# Preexistence revisited

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

Julien Pagès
LIRMM – Université de Montpellier, France
julien.pages@lirmm.fr

Jean Privat
Université du Québec à Montréal, Canada
privat.jean@uqam.ca

Colin Vidal
LIRMM – Université de Montpellier, France
colin.vidal@lirmm.fr

## ABSTRACT

*Preexistence* is a property which asserts that the receiver of a given call site has been instantiated before the current invocation of the considered method [Detlefs and Agesen, 1999]. Hence, preexistence is a mean to avoid *on-stack replacement* when a method must be recompiled during its own activation.

In the original proposition, preexistence is an immutable property, its analysis is purely intra-procedural, and it considers only values. In this paper, we propose to extend it to a simple inter-procedural static analysis that considers types too, not only values. A consequence of this extension is that preexistence is no longer immutable, hence the analysis is not monotonous.

## Keywords

object-oriented programming, just-in-time compilation, on-stack replacement, code patching, preexistence, inlining, devirtualization, multiple inheritance, perfect hashing

## 1. INTRODUCTION

### *Just-in-time recompilation and repair*

The efficiency of modern runtime systems like the Java virtual machine (JVM) relies on dynamic, just-in-time compilation, which is generally triggered by dynamic class loading. Two optimizations are essential: (i) *devirtualization* involves substituting a static call to a more complex late-binding sequence; (ii) *inlining* can then be applied to a static call, in order to inline the callee in the caller. It may occur, in these systems, that the dynamic loading of a class invalidates an assumption underlying the optimized compilation of a method which, thus, must be deoptimized and recompiled. A problem arises when the invalidated method is currently active, for instance when its current invocation has provoked the aforementioned class loading. Let us consider the following Java-like example

```
main() {
    new A().baz();
}
 class A {
     void baz() { this.foo().bar(); }
     void bar() { ... }
     A foo() { return new B(); }
}

class B extends A {
     void bar() { ... }
}
```

In face of such a program, a plausible sequence of actions might be as follows:

1. `main` is compiled, then invoked;
2. `new A` provokes the loading of class `A`;
3. `A.baz` is then compiled with an aggressive optimization of both `foo` and `bar` invocation, on the basis of the currently valid assumption that both methods are not redefined in any loaded subclass of `A`; for instance, the calls are devirtualized, i.e. compiled as a static call, respectively, to `A.foo` and `A.bar`;
4. then, the invocation of `foo` provokes the loading of class `B`, thus invalidating the previous assumption: when returning from `foo` an instance of `B`, the static call to `A.bar` will be erroneous, and it must be *repaired.*

In the following, we will use the *repair* term for the emergency recompilations that must be done while the code to recompile is active.

This is, of course, a simplified example which can be complicated in several directions, for instance with inlining, multi-threads or reflection. Anyhow, the compiled code of `A.baz` must be repaired, and there are roughly three ways to do it:

**Guards:** A first approach just involves avoiding any repairing, by guarding the optimized version and executing the non-optimized one when the guard fails. *Class guards* and *method guards* have been considered [Detlefs and Agesen, 1999]. The former is costless but too narrow, while the later is accurate, but more costly. However, this drawback has been mostly fixed with *thin guards* [Arnold and Ryder, 2002].

**On-stack replacement:** In a second approach, the optimization is not guarded, and the `A.baz` method is recompiled in order to deoptimize the call to `bar`; then, the new version is assigned to the corresponding entry of the method table of `A`. It can be done lazily, with a *trampoline*. This recompilation will account for future invocations of `baz`, but it cannot solve the point of the current invocation. Then, *on-stack replacement* [Hölzle et al., 1992, Fink and Qian, 2003, Steiner et al., 2007] involves patching the stack in order to replace the return address of the call to `foo` by the corresponding address in the newly compiled code.

**Code patching:** An alternative, called *code patching* [Ishizaki et al., 2000], avoids the complete recompilation of a method by just patching the current compiled code of `A.baz`, replacing the static call to `A.bar` by a call to a stub function implementing the virtual call that dispatches to either `A.bar` or `B.bar`. This stub function can be generated on the fly, or may have been precompiled at the first compilation of `A.baz`. Thus, code patching avoids on-stack replacement, too.

