An Analytical Comparison of Generative Programming Technologies Technical Report B-04-02

Dirk Draheim, Christof Lutteroth
Institute of Computer Science
Free University Berlin
email: {draheim,lutterot}@inf.fu-berlin.de

Gerald Weber
Department of Computer Science
The University of Auckland
email: weber@cs.auckland.ac.nz

January 2004

Abstract

In this paper we analyze existing generative programming technologies with respect to prototypical example problems. We point out their benefits and shortcomings, introduce the notion of generator type safety and eventually propose a new approach, which integrates introspection with parametric polymorphism and statically ensures generator type safety.

Contents

1 Introduction

Generative programming is about the idea to automate parts of the software generation process. Usually there are programming tasks in software projects which are so regular that we can make the computer do them for us by programming a generator. A common example is a compiler, which generates the representation of a program in one language from its representation into another.

Many generators are very specialized stand-alone programs dedicated to some particular function, but the idea of generation can also be supported in a programming language itself. A programming language can have features that facilitate or explicitly support the automation of the programming process.

A generator usually needs certain parameters in order to generate the desired result. These parameters may be just primitive values, objects or more complex constructs like classes or the abstract representation of executable code. As a result, features for generative programming usually are intimately related to the programming language itself. In a broader sense, we are looking at the capability of a language to write metaprograms, i.e. programs that represent and deal with other programs or sometimes even with themselves. Those metaprograms are our generators.

In section 2 we outline two example problems for generative programming and analyze the demands they make on a generator. Then we discuss in section 3 in how far these demands are met by different technologies in different programming languages and summarize their advantages and disadvantages. Finally, in section 4, we outline our own approach for a generative programming technology and demonstrate how the two examples can be realized in a statically type safe way.

2 Aims of Generative Programming

First, generative programming is about saving time in the process of software production. This is done by reusing common parts and adapting them to the place they are needed. Through reuse and adaptation we can reduce the complexity of our system, keep a cleaner structure in our design and potentially handle many variants in a better way.

Then, when the software product is in use, we can save time by having an adaptable or even adaptive system. "Adaptable" means that we can adapt it to our needs, "adaptive" goes one step further and means that the system can adapt to its environment by itself.

Eventually, a gain of performance in the software product is possible. When our system is adapted to its environment, it means also that it does not more than necessary, needs not more resources than necessary and maybe uses certain optimizations that are possible for the given circumstances.

3 Example Problems

3.1 Example: Getter- and Setter-Methods

Full object oriented data encapsulation requires the programmer to supply methods `getX` and `setX` for each externally accessible member variable `X`of a class. This is a routine task and just typing work, so we could write a generator that generates those getters and setters for us. This generator would have to operate in the following way:

```
class Person {
    String name;
    int    phone;
}

== Generator ==>

class PersonWithGetterSetter {
    private String name;
    private int    phone;

    public String getName()         { return name; }
    public void   setName(String v) { name = v; }

    public int    getPhone()        { return phone; }
    public void   setPhone(int v)   { phone = v; }
}
```

3.2 Example: Database Interface

In many applications we want some of the objects used to be persistent. Often we want the objects to be stored in a relational database, which is accessed by the use of a special database driver and a special database language (usually SQL). Once we know the structure of the objects that we want to store, we can derive from that a definition for a corresponding database table and code to store our objects. There are also certain typical search queries, which we can program immediately to retrieve our objects from the database table. All this, of course, needs some expertise of database programming and knowledge of the respective database language, but can be done as a routine job for many cases.

There exist tools, e.g. Toplink and in the framework of Enterprise Java Beans, which generate such database interfaces for given classes and thus allow programmers to stay on an abstract object oriented level. Such a generator might, for example, do the following:

```
class Person {
    String name;
    int    phone;
}

==Generator==>

class PersonTable {
    String dbname;

    void insert(Person p) {
        // insert p into table
    }

    Person[] selectWhereName(String s) {
        // return (all entries with name==s);
    }

    Person[] selectWherePhone(int i) {
        // return (all entries with phone==i);
    }
}
```

3.3 Demands on the Generator

Our two examples make certain demands on a generator:

First of all, the generator gets an unknown class as its argument, and in order to deal with that class it has to introspect its inner structure. That is, it has to know all of its externally accessible variables and methods with their types.

Afterwards it has to generate a new class. This class has a new signature, i.e. new externally accessible variables and methods, and it has a new internal structure, i.e. new private variables and code.

