This paper should be referenced as:

Kirby, G.N.C. & Morrison, R. “Variadic Genericity Through Linguistic Reflection: A Performance Evaluation”. In Proc. 8th International Workshop on Persistent Object Systems (POS8), Tiburon, California (1998).

Variadic Genericity Through Linguistic Reflection: A Performance Evaluation

G.N.C. Kirby and R. Morrison

School of Mathematical and Computational Sciences, University of St Andrews, North Haugh, St Andrews, Fife KY16 9SS, Scotland

Email: {graham, ron}@dcs.st-and.ac.uk

Abstract

The use of variadic genericity within schema definitions increases the variety of databases that may be captured by a single specification. For example, a class of databases of engineering part objects, in which each database instance varies in the types of the parts and the number of part types, should lend itself to a single definition. However, precise specification of such a schema is beyond the capability of polymorphic type systems and schema definition languages. It is possible to capture such generality by introducing a level of interpretation, in which the variation in types and in the number of fields is encoded in a general data structure. Queries that interpret the encoded information can be written against this general data structure.

An alternative approach to supporting such variadic genericity is to generate a precise database containing tailored data structures and queries for each different instance of the virtual schema.[1] This involves source code generation and dynamic compilation, a process known as linguistic reflection. The motivation is that once generated, the specific queries may execute more efficiently than their generic counter-parts, since the generic code is "compiled away". This paper compares the two approaches and gives performance measurements for an example using the persistent languages Napier88 and PJama.

1 Introduction

In some applications it may be necessary to access multiple databases, the schemas of which differ in their details but share some higher-level structure. In such cases it is convenient if the application code can be written in terms of that higher-level structure, with the differences abstracted over. One example is a CAD application that manipulates objects representing engineering parts, in which the types of the parts, the number of different part types, and the pattern of inter-connections among parts all vary between databases. Here the common structure is that each database contains inter-connected part objects, and it is useful for the application to be able to access any database with such a schema. We use the term *variadic genericity* to denote this property: *generic* since the application may access databases of any conforming schema, and *variadic* since the number of types and inter-connections may vary.

Schema definition languages can capture individual schemas but not such variadicity. Polymorphic type systems, which are designed to express genericity, can only capture variadic genericity to a limited extent. For example, parametric polymorphism [Str67] does not capture it at all—since a fixed number of types are abstracted over—and inheritance [Car84] can only capture the special case where the schemas are in the inclusion relation.

Given that such an application should be able to operate with any database that shares the virtual schema, this leads to the problem of how the set of databases should be designed. One approach is to represent all databases using a single sufficiently general data structure. Another is to automate the definition of precise schemas and queries for each database, and to provide a common API through which the databases may be accessed by the application. The first approach has the advantages of simplicity and flexibility, while the second allows greater type precision and, potentially, performance benefits. This paper outlines how an example application can be constructed using the two

[1] We use the term *virtual schema* here to indicate that only concrete versions of the schema exist and that they all share a common form not definable in the language.

approaches, and assesses whether the expected performance gains are realised using the automated precise definition approach in the persistent programming languages Napier88 [MBC+96] and PJama [ADJ+96].

The programming technique used to implement the variadic genericity in the second approach is called linguistic reflection [SSS+98]. This may be defined as the ability of a running program to generate new program fragments and to integrate these safely into its own execution. The technique involves calling a compiler dynamically to compile newly generated program fragments, and using a linking mechanism to bind these new program fragments into the running program. The process may be recursive, since the generated source fragments may contain reflective code that causes further reflection when it is executed.

Linguistic reflection provides the ability to implement highly abstract (generic) specifications, such as those used in query languages and data models, using a meta-level description of types within a strongly typed programming language. It thus extends the data modelling of the type system—in this case to support variadic genericity. A related use of linguistic reflection to generate tailored relational data representations is described in [CAD+87].

The provision of linguistic reflection in Napier88 and Java (and hence PJama) are discussed in [Kir92] and [KMM97, KMS98] respectively. Here we focus on a performance evaluation of linguistic reflection in providing variadic genericity. Section 2 now describes an example application using variadic genericity, followed by the results of experiments comparing the performance of the two implementation approaches in Section 3.

2 The Motivating Application

2.1 The Virtual Schema

The example used in this paper is that of a group or class of object databases, each instance of which contains a set of inter-connected objects representing engineering parts. The schemas of the individual databases differ, but they all conform to the following informal virtual schema definition:

> *A database contains a set of part objects which are instances of* n *types named* Part1 *to* Part*n. All of these types share a number of common scalar fields. Each type also contains a number of individual scalar fields and reference fields. Each reference field refers to a set of instances of a different part type.*

Thus individual database schemas may vary in the following aspects:

- the number of part types;
- the number and types of the scalar fields of each part type;
- the number and types of the reference fields of each part type.