Guards are robust and safe, but they suffer from two drawbacks: (i) critical parts of the code must be duplicated; (ii) whereas one may expect that the need for dynamic repair should decrease asymptotically during the execution of a program, guards will be checked perpetually. The two other guardless techniques are complementary and fairly delicate, since both of them involves patching, either the stack or the code, and patching is not that portable. Moreover, choosing between them is not obvious. Overall, all three techniques work well, but it would be far better to be able to avoid all of them.

### *Preexistence*

*Preexistence* is a property which asserts that the receiver of a given call site has been instantiated before the current invocation of the considered method [Detlefs and Agesen, 1999]. Preexistence offers an important guarantee on compiled methods: when a method must be recompiled, the current compiled code of the call sites whose receivers is preexisting to the method invocation remains correct and doesn't need to be repaired. The consequence of the guarantee is that method repair (on-stack replacement or code patching) remains needed only for methods (i) that are currently active, and (ii) contain an invalidated call site with a non-preexisting receiver.

In [Detlefs and Agesen, 1999], the authors used preexistence to decide on inlining for static calls with a preexisting receiver, and the proposed preexistence analysis was purely intra-procedural. In the context of the Java language, two kinds of receiver were considered, namely *invariant arguments*, i.e. the formal parameters of the method to which no assignments are made, and *immutable private fields*, i.e. private fields that are only assigned to in a constructor. Their experiments on a small set of Java benchmarks showed that the proportion of call sites with a preexisting receiver was highly significant. Overall, preexistence analysis is very often cited in the literature, and considered as a well-tried technique, complementary to the repair techniques proposed by the various authors, e.g. [Ishizaki et al., 2000, Arnold and Ryder, 2002].

### *The need for better preexistence*

In this paper, we enlarge preexistence in both its usage and its analysis. We consider languages with full multiple inheritance, implemented with *perfect hashing* [Ducournau and Morandat, 2011, 2012]. With multiple inheritance, especially with such an implementation, one has to consider three kinds of object-invocation sites, for method invocation, attribute (aka field) access and subtyping tests (e.g. casts). Each one can be implemented in a direct, efficient way, like in single inheritance, or in an indirect way implying perfect hashing, and involving a longer, less efficient code sequence. The details are irrelevant, here, and the only point to keep in mind is that repairing a method could concern all of these three kinds of sites. In contrast, in languages with multiple subtyping, like Java, the considered optimizations don't concern at all attribute access. Moreover, the literature on these dynamic optimizations doesn't consider subtyping tests, whereas they are eligible to optimizations similar to devirtualization. Indeed, when the test is always true or always false, it is similar to a static call for method invocation. Otherwise, when the target type is an interface, it can be optimized as a class test if the interface is directly implemented by a single class.

Whereas code patching is a practical solution that deserves to be considered for non-inlined method invocation, because it just involves replacing a function call with another function call, it seems to be markedly less efficient with attribute access and subtyping tests because both are compiled into inlined sequences of code, and the point is really crucial for attribute access. There is thus a need for a more accurate preexistence analysis, which would increase the proportions of receivers that are proven to be preexisting. Therefore, in this paper, we extend preexistence analysis to other forms of expressions, especially through inter-procedural analysis. This makes us distinguish between *value-preexistence* and *type-preexistence*. As it will appear that the new inter-procedural type-preexistence analysis has the drawback to make preexistence volatile, we also present the principle of the experiments that would allow us to specify a protocol of method repair and recompilation offering a good trade-off between the efficiency of the generated code and the efficiency of the protocol.

### *Outline*

Section 2 presents the preexistence principle and our extended preexistence analysis. The next section, a work in progress, will present the results of our first experiments in the interpreter of the Nit language [Privat, 2008]. The paper ends with conclusion and a few prospects.

