Generic Typing Policy Impacts on Program Performances

Alexandre Terrasa — Jean Privat

Université du Québec à Montréal

ICOOOLPS'13
2013–7–2
Montpellier

Summary

Generic typing policies

Generic typing policies

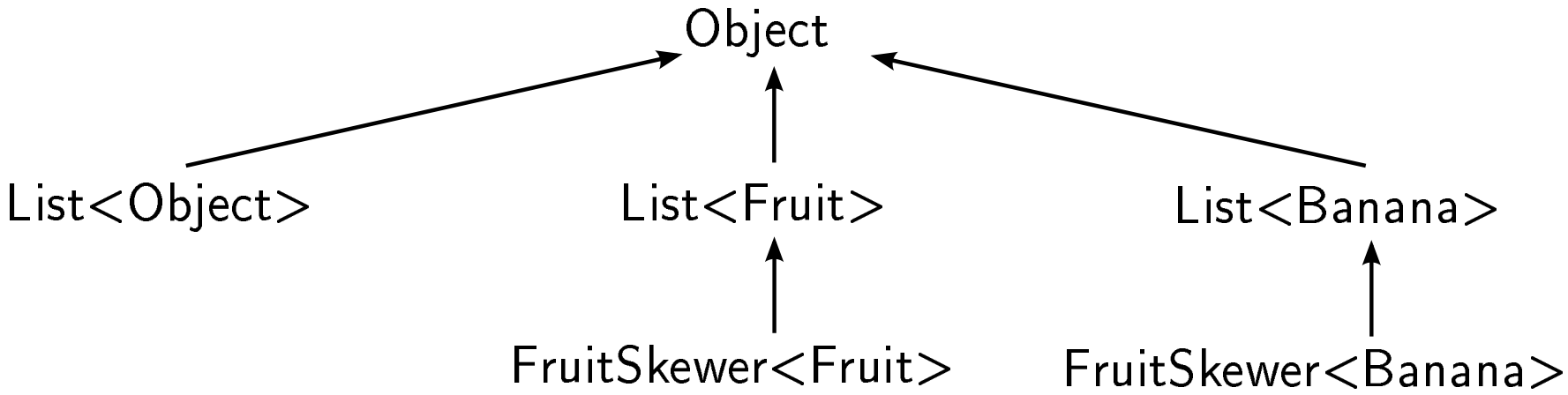
- rules regarding formal types and generic classes
- relationships between generic types
- type verifications made at runtime