Figure 1 shows an example individual schema in Java, conforming to the virtual schema. Here the common scalar fields are grouped into a superclass *Part* from which the specific part types inherit. The sets of references in each specific part class are represented as arrays of the appropriate class. Finally, the root of the database is an instance of class *DB*, containing an array of instances of a particular part class, *Part1*. The diagram shows a small example instance of the schema, with a single instance of *Part1* in the root array. Scalar fields are indicated by dots, and references to empty arrays by diagonal lines.

```
public class DB {
       public Part1[] root; }

public class Part {
       public int id;
       public int modifyDate;
       public String info; }

public class Part1 extends Part {
       public int scalar1;

       public Part2[] refs1;
       public Part3[] refs2;
       public Part4[] refs3; }

public class Part2 extends Part {
       public Part1[] refs1;
       public Part3[] refs2; }

public class Part3 extends Part {
       public String scalar1;
       public int scalar2;

       public Part2[] refs1; }

public class Part4 extends Part {
       public float scalar1; }
```

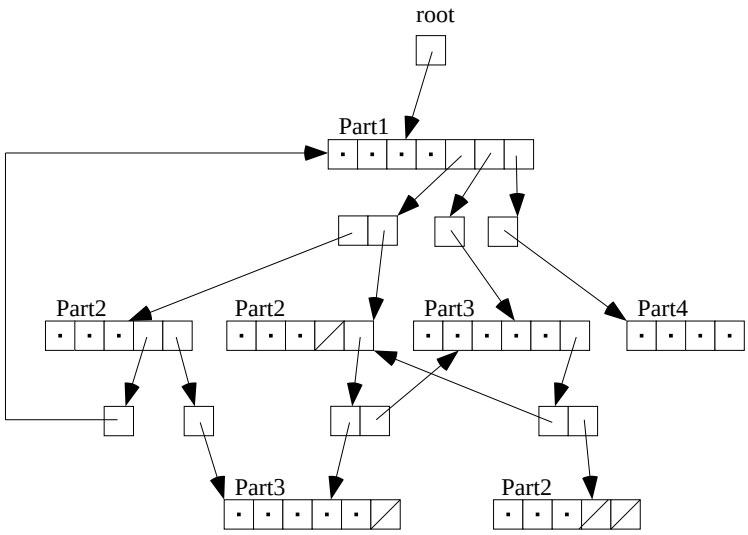

Figure 1. Example schema and database instance conforming to the virtual schema

The following queries and updates are defined for all instances of the virtual schema:

Query 1: traverse all parts; return the number of parts read.
Query 2: traverse all parts; return the sum of all non-common integer fields (those fields not in *Part*).
Query 3: find all the instances of *Part2* modified since a given date.

Update 1: traverse all parts and update the common field *modification date*; return the number of updates made.
Update 2: traverse all instances of *Part3* and in each insert a reference to a new instance of *Part2*; return the number of updates made.

Although these queries and updates are applicable to all databases conforming to the virtual schema, they differ in the degree to which they operate on particular part types. For example, Query 1 accesses all part instances regardless of their types, whereas the actions of Query 2 and Query 3 vary depending on the details of the specific part types. This illustrates why inheritance on its own is not sufficient to model the required variadic genericity: that would only allow queries to access the fields common to all part types.

Figure 2 shows a possible simple implementation of Query 2 for the schema of Figure 1. The superclass *Query*, omitted to save space, initialises the variable *db* to refer to the root array of *Part1* instances, and defines the methods *visited* and *remember* which keep track of which parts have been encountered previously.

```
public class Query2 extends Query {
	private int count = 0;

	public Object runQuery () {
		count = 0;
		for (int i = 0; i < db.root.length; i++) { travPart1 (db.root[i]); }
		return new Integer (count); }

	private void travPart1 (Part1 p) {
		if ( visited (p) ) return;
		remember (p);
		count += p.scalar1;
		for (int i = 0; i < p.refs1.length; i++) { travPart2 (p.refs1[i]); }
		for (int i = 0; i < p.refs2.length; i++) { travPart3 (p.refs2[i]); } }

	private void travPart2 (Part2 p) {
		if ( visited (p) ) return;
		remember (p);
		for (int i = 0; i < p.refs1.length; i++) { travPart1 (p.refs1[i]); }
		for (int i = 0; i < p.refs2.length; i++) { travPart3 (p.refs2[i]); } }

	private void travPart3 (Part3 p) {
		if ( visited (p) ) return;
		remember (p);
		count += p.scalar2;
		for (int i = 0; i < p.refs1.length; i++) { travPart2 (p.refs1[i]); } }
}
```

Figure 2. Query 2 code specialised for the example schema

2.2 Generic Database Representations

Consider a generic application that can use any database conforming to the virtual schema. Since the variation between schemas is too great to capture using polymorphism, the simplest way to implement such an application is to define a class or type that is sufficiently general to represent any instance of the virtual schema. Queries and updates are written to iterate over this general representation, and will thus work for any conforming database.

One possible general representation is the class *GenericSchema* shown in Figure 3. The arrays *commonFieldNames* and *commonFieldTypes* represent the names and types of the fields common to all part types. The array *partScalarFieldTypes* represents the types of the scalar fields in each part type. Each element is itself an array containing the types for a particular part type. The array *partCrossRefs* represents the inter-connections between part types: each element corresponds to a particular part type, and is itself an integer array. That array contains an element for each reference field of the part type, recording the index of the part typepointed to by that field. Assuming that the fields in each part type are named systematically *scalar1…scalar*n and *refs1…refs*m, there is no need to represent the field names. Finally, the integer *rootTypeNo* represents the particular part type of the persistent root array elements.

```
public class GenericSchema {
	public String[]  commonFieldNames;
	public Class[]   commonFieldTypes;
	public Class[][] partScalarFieldTypes;
	public int[][]   partCrossRefs;
	public int       rootTypeNo }
```