On top of this, we would like the generator to generate only type correct outputs. We would like to ensure at generator-definition time that the outputs will always be type correct.

4 Existing Concepts for Generative Programming

Our focus is on the support for generative programming in programming languages. That is, we want to examine how our example problems can be solved using the generative features of existing languages. The following list shows an overview of well-known features that can be used for generative programming:

- Inheritance (C++, Java)
- (Bounded) Parametric polymorphism (C++, GJ)
- Partial evaluation (new C++ Standard)
- Aspect oriented programming (AspectJ)
- Introspection (Java)
- Reflection, meta object protocol MOP
  runtime (Lisp, CLOS, MetaJ)
  compile-time (OpenC++, Jasper)

4.1 Bounded Parametric Polymorphism in C++ (Old Standard)

In C++ parametric polymorphism (also known as parametric types) is realized by the concept of templates. This means, a class and also a function can be parameterized by a type variable, which can be used in all places where a usual type can be used. When we use a template by filling in actual parameters, the compiler generates us a new class according to the template, with the type variable replaced by the actual parameter. Parametric polymorphism is used for example for parameterized container classes.

But this is already the only degree of flexibility that is allowed. It does not suffice for the generic database table class:

```
template <class A>
class Table {
    void    insert(A& v)
    {
        // We cannot read the fields of A,
        // because we don't know them.
    }

    A[]     selectWhere???(??? x)   // The name of each function cannot
vary.
    {
        // Type of the corresponding field ?
    }
}
```

Neither can we introspect the internal structure of the type parameter, nor can the type parameter influence the class we want to generate in other ways, e.g. we cannot influence its signature.

4.2 C++ Template Metaprogramming (New Standard)

The new standard makes C++ a 2-level-language. We have regular dynamic code, which is executed at runtime, and we have static code, which is executed at compile-time. This partial evaluation at compile-time is often used to implement specializers, that is for metaprograms that specialize some other program with regard to given actual parameters. In C++ this is made possible by the use of templates, which can perform calculations, distinguish between cases and use recursion. They give us a Turing-complete way of functional programming. For a more detailed account on C++ template metaprogramming see [1].

In [2] Attardi and Cisternino used template metaprogramming to implement static and dynamic reflection in C++. On top of this they implemented an interface for relational database tables, which is used like this:

```
class DocInfo {
    char const* name;
    char const* title;
    META( DocInfo,
         (VARKEY(name, 2048, Field::unique),
          VARFIELD(title, 2048)));
};

Table<DocInfo> table("db/table");
DocInfo a;
Table.insert(a);
```

For each actual parameter of `Table`, adapted code for the insertion of a correspondingly structured row into the given database table is generated. But a class cannot be introspected automatically. The macro `META` (and also another macro `REGISTER`) have to be used for the generation of static metainformation about the class. Furthermore, it is not possible to generate function names depending on the parameter.

4.3 AspectJ

AspectJ is a Java extension for aspect oriented programming, which is an attempt to systematize and simplify the handling of crosscutting concerns in programs. Crosscutting concerns are functional parts that spread over multiple logical units of a program and that therefore cannot be modularized in a traditional way. AspectJ offers two approaches, dynamic and static crosscutting, in order to deal with crosscutting concerns.

Dynamic crosscutting is a mechanism that allows extending and modifying a program's functionality by inserting pieces of code (advice) at well-defined points in the execution trace (join points). Join Points and advices are modularized in so called aspects. Let us look at a typical crosscutting concern, an error logging functionality:

```
aspect SimleErrorLogging {
    Log log = new Log();
    pointcut publicMethods(): receptions(public * myPackage.*.*(..));

    after() throwing (Error e): publicMethods() {
        log.write(e);
    }
}
```

This aspect logs every Error exception thrown in any public method of `myPackage`. As useful as it may be — the developers of AspectJ are still researching on the practical usability issue — dynamic crosscutting does not help us with any of our example problems, because aspects are of a quite static nature. Our main parameter is a set of join points, also called a point-cut, i.e. where to link in an advice, but we can not let the aspect adapt to a given class.

Static crosscutting allows to extend the signature of classes and interfaces. We can add a new Method to a class from within an aspect (member introduction):

```
Modifiers Type TypePattern.Id(Formals) {
    Body
}
```

Following this pattern we can add a new method to the class matching `TypePattern.Id`, but still we have to specify the method literally and cannot let it adapt dependent on some parameter.

