Efficiency of Subtype Test in Object Oriented Languages with Generics

Alexandre Terrasa
Université du Québec à Montréal
terrasa.alexandre@courrier.uqam.ca

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

ABSTRACT

In a programming language, the choice of a generics typing policy impacts both typing test semantics and type safety. In this paper, we compare the cost of the generic policy chosen and analyze its impacts on the subtype test performance.

To make this comparison we implement two compilers for the NIT language. One applies covariant policy using a homogeneous implementation and the second applies an erased policy and implementation.

We compare the time e ciency of our solution against engines for C++, C#, EIFFEL, JAVA and SCALA. Results show that our implementations give the best performances compared to existing solutions.

We also compare covariant and erased policies on existing NIT programs using our two compilers. Results show that covariance does not imply signi cant overhead regarding subtype test performances.

Due to the small di erence between the costs of the di erent policies, the choice of a policy by a language designer should not be in uenced by performance considerations.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors | *Compilers*

General Terms

Experimentation, Languages, Subtyping, Performance

Keywords

generics, typing policy, covariance, erasure, generics implementation, homogeneous, heterogeneous, subtype test, multiple inheritance

1. INTRODUCTION

Genericity is an important feature of statically typed object oriented languages, especially for libraries and reusable components. Several languages provide genericity features and generic typing policy vary between them.

In a programming language, the generics typing policy speci es the rules regarding type parameters, type arguments and generic classes. The choice of a typing policy impacts both typing test semantics and type safety.

In this paper, we compare the runtime cost of the chosen generics policy and its impacts on the subtype test performances. Comparing objectively the typing policies is not an easy task since each policy has di erent rules. Indeed, changing a language policy has dramatic e ects because: i) it changes the expressiveness of the language; ii) it changes the semantics of the program: implicit casts can be removed or added at di erent places; iii) the semantics of the subtype test `instanceof` or explicit casts depends on the policy.

1.1 Generics Typing Policies

Basically, at runtime, one can consider three families of typing policies: invariance, covariance, and erasure. The choice of a policy a ects the semantics and the number of subtype tests to be performed at runtime.

Invariance. There is no subtyping relationship between the generic variations of a class. Invariant typing policy guarantees that static typing is enough to ensure the dynamic type safety. This is the policy applied by C++:

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

Covariance. A subtyping relationship exists between two generic variations based on the relationship of the type arguments:

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

To ensure type safety, languages may use two di erent approaches. The rst, used by C# with its variant generic interfaces, is to introduce limitations and static rules to ensure type safety at compile time. The second, used by EIFFEL, delegates these checks at runtime and add some *covariant casts* to ensure type safety.

Erasure. The types arguments are not considered and all generic variations of a class are subtypes of each other. This is the policy applied at runtime by JAVA and SCALA:

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

Erasure can be seen as the way to do genericity without generic classes. To ensure type safety, the compiler automatically adds *erasure casts* to change the type of the method argument or the return type according to the static bounds specified. While erasure can avoid some covariance casts, erasure casts represent an overhead that is not required by the covariant policy.

1.2 Generics Implementations

Invariant and covariant generic policies can be implemented by both heterogeneous and homogeneous implementation. The main difference between these approaches is whether the generic variations of a class share the same implementation or not.

Heterogeneous. With the heterogeneous implementation the compiler generates a *customized* version of the compiled class for each generic variation. In each customized variation of a class, types parameters are replaced by concrete types. Since each customized version is considered as a distinct class subtypes tests can be resolved in the same way than with non-generic types. This is the implementation used for C++ templates and SMARTEIFFEL.

Homogeneous. With the homogeneous implementation all the generic variations of a class share the same compiled code. Therefore, references to type parameters and open types [7] must be resolved at runtime. This is the implementation used by EIFFELSTUDIO and C#.

Erased. The erased implementation behave like implementations without genericity. At compile time, all references to type parameters in the class body are replaced by their static bounds. The same compiled version of a generic class is shared by all its instances. Erasure casts are done using the static bounds in the compiled class. Subtyping tests with erased genericity behave exactly like non-generic subtype tests, only the class type is taken into account in subtype test. The erased policy results from an erased implementation. This is the implementation used by JAVA and SCALA.

1.3 The Nit Language

Comparing generics typing policies and implementations is a difficult task and we had two difficult constraints to solve:

- Having real-world programs that behave exactly the same way with both policies. This is a difficult task.
- Having two compilers that generate the same machine code except for what is related to the policies and the typing implementation. This is easier because we can implement them.