Figure 3. Generic representation of schemas

Figure 4 shows an example instance of *GenericSchema* that represents the particular schema shown in Figure 1. The arrays *afn* and *aft* represent the fields of *Part*. The two-dimensional array *psft* represents the types of the scalar fields of each part type: for example, its third element contains representations of the types *String* and *int*, corresponding to

the fields *scalar1* and *scalar2* of *Part3*. The two-dimensional array *pcr* represents the types of the cross-references between part type: for example, its second element contains 1 and 3, corresponding to the element types (*Part1* and *Part3*) of the reference fields of *Part2*.

```
Class StringTYPE = "".getClass();
String[]  afn = { "id", "modifyDate", "info" };
Class[]   aft = { Integer.TYPE, Integer.TYPE, StringTYPE };
Class[][] psft  { {Integer.TYPE}, {}, {StringTYPE, Integer.TYPE}, {Float.TYPE} };
int[][]   pcr = { {2, 3, 4}, {1, 3}, {2}, {} };

GenericSchema gs = new GenericSchema (afn, aft, psft, pcr, 1);
```

Figure 4. A particular schema representation

Given the requirement for a general representation, it is not possible to define specific classes for the various part types. Instead, the single class *GenericPart*, shown in Figure 5, is used to represent all parts. The field *typeNo* is a tag indicating the part type; the arrays *commonFieldValues* and *partScalarFieldValues* represent the inherited and local scalar field values as instances of the general class *Object*; the array *partRefFieldValues* contains the cross-references to other parts.

```
public class GenericPart {
       public int             typeNo;
       public Object[]        commonFieldValues;
       public Object[]        partScalarFieldValues;
       public GenericPart[][] partRefFieldValues }
```

Figure 5. Generic part type

A complete database is represented as an instance of the class *GenericDB* shown in Figure 6. This contains a reference to a particular virtual schema instance and to an array of generic part objects. Queries and updates operate on instances of *GenericDB*.

```
public class GenericDB {
       public GenericSchema genericSchema;
       public GenericPart[] root }
```

Figure 6. Generic database instance

Figure 7 shows a possible implementation of Query 2 using the general representation. In order to avoid having to interpret the type of every scalar field of every object as it is encountered, the query constructor builds the bitmap *intMap* recording whether or not the fields of each type hold integers, based on the *GenericSchema* instance passed to it. This bitmap is then used during traversal of the part objects to determine which fields should be read and added to the accumulated sum.

```
public class Query2 extends Query {
    private boolean[][] intMap;
    private int count = 0;

    public Query2 (GenericDB db) {
        // Build bit-map of integer fields.
        ... }

    public Object runQuery () {
        count = 0;
        for (int i = 0; i < db.root.length; i++) { travPart (db.root[i]); }
        return new Integer (count); }

    private void travPart (GenericPart p) {
        if ( visited (p) ) return;
        remember (p);
        boolean[] map = intMap[p.typeNo];
        for (int i = 0; i < p.partScalarFieldValues.length; i++) {
            if (map[i]) count += ((Integer) p.partScalarFieldValues[i]).intValue(); }

        for (int i = 0; i < p.partRefFieldValues.length; i++) {
            for (int j = 0; j < p.partRefFieldValues[i].length; j++) {
                travPart (p.partRefFieldValues[i][j]); } } }
}
```

Figure 7. Generic implementation of Query 2

A generic database representation of this form has the advantage of flexibility. It is simple to introduce new part types by creating suitable instances of *GenericPart*. Since the part types are not represented by specific types, it is also straight-forward to evolve the "types" of existing part objects by updating their fields appropriately.

This form of genericity also has several disadvantages. The database contents are less strongly typed than with the specific representation shown in Figure 1, in the sense that the part types are encoded in the *GenericPart* values rather than in the type definitions. This means that errors such as attempting to assign a part of the wrong part type into a reference field can only be detected by dynamic checks coded explicitly by the application programmer, rather than being detected statically.

Secondly, the storage of the database in the generic form results in additional costs during query execution, compared with the specific representation. This is because a query must determine which type each part object belongs to as it is encountered, by examining its *typeNo* tag, and act accordingly. In contrast, with the specific representation the query contains separate code specific to each part type and there is no need for dynamic checking.

2.3 Genericity Through Linguistic Reflection

The previous section showed one way to achieve genericity for applications that need to operate on multiple databases conforming to the virtual schema. An alternative approach is to use the technique of linguistic reflection. This involves the automatic generation of appropriate specific class and query definitions, based on the specification of a particular schema [KMS98, SSS+98]. For example, from the schema representation *gs* of Figure 4, the class definitions of Figure 1 would be generated, as would the source code of the queries and updates to operate over them.

The advantage of this approach is that applications may access multiple databases with different schemas without each one having to be coded separately, while the data within each database is held in a specifically typed form. Thus more errors may be detected statically and queries may operate more efficiently than with the general storage form. Effectively the genericity is moved from the database itself to the code that generates it.

There are of course disadvantages to using linguistic reflection in this manner. The principal difficulty is an increase in complexity in the application, which is now concerned with the generation of new program code. Another problem is that changing the types of existing part objects becomes more difficult due to the stronger typing. For applications where these factors are considered acceptable, however, the use of linguistic reflection may bring significant

performance benefits. The next section presents some measurements comparing two generic implementations of an example database, one using a general representation and one using linguistic reflection, for the persistent programming languages Napier88 and PJama.