3 common policies

- invariant
- covariant (and other *-variants)
- erased

Invariant policy

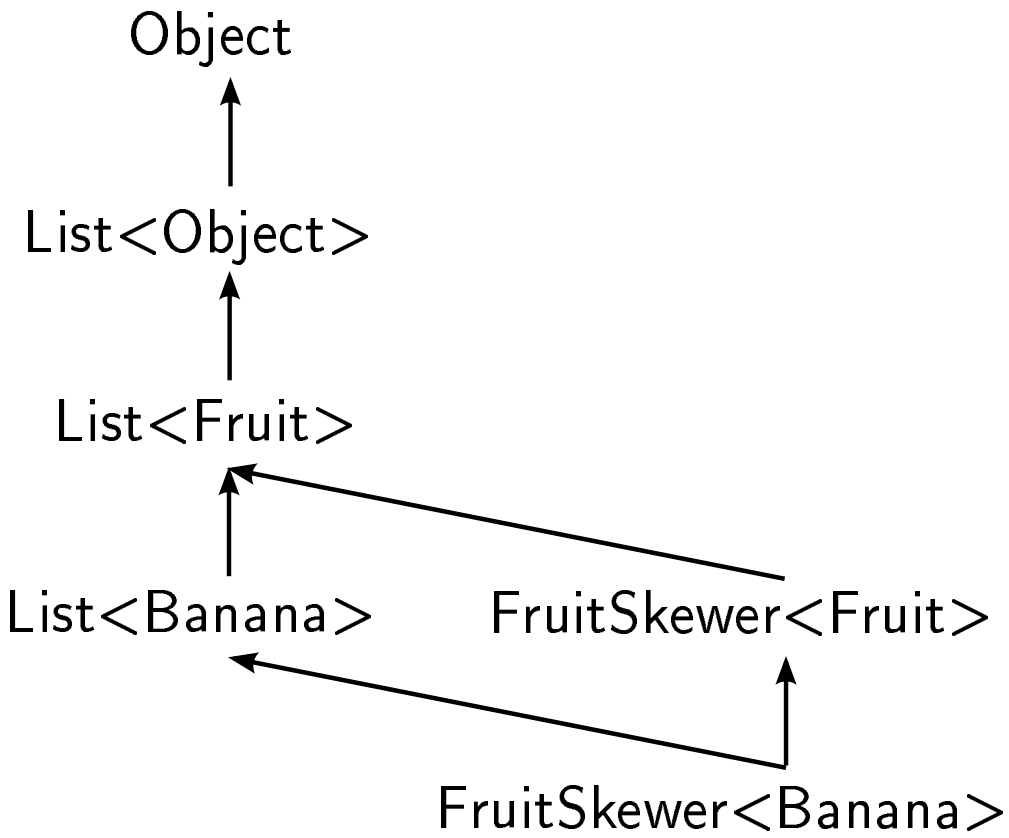
Invariant policy: C++



$$\forall A, B, G\langle A\rangle, G\langle B\rangle : G\langle A\rangle <: G\langle B\rangle \Leftrightarrow A = B$$

Covariant policy

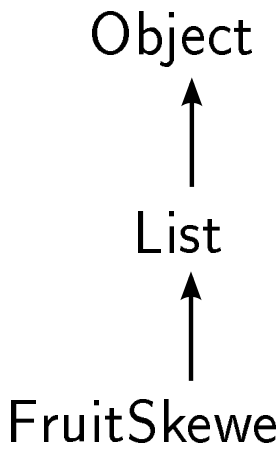
Covariant policy: Eiffel, C#, Nit



$$\forall A, B, G\langle A\rangle, G\langle B\rangle : G\langle A\rangle <: G\langle B\rangle \Leftrightarrow A <: B$$

Erased policy

Erased policy: Java, Scala



$$\forall G\langle A\rangle, G\langle B\rangle : G\langle A\rangle <: G\langle B\rangle$$

Example of a statically correct program with each policy

```
1  List<Fruit> lf;
2  List<Banana> lb = new List<Banana>;
3  Object o = lb;
4
5  lf = (List<Fruit>)o;
6
7  lf.push(new Coconut());
8
9  Banana b = lb.pop();
10
11 b.peel();
12
```

- This program is statically correct with the three policies
- But the dynamic behaviors differ

Example of a statically correct program with each policy

```
1  List<Fruit> lf;
2  List<Banana> lb = new List<Banana>;
3  Object o = lb;
4
5  lf = (List<Fruit>)o;
6       ↑ Invariance: Expected ‘List<Fruit>‘, got ‘List<Banana>‘
7  lf.push(new Coconut());
8
9  Banana b = lb.pop();
10
11 b.peel();
12
```

- This program is statically correct with the three policies
- But the dynamic behaviors differ

Example of a statically correct program with each policy

```
1  List<Fruit> lf;
2  List<Banana> lb = new List<Banana>;
3  Object o = lb;
4
5  lf = (List<Fruit>)o;
6
7  lf.push(new Coconut());
8        ↑ Covariance: Expected a ‘Banana‘, got a ‘Coconut‘
9  Banana b = lb.pop();
10
11 b.peel();
12
```

- This program is statically correct with the three policies
- But the dynamic behaviors differ