We base our study on the NIT language[1]. NIT is an evolution of the PRM language which was dedicated to exhaustive assessments of various implementation techniques and compilation schemes [4, 8]. NIT is an object oriented language with static typing. Its features include nullable types [6] and virtual types [9].

The NIT specification states that the policy involved is covariant (the same was true in PRM). However, the main separate compiler, `nitc`, implements genericity with an unsafe erasure policy (i.e., without automatic casts) and has a lot of `TODO` and `FIXME` in its source-code. There are two other engines, used for education purposes, `nit`, a naive and really inefficient interpreter, and `nitg`, a global compiler with heavy customization. Both implement the real covariant semantic but on a subset of the language. Consequently, most of the existing Nit programs, including nitg and nit, behave exactly the same way either with a covariant policy or with an erased policy.

With the NIT language, we have a test platform to compare policies and implementations. To make this comparison we created two new implementations of the NIT compiler. `nitg-s` follows the covariant policy using a homogeneous implementation. `nitg-e` uses an erased implementation and policy. Both engines were designed to provide maximum scalability for testing.

1.4 Comparing Policies and Implementations

In the rest of this paper, Section 2 presents the micro benchmarks used to verify that our solution is in the same magnitude of existing implementations. We compare the time efficiency of our solution against implementations used in other languages and compilers using generated type hierarchies. Results with compilers for C++, C#, EIFFEL, JAVA, SCALA and NIT show that our implementations give good performances compared to existing solutions.

Section 3 compares covariant and erased policies on NIT programs and discuss the performances of the covariant compiler versus an erased compiler. Results show that covariance does not imply significant overhead regarding subtype test performances.

Finally, Section 4 presents our conclusions and future work.

2. LANGUAGES COMPARISON

In this section we compare the behavior of multiples engines (10 compilers and 2 virtual machines) for the languages C++, C#, EIFFEL, JAVA, SCALA and NIT on some unrealistic micro-benchmarks.

Unlike the study proposed by Garcia [5] that provides a comprehensive study of facilities for generic programming in these languages, this evaluation aims to compare the implementation of subtype test for generics with all other mechanisms being different.

This evaluation aims to answer several questions: i) Do the NIT implementations have performances of the same order compared to other existing solutions? ii) How different are

[1] `http://nitlanguage.org`

the measured behaviors with a large variety of policies, implementations, languages, and engines? iii) How does a speci c execution engine respond to a stress test on subtyping mechanisms for generics?

2.1 The Engines

We compare the following compilers and virtual machine.

g++ provides the heterogeneous implementation of the invariant policy of C++. We use the version *GNU gcc g++ (Debian 4.7.2-5) 4.7.2 - linux-x86-64 with GNU*.

clang++ also provides the heterogeneous implementation for C++. We use the version *Debian clang version 3.0-6.2 (tags/RELEASE_30/final) (based on LLVM 3.0) - linux-x86-64*.

Both *g++* and *clang++* use the same implementation of the subtype test shared in *libstdc++* version *4.7.2-5 Debian GNU linux-x86-64*. Both are also used with the `-O2` optimization level.

javac version *1.7.0_03* is used to compile JAVA programs. They are run on the *OpenJDK Runtime (IcedTea7 2.1.2) (7u3-2.1.2-2)* following the erased policy and implementation of Java.

gcj, the GNU compiler for JAVA, version *4.7.2* is also used to compile JAVA programs.

scalac version *2.9.1*. SCALA programs are run on the same runtime environment than JAVA programs. The policy used is covariant but uses an erased implementation.

gmcs is the compiler for MONO C# version *2.10.8.1*. C# programs are executed on the MONO JIT compiler version *2.10.8.1*. C# provides a covariant and homogeneous implementation of the covariant policy.

es is the EIFFEL STUDIO compiler. It provides a homogeneous implementation of the EIFFEL covariant policy. We use the version *ISE EiffelStudio 7.1.8.8986 GPL Edition - linux-x86-64*. It is used with the `-finalize` optimization level.

se is the SMARTEIFFEL compiler. It provides an heterogeneous implementation of the EIFFEL covariant policy. We use the version *Release 2.4 (svn snaphsot 9308) - linux-x86-64*. It is used with the `-no_check` optimization level.

nitg compiler provides a heterogeneous implementation for NIT covariant policy using a global compilation process and customization.

nitg-s compiler provides an homogeneous implementation of the NIT covariant policy using a separate compilation process with global linking.