3 Experiments

3.1 Overview

The experiments compare two alternative implementations of an example generic application, the parts database described in the previous section. One implementation uses linguistic reflection to achieve genericity, and one a non-reflective technique. A non-generic implementation was also measured, to provide a base-line against which to assess the genericity costs. Each experiment was performed in both Napier88 and PJama.[2]

In the linguistic reflective implementation, a general type to represent any conforming schema was defined, as illustrated in Figure 3, and a generator used to generate specific types and queries as needed, from a schema representation. In the non-reflective implementation, general types as illustrated in Figure 5 and Figure 6 were used to represent the database and its parts, and queries were written to operate over those general types. Finally, the non-generic implementation, giving the base-line, defined types and queries to represent one specific database schema, with no attempt to accommodate alternative schemas, as illustrated in Figure 1.

The times to generate a database and to execute the queries and updates described in section 2.1 were measured in each case. By referring to the non-generic implementation, the costs of providing genericity by the two alternative mechanisms were compared. It should be emphasised that the experiments were conducted to compare the genericity mechanisms and were not concerned with absolute efficiency—for example, no effort was made to optimise the implementations of the queries.

Where a database part object referred to a set of other parts, that set was chosen by selecting between 0 and 3 parts from the set of all existing parts of the appropriate type. The number of parts and their identity were determined by a pseudo-random number (PRN) generator, using the same initial seed for each test. The PRN generator used in the Napier88 tests was different from that used in the PJama tests, thus the connectivity of all the Napier88 databases was the same, but differed from that of all the PJama databases. Furthermore, since the Napier88 tests were conducted on a different platform from the PJama tests, as described in section 3.2, no direct comparison of the performance of the two languages can be made. For this reason, the timing figures plotted in the graphs in section 3.3 are normalised relative to the times taken to generate the smallest non-generic database on each platform, allowing the relative performance of the various implementation techniques to be compared for each platform separately.

3.2 Napier88 and PJama Platforms

The Napier88 timings were obtained using Napier88 release 2.2.1 on a 50 MHz Sun SPARC 10 with 64 MB RAM running SunOS 4.1.3. The Napier88 interpreter and store were held on a 2.1 GB Seagate Barracuda disk. Although unloaded the machine was running multi-user and was connected to the network. The standard release "compiler only" store image was used, with the Napier88 parameter *NPRHEAP* (object cache size) set to 32 MB. The timing figures were obtained by instrumenting the programs with calls to the Napier88 standard library procedure *time*, which returns the Napier88 system elapsed time with 60 Hz resolution.

The PJama timings were obtained using PJama release 0.4.6.12 on a 167 MHz Sun Ultra 1 with 128 MB RAM running Solaris 2.6 in single user mode. The PJama interpreter and store were held on a 4.2 GB Sun fast wide SCSI disk. The following Java/PJama parameters were set: *-mx32m -oss2m -ocs32m -bfs16m* (heap size 32MB; maximum thread stack size 2MB; object cache size 32MB; page buffer pool size 16MB). The timing figures were obtained by instrumenting the programs with calls to *System.currentTimeMillis()*, which returns the time with 1 KHz resolution.

---

[2] Source code and raw timing data are available on the web at: *http://www-ppg.dcs.st-and.ac.uk/Research/LinguisticReflection/*

3.3 Methodology and Results

Each figure plotted in the following charts is the mean of 5 consecutive measurements, each of which was obtained using a fresh instance of the virtual machine. For each cold timing the query or update operation was performed directly after VM initialisation, whereas for each warm timing the operation was performed twice and the results of the first run discarded. The following sequence was performed for every combination of platform, implementation and operation:

```
for i = 1 to 5 do {
    initialise VM
    perform operation and record time elapsed
}

take mean of 5 times to give cold measurement
```

```
for i = 1 to 5 do {
    initialise VM
    perform operation and discard time elapsed
    perform operation again and record time elapsed
}
take mean of 5 times to give warm measurement
```

Note that the times recorded include the time taken to stabilise the persistent store after performing the operation, but not the time to initialise the VM on start-up.[3] In each of the following charts the three connected series of points show the times for the various implementations: *non-reflective*, *reflective* and *specific.* As explained earlier, the times, plotted on the left vertical axes, are normalised relative to the generation time for the smallest non-generic database, which contained 100 parts—this time was 0.15s for Napier88 and 0.49s for PJama. The horizontal axes show the number of parts in each database. The non-connected points, plotted against the right vertical axes, show the percentage speed-up exhibited by the reflective implementation over the non-reflective implementation, calculated as *((non-reflective / reflective) - 1) * 100*.

Figure 8 and Figure 9 show the normalised times taken to generate a database of various sizes; these include the time taken to generate, compile and load the new source code. For small numbers of parts, the initial database generation costs of the reflective implementation are much greater than those of the non-reflective implementation. This reflects the overheads of generating and compiling the database representation types and queries; these overheads appear to be relatively greater for Napier88 than for PJama. As the size of the database increases the difference becomes less marked and eventually the reflective implementation becomes more efficient. This is because the generation and compilation costs remain constant, while the saving due to the greater efficiency of the generated code increases with database size. The break-even points are at around 60,000 parts for Napier88 and 10,000 for PJama.

