View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by St Andrews Research Repository

Linguistic Reflection in Java: A Quantitative Assessment

Graham Kirby, Ron Morrison
School of Mathematical and Computational Sciences, University of St Andrews, North Haugh,
St Andrews KY16 9SS, Scotland
(email: {graham, ron}@dcs.st-and.ac.uk)

David Stemple
Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA
(email: stemple@cs.umass.edu)

Abstract

Linguistic reflection allows a running program to generate new program fragments and to integrate these into its own execution. The advantages of the technique include attaining high levels of genericity and accommodating system evolution. Here its use to provide generic programs in Java is compared for a particular example, natural join, against alternative implementation approaches.

Introduction

Linguistic reflection may be defined as the ability of a running program to generate new program fragments and to integrate these into its own execution. We have described elsewhere how this style of reflection may be provided in the compiled, strongly typed language Java and used as a paradigm for program generation [KMS98]. The motivation for this work comes from the desire for two advanced programming capabilities. The first is the ability to implement highly abstract (generic) specifications, using a meta-level description of types, within a strongly typed programming language. The second is the ability to accommodate some of the continual changes (evolution) in data-intensive applications without resorting to highly interpretative approaches or *ad hoc* restructuring methods. Both capabilities involve reflective access to the types (classes) of a system that is changing itself.

These motivations are particularly relevant in persistent systems since such systems allow data to retain its type structure throughout its lifetime. Our own experience brought to the Java programming context is with strongly typed run-time linguistic reflection in the persistent languages PS-algol [PS88] and Napier88 [MBC+96]. So although the work is described here in the context of standard Java, it has been applied to a persistent version of Java, PJama [ADJ+96], for maximum effect.

The motivating example, to write a strongly typed generic natural join function, is taken from persistent programming. The subtlety of the example is that the function cannot be exactly typed by conventional polymorphic type systems since the code for the function and its result type depend upon the types of the parameter values. The difficulty of typing such a generic function can be avoided by using linguistic reflection to generate specific, simply typed, functions on demand. Once generated the functions are compiled and bound to the executing program and applied.

Run-Time Linguistic Reflection

In run-time linguistic reflection, the executing program generates new program fragments in the form of source code, invokes a dynamically callable compiler, and finally links the results of the compilation into its own execution. The generation of the source code is most effective where the values and types of existing data are used to tailor the generated code to that data. This requires three distinct facilities:

- a meta-level mechanism that provides descriptions of the types of existing data;
- a compiler that can be invoked dynamically; and
- a dynamic incremental loader.

A generic relational natural join function provides our motivating example. This represents an abstraction over types that is beyond the capabilities of most polymorphic type systems. The details of the input types, particularly the names and domains of the attributes, significantly affect the algorithm and the output type of the function, determining:

- the result type;
- the code to test whether a given pair of tuples matches on the overlapping attributes; and

- the code to build a relation having tuples with the aggregation of attributes from both input relations but with only one copy of the overlapping attributes.

The type of a polymorphic natural join function might be approximated as:

$$\forall a.\forall b.(\text{relation } [a] \times \text{relation } [b] \rightarrow \exists c.\text{relation } [c])$$

where *c* is some type related to *a* and *b* by the rules of natural join

That is, the function takes as parameters two relations, with tuple types *a* and *b* respectively, and returns a third relation with a related tuple type *c* as a result. Type systems that can accommodate such functions are generally higher order and dynamically checked, since the relationship between the input types (which are unknown statically) and the output type precludes static checking.

The function cannot be written as a statically-typed polymorphic procedure since it requires knowledge of the type structure of *a*, *b* and *c* and they are explicitly abstracted over here. The linguistic reflection technique allows a meta-function to be written, the generic natural join, that can examine the structure of the input types. While this generic meta-function cannot be statically typed in general, for any one particular call of the generic meta-function the type structures *are* known. At the point of use of the natural join, the generic meta-function is called and given as input the type structure. It then computes a specific natural join function for these input types together with a call to the specific natural join. This generated code is then fed into the compiler dynamically for checking and code generation, then linked and executed.