## 2. PREEXISTENCE ANALYSIS

### 2.1 Vocabulary

We consider a simplified intermediate representation (IR) involving, for each method, the following items:

- a list of *input parameters*, including a *receiver* (`this`);
- a set of *literals* (mostly for `null` and enumerated values);
- sets of *object-invocation sites*, splitted in four kinds:
  - *call sites*, for method invocation,
  - *read sites*, for attribute reading,
  - *write sites*, for attribute assignment,

  - and *cast sites*, for subtyping tests or casts (we assume here that the cast site returns its receiver, typed with the cast target type);

  each site has a *receiver*, and a call site has a list of *extra arguments*, and both receiver and arguments are *expressions*;

- a set of *instantiation sites* (they are not object-invocation sites);

- a set of *variables*, each variable depending on one or more *expressions*; for the sake of simplification, dependence between variables is assumed to be acyclic;

- a distinguished variable for the *value returned* by the method;

- an *expression* is either a parameter, a literal, a variable, a call site, an instantiation site, a cast site or a read site.

Finally, the *dispatched methods* of a call site are the methods that could be invoked according to classes currently loaded. They might be computed with *class hierarchy analysis* (CHA [Dean et al., 1995]). In this intermediate representation, everything is static and immutable, apart from the dispatched methods which form ever-increasing sets.

It is worth noting that the two terms *invocation expression* and *invocation site* denote the same entity from different points of view: the former considers the callees and their returned value which may be preexisting, or not, according to the call arguments and the dispatched-method returned values, whereas the latter considers the caller and denotes a site whose receiver may be preexisting, which has an influence on the way it must be compiled and repaired. For instance, in the `x.foo().bar()` expression of our introductory example, the preexistence of the invocation expression `x.foo()` is a condition of the preexistence of the site calling `bar()`.

We don't consider any intra-procedural control flow analysis apart from the dependence between variables. Especially, we don't mind whether `foo` is invoked before or after `bar`.

## 2.2 Load and compile protocol

In order to be able to specify our limited form of interprocedural analysis, which is static although performed dynamically, at runtime, we need to specify how the code will be incrementally analysed. We thus assume some abstract load and compile protocol based on the general idea that both class loading and method compilation are lazy and performed just-in-time, via trampolines. Furthermore, we assume that the preexistence analysis will take place when a method is compiled.

In order to make things more precise, one might imagine several variants.

### *Variant 1*

1. a class `A` is loaded when an instantiation site `new A` is *executed* for the first time;

2. a method `foo` defined in class `A` is compiled when executing a call site dispatches to this method for the first time;

3. a method is recompiled the first time a call site dispatches to it after the invalidation of one of its invocation sites.

### *Variant 2*

The first rule is replaced with the following:

1. a class `A` is loaded when an instantiation site `new A` is *compiled* for the first time;

In both variants, (i) a call site may dispatch to a method that is not compiled yet; (ii) loading a class doesn't compile anything, and it just implies computing the layout of its future instances[1]. Moreover, in Variant 1, (iii) an instantiation site `new A` may be compiled when the `A` class is not loaded yet.

The benefits of Variant 2 would be to avoid extra recompilations, in the cases where the instantiation is effectively done. In contrast, the extra costs would be to load a class that is not used because its instantiation is guarded, and loading this class would enlarge the call graph with never-compiled methods. The benefit/cost balance of these variants should be tested.

### *Trampolines*

Without loss of generality, we assume that the underlying implementation of objects associates with each object a method-table containing, for each method known by the object class, the address of the implementing code. Then, lazy compilation involves filling the still uncompiled method entries with a *trampoline*, i.e. the address of a stub function which will compile the method, fill the method entry with the resulting address, then jump to it. Thus, lazy compilation of a method which is called via late-binding is both simple and efficient, and can be done at a reasonably high level.

Strangely enough, the lazy compilation of a function called statically is less simple as it requires some low-level code patching, for instance by replacing, in the original compiled code, a jump to the trampoline address by a jump to the newly compiled address. This would be the case for an instantiation site in Variant 1, for a monomorphic call site in all variants, and for calls to `super` and to constructors. Anyway, we assume that we are able to do it, and we will assume, for Variants 1 and 2, that there is some blackbox mechanism for calling a still unknown function.

### *Variant 3*

Alternatively, one might imagine a third variant, in which all methods called via a static call would be compiled when their caller is compiled. Then, there is no need to resort to obscure blackboxes. This would still preserve the dynamic aspect of class loading, provided that all call sites are not static.