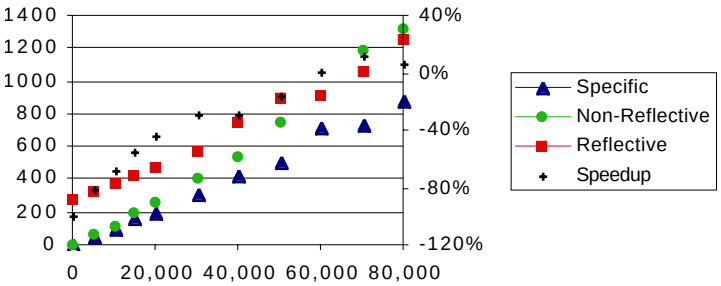

Figure 8. Generation in Napier88

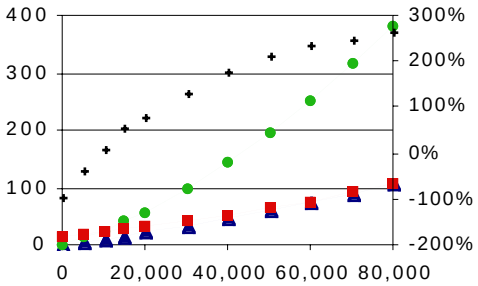

Figure 9. Generation in PJama

The queries and updates can be categorised into two groups. Query 1 (count all parts) and Update 1 (touch all parts) both perform a particular operation on all part objects regardless of their type. Query 2 (sum all non-inherited integer fields), Query 3 (find all recently modified *Part2* instances) and Update 2 (insert a new reference into every *Part3* instance) access only certain part objects, depending on their type. There are also two separate factors that may affect the relative performance of the reflective and non-reflective implementations. Firstly, relevant to all queries and updates, the part types generated by the reflective implementation produce a smaller database representation than that of the non-reflective implementation (as described in the next section), resulting in less data being faulted in. Secondly, where the operation of the query or update depends on the part types, the necessary checks are factored

[3] Stabilisation costs are an intrinsic part of the timings for updates. They were also included in the query timings for simplicity. For all but the smallest databases stabilisation was insignificant compared to the overall costs—see the raw data on the web for details.

out of the generated source code. Further, in the generated queries and updates some inter-object references need not be traversed, where it is known statically that they lead to parts of non-relevant types.

Figure 10 and Figure 11 show the normalised times to execute Query 1 in Napier88 and PJama. In each case the cold times are shown in the chart on the left, and the warm times on the right. Napier88 shows no appreciable difference between any of the implementations, whereas PJama shows considerable speed-ups for the reflective implementation over the non-reflective in the cold case, and a small but consistent gain for the reflective implementation in the warm case. Since Query 1 accesses all part objects, any differences between the reflective and non-reflective implementations are due solely to the size of the database representation. The timings thus indicate that the object faulting costs relative to query execution costs are less significant in the Napier88 case than in the PJama case, even though, as described in the next section, the saving in database size using reflection is actually greater for Napier88. Note also the high normalised execution times in Napier88 compared to PJama.

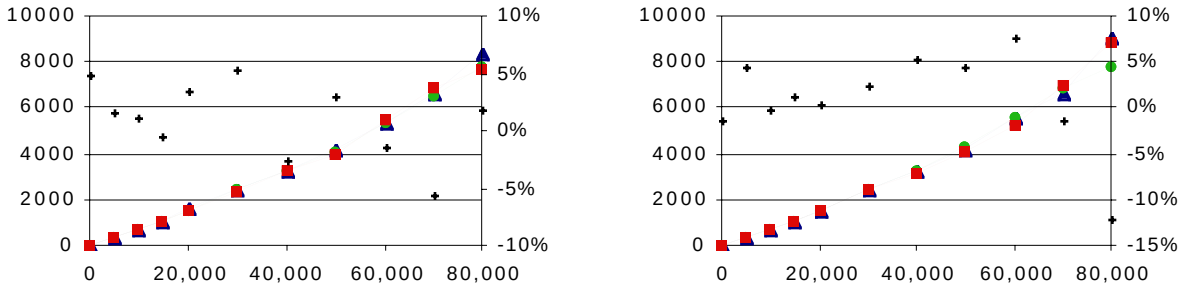

Figure 10. Query 1 in Napier88

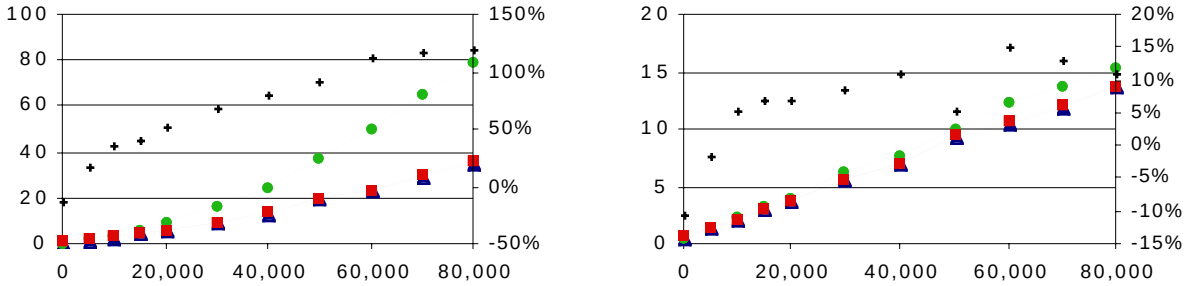

Figure 11. Query 1 in PJama

Figure 12 and Figure 13 show the times for Query 2. The Napier88 times show consistent speed-ups of around 30-40% for the reflective implementation in both cold and warm cases, these times being practically indistinguishable from the non-generic implementation. On the basis of the Query 1 times, the savings over the non-reflective implementation are likely to be due mainly to the greater efficiency of the generated query: all instances of *Part4* can be ignored, since they contain neither integer fields nor references to part of other types. The PJama warm times show similar savings, while in the cold case the speed-up exceeds 200% for the larger databases. This indicates again that the object faulting costs relative to query processing are more significant in the PJama case.