The generated C by both NIT engines is compiled with *gcc* version *GNU gcc g++ (Debian 4.7.2-5) 4.7.2 - linux-x86-64 with GNU* with the `-O2` optimization level. The version is the Git commit `7e44894eb766bf5d4d58c2a45356c6344410248f`.

2.2 The Micro-Benchmarks

The objective of the micro-benchmarks is to measure subtyping mechanisms of generics. In order to do so, the programs must, during their execution, use subtyping for generics heavily, and the other mechanisms of the language minimally.

Our main constraint is that the programs must be equivalent as much as possible for all languages involved. For this reason, we use the lowest common denominator for the policy and keep to minimum the use of libraries and other speci c mechanisms.

The test programs are automatically generated by a script. The generation is deterministic and the same program is generated in all languages. Each generated program is made of three parts:

- A simple type hierarchy. The height of the hierarchy is determined by a parameter of the script.
- A loop of 2.500.000.000 iterations.
- A subtype test done at each iteration.

The test protocol. Micro-benchmarks are launched on a Intel i7-2640M with a CPU@2.80GHz x86_64 machine installed with a Debian GNU/Linux operating system.

The test protocol is the following: 6 consecutive executions of each con guration are performed, the rst execution is discarded, and the minimum, maximum and average user time of the 5 executions are kept. User time is measured with the GNU `time` command.

The big loop. The big loop is done by a function and is divided into two `for` loops doing 50.000 iterations each. The `else` part is not executed but prevents loop optimizations.

```
static public void test(Root a, Root b, int
    loops, int start) {
  int x = start;
  for(int i = 0; i < loops; i++) {
    for(int j = 0; j < loops; j++) {
      if(TYPE_TEST && x >= 0) {
      } else { x = x - i + j; a = b;}
    }
  }
  System.out.println(x);
}
```

Figure 1 gives the performances of each engines with a dry loop: no subtype test is done in the loop. The `TYPE_TEST` placeholder is replaced by `true`. We tried to put every engine on the same footing but the JAVA JVM applies hard to control common cases optimizations [2]. Other engines reach the same performances on the dry loop. These results measure only the time used by the dry loop and can be used to approximate time consumed by subtype tests in further results.

The type test. Three kinds of type hierarchies are generated depending on the micro-benchmark objectives. The generated type hierarchies are simple. The h parameter of the script determines the height of the hierarchy. More specifically, the following classes are generated: A root class R, without attribute. A total of h linear generic subclasses C_i of R. Each generic subclass take only one type parameter. For the experimentation, we used three configurations: h=2; h=4 ; h=8.

In the loop, the TYPE_TEST placeholder is replaced by a test between an instance of the deepest type in the hierarchy against the middle type of the type hierarchy.

Results given in Figure 2 show subtype test performances regarding the height of the hierarchy:

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

The same type hierarchy is used to bench failed subtype tests. In the loop, the TYPE_TEST placeholder is replaced by the negation of the test between an instance of the penultimate type against the last type. Results are given in Figure 3.

Results given in Figure 4 show subtype test performances regarding covariance:

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

Results given in Figure 5 show subtype test performances regarding generic type nesting level:

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

Note: In $C{+}{+}$, classes are generated fully virtual: method and specialization are all declared using the `virtual` keyword. In $Java$ and $C\#$, we use interfaces instead of classes to ensure the virtual machine will rely on its implementation of multiple-inheritance type tests. Same for $Scala$ using traits.

2.3 Discussion

Implementations based on function calls are slower. The functions used in the subtype test implementation of $g{+}{+}$, $clang{+}{+}$ and EiffelStudio are less efficient than switches used in $nitg$ and SmartEiffel or than the tables used in $nitg\text{-}s$.