4.4 MetaJ

MetaJ [5] is a reflective interpreter with Java syntax, i.e. an interpreter that features reflection. According to [4] reflection can be defined as "the ability of a program to manipulate as data something representing the state of the program during its own execution". Reflection can be subdivided into introspection and intercession and orthogonally into structural and behavioral reflection.

Structural reflection means that the structure of the program, its data structures and its code, can be read by introspection and modified by intercession during execution time. Introspection is the representation of parts of the program structure as data (also known as "reification") that can be read by the program. A suitable representation of a method, for example, would be its abstract syntax tree (AST). Intercession means that such data can be modified and brought back into the internal level (sometimes just called "reflection"). By that a reified method may be modified and then executed in its modified version. Behavioral reflection means that parts of the runtime system, which determines the behavior of the program, can be represented through introspection and modified through intercession.

In MetaJ a program's structure is accessible through a metaobject protocol (MOP), i.e. an object oriented runtime interface to parts of a program. Objects, classes and methods can be represented as metaobjects (introspection) and modifications of the metaobjects cause corresponding modifications in the represented internal entities (intercession). Not only the running program itself can be modified (structural reflection), but also the runtime system (behavioral reflection), that is the interpreter, according to the "Reflective Towers" meta-architecture. The "Reflective Towers" meta-architecture is an architecture for reflective systems (that is where the "meta" comes from) which works the following way: First, there is a base interpreter and on top of it the program that is to be interpreted. This base interpreter cannot be modified, but it is possible to insert a new interpreter between the base interpreter and the program which can be modified and change the operational semantics of the language. In MetaJ such an intermediate interpreter is accessible like any program through the metaobject interface. Now, the base interpreter interprets the intermediate interpreter on top of it which in turn interprets the program. And we can insert as many intermediate interpreter levels as we like, so that we have a tower of them, each interpreting that which is on top of it.

Such a metaobject protocol is a very powerful concept. We can change more or less whatever we want dynamically, and this can be used to implement adaptive systems. But there are also some drawbacks: First of all, it is difficult to have a reflective system work efficiently. In general, we have a significant performance overhead in such systems. Then, there are security issues: the Turing complete manipulation of metainformation makes also type soundness much more complicated. Most of the case static (i.e. compile-time) reflection is sufficient, and here errors can be detected at compile-time.

4.5 Jasper

Jasper [6] is a reflective syntax processor for Java. It provides mechanisms for static reflection. It gets some source code as input and builds, by the use of a metaobject protocol, an internal representation of it. Once this is done, it can transform and expand the syntax and eventually generate conventional Java source code.

The idea is, to allow metaprogramming through the extension/modification of the syntax processor itself - an architecture that is known as "open compiler". These processor extensions can have various forms:

First, the way the processor works can be modified by extending the metaclasses of its metaobject protocol. Since all objects in the processor are created by using a factory class, a metaclass can centrally be substituted by a subclass of it. This subclass may, for example, override the print-method that outputs the readily transformed AST-representation of the parsed input source code and thus modify the processors output.

Syntax-to-syntax transformations can also be implemented by extending a traversal class which traverses the AST and a visitor class that defines a transformation that should be performed by a visitor object on every traversed AST-node.

Eventually we are able to extend the parsed syntax by means of parser-plugins. A parser-plugin is a class that can be hooked into the processor's parser. It registers itself by handing a keyword and a context description to the parser, and when the parser is in this context, i.e. in this given parse rule, and matches the plugin's keyword, then the parsing process is handed over to the plugin's parse-method.

To facilitate the implementation of parser-plugins, Jasper provides a mechanism that generates those plugins from macro-descriptions and templates, i.e. Jasper has a generator for parser-plugins, which are generators themselves).

Let us look at an example macro, the `TRIVIAL` macro which is the identity function for statements. It defines a parser-plugin that tries to match the keyword `TRIVIAL` in the parse context of a statement, parses a statement afterwards by using the standard method and, as a syntax expansion, simply hands back the parsed statement:

```
MACRO Statement TRIVIAL (Statement s) { return s; }

===Generator===>

class TRIVIAL_Statement extends StatementParserExtension {
    public void init(Parser p) {
        init(p, "TRIVIAL");
    }

    public Statement parse(Identifier kwd) {
        return new TRIVIAL_StatementExpander(p.parseStatement());
    }
}

class TRIVIAL_StatementExpander extends StatementExpander {
    Statement s;
    TRIVIAL_StatementExpander(Statement s) {
        this.s = s;
    }

    public Statement expand() {
        return s;
    }
    ...
}
```

As we can see, two classes for the plugin are generated: `TRIVIAL_Statement`, which contains the initialization method that registers the plugin at the parser and the plugin's parse-method that gets called when the keyword `TRIVIAL` was matched in the parse context of a statement and `TRIVIAL_StatementExpander`, which does the actual syntax expansion work in the expand-method.