Example of a statically correct program with each policy

```
1  List<Fruit> lf;
2  List<Banana> lb = new List<Banana>;
3  Object o = lb;
4
5  lf = (List<Fruit>)o;
6
7  lf.push(new Coconut());
8
9  Banana b = lb.pop();
10                 ↑ Erased:  Expected a ‘Banana‘, got a ‘Coconut‘
11
```

Example of a statically correct program with each policy

```
1  List<Fruit> lf;
2  List<Banana> lb = new List<Banana>;
3  Object o = lb;
4
5  lf = (List<Fruit>)o;
6
7  lf.push(new Coconut());
8
9  Banana b = lb.pop();
10
11 b.peel();
12  ↑ Dynamic:  no method ‘peel‘ on ‘Coconut‘
```

- This program is statically correct with the three policies
- But the dynamic behaviors differ

Comparing generic typing policies

What is the impact of these policies on program performances?

Comparing generic typing policies: a difficult task

- changing the policy = changing the language semantic
- need to find valid programs with each policy
- need to find compilers for each policy
- compared compilers must (idealy) only differs on the policy

Comparing generic typing policies

What is the impact of these policies on program performances?

Comparing generic typing policies: a difficult task

- changing the policy = changing the language semantic
- need to find valid programs with each policy
- need to find compilers for each policy
- compared compilers must (idealy) only differs on the policy

We are lucky

- Most Nit programs have the same behavior with either the erased or the covariant policy

The Nit language

Nit — `http://nitlanguage.org`

- an evolution of PRM
- dedicated to exhaustive assessments of various implementation techniques and compilation schemes

Some Features

- oriented language with static typing
- multiple inheritance
- virtual and nullable types
- generics following a covariant policy (per spec.)

Micro-benchmarks

Objectives

- check that Nit performances are reasonable
- observe behaviors across a variety of implementations

What will vary?

- generic typing policies
- compilation scheme (global, separated, etc.)
- generics impelentation thenique (homogenous, heterogenous, etc.)
- OO implementation technique (binary matric, coloring, binary trees, etc.)

Micro-benchmarks

Policies, Languages and Engines

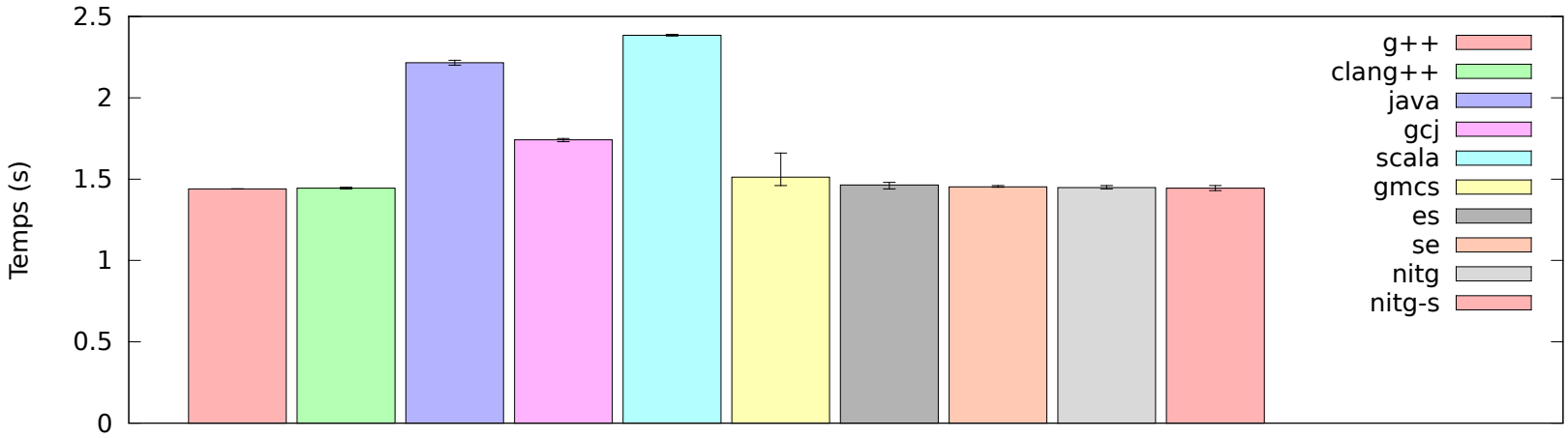
Invariant C++: g++ &g++nv&&iant +g++ &g+riant