Not all the implementations of the subtype test are time constant. $g{+}{+}$, $clang{+}{+}$ and EiffelStudio performances vary with hierarchy depth. EiffelStudio performances vary with generic types nesting and covariance depth. Failed tests are slower than successful ones in the Java JVM because of common cases optimizations. Java uses a last type checked cache for subtyping tests against interfaces.

g++	clang++	java	gcj	scala	gmcs	es	se	nitg	nitg-s
1.44	1.44	2.21	1.74	2.38	1.51	1.46	1.45	1.44	1.44

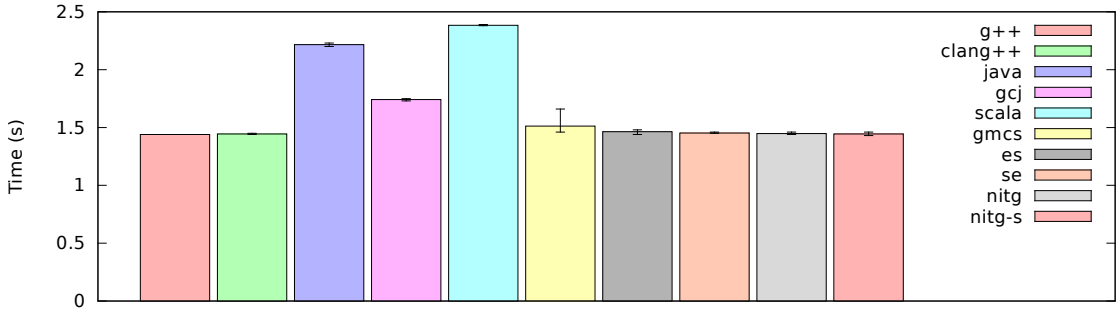

Figure 1: Time efficiency of each engine on the dry loop.

	g++	clang++	java	gcj	scala	gmcs	es	se	nitg	nitg-s
$h = 2$	104.82	105.44	3.97	12.94	4.10	11.67	20.19	3.62	2.21	2.88
$h = 4$	188.23	213.00	4.02	12.98	4.12	11.38	28.88	3.62	2.19	2.88
$h = 8$	299.21	299.21	3.96	12.96	4.12	11.55	28.92	3.61	2.18	2.88

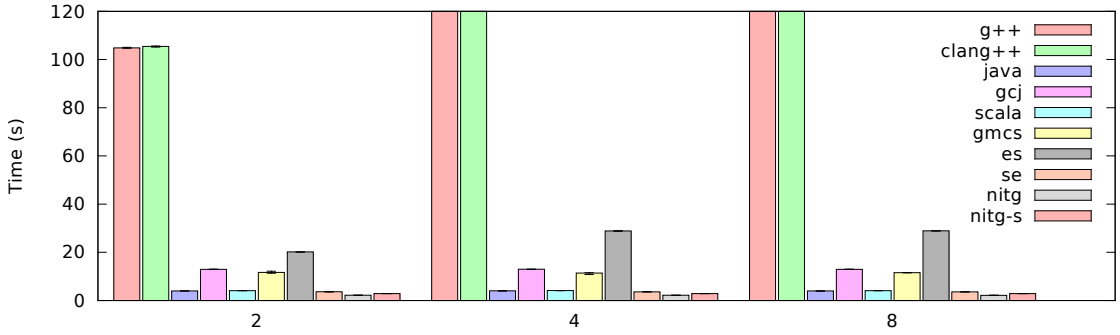

Figure 2: Time efficiency of each engine considering variations of the depth of the type hierarchy.

	g++	clang++	java	gcj	scala	gmcs	es	se	nitg	nitg-s
$h = 2$	69.13	86.88	48.11	12.34	47.87	11.54	30.43	3.81	2.17	2.16
$h = 4$	258.95	237.06	51.35	12.43	50.95	11.16	30.32	3.75	2.16	2.16
$h = 8$	299.21	299.21	56.76	12.33	56.83	11.50	33.61	3.67	2.16	2.16

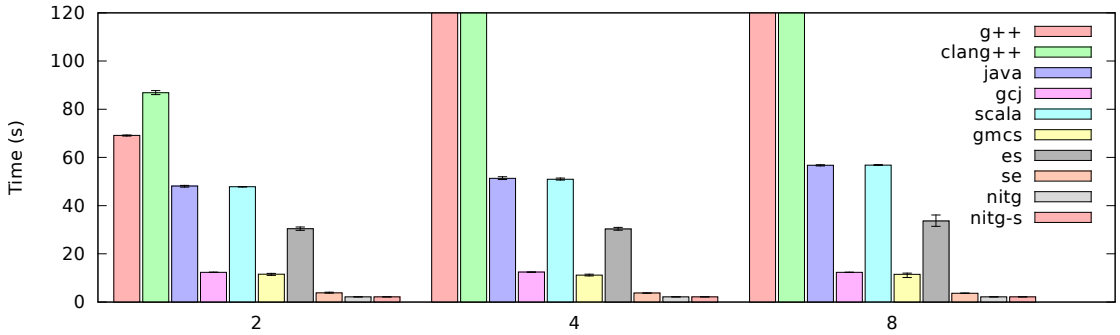

Figure 3: Time efficiency of each engine considering variations of the depth of the type hierarchy with failing subtype tests.