## 2.3 Preexistence of expressions

In principle, as preexistence is used only to select how an invocation site is compiled, it should be a matter of dynamic types, not of values. However, the original definition was formulated only in terms of values, as if the only way to assert that the type of an expression is preexisting would be

[1]We don't consider, here, such things as the initialization of static variables in Java.

to assert that its value is preexisting. Moreover, there is at least one case where the preexistence of a value is needed. Hence, our definition of preexistence will be two-fold:

- an expression is *type-preexisting* if its *dynamic type* must necessarily have been *loaded* before the current invocation of the including method;
- an expression is *value-preexisting* if its *value* must necessarily have been *instantiated* before the current invocation of the including method.

Of course, value-preexistence implies type-preexistence. The rules for proving preexistence are the following:

**Literal:** all literals are value-preexisting;

**Parameter:** all input parameters are value-preexisting;

**Variable:** a variable is value (resp. type) preexisting iff all the expressions which it depends on are value (resp. type) preexisting;

**ReadSite:** a read site is preexisting if the receiver is value-preexisting and the read attribute is guaranteed to be immutable; this guarantee can be offered by specific language rules like `final` in Java or `val` in Scala; then, the read-site is value-preexisting, too;

**Return:** the return of a method is value (resp. type) preexisting iff its distinguished return variable is value (resp. type) preexisting (and, of course, the method has been already compiled)[2].

**NewSite:** an instantiation site is type-preexisting iff the instantiated class is preexisting, i.e. it has been already loaded;

**CallSite:** a method-invocation expression is value (resp. type) preexisting if

- its receiver and arguments are all value (resp. type) preexisting, and
- all its dispatched methods have a value (resp. type) preexisting returned value;

**CastSite:** a cast-site expression is value (resp. type-preexisting) if its receiver is value (resp. type-preexisting);

**FinalTypeSite:** an object-invocation expression is type-preexisting if the static type of the expression[3] cannot be specialized (e.g. it has been declared `final` in Java), and the corresponding class is already loaded;

**Otherwise:** if none of the previous rules applies, one must conclude to non-preexistence.

For the sake of simplification, we don't consider, here, all expressions returning primitive types: such literals, parameters and variables are simply removed, and method invocations are just considered as returning nothing. For the same reason, we don't distinguish specifically the call sites that can only invoke a single method, because the invoked method has been declared `static` or `final`, or the dynamic type of the receiver is statically known: they are just special cases of the general CallSite rule.

[2]The Return rule assumes that the code of the method is available to analysis. If it is not the case, e.g. for precompiled methods, the return must be assumed to be non-preexisting. A manual preexistence tag might be envisaged, too.

[3]This is the only need for static types in the intermediate representation.

The original proposition considered only value-preexistence, through the Parameter and ReadSite rules. The essence of type-preexistence is captured by the NewSite rule. While this rule is obvious, we must confess that it took us a long time before identifying it. Besides, it is worth noting that this rule is absolutely useless for a purely intra-procedural analysis like the original preexistence proposition. Indeed, since the dynamic type of an instantiation site is known at compile-type, static optimizations can be done that will never require to be repaired. Hence, the main novelty, here, comes from coupling both NewSite and CallSite rules. Besides, the CastSite and FinalTypeSite are also new. The impact of the compilation protocol lies in the NewSite rule. With Variant 2, all instantiation sites are type-preexisting.

Overall, non-preexistence originates from three possible causes:

- instantiating a non-loaded class (only in Variant 1);
- reading of a mutable attribute;
- a call-site expression whose all dispatched methods are not compiled yet.

### *Preexistence of object-invocation sites*

Besides expressions, we also consider preexistence for object-invocation sites.

**Site:** An object-invocation site is preexisting iff its receiver is type-preexisting.

We don't mind whether a preexisting site is value- or type-preexisting. Indeed, both would be optimized in the same way.

Without loss of generality, a method may contain both preexisting and non-preexisting object-invocation sites. A conservative recompilation policy would restrict optimizations to preexisting sites. Then, if the recompilation of a method is triggered during its activation, it will concern only its preexisting sites, and the execution of all the sites will safely continue.

