A Type System for Reflective Program Generators

Dirk Draheim[1], Christof Lutteroth[2], and Gerald Weber[2]

[1] Institute of Computer Science,
Freie Universität Berlin,
Takustr.9, 14195 Berlin, Germany
draheim@acm.org
[2] Department of Computer Science,
The University of Auckland,
38 Princes Street, Auckland 1020, New Zealand
lutteroth@cs.auckland.ac.nz, g.weber@cs.auckland.ac.nz

Abstract. In this paper we describe a type system for a generative mechanism that generalizes the concept of generic types by combining it with a controlled form of reflection. This mechanism makes many code generation tasks possible for which generic types alone would be insufficient. The power of code generation features are carefully balanced with their safety, so that we are able to perform static type checks on generator code. This leads to a generalized notion of type safety for generators.

1 Introduction

Generators are a cornerstone of today's software engineering, especially in the area of enterprise application development [1]. There exists a large variety of tools for the generation of database interfaces, GUIs and compilers, and even CASE tools can be subsumed under the notion of generators. Besides these very specialized examples of code generation technology, many systems have been developed that offer a more generic approach toward code generation. Some of these systems allow the user to extend a programming language with new constructs which trigger the generation of customized code.

In many cases it is not easy for a user to develop own code generators, even when using systems that support this explicitly. The user has to have knowledge about how a generator receives its parameters, how code is represented and processed, how code is emitted, and how a generator is deployed. The answers to these questions vary greatly from technology to technology. Code generation is a sensitive area because it depends on parameters, and the usual data structure involved, a syntax tree, is not trivial. A generator may work well most of the time but can potentially fail with some rare actual parameters, and an error may not be obvious but express itself in some slightly malformed parts of generated code. Using generators always bears the risk of introducing hard to find bugs, while a good generator has the potential to provide an economic and solid solution to

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 327–341, 2005.

a common problem. Complexity in the development of code generators leads to generators that are more error-prone.

In this paper we show how the concept of code generators can be made accessible to the user directly in object-oriented languages and how a type system can be extended to take generators into account. The aim is to make generators part of a program and not of the compiler while retaining the safety properties of a typed language. No internal knowledge of the compiler should be required, and the generation process should be transparent for the user. Placing generators into the language itself instead of into a compiler affects the language syntax as well as its semantics and safety; the challenge lies in integrating the new constructs syntactically without interfering with existing semantics. Typed languages usually offer a high degree of safety through the use of type systems, and type checkers are able to detect many potential execution errors statically. With the new concept of generators, however, new types of potential execution errors are introduced, namely those that happen when code generation produces ill-typed code. Consequently, code generation poses new challenges to type systems.

In Sect. 2 we introduce the Genoupe language, which integrates code generators into the C# language, by looking at source code examples. We also discuss its general applicability to different problems. Section 3 presents the novelties of Genoupe's type system and discusses some malformed examples of Genoupe code that cannot be given a correct type. Section 4 looks at related work and explains how Genoupe is different to similar approaches. The paper concludes with Sect. 5.

2 Object-Oriented Programming with Parameterized Generators: The Genoupe Language

Our concept for the integration of generators into object-oriented programming is called Genoupe. It was developed from the language Factory [2], which integrated reflective generators into Java, and implements a similar but strongly revised concept for C#. Genoupe introduces a syntax that is reminiscent of that of generic types, although it is not limited to classes or interfaces. Like for generic types the template paradigm is used, but in contrast to simple genericity, the template can contain generator code written in a special compile-time level language. This sublanguage is kept in an imperative style and along the lines of the C# language itself, so that a C# programmer will intuitively understand its meaning. Also the type system is analogous to the runtime one, but simpler for ordinary types, since we usually do not need as many features here for generation as we usually want for runtime code. With respect to generated types the type system gets somewhat more sophisticated, and we need a whole set of essentially new type rules. However, this is well worth it because, as we will see in Sects. 3 and 4, the new type system makes it possible to detect parts of a generator that can potentially generate malformed code, in contrast to just detecting code that is malformed itself.

In the Genoupe language a generator can be embedded into the source code like an ordinary type definition. Source code files written in the Genoupe language have the name suffix `.genoupe` and are compiled to ordinary C# source files with the same name but `.cs` suffix (see Fig. 1). Each time a generator is applied with new arguments, new types with unique names are created. If a generator is applied more than once with the same arguments in a compilation run, the corresponding code is generated only once.