	gmcs	es	se	nitg	nitg-s
$h = 2$	11.46	20.56	3.77	2.19	2.90
$h = 4$	11.55	29.41	3.72	2.19	2.89
$h = 8$	11.11	29.41	4.41	2.18	2.90

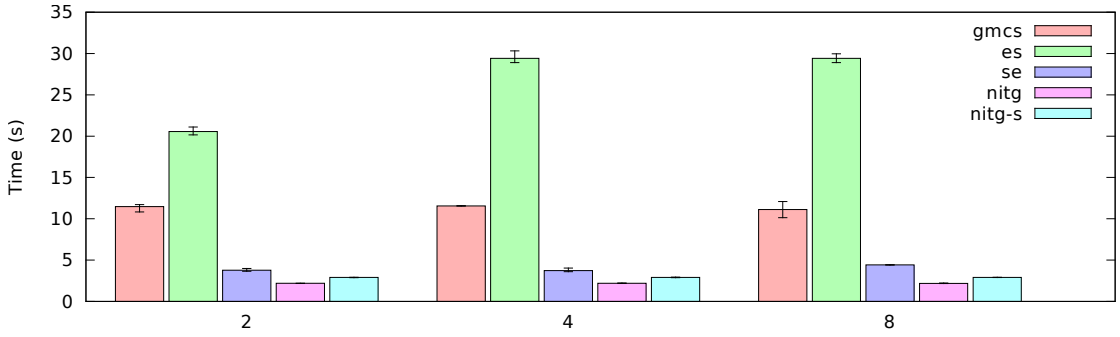

Figure 4: Time efficiency of each engine considering covariant subtype tests.

	gmcs	es	se	nitg	nitg-s
$h = 2$	11.27	28.88	2.17	2.88	2.92
$h = 4$	11.26	29.13	3.66	2.20	2.90
$h = 8$	11.55	30.47	3.70	2.16	2.88

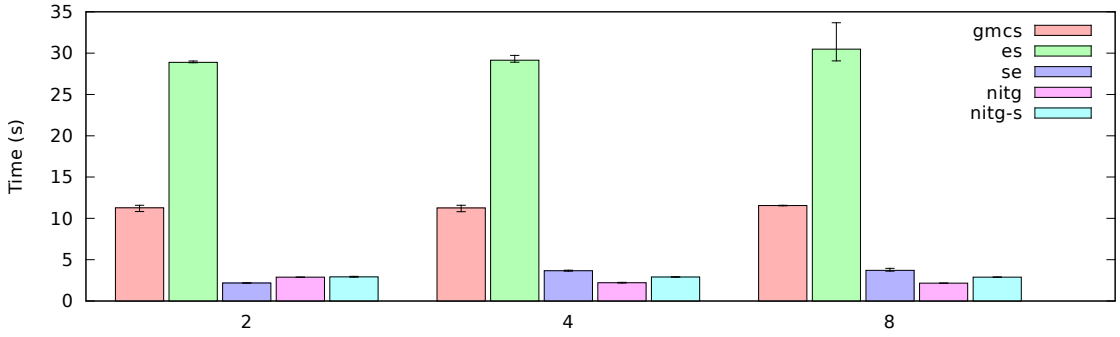

Figure 5: Time efficiency of each engine considering different generic type nesting levels.

larger; (iii) the existing programs keep their expressiveness and really adapting them to a safe policy may require complex rewriting, some workarounds and numerous additional explicit casts.

3.2 The Test Corpus

The test corpus consists of ve programs (most of them are available in the NIT project repository): `nitg`, `nit`, `shoot`, `bintrees` and `pep8analysis`.

nitg is the compiler used to implement erased and covariant genericity. The program is compiled once but executed in two di erent ways.

