Preexistence and concrete type analysis in the context of multiple inheritance

Roland Ducournau, Julien Pagès, Jean Privat

LIRMM, University of Montpellier and UQAM

August 30, 2016

Context

Virtual machine for Java-like languages

- Dynamic class loading
- Lazy just-in-time compilation

Aggressive optimizations

- Devirtualization
- Inlining

Deoptimizations are needed

Hot-repair problem

Recompilations are complex when the method is **active**

- Guards
- Code-patching
- On-stack replacement
- Preexistence

Motivations

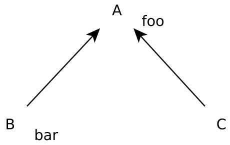
Two main objectives :

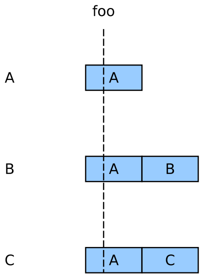
1. Apply existing techniques to **multiple inheritance**
2. **Extend preexistence** to avoid hot-repairs

Multiple inheritance

Three object mechanisms to implement (see [Ducournau and Privat, 2011] for the semantics):

- Method call
- Attribute access
- Subtyping test

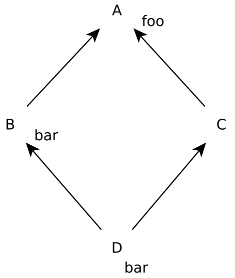


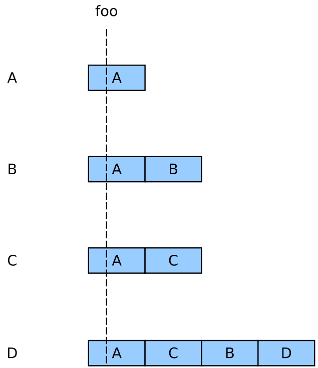


Multiple inheritance

Three object mechanisms to implement (see [Ducournau and Privat, 2011] for the semantics):

- Method call
- Attribute access
- Subtyping test

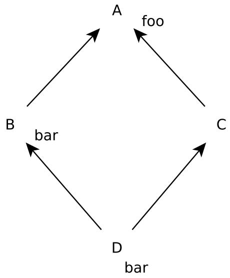


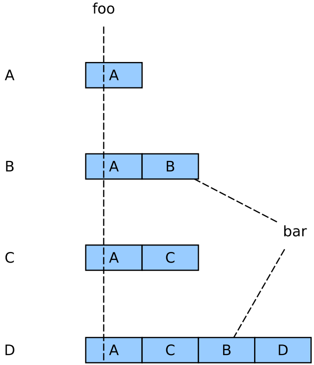


Multiple inheritance

Three object mechanisms to implement (see [Ducournau and Privat, 2011] for the semantics):

- Method call
- Attribute access
- Subtyping test





Implementations and multiple inheritance

Three available implementations:

In decreasing efficiency

- Static call (not for attribute accesses)
- Single-inheritance
- Perfect-hashing [Ducournau and Morandat, 2011]

Optimizing a site by generalizing devirtualization:

- Method call: static call
- Attribute access: as in single-inheritance with a unique position
- Subtyping-test: the target is final or the result is known statically

Preexistence

Introduced by Detlefs and Agesen (ECOOP 1999) to avoid hot repair

Definition

A property of the receiver of a method call site.
The receiver was created before the current call.

```
fun foo(x: A)
do
    x.bar() // x is preexisting
end
```

`x.bar()` can be devirtualized and inlined without guard or patch

Rules of original preexistence

Rules of preexistence

- **Parameter-P**: a parameter is preexisting.
- **ImmutableAttribute-P**: an immutable-attribute reading is preexisting if its receiver is.

Extended preexistence

Extend the definition of preexistence to:

- Any expressions (not only a receiver)
- All object-invocation sites (not only method call)
- Type-preexistence (not only value-preexistence)

Extended preexistence

Value-preexistence

The **value** was created *before* the current invocation of the including method

Type-preexistence

The **dynamic type** was instantiated (*new*) *before* the current invocation of the including method

Value-preexistence $\Rightarrow$ **Type-preexistence**.
Both lead to safe optimizations without hot repairs.