Micro-benchmark code in Java

```
1  static public void test(Root a, Root b, int loops, int
       start) {
2    int x = start;
3    for(int i = 0; i < loops; i++) {
4      for(int j = 0; j < loops; j++) {
5        if(TYPE_TEST && x >= 0) {}
6        else { x = x - i + j; a = b;}
7      }
8    }
9    System.out.println(x);
10 }
```

- two `for` loops of 50.000 iterations $\Rightarrow$ 2.5G iterations
- the `else` part is dead code (avoid loop optimizations)
- 6 consecutive executions
  - the first is discarded
  - keep minimum, maximum, and average user time

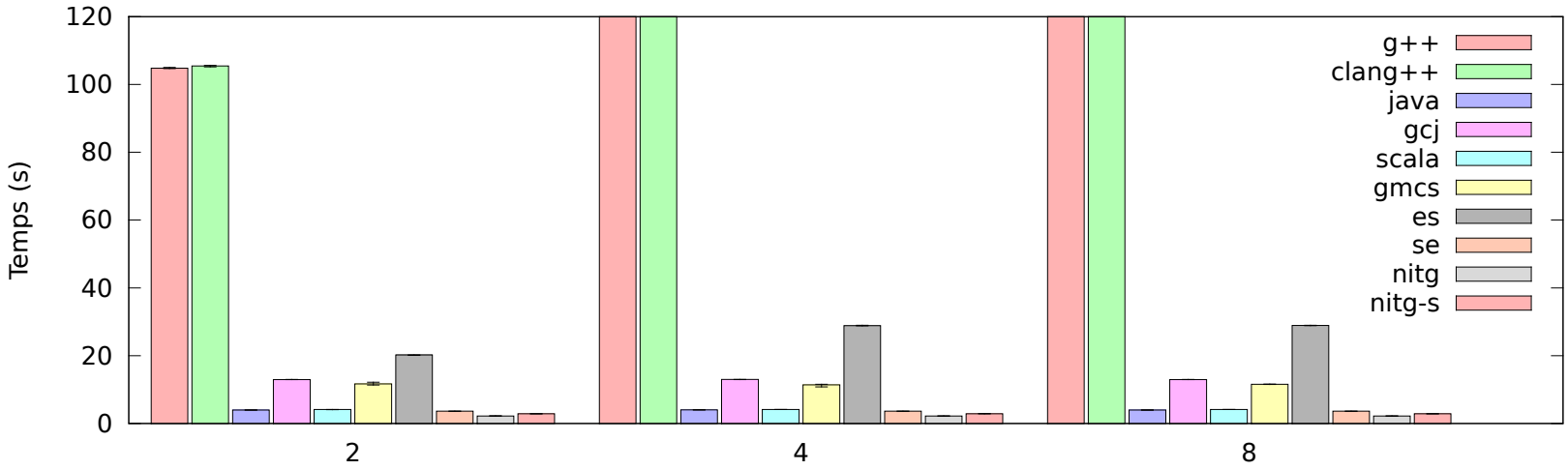
Execution time of programs on the raw loop