Implementation of Generic Natural Join Using Linguistic Reflection

We now describe how the motivating example of generic natural join can be implemented in Java. The implementation involves two main parts:

- Selecting a representation of relations as Java classes. There is a choice between using a single class which is sufficiently general to represent all relational types, or using a different class for each different type. The second option is chosen here.
- Defining a class with a generic static method to perform the natural join operation on any pair of relations, using *Just-In-Time* generation of specific source code. Since the method accepts any relations, its two formal parameters and its result are typed as the general class *Object*, while its actual parameters on a call will have classes representing specific relational types.

The technique chosen for representing relations is to define a class for each relational tuple type, specifying the attributes of that type. Each relation is then represented as an array of tuples, each of which is an instance of the class. Although an ordering on the tuples is introduced which is not part of the relational semantics, this can be ignored.

The representation is illustrated in Figure 1, in which lines 1-20 define the class *Employee*. The four public methods *name()*, *title()*, *department()* and *jobId()* correspond to the attributes of the relation, and each relation instance is represented as an array of type *Employee[]*. The classes *Job* and *Salary* are defined similarly.

```
1   public class Employee {
2
3       private String name;
4       private String title;
5       private String department;
6       private int    jobId;
7
8       public String name()       { return name; }
9       public String title()      { return title; }
10      public String department() { return department; }
11      public int    jobId()      { return jobId; }
12
13      public Employee( String name, String title, String department, int jobId ) {
14
15          this.name =       name;
16          this.title =      title;
17          this.department = department;
18          this.jobId =      jobId;
19      }
20  }
21
22  // Attributes 'post', 'duties' and 'jobId'. Details omitted to save space.
```

```
23  public class Job {…}
24
25  // Attributes 'post' and 'salary'. Details omitted to save space.
26  public class Salary {…}
```

Figure 1. Classes representing relational tuple types

Figure 2 shows two invocations of the reflective generic natural join method. The relations are formed in lines 2-4, calling utility methods to read data from files into arrays of the appropriate types. An instance of the generic natural join class *NatJoin* is then created. In line 8 a call is made to the method *natJoin* which performs the natural join on the relations *emps* and *jobs*. The specific type of the resulting relation *employeesAndJobs* is not known statically, thus it is typed as *Object*. The contents of the relation are printed out by a call to the method *printRelation* in line 9; this method is implemented in a similar way to *natJoin* in that it inspects the type of the relation passed to it and then generates specific code to print out relations of that type. Finally, in lines 11-12 the result relation is joined with a third relation *salaries*, and the result of that join is printed out. Note that since the input and output types of *natJoin* are *Object*, the same method is used to perform both example joins which have different actual parametric types. Also, in this example the result types of the joins need not be specified statically.

```
1   try {
2       Employee[] emps =     loadEmployees( "employees.txt" );
3       Job[]      jobs =     loadJobs( "jobs.txt" );
4       Salary[]   salaries = loadSalaries( "salaries.txt" );
5
6       NatJoin natJoinPack = new NatJoin();
7
8       Object employeesAndJobs = natJoinPack.natJoin( emps, jobs );
9       natJoinPack.printRelation( employeesAndJobs );
10
11      Object empsJobsAndSalaries = natJoinPack.natJoin( employeesAndJobs, salaries );
12      natJoinPack.printRelation( empsJobsAndSalaries );
13  }
14  catch (Exception e) { System.out.println( "join failed: " + e.getMessage() ); }
```