The rst way, `nitg nit_metrics.nit`, does the global compilation of the `nit_metrics` program (a program that computes and displays static metrics about programs). The second way, `nitg --separate nitg.nit`, is the separate compilation of `nitg` itself using the covariant-homogeneous implementation.

Note: the two executions share the same front-end (parsing, model building, semantic veri cation) but this whole front-end process only requires, in both cases, less than 15% of total compilation time. The program is the same but the two executions are quite di erent.

nit is the really slow and naive interpreter executed with `nit -- test_parser.nit -n rapid_type_analysis.nit`. First argument, `test_parser.nit`, is a small test program that parses a NIT source le, then builds and displays its AST. The source le tested here is the Nit module implementing the RTA algorithm. Note: `-n` is intended to `test_parser` (not the interpreter) and skips the display part.

shoot is a basic *shoot'em up* game with a complete OO game logic in a 2D environement. This program extensively uses oating-point arithmetic and nested heterogeneous collections to store the game elements. For obvious reason, the version executed is headless (no display) and has no frame rate limitation. A total of 300.000 frames are computed. The full graphical version is not yet published.

bintrees is a simplistic adaptation of GCBench[2] of Hans

[2] `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench/applet/GCBench.java`

Boehm except that the trees are implemented by a generic class.

pep8analyzer[3] is a static analyzer that detects bugs and bad programming practices in Pep/8[4] programs. The arguments of the execution is a set of programs written by students of the course on assembly languages.

3.3 Generic Policy Impact

We consider here the impact of the chosen policy on the amount of subtype tests compiled according to the policy, and performed at runtime. We first compare the number of subtype tests compiled by both engines including implicit and explicit tests. Then we execute the compiled program and count the number and the kinds of tests performed during execution. Figure 6 gives the number of tests or casts performed at runtime.

With `nitg-s`, an important proportion of all tests are generated by the compiler to ensure safety in generic covariant calls. Since the NIT specification does not allow covariance on a parameter of a method types with a resolved type, all implicit tests are against unresolved generic types.

`nitg-e` requires more subtype tests than `nitg-s` because of erasure casts. This makes sense, because reads are more numerous than writes in real programs. Only a few tests on unresolved types are required, which is because unresolved types for `nitg-e` consist only of type tests against virtual types, all other types being erased.

Metrics for `nitg-eu` and `nitg-su` can be derived by removing all the implicit tests while keeping the explicit ones.

3.4 Results and Discussion

The experimental protocol used is the same than for micro-benchmarks (Section 2.2). As shown in figure 7, without any consideration of generic policy, performances are comparable between all engines. `nitg-e` performs better than `nitg-s` in almost all cases, but only for a maximum of 5.8% in `nit`. This behavior can be explained because while `nitg-e` has the most dynamic type tests, those are essentially on resolved types. However, in `shoot`, there is so much erasure tests that `nit-e` is 9,5% slower than `nitg-s`.

On the unsafe versions for `nitg-e` and `nitg-s`, we can conclude that the cost of the checks are not a major overhead. For `nitg-su`, it goes from 1.4% in `shoot` to 9.1% in `bintrees`. Since the latter case is the least realistic program, this is not a major concern.

4. CONCLUSION

In this paper, we compare the cost of the generics policy chosen by a language and analyze its impacts on the subtype test performances. To make this comparison we create two compilers for the NIT language: `nitg-s` applying covariant policy using a homogeneous implementation. `nitg-e` applying an erased policy with erased implementation.

We compare the time efficiency of our solution in NIT against engines for C++, C#, EIFFEL, JAVA and SCALA. Results show that our implementations give the best performances compared to existing solutions. Subtyping test implementations based on function calls are slower than switches or tables based implementations. Unlike our implementations, not all the other implementations of the subtype test are time constant with generics.

We compare covariant and erased policies on existing NIT programs using `nitg-s` and `nitg-e`. We first look at the number of subtype tests compiled by both engines including implicit and explicit tests and the number of tests actually executed. Results show that erased `nitg-e` requires more subtype tests than covariant `nitg-s` because of erasure casts. We then look at time efficiency between the two engines and results show that covariance does not imply significant overhead regarding subtype test performances.

Due to the small difference between the costs of the different policies, the choice of a policy by a language designer should not be influenced by performance considerations.

5. REFERENCES

[1] H.-J. Boehm and M. Spertus. Garbage collection in the next C++ standard. In *Proceedings of the 2009 international symposium on Memory management*, ISMM '09, pages 30–38, New York, NY, USA, 2009. ACM.