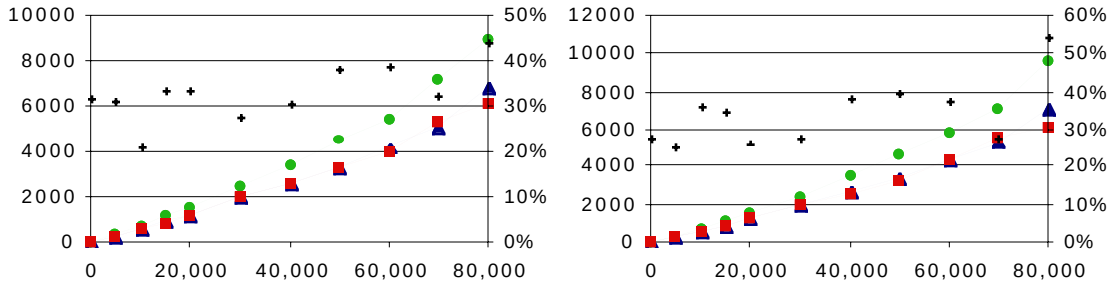

Figure 12. Query 2 in Napier88

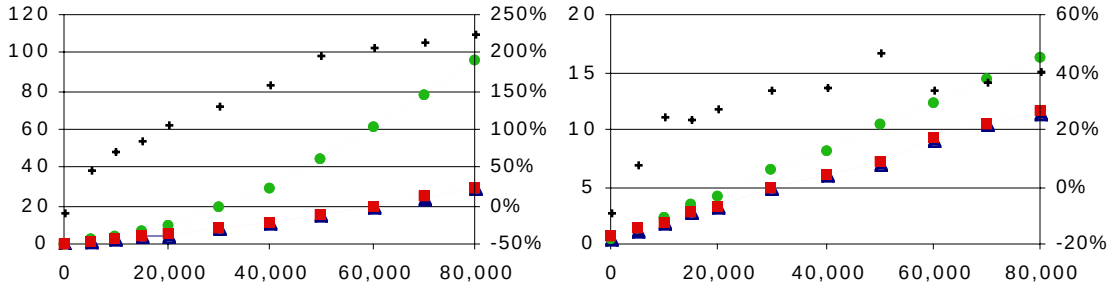

Figure 13. Query 2 in PJama

Figure 14 and Figure 15 show a similar pattern for Query 3, as might be expected since this query also accesses only a subset of the part objects depending on their type.

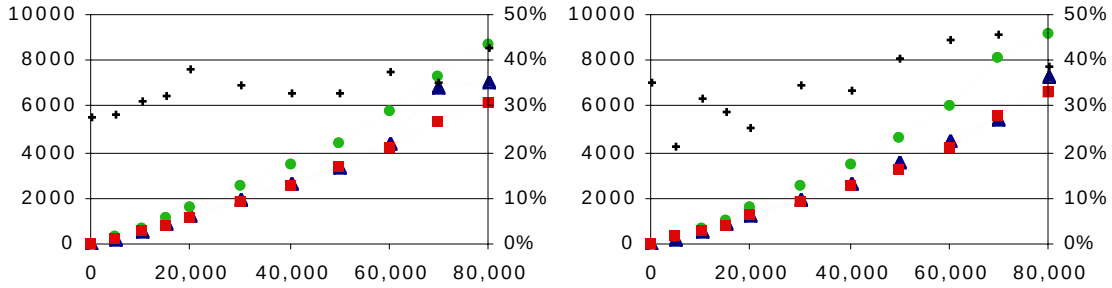

Figure 14. Query 3 in Napier88

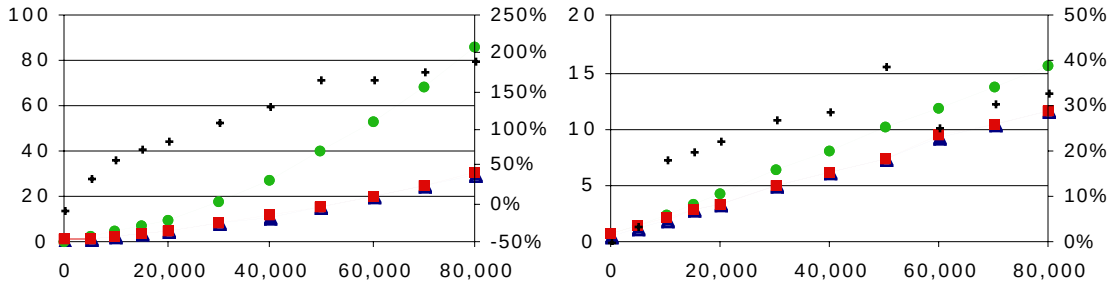

Figure 15. Query 3 in PJama

Figure 16 and Figure 17 show the times for Update 1. These exhibit a similar pattern to that of Query 1: there is essentially no difference among the Napier88 implementations, whereas the reflective PJama implementation shows a significant gain over the non-reflective implementation in the cold case, increasing with database size, and a small gain in the warm case. As noted earlier, Update 1 is similar to Query 1 in that all part objects are accessed, so differences are due to the sizes of the databases rather than any difference in the algorithms.