Figure 2. Invoking linguistic reflective Natural Join

Measurements

To assess the utility of linguistic reflection, the reflective implementation of generic natural join described in the previous section is evaluated in comparison with three other implementation techniques. These other techniques are as follows:

tailored uses a non-generic function which operates over one specific pair of relation types, thereby fixing the input types at compilation time. Since no attempt is made to write generic code, this serves as a base for comparison with the other techniques. Any differences measured between this and the other, generic, functions thus represent overheads due to genericity.

interpretive uses a single general type for relations and a single generic function which operates over that relation representation. With this solution, all relations are represented as instances of the same class. This corresponds to conventional relational systems.

core reflective uses a generic function which operates over relations represented as typed arrays of tuple objects, in the same style as the linguistic reflective solution. The generic join function works by inspecting the types of the input relations, its subsequent actions depending on the structure of those types. The result type is passed to the function as a parameter. This solution uses a similar algorithm to the linguistic reflective solution, but executes it directly rather than generating new code that performs it.

The criteria compared are:

- the overall execution times for an example workload;
- the specific execution times for forming relation representations, generating, compiling and loading code where applicable;
- the relative code sizes; and
- the amortisation of generation and compilation costs over multiple executions for the same input types using persistent caching of intermediate results.

Due to lack of space only the overall execution times are given here. Details of the other comparisons are given in [KMS98]. The measurements are performed using the persistent Java system PJama Release 0.3.4.6* in the default configuration. The platform is a 110 MHz Sun SPARCstation-4 with 32 MB RAM and 68 MB swap space, running Solaris 2.5.1 in single user mode. The timings are obtained by inserting calls to the method *System.currentTimeMillis()* in the Java source code immediately before and after each operation to be measured. This method returns the current "wall clock" time with milli-second resolution.

To identify effects due to objects being brought into the PJama object cache, both cold and warm measurements are made. The cold timings correspond to the start of a new PJama invocation, whereas the warm timings are made after executing a number of iterations. Each individual cold measurement is obtained using a new invocation of the PJama abstract machine. The warm measurements are obtained by running the join 20 times in succession on the same invocation of the abstract machine, discarding the first 10 measurements and recording the following 10. Each timing figure shown in the tables is the mean of 10 measurements unless noted otherwise, with the standard deviation shown in brackets.

For each algorithm, timings are obtained for the natural join running on the same three data sets, each consisting of a pair of relations. The data sets all contain the same relational types (*Employee* and *Job*), and thus the same number of attributes (4 and 3 respectively), but vary in the number of tuples. Table 1 shows the number of tuples in each relation, and their overall sizes.

	relation 1		relation 2		result relation	
	tuples	size	tuples	size	tuples	size
join 1	100	3.2 KB	15	360 B	153	7.3 KB
join 2	1000	32 KB	150	3.6 KB	12612	605 KB
join 3	3000	96 KB	700	16.8 KB	175433	8.4 MB

Table 1. Sizes of relations used

The bar chart in Figure 3 shows an overview of the measured total mean cold execution times. The inset contains a magnified view of the bars for the *join 1* data set. For that data set—the smallest—the cost of the linguistic reflective solution without cacheing is around 10 times higher than any of the others, and 100 times higher than the tailored solution. For the largest data set, however, the two linguistic reflective solutions are cheaper than any of the other generic solutions, and fairly close to the tailored solution which represents the lowest cost attainable using the naive join algorithm.

* Further information on PJama is available at:
`http://www.sunlabs.com/research/forest/opj.main.html`

