Preexistence revisited

R. Ducournau, J. Pagès, J. Privat, C. Vidal

LIRMM – Université de Montpellier & CNRS
Université du Québec à Montréal

ICOOOLPS, Prague, July 2015

Plan

LIRMM

Motivations : Virtual Machines

Work under the Open-World Assumption

- dynamic class loading
- lazy method compilation

Performance results from greedy Current-World optimizations

- devirtualization
- inlining

Consequence

- need for dynamic recompilations

LIRMM

Motivations : the Repair Problem

How to recompile a method ...

... while it is running

Motivations : the Repair Problem

3 well-known techniques

- guards
- stack-patching (aka OSR)
- code-patching

Still better to avoid them

Motivations : Multiple Inheritance

In a Java-like language

Optimisations apply to

- method invocation (mainly)
- subtyping tests (marginally)
- and interfaces

In multiple inheritance

Optimisations apply to

- attribute access, too

LIRMM

Motivations : Multiple Inheritance

Inlined mechanisms

- attribute access
- subtyping tests

Repair techniques

- stack-patching : does not apply at all
- code-patching : does not apply efficiently

Motivations : Multiple Inheritance

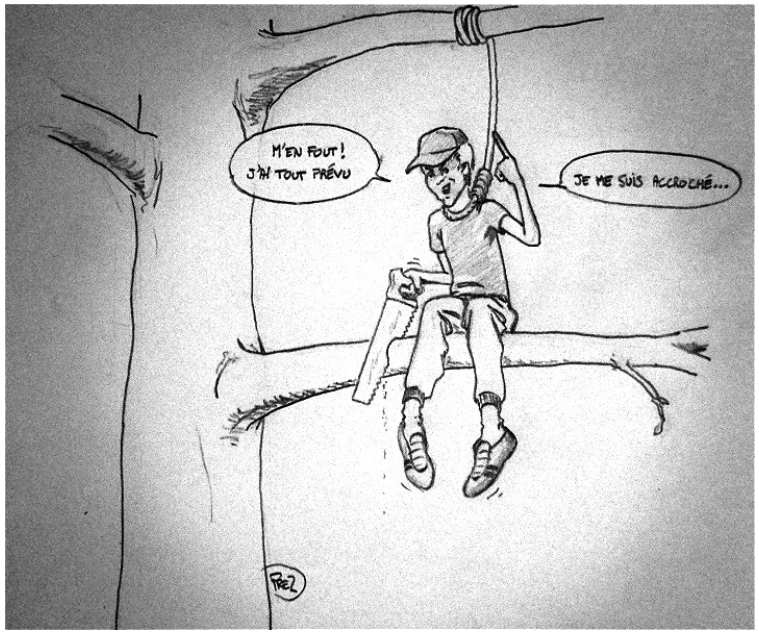
Object representation

implementation	method	attribute	subtyping
inlining	x		
static	x		x
single subtyping (SST)	x	x	x
perfect hashing (PH)	x	x	x
unknown	x	x	x

Optimizations involve substituting

- inlining to static (methods only)
- static to SST (except attributes)
- SST to PH

Plan



LIRMM

Preexistence

A property ensuring that

- a reference will remain compatible
- with the current compiled code of a method
- during the current activation of this method

Preexistence of the receiver avoids the need for hot repair

Original preexistence of value

The referenced object has been created before entering the method

2 original rules [Detlefs and Agesen, Ecoop'99]

- an **input parameter** is preexisting,
- an **immutable attribute** of a **preexisting object** is preexisting

Example

def bar(x) { x.foo()}

x is preexisting and the call to **foo** can be safely optimized

Assessment

Between 20% and 60% of call sites have a preexisting receiver

LIRMM

Extended preexistence of type

The object's class has been loaded before entering the method

The main rule

- **new A()** is preexisting iff **A** is already **loaded**

Example

```
def bar(x) {
    if condition then y=x else y=new A() end
    y.foo() }
```

y is preexisting and the call to **foo** can be safely optimized

Extended preexistence of type

Auxiliary rules

- any expression typed with a **final** type is preexisting
- a **method-invocation** expression is preexisting iff
  - each invoked method has a preexisting return
  - each argument corresponding to a returned parameter is preexisting

Consequence

- a call to a **factory** method is preexisting
- provided that all the invocable methods are compiled !