- `TYPE_TEST` is `true`

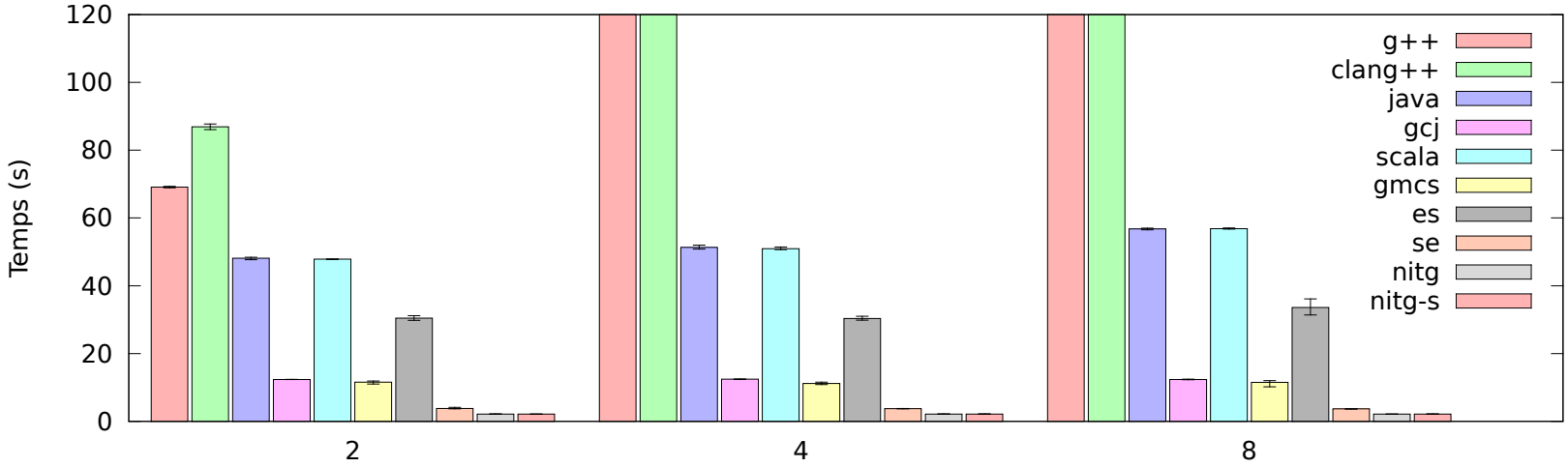
Variations on the depth of the type hierarchy

$$R :> C_1\langle R\rangle :> C_2\langle R\rangle :> \cdots :> C_h\langle R\rangle$$



- `TYPE_TEST` is a successful subtype test
- C++ is $\approx$ 200s for depth=4 and $\approx$ 300s for depth=8