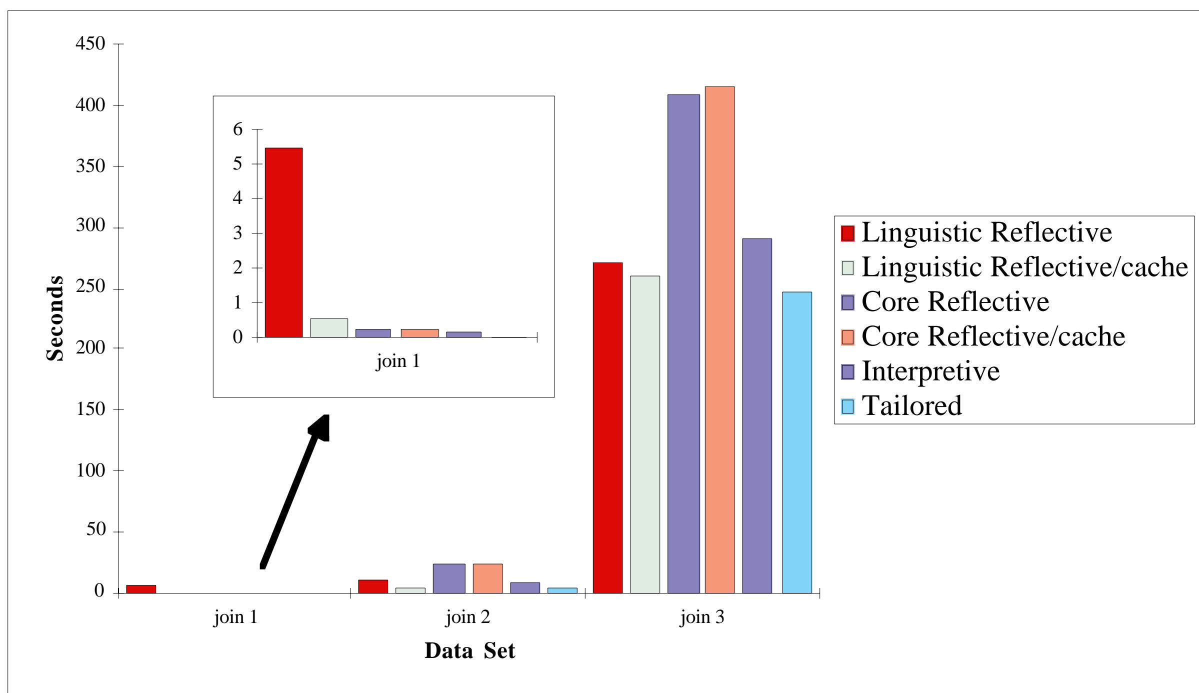

Figure 3. Overview of join execution times

In summary, the timing measurements compare the execution costs of a linguistic reflective implementation of a generic operation with those of a non-generic solution, and with an interpretive and core reflective generic solution. The results obtained indicate that linguistic reflection is expensive where the amount of computation performed by the generated code is small. As the amount of computation increases the overheads of linguistic reflection become less significant. Using a persistent cache also allows those overheads to be amortised over multiple calls which use the same input types. Furthermore, for large data sets the linguistic reflective solution is cheaper than the alternative generic solutions.

Conclusions

Strongly typed run-time linguistic reflection allows programs to generate new programs safely. The addition of type-safety to linguistic reflection yields the following benefits:

- More information is available to the reflective computation, in the form of systematically required types. This information can be used to automatically adjust to implementation details and system evolution.
- The type safety of all newly generated program fragments is checked before they are allowed to be executed. Such type discipline is highly advantageous in a programming environment in which the integrity of data must be supported.

It is, therefore, somewhat ironic that strong typing, which makes it difficult to integrate linguistic reflection with typed programming languages, also makes linguistic reflection effective as an amplifier of productivity. Conversely, linguistic reflection returns some of the expressibility to the language that was lost with the introduction of a type system (a major motivation for polymorphism). The importance of strongly typed linguistic reflection is that it provides a uniform mechanism for genericity and evolution that exceeds the capabilities of current non-reflective programming languages.

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.

[KMS98] Kirby, G.N.C., Morrison, R. & Stemple, D.W. “Linguistic Reflection in Java”. Submitted for publication (1998).

[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).

[PS88] PS-algol "PS-algol Reference Manual, 4th edition". Universities of Glasgow and St Andrews Technical Report PPRR-12-88 (1988).