LIRMM

Extended preexistence

Pros

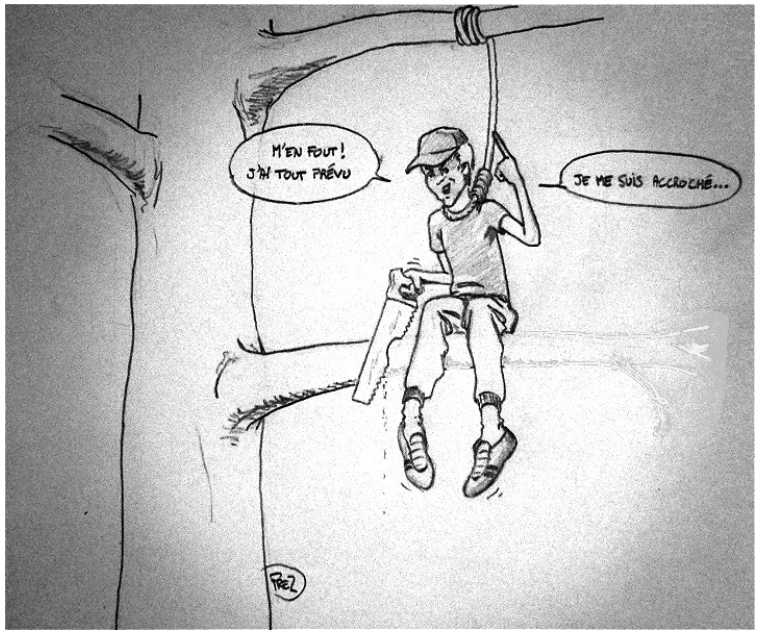
- extended ⇒ (hopefully many) more preexisting receivers
- applied to attribute access and subtyping tests, too

Cons

- preexistence is no longer immutable
- a preexisting method-invocation becomes non-preexisting when a class redefining this method is loaded
- a method must be recompiled when a site of it switches to non-preexistence

LIRMM

Plan



Experiments with preexistence

The testbed

- the **Nit** language (Jean Privat, UQAM, formerly **Prm**, LIRMM)
- the **Nit** Closed-World interpreter
- the **Nit** Open-World VM, based on the interpreter
- a meta-evaluator benchmark
  - the Nit interpreter
  - run in the Nit VM
  - on a small Nit program (eg **fibonacci(4)**)
- statistics at the end of the computation

Statistics of preexistence

Original preexistence

	method	attribute	subtyping	total	%
preexisting	4044	3802	248	8094	58%
non preexisting	4216	916	734	5866	42%
total	8260	4718	982	13960	

- for methods, preexistence rate in the middle upper range of the original paper
- even higher for attributes (80%)
- there is potential for improvement

LIRMM

Statistics of preexistence

Original non-preexistence

	method	attribute	subtyping	total	%
potential	4216	916	734	5866	100%
NewSite	1331	80	0	1411	24%
CallSite	1388	255	663	2306	39%
ReadSite	1426	551	68	2045	35%

LIRMM

Statistics of preexistence

Original non-preexistence

	method	attribute	subtyping	total	%
potential	4216	916	734	5866	100%
NewSite	1331	80	0	1411	24%
CallSite	1388	255	663	2306	39%
ReadSite	1426	551	68	2045	35%
improvable	2719	335	663	3717	63%