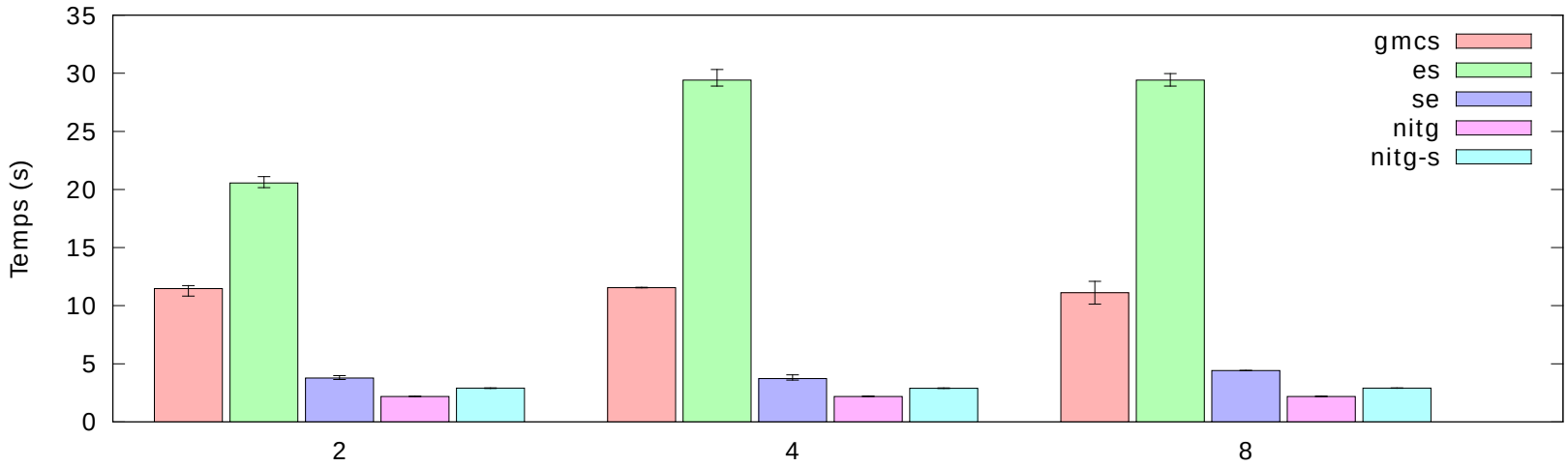
Failed subtype test



- `TYPE_TEST` is the negation of a failed subtype test
- C++ is $\approx$ 250s for depth=4 and $\approx$ 300s for depth=8

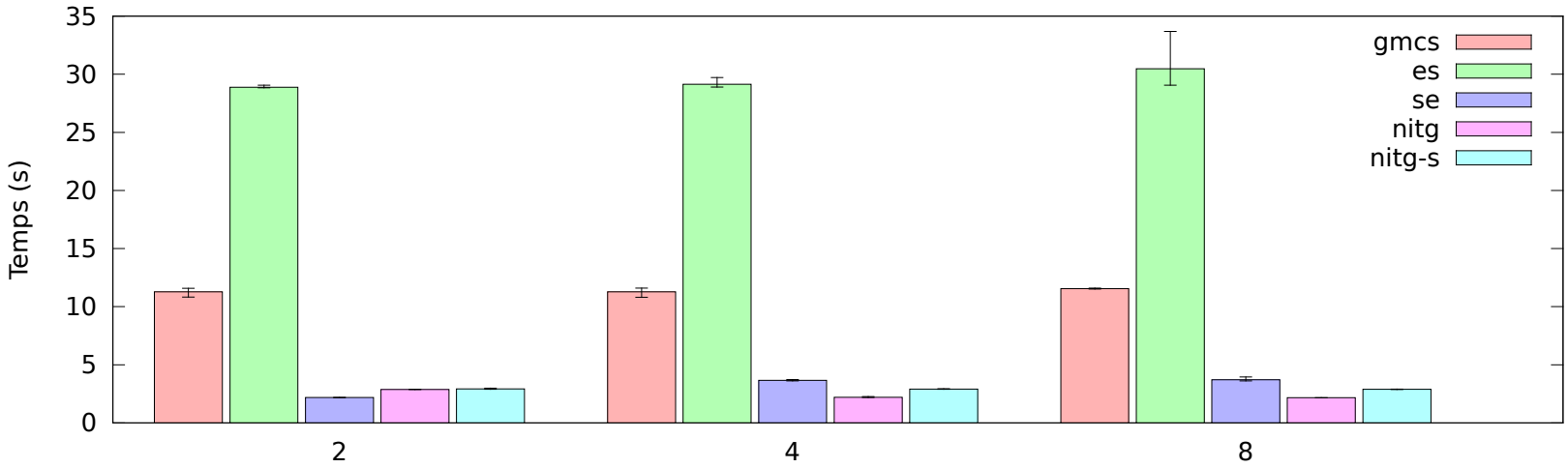
Generic covariant subtype test

$$R :> C_1\langle C_1\langle R\rangle\rangle :> C_1\langle C_2\langle R\rangle\rangle :> \cdots :> C_1\langle C_h\langle R\rangle\rangle$$



Nested generic types in subtype test

$$R :> C_1\langle R\rangle :> C_1\langle C_1\langle R\rangle\rangle :> \cdots :> C_1\langle C_1\langle C_1\langle \cdots\rangle\rangle\rangle$$