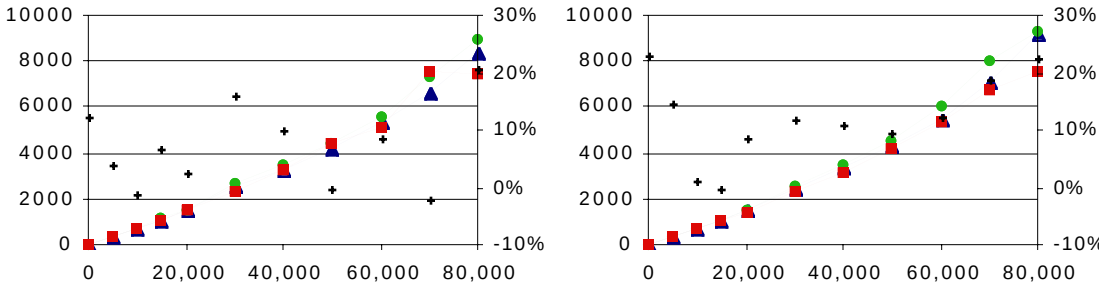

Figure 16. Update 1 in Napier88

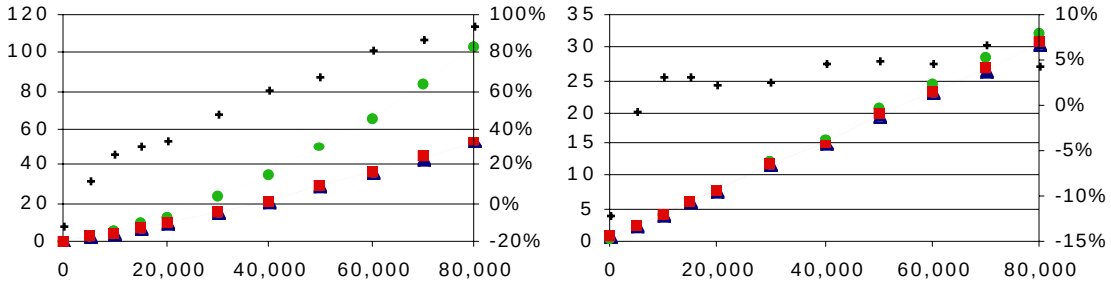

Figure 17. Update 1 in PJama

Finally, Figure 18 and Figure 19 show the times for Update 2, the pattern of which is similar to those of Query 2 and Query 3. The reflective Napier88 implementation shows gains of around 30% over the non-reflective implementation in both cold and warm cases. For PJama the gains range from 50% to 170% in the cold case, and remain around 100% for most database sizes in the warm case. Like Query 2 and Query 3, the actions of Update 2 depend on the part types, so there is scope for greater efficiency in the reflectively generated algorithm. Note the relatively larger gains in the PJama warm case, compared to those of Query 2 and Query 3. This is due to the new and updated objects being committed to the persistent store in Update 2, whereas no objects were updated in the queries, thus the smaller database size gives an advantage even in the warm case.

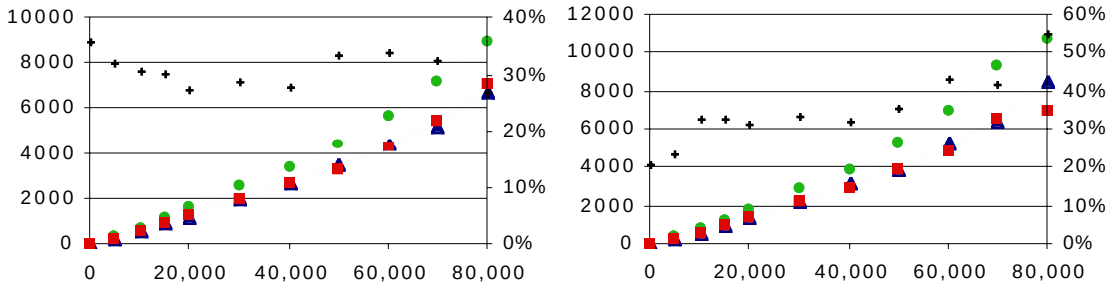

Figure 18. Update 2 in Napier88

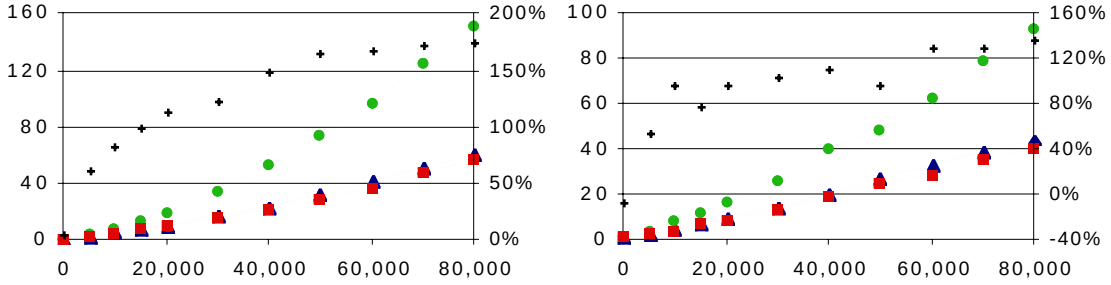

Figure 19. Update 2 in PJama

3.4 Space Issues

The non-reflective and reflective implementations differ in the number and size of the objects used to represent the database parts. Knowledge of the object formats used in the Napier88 [BCC+94] and PJama implementations can be used to derive the following estimates of the number of words required for a database of *n* parts, where *c* is the average number of cross-references of each type, and *s* is the average string length in characters. For simplicity, an equal number of each part type is assumed.