LIRMM

Statistics of preexistence

Extended preexistence

	total	%
improvable	3717	100%
NewSite	1390	
CallSite	16	
improved	1398	38%

Experiments with preexistence

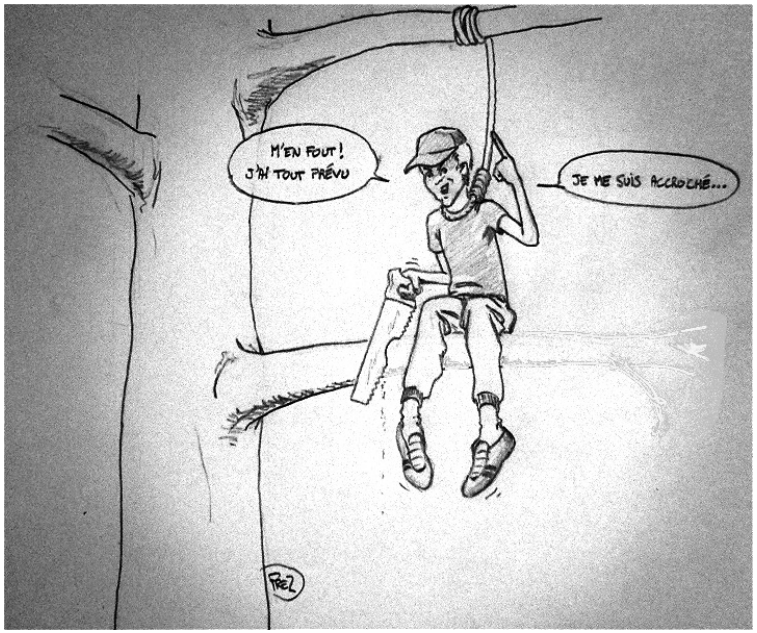
Pros and cons

- most of the improved sites have static concrete types
- inter-procedural analysis has marginal effect

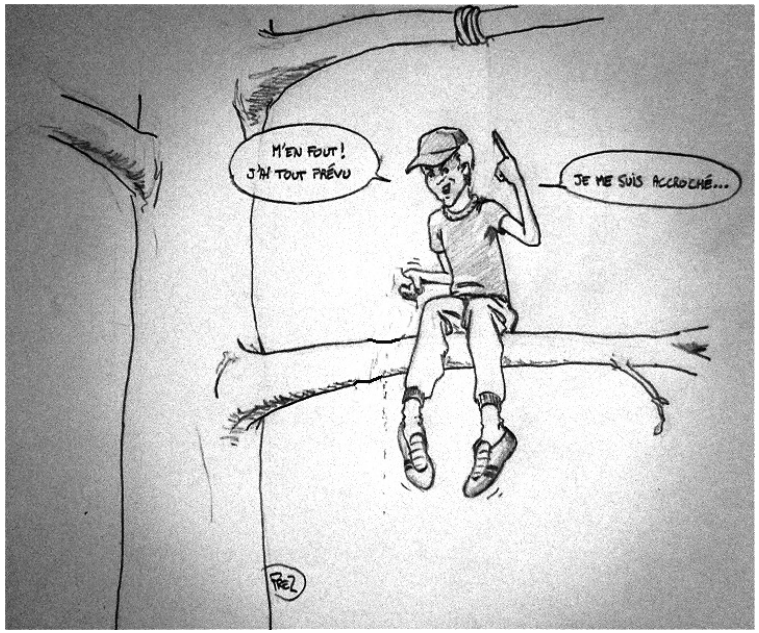
Experiments with preexistence

Pros and cons

- most of the improved sites have static concrete types
- inter-procedural analysis has marginal effect



Plan



Experiments with inlining

But preexistence rate is meaningless

- preexistence depends on programming style
- any program can be transformed into a 100%-preexistence program
- preexistence is not preserved by inlining

```
def bar(y) { y.baz() }
bar(x.foo())
```

$\Longleftrightarrow$

```
x.foo().baz()
```

Next step involves experimenting inlining

- inline bar or baz, not both

Other perspectives

Assessing the recompilation cost

- method recompilations
- transitions between implementations
- transitions between preexistence and non-preexistence

Extended protocols

- with guards or patches ?

Other benchmarks

Conclusion

Extended preexistence : an interesting idea

which needs a deeper study