In contrast, if one wants to follow a more aggressive optimization policy, by optimizing the non-preexisting sites, on-stack replacement or code patching will be required.

Therefore, the middle course might be to

- aggressively optimize all preexisting object-invocation sites;
- optimize, but not inline, all non-preexisting call sites; code patching can be efficiently used for repairing those sites;
- not optimize at all read, write and cast sites that are not preexisting.

It must be noticed that a method must be recompiled when one of its optimized preexisting sites becomes non-preexisting, even if the site itself is not invalidated. Conversely, when a non-preexisting site becomes preexisting, its recompilation might be considered in order to optimize it, but this recompilation is not mandatory. The transition of a non-preexisting site to preexistence may have two possible causes, a new class loading (NewSite rule), or the completion of the call-graph by compiling a dispatched method (CallSite rule).

### An example

Let us consider the following sketch of code:

```
class A { void foo() {..} }
class B extends A { .. }
class C extends B { void foo() {..} }

class Amaker {
     A factory() { return new A() }}
class Bmaker extends Amaker {
     A factory() { return new B() }}

void bar(Amaker mk) { mk.factory().foo() }

main () {
     step1();
     step2();
}

void step1() { bar(new Amaker()); }
void step2() { bar(new Bmaker()); }
```

Then the overall scenario of Variant 2 would be as follows:

1. the `main` function is compiled with two blackbox calls to `step1` and `step2`;

2. the `step1` function is first compiled, which implies loading `Amaker`, then executed;

3. `bar` is then compiled: as `mk` is preexisting, the call to `factory` can be static, but `foo` is not compiled yet, which implies a blackbox;

4. the call to `factory` is executed: `Amaker.factory` is compiled, `A` is loaded, and the return of `factory` is preexisting; `factory` is executed, before `A.foo` is compiled then executed; now, `bar` could be recompiled because both calls to `factory` and `foo` can be optimized;

5. `step2` is then compiled and executed; `Bmaker` is first loaded; as a new dispatched method is now possible for the `factory` call-site, both `factory` and `foo` sites must be deoptimized, the latter because its receiver is no longer preexisting; luckily, there is nothing to do because the optimization was still waiting a new call to `bar`;

6. `bar` is then recompiled and executed, thus loading `B`; at the end, the `factory` call-site has a preexisting return, and `bar` could be recompiled in order to optimize the `foo` call site.

It is worth to examine a few variations:

- if `step1` and `step2` were inlined in `main`, then `Bmaker` would be loaded at the same time as `Amaker`; hence, the call to `factory` would never be static

- if `bar` was inlined in `step1` or `step2`, the call to `factory` would be a static call, even after the loading of `Bmaker`, and the resulting expression would be preexisting.

Indeed, preexistence is always relative to the enclosing method, and inlining doesn't preserve it. Hence, the arithmetics of inlining may be rather counter-intuitive, since it is possible that $1 + 1 = 1$, when one inlining is possible in both the caller and the callee, but not at the same time. The inlining decision may thus be crucial, since one inlining in the caller might prevent several inlinings in the callee.

### Preexistence attribute analysis

The *immutable field* analysis of [Detlefs and Agesen, 1999] is translated, here, into the ReadSite rule. This particular case is the only reason of the need for distinguishing value-preexistence from type-preexistence. Immutability can be deduced from specific kanguages features, with such keywords as `final`, `read-only` or `val`. Otherwise, as discussed in [Detlefs and Agesen, 1999], immutability could be deduced from the static analysis of a class, as the fact that the attribute is not assigned, apart from in a constructor, and a constructor cannot be applied twice to the same object. Of course, assigning the attribute must be reserved to the considered class, for instance it is declared `private`.

However, immutable fields don't fit well to our target language, Nit [Privat, 2008], because both privacy and immutability cannot be expressed strictly. Therefore, for the sake of the experiment, we introduced special manual annotations to allow programmers to assert that an attribute can be considered as immutable.