Napier88:	**non-reflective**:	(46 + 1.5c + 0.31s)n
	reflective:	(24 + 1.5c + 0.31s)n

PJama:	**non-reflective**:	(30 + 1.5c + 0.31s)n
	reflective:	(20 + 1.5c + 0.31s)n

Since each expression contains a constant factor, the difference in storage used by the non-reflective and reflective implementations is greatest when the average connectivity and string size is low. At the limit the non-reflective implementation requires around twice as much storage for Napier88, and 1.5 times as much for PJama.

3.5 Further Work

The example application exhibits several degrees of freedom, which could be explored more fully. One is the implementation of the queries and updates: a simple recursive implementation was used in the experiments whereas a more complex iterative algorithm might be more efficient. Other variables include the database sizes, the degree of connectivity between parts, the relative distribution of part types, and the various VM settings.

More significantly, the application measured here is highly artificial and it is unclear how far if at all the benefits of using linguistic reflection demonstrated here will extend to more realistic applications, particularly in the presence of query optimisation.

4 Conclusions

Linguistic reflection has previously been discussed by the authors and others as a programming technique that is interesting in its own right, and which can be used to support software evolution and genericity. Here, however, we have focused on a particular application area, object databases that exhibit variadic genericity, and presented linguistic reflection as an implementation mechanism that out-performs the alternatives on certain workloads.

We implemented versions of an example application using three different representations: a precisely typed representation generated using linguistic reflection; a more loosely typed non-reflective representation; and a precisely typed, non-generic representation. The non-generic implementation was used as a base to indicate the intrinsic costs of the algorithms used; the relative costs of the two genericity mechanisms were then compared for both Napier88 and PJama. The reflective implementations performed equivalently to or better than the non-reflective implementations in all cases apart from initial database generation in Napier88. The greatest improvement was for a cold run of a type-dependent read-only query in PJama (Query 3), which gave speed-ups of over 200% for the larger databases.

It is not clear how widely applicable linguistic reflection is to real problems. We may conclude though that it is worth considering in cases where the following hold:

- the persistent data is naturally represented by a relatively complex graph structure;
- there is a requirement for genericity; and
- the speed of operations on the data, and the storage occupied by it, is more important than implementation costs and the speed of initial generation of the data structure.

5 Acknowledgements

This work is partially supported by the EPSRC through Grant GR/L32699 “Compliant System Architecture” and by ESPRIT through Working Group EP22552 “PASTEL”. We thank Laurent Daynès and Mick Jordan for their help with PJama, and Dave Munro and the referees for their helpful suggestions.

6 References

[ADJ+96] Atkinson, M.P., Daynès, L., Jordan, M.J., Printezis, T. & Spence, S. “An Orthogonally Persistent Java™”. SIGMOD Record 25, 4 (1996) pp 68-75.

[BCC+94] Brown, A.L., Carrick, R., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C., Morrison, R. & Munro, D.S. “The Persistent Abstract Machine Version 10 / Napier88 (Release 2.0)”. Universities of St Andrews and Adelaide (1994).

[CAD+87] Cooper, R.L., Atkinson, M.P., Dearle, A. & Abderrahmane, D. “Constructing Database Systems in a Persistent Environment”. In Proc. 13th International Conference on Very Large Data Bases (1987) pp 117-125.

[Car84] Cardelli, L. “A Semantics of Multiple Inheritance”. In **Lecture Notes in Computer Science 173**, Kahn, G., MacQueen, D.B. & Plotkin, G. (ed), Springer-Verlag, Proc. International Symposium on the Semantics of Data Types, Sophia-Antipolis, France, Goos, G. & Hartmanis, J. (series ed) (1984) pp 51-67.

[Kir92] Kirby, G.N.C. “Reflection and Hyper-Programming in Persistent Programming Systems”. Ph.D. Thesis, University of St Andrews. Technical Report CS/93/3 (1992).

[KMM97] Kirby, G.N.C., Morrison, R. & Munro, D.S. “Evolving Persistent Applications on Commercial Platforms”. In **Advances in Databases and Information Systems**, Manthey, R. & Wolfengagen, V. (ed), Springer-Verlag, Proc. 1st ACM SIGMOD East-European Symposium on Advances in Databases and Information Systems, St Petersburg, Russia, In Series: Electronic Workshops in Computing, van Rijsbergen, C.J. (series ed), ISBN 5-7940-0004-X (1997) pp 170-179.

[KMS98] Kirby, G.N.C., Morrison, R. & Stemple, D.W. “Linguistic Reflection in Java”. Software—Practice & Experience 28, 10 (1998) pp 1045-1077.

[MBC+96] Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C. & Munro, D.S. “Napier88 Reference Manual (Release 2.2.1)”. University of St Andrews (1996).

[SSS+98] Stemple, D., Stanton, R.B., Sheard, T., Philbrow, P., Morrison, R., Kirby, G.N.C., Fegaras, L., Cooper, R.L., Connor, R.C.H., Atkinson, M.P. & Alagic, S. “Type-Safe Linguistic Reflection: A Generator Technology”. To Appear: The FIDE Book, Atkinson, M.P. (ed), Springer-Verlag (1998), Technical Report ESPRIT BRA Project 3070 FIDE FIDE/92/49.

[Str67] Strachey, C. **Fundamental Concepts in Programming Languages**. Oxford University Press, Oxford (1967).