Discussion

Constance in time

- hierarchy depth impacts *g++*, *clang++* and EiffelStudio
- nesting level impacts EiffelStudio
- successful vs. failed tests impacts *java*

Performance

- Nit implementations are good
- calling (complex) functions for type test is inefficient

Policies

Nothing conclusive

Policies comparison in Nit

4 compiler variants for Nit

nitg-s implements covariant policy

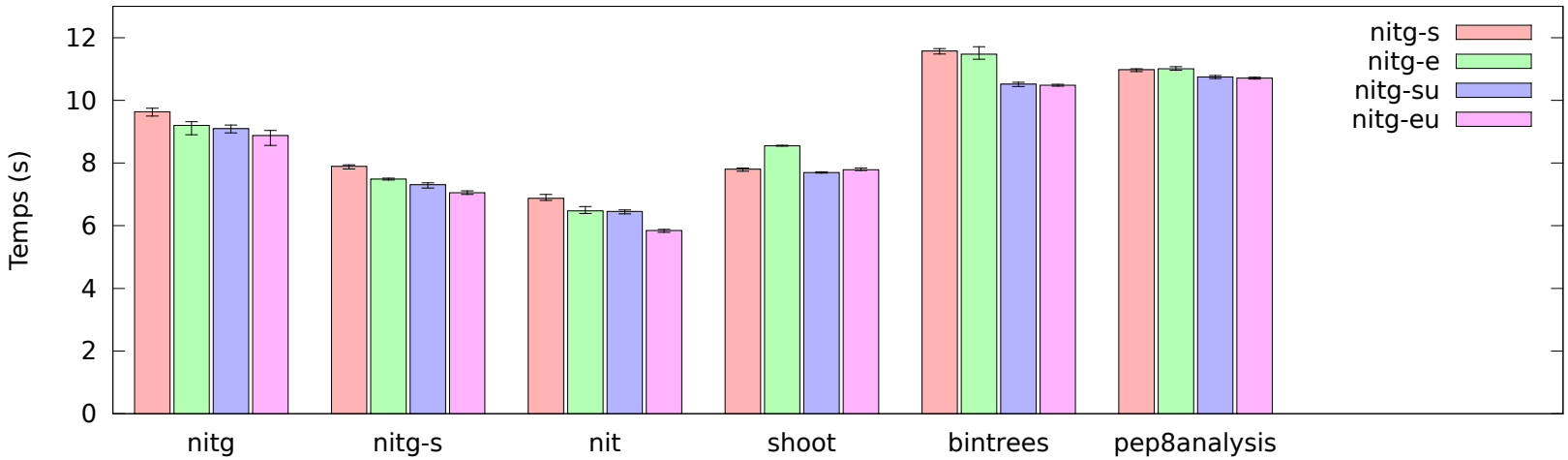
nitg-e implements erased policy

nitg-su and **nitg-eu** are unsafe variations: no checks are done at runtime

Corpus : 5 real Nit programs

nitg	the new Nit compiler (2 distinct execution runs)
nit	the naive Nit interpreter
shoot	an old-school 2D soot-them-up (headless version)
bintree	Boehm's bintree benchmark w/ generics
pep8analysis	static analyzer on Pep/8 programs

Performances considering generics policies in Nit



Discussion

Typing policies: covariance vs. erased

Performances are quite comparable

- `nitg-e` produces often faster programs than `nitg-s`
- but only with a maximum of 5.8% with `nit`

Checks at runtime : safe vs. unsafe policies

Performances are also quite comparable

- subtype tests required by policy do not represent a significant overhead
- only with a maximum of 9.1% with `bintrees`

Conclusion

Impacts on performances of the generics typing policies

- 4 near-identical compilers for the Nit language
- similar performances with existing solutions
- compared on a corpus of reals programs written in Nit

Our Conclusion

- covariant typing policy implies an insignificant performance overhead
- the choice of a generics typing policy by a language designer should not be done on performances considerations

Questions

Thank you for your attention

Any questions ?