### More accurate inter-procedural preexistence analysis

The formulation of the CallSite rule is very restrictive, since it forces the arguments of the call-site to be preexisting, even when the value returned by the dispatched methods does not depend on their input parameters. Therefore, besides returning just a boolean value, the preexistence analysis might return, for each value-preexisting expression, a *dependence set* consisting of the subset of input parameter positions the expression depends on. The rules are modified as follows:

**Parameter:** an input parameter at position `p` has the dependence set `{p}`. This rule is the source of the information, and the following rules will compute which parameters (identified by their position) can flow down to the method return, or to other method invocations.

**Variable:** the dependence set of a variable is the union of the dependence sets of all the expressions it depends on;

**Return:** the dependence set of a method return is that of the distinguished variable.

**ReadSite:** a preexisting immutable read site has the dependence set of its receiver;

**CallSite:** a method-invocation expression is preexisting iff, for each dispatched method,

- the returned value is preexisting
- all the arguments of the call site corresponding to the returned dependence set are preexisting;

moreover, the dependence set of the expression is the union of the dependence sets of the call-site arguments whose corresponding input parameter belongs to the returned dependence set;

**CastSite:** the dependence set of the expression is that of its receiver.

All other dependence sets are empty. The interest of this extension is that the new CallSite rule can conclude to the preexistence of a call-site expression even when one of its arguments is not preexisting, because the corresponding parameter is not used in the returned values of the dispatched methods. The CallSite rule can be stated more formally as follows. Let $DSet(expr)$ denotes the dependence set of the *expr* expression, and $DSet(fun)$ denotes the dependence set of the return of the *fun* method. Let us assume that the call-site `foo(arg`$_1$`,..,arg`$_k$`)` dispatches to the set $dispatch(\texttt{foo})$. Then, this call-site expression is preexisting iff

$$\begin{aligned}&\forall \texttt{x.foo} \in dispatch(\texttt{foo}),\\ &\quad\text{the return of } \texttt{x.foo} \text{ is preexisting}\\ &\quad\text{and } \forall i \in 1..k, (i \in DSet(\texttt{x.foo}) \Rightarrow \texttt{arg}_i \text{ is preexisting})\end{aligned}$$

Moreover,

$$DSet(\texttt{foo}(\texttt{arg}_1, .., \texttt{arg}_k)) = \bigcup_{\texttt{x.foo} \in dispatch(\texttt{foo})} \Big( \bigcup_{i \in DSet(\texttt{x.foo})} DSet(\texttt{arg}_i)\Big)$$

Dependence sets must also be used for type-preexistence, because the combination of type-preexistence and value-preexistence gives type-preexistence, along with the dependence set of the latter. For instance, let us consider the following function:

```
A foo (A x, B y) {
     if <some condition>
        return x;
     else return new A(y);
}
```

Its return depends on `x` and `new A(y)`. The former is value-preexisting, with `{1}` as dependence set, while the latter is only type-preexisting, with an empty dependence set. Therefore, the return of `foo`, is only type-preexisting, but its dependence set is `{1}`. An expression calling `foo` can be preexisting only if its first argument is preexisting, and even if its second argument is not preexisting.

### *About recursion*

The above set of rules is correct were it not for recursion. In case of recursion, when the preexistence of an expression depends on itself, one must follow an optimistic approach: if non-preexistence cannot be proven, then one can conclude to preexistence.

Therefore, in practice, the computation of the preexistence of an expression is based on a 3-value recursive function, which returns

- `non-preexisting`, if a rule concluding to non-preexistence has been encountered;
- `recursive`, if no rule concluding to non-preexistence has been encountered, but a recursion has been met;
- `preexisting`, otherwise.

The top-level function which computes the preexistence of a site calls the recursive function on the receiver, and replaces its hypothetical `recursive` value with `preexisting`. What about dependence sets in a recursive call? An exact computation might be more difficult, and a conservative assumption would be to consider that the dependence set of a recursive call contains all the input parameters.

## 2.4 Computing preexistence

In the original preexistence analysis, preexistence and non-preexistence were immutable properties, which were computed only once. However, in the extended preexistence analysis, the preexistence of an entity is no longer immutable. Indeed, class loading can enlarge the call graph and add to a currently preexisting method-invocation expression a dispatched method whose return is not preexisting. In the opposite direction, a non-preexisting method-invocation expression may become preexisting when a still uncompiled dispatched method is compiled and has a preexisting return.