Example of extended preexistence

```
foo (x: A)
do
    var y: A
    if <some condition> then
        y = new B()
        y.bar1() // Type-preexisting if B is already loaded
    else
        y = x
        y.bar2() // Value-preexisting
    end
    y.bar3()
end
```

Rules of extended preexistence

- **Literal-P, Parameter-P**: literals and input parameters are value-preexisting.
- **ImmutableAttribute-P**: an immutable-attribute reading is value-preexisting if its receiver is.
- **Variable-P**: a variable is preexisting if all the expressions which it depends on are preexisting.
- **Cast-P**: a cast expression is preexisting if its receiver is preexisting.

Concrete types and preexistence

ConcreteType-P:

The concrete type of an expression is the set of its possible dynamic types, if this set is known statically.
The expression is type-preexisting if all types/classes are loaded.

Main rules of concrete types

- **FinalType-CT**: when the static type of an expression cannot be specialized, the concrete type of the expression is the static type.
- **New-CT**: a New expression has the immutable concrete type of its instantiated class.

Extension to inter-procedural analysis

- **Return-P**: the return of a method has the preexistence of its return variable
- **Call-P**: a method-invocation expression is preexisting if:
  1. its receiver and arguments are all preexisting
  2. the returned values of all its dispatched methods are preexisting
- **Call-CT**: similar to the previous **Call-P** rule but provides concrete type

Extension to inter-procedural analysis

- **Return-P**: the return of a method has the preexistence of its return variable
- **Call-P**: a method-invocation expression is preexisting if:
  1. its receiver and arguments are all preexisting
  2. the returned values of all its dispatched methods are preexisting
- **Call-CT**: similar to the previous **Call-P** rule but provides concrete type

The **Call-P** and **Call-CT** rules yield mutability in the preexistence of these expressions: adding a new dispatched method can break the preexistence.

Virtual machine and multiple inheritance

The Nit [Privat, 2008] language

- A full multiple-inheritance object-oriented language
- In static typing

The virtual machine

- Developed from the Nit interpreter
- Dynamic loading
- With (only) a simulation of JIT-Compilation
- With the perfect hashing technique for implementing multiple inheritance

Principle of experiments

The benchmarks

- Nit programs: the Nit compiler and interpreter and various tools

Statistics and measures

The principle: counting elements

1. Static counters, number of sites according to:
   - implementations
   - preexistence
2. Dynamic counters
   - executed sites and their implementations
   - number of recompilations or patches

Compilation strategies

Three strategies to evaluate recompilations and preexistence:

1. **Pure code-patching**: each method is compiled lazily with the best available implementations. An implementation change is propagated to the sites.
2. **Pure preexistence**: only preexisting sites used an optimized implementation. Recompilation of whole method when needed.
3. **Mix** of code-patching for method calls and preexistence for attribute and casts.

Results of experiments

Percentages of preexisting optimized sites

	Original	Extended	Improvement
Method calls	48%	60%	25%
Attribute accesses	66%	71%	7%

Conclusion

Two contributions:

1. **Extended preexistence** increases the preexistence rate and **concrete types analysis** increases optimization opportunity
   - Applicable to all Java-like OO languages
2. Application to **multiple inheritance**
   - In pure-preexistence at runtime: only 5% of attribute accesses require perfect hashing $\Rightarrow$ low overhead of multiple inheritance

Perspectives

1. Improve the virtual machine (more realism) with a real JIT-compiler with code-production.
2. Study the effect of inlining on preexistence.
3. More benchmarks

Bibliography

Ducournau, R. and Morandat, F. (2011).
Perfect class hashing and numbering for object-oriented implementation.
Software: Practice and Experience, 41(6):661–694.

Ducournau, R. and Privat, J. (2011).
Metamodeling semantics of multiple inheritance.
Science of Computer Programming, 76(7):555 – 586.

Privat, J. (2008).
Nit language.
`http://nitlanguage.org/`.

Thank you