To facilitate the implementation of those parser-plugins even more, Jasper provides a template mechanism for building new AST-nodes in the expansion part of a macro. The template mechanism consists of a built-in macro NEW, which gets some syntax and generates code that creates an AST-representation of the given syntax (i.e. it performs a reification of the given syntax to the metaobject level). Let us look at the example macro IFNOT, which implements a conditional with negated condition:

```
MACRO Statement IFNOT (Expression test, Statement body) {
    return NEW Statement (Expression test, Statement body)
             if(test==false) body;
}

===Generator===>

class IFNOT_Statement extends StatementParserExtension
{
    ...
}

class IFNOT_StatementExpander extends StatementExpander {
    Expression test;
    Statement body;

    public Statement expand() {
        return new IfStatement(
          new EqualExpression(test, new BooleanExpression(false)),
body);
    }
}
```

We see how language extensions can be implemented as syntax processors through a compile-time MOP. Structural type safety of the extensions is ensured, because they get compiled before they are used. Furthermore, the implementation of extensions is facilitated by macros and templates.

5 An Approach towards Generator Type Safety

Our idea is to formulate generators as templates. The template approach means that as much as possible of the desired output is written down directly. Often, this approach is already used for generic polymorphism, so it seems straightforward to integrate further support for generative programming into the feature of parametric types, since our generators are parameterized, too. In order to do advanced generation work we need partial evaluation with the capability to introspect our parameters and to generate new signature elements and code.

- Our mechanism should be powerful enough to handle all the common problems.
- It should be simple enough to be intuitive for the user.
- It should provide generator type safety. That is, we want the compiler to be able to guarantee us that our generator will only generate type safe programs.

The main construct is a FOREACH loop, which allows to introspect a type parameter and iterate over its constituents. Everything within the loop is generated for each iteration, so that adequate signature elements and code can be generated for each constituent of the parameter. The loop-variable can be used in the generation part to insert constituent-specific information. The following illustrates this idea for the example of the getter and setter generator. We iterate over the fields of the class parameter and use its type and identifier name.

```
class WithGetterSetter<A> {
    FOREACH(field in A.fields) {
        field.type "get"+field.name () {
            return field;
        }

        void       "set"+field.name (field.type v) {
            field = v;
        }
    }
}
```

To implement the database access generator, we can place the FOREACH loop within a method's body:

```
class Table<A> {
    String dbname;
    ...
    void insert(A x) {
        // create new record in DB dbname
        FOREACH(field in A.fields) {
            // insert x.field into corresponding
            // column of new record
        }
    }
}
```

6 Conclusion

We depicted some example problems for the use of generative programming and gave an overview of different generative features in programming languages. Analysis showed that some of them were not flexible enough to handle our examples whereas some were so flexible that issues of type safety and usability arose.

Eventually, we pointed out a new approach that allows to write powerful generators which can guarantee the type-safety of all their outputs at definition-time.

References

[1] Czarnecki, K., Eisenecker, U. "Generative Programming - Methods, Tools, and Applications". Addison Wesley Publishers, 2000.

[2] Attardi, G., Cisternino, A. "Reflection Support by Means of Template Metaprogramming". In: LNCS 2186, pp. 118, Springer-Verlag, Berlin, 2001.

[3] Kiczales et al. "An overview of AspectJ". In: Proceedings of the European Conference on Object-Oriented Programming, Budapest, Hungary, 18-22 June 2001.

[4] Gabriel, R. G., Bobrow, D. G., White, J. L. "CLOS in Context-The Shape of the Design Space". In: Object Oriented Programming-The CLOS perspective. The MIT Press, Cambridge, MA, 1993, pp. 29-61.

[5] Douence, R., Südholt, M. "A Generic Reification Technique for Object-Oriented Reflective Languages". In: Higher Order and Symbolic Computation, 14(1), 2001.

[6] Nizhegorodov, D. "Jasper: Type-Safe Compile-Time Reflection Language Extensions and MOP-Based Templates for Java". In: Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures, 2000.