Therefore, for the sake of performance, it is essential to try to avoid to recompute preexistence, hence to memoize it. However, preexistence dependences may be tricky to maintain, and it might be simpler to memoize preexistence only in the immutable cases.

Preexistence and non-preexistence are immutable in only a few cases:

**Parameter, Literal, ReadSite:** the conclusions of these rules are immutable in all cases;

**NewSite:** preexistence is immutable, but non-preexistence is not (only in Variant 1);

**CallSite:** non-preexistence may be immutable, if it results from the immutable non-preexistence of an argument or of a dispatched method return; in contrast, preexistence is never immutable, since the call graph can be enlarged (apart from the call sites that are immutably static);

**FinalTypeSite:** preexistence is immutable, but non-preexistence is not;

**CastSite:** preexistence is immutable if it is immutable for the receiver;

**Variable:** the preexistence of a variable is immutable if it depends only on immutably preexisting expressions; in particular, the preexistence of a method return may be immutable; in contrast, its non-preexistence is immutable, if it depends on at least one immutably non-preexisting expression;

**Site:** invocation sites have immutable preexistence iff their receiver has.

Immutable non-preexistence originates from a single cause, mutable-attribute read sites. Immutable preexistence originates from parameters, literals, immutable-attribute read sites, instantiation sites and final return types.

## 2.5 The case of reflection

Reflection is usually an obstacle to static analysis. However, for preexistence, the obstacle is not insuperable, at least when reflection is specified as in Java. By default, one might consider that all reflective methods are precompiled since their code is not available: hence their return would be non-preexisting. It is however possible to extend the preexistence analysis to the main methods of the reflection API.

For Java, one first observes that all attributes and methods typed by `Class` resort to the FinalTypeSite rule, since `Class` has been declared `final`. Moreover, one can complete the preexistence analysis of reflection with the following rules:

**class:** the `A.class` expression is value-preexisting iff class `A` is already loaded; otherwise, this expression is a black-box, and its execution will trigger the loading of `A`, like `new A`;

**getClass:** an invocation expression of this method is value-preexisting iff its receiver is type-preexisting;

**newInstance:** an invocation of this method is type-preexisting iff its receiver is value-preexisting.

Finally, it is impossible to say anything special about `Class.forName`, which can provoke class loading, hence cannot be value-preexisting.

Both `getClass` and `class` rules assume that reflection is not lazy. If the instantiations of `Class` were lazy, i.e. if the `A` class could be loaded without creating the value of `A.class`, we could never conclude to value-preexistence, and both rules would be invalid. Moreover, in a fully reflective setting where `Class` could have subclasses, even type-preexistence would be unreachable.

## 3. EXPERIMENTS

We are currently implementing the proposed preexistence analysis in the Nit interpreter, and we plan a series of experiments in order to assess the benefits of the approach and the effect of the discussed variations.

Up to now, our inter-procedural preexistence analysis can detect preexisting method-call expressions. The conditions are not too restrictive, since the considered methods must return either `null` (or some hypothetical literal, e.g. instances of Java `enum`), an input parameter, an immutable attribute, an instantiation, a final-type expression, or a call-site expression returning, recursively, one of them. One might think that many call-sites are concerned. However, an extra condition might dramatically restrict the scope of the inter-procedural preexistence analysis: a call-site expression can be considered as preexisting only if all its dispatched methods have been compiled, hence executed at least once. Typically, a factory method would have a preexisting return, but the expressions calling this method would be preexisting only if all the method implementations have been called at least once.

In its original specification, preexistence had a nice property, namely it was immutable and was computed only once for each invocation site. Moreover, apart from inlining, the possible optimizations for a given call site were essentially monotonic. Indeed, a call site could be successively implemented as a static call, then as in single inheritance, and finally, in a way compatible with full multiple inheritance. Overall, everything was monotonic, hence acyclic, with a small number of possibilities, and one might expect high efficiency, i.e. a low number of recompilations. Apart from preexistence which was not considered, it was the main conclusion of our first experiments [Ducournau and Morandat, 2012].