[2] C. Click and J. Rose. Fast subtype checking in the HotSpot JVM. *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande - JGI '02*, pages 96–107, 2002.

[3] R. Ducournau. Coloring, a versatile technique for implementing object-oriented languages. *Software: Practice and Experience*, 41(6):627–659, May 2011.

[4] R. Ducournau, F. Morandat, and J. Privat. Empirical assessment of object-oriented implementations with multiple inheritance and static typing. *ACM SIGPLAN Notices*, 44(10):41–60, Oct. 2009.

[5] R. Garcia, J. J. Arvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17(2):145–205, 2007.

[6] J. Gelinas, E. Gagnon, and J. Privat. Prevention de dereferencement de references nulles dans un langagea objets. *Langages et Modèles à Objets*, Volume L-3:5–16, 2009.

[7] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common language runtime. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation - PLDI '01*, pages 1–12, New York, New York, USA, 2001. ACM Press.

[8] J. Privat and R. Ducournau. Raffinement de classes dans les langages a objets statiquement types. *RSTI-L'OBJET*, 2005.

[9] M. Torgersen. Virtual types are statically safe. *In 5th Workshop on Foundations of Object-Oriented Languages*, pages 1–9, 1998.

[3] `http://github.com/xymus/pep8analysis`

[4] Pep/8 is a virtual CISC processor designed to teach computer architecture and assembly language programming principles. `https://code.google.com/p/pep8-1/`

	nitg		nitg-s		nit		shoot		bintree		pep8analysis	
	res.	unres.	res.	unres	res.	unres	res.	unres.	res.	unres.	res.	unres.
explicit	153M	10M	269M	15M	54M	14M	38M	0	143	0	4.5M	17M
covariant	0	149M	0	201M	0	156M	0	5.8M	0	10M	0	98M
explicit	159M	0.7M	280M	0.3M	64M	3.1M	38M	0	143	0	14M	379
covariant	66M	0.08M	70M	0.1M	53M	0.08M	5.4M	0	70M	40M	58M	0.4M
erasure	956M	0.4M	131M	0.3M	140M	0.06M	134M	0	50	0	220M	0

Figure 6: Metrics on dynamic executed subtype tests. The first two lines are for nitg-s and the last three lines are for nitg-e. *explicit* corresponds to the number of explicit subtype tests and casts requested by the developer; *covariance* represents the covariance casts inserted by the compiler to ensure safety in covariant generic calls; *erasure* are casts added by the erasure compiler; *res.* corresponds to subtype tests against resolved types such as List<Integer> or Map<String, Integer>; *unres.* is for tests against open types [7] such as List<E> or E.

	nitg-s	nitg-e	nitg-su	nitg-eu
nitg	9.63	9.19	9.09	8.87
nitg-s	7.89	7.49	7.30	7.05
nit	6.87	6.47	6.45	5.84
shoot	7.80	8.55	7.69	7.78
bintree	11.57	11.47	10.51	10.48
pep8analysis	10.97	11.01	10.74	10.71

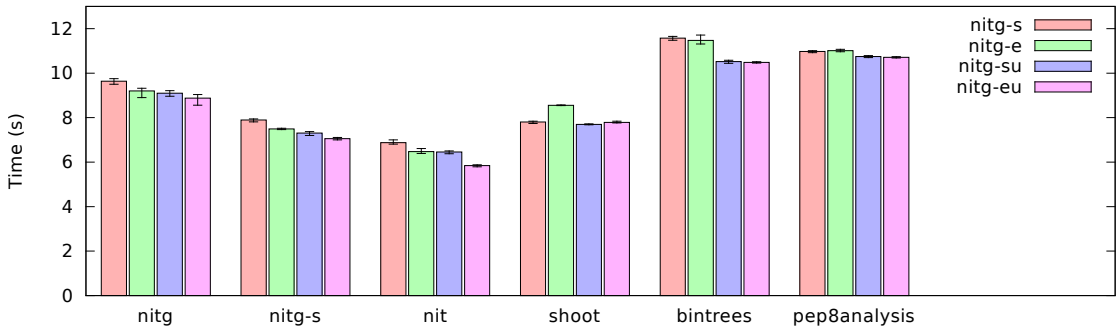

Figure 7: Time efficiency of compiled Nit programs according to the used generics policy.