions on Architecture and Code Optimization (TACO) 9(4), 30 (2013)
22. Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: One vm to rule them all. In: Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13. pp. 187–204. ACM (2013)
23. Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D., Wimmer, C.: Self-optimizing ast interpreters. In: Proceedings of DLS. pp. 73–82. ACM (2012)