In contrast, the extended preexistence analysis proposed here is no longer monotonic or acyclic. In the inter-procedural analysis, preexistence is rarely immutable. For instance, a call-site expression might alternate preexistence and non-preexistence as the call graph grows up, a class loading switching the expression to non-preexistence, then a method compilation switching it back to preexistence. Therefore, a call site might alternate, too, optimizations and deoptimizations. Therefore, there is an obvious threat to efficiency in this proposition, and experiments are required in order to specify a precise protocol that would keep the original efficiency while providing an effective improvement in the number of optimized preexisting sites.

So, our experiments will compute various statistics during the execution of a program:

- the number of preexisting sites that can be optimized;
- the number of invocation-site expressions that can be considered as preexisting;
- for each of the previous sttaistics, the number of time a given rule applies;
- the number of recompilations triggered by an invocation-site invalidation, or by a transition from preexistence to non-preexistence;
- the number of transitions from preexistence to non-preexistence, and vice-versa.

Various protocols (e.g. the variants described in Section 2.2) will be compared with respect to theses statistics. We expect to be able to present our first results at the workshop.

## 4. CONCLUSIONS AND PROSPECTS

This paper has presented a preexistence analysis which extends previous work by distinguishing between type- and value-preexistence. The introduction of type-preexistence allows us to take advantage of instantiation sites and final types, through a simple dynamic, inter-procedural control-flow analysis. We are currently experimenting this extended preexistence analysis in the Nit interpreter, in order to simulate the behaviour of a JIT compiler based on this preexistence analysis, by counting the number of entities that are preexisting or non-preexisting, optimized, or non-optimized, memoized or not memoized, according to various protocols of compilation, recompilation and repair. Our work has been motivated by multiple inheritance, for which a more accurate preexistence analysis was required, but the proposed preexistence analysis could be used as well with languages like Java or C#. Actually, as the static types are used only marginally in the FinalTypeSite rule, the extended preexistence analysis could be used, as well, for languages like Smalltalk or Pharo, for instance with the implementation proposed in [Ducournau, 2012].

## References

M. Arnold and B.G. Ryder. Thin guards: a simple and effective technique for reducing the penalty of dynamic class loading. In B. Magnusson, editor, *Proc. ECOOP'2002*, LNCS 2374, pages 498–524. Springer, 2002.

J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proc. ECOOP'95*, LNCS 952, pages 77–101. Springer, 1995. doi: 10.1007/3-540-49538-X_5.

D. Detlefs and O. Agesen. Inlining of virtual methods. In R. Guerraoui, editor, *Proc. ECOOP'99*, LNCS 1628, pages 258–277. Springer, 1999. doi: 10.1007/3-540-48743-3_12.

R. Ducournau. Perfect hashing for method dispatch with dynamic typing and dynamic compilation. In *ICOOOLPS Workshop*, 2012.

R. Ducournau and F. Morandat. Perfect class hashing and numbering for object-oriented implementation. *Softw. Pract. Exper.*, 41(6):661–694, 2011. doi: 10.1002/spe.1024.

R. Ducournau and F. Morandat. Towards a full multiple-inheritance virtual machine. *Journal of Object Technology*, 12:29, 2012. doi: 10.5381/jot.2012.11.3.a6.

S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proc. CGO'03*, pages 241–252. IEEE Computer Society, 2003. ISBN 0-7695-1913-X. doi: 10.1109/CGO.2003.1191549.

Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proc. PLDI '92*, pages 32–43. ACM, 1992. ISBN 0-89791-475-9. doi: 10.1145/143095.143114. URL `http://doi.acm.org/10.1145/143095.143114`.

K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In *Proc. ACM OOPSLA '00*, pages 294–310, 2000. ISBN 1-58113-200-X. doi: 10.1145/353171.353191.

Jean Privat. Nit language. `http://nitlanguage.org/`, 2008.

E. Steiner, A. Krall, and C. Thalinger. Adaptive inlining and on-stack replacement in the Cacao virtual machine. In *Proc. PPPJ '07*, pages 221–226. ACM, 2007. ISBN 978-1-59593-672-1. doi: 10.1145/1294325.1294356.