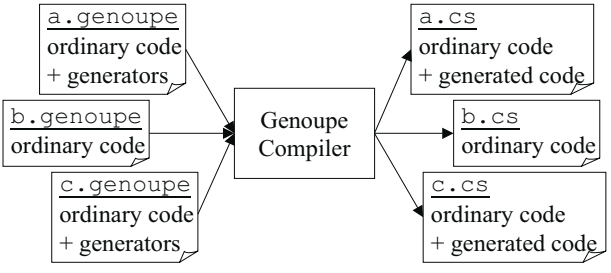

Fig. 1. The Genoupe compilation process

In a generator actual type parameters can be accessed through so called *generator variables*. These are variables that, in contrast to runtime variables, hold objects at generation-time and make them accessible in the generator code. Analogous to the parameters in an ordinary method, each declared generator parameter creates a generator variable, which can be used in *generator expressions*. A generator expression describes a values that is used at generation-time, just as an ordinary expression describes a value that is used at runtime. It is very similar to an ordinary C# expression in the sense that most generator expressions are valid C# expressions. One speciality of generator expressions is that, with the same values assigned to the generator variables, two structurally equal generator expressions describe the same value. We do not have non-deterministic effects like, e.g., random values, which are not needed in code generators. As we will see in Sect. 3.1, this will help us to rule out some potential generation errors statically.

Usually generator expressions are used to introspect type parameters and extract or construct the information that is needed for intercession, i.e., information that represents code that should be made part of the generator output. In order to make the value of a generator expression part of the generated code, the generator expression is enclosed in `@` characters and placed into the code template at a position where the entity that is represented by the expression's value is allowed to occur. If we want, for example, to generate a certain type in a declaration of a generated class, we would create a generator expression that evaluates to a `Type` object representing the desired type. This generator expression would be placed, enclosed in `@` characters, at the position in the source code where we would normally place a type name. At generation time all generator expressions are evaluated and substituted by the code represented by their values.

That is, if we had a generator expression of type `Type`, i.e., one that evaluated to a `Type` object, Genoupe would substitute the generator expression by the name of the type represented by the type object in the generated code. Genoupe makes use of the standard C# metaobject protocol, so that it is obvious in most cases which type represents which language entity.

In the following subsections we will consider some simple examples of Genoupe source code, which will point up how Genoupe can be used. Some applications for Genoupe, e.g., the generation of interfaces like GUIs or APIs, are not discussed here. Information on those and further examples can be found in [3,4].

2.1 Parametric Polymorphism

One of the simplest applications for Genoupe is parametric polymorphism. The following generic stack generator has a single parameter `T` of type `Type` and generates a stack class for elements of type `T`:

```
1  public class Stack(Type T)
2  {
3      private Stack s = new Stack();
4
5      public void push(@T@ x) {
6          s.push(x);
7      }
8
9      public @T@ pop() {
10         return (@T@) s.pop();
11     }
12 }
```

The generator parameter declaration in line 1 looks a bit similar to a method declaration, and like in a method declaration, a generator can have an arbitrary number of parameters with arbitrary type. In lines 5, 9 and 10 we insert generator expressions containing only the generator parameter in order to generate correct type declarations and type casts.

2.2 Class Extensions

Genoupe can be used for the generation of useful extensions. In contrast to ordinary inheritance mechanisms, which also extend classes, a generator can adapt the extension it generates to the class that is extended. This makes it possible to address static crosscutting concerns [5].

The following code snippet shows a generator that takes a class `T` and an array of field names `FNames` for that class. It generates a subclass of `T` that extends it by a new method `Randomize` that assigns random values to the fields of `T`. This can be useful, for example, for the generation of test data.

```
1  public class Randomizeable(Type T, String[] FNames) : @T@
2  {
3      public void Randomize() {
4          Random r = new Random();
5          @if(FNames!=null) {
6              @foreach(FName in FNames) {
7                  @const F = T.GetField(FName);
8                  @if(F.FieldType==Double)
9                      this.@F.Name@ = r.NextDouble();
10                 else @if(F.FieldType==Boolean)
11                     this.@F.Name@ = (r.NextDouble()>=0.5);
12                 // ...handle other data types...
13             }
14         } else {
15             @foreach(Field in T.GetFields()) {
16                 // ...generate assignments for all fields...
17         }   }
18 }   }
```

In line 5 we see the `@if` control construct of the generator language for conditional generation. It checks if an array of field names has been given at all, and only then the `FNames` array is used. In line 6 we see the `@foreach` construct, which is used for iterative generation. Its only difference to the `foreach` construct of C# is that the static type of the iterator variable needs not to be declared. In line 7 we define a new generator variable with a constant value, which is just syntactic sugar for our convenience. In the following lines, depending on the type of the respective field, we generate a statement that assigns to the field a compatible random value. The field's identifier is generated with a corresponding generator expression of type `String`. In the else-clause of the outermost `@if`, which is analogous to the aforementioned code, we handle the case that an array of field names was not given by generating code that assigns random values to all fields of `T`.

2.3 Proxies and Wrappers

A common pattern for modifying the behavior of existing classes or bridging incompatibility is the use of proxies [6] and wrappers. With Genoupe both of these can be generated automatically, which makes it possible to address dynamic crosscutting concerns [5].

The following class generator takes a type parameter `T` and creates a subtype of `T` that overrides and wraps `T`'s methods. A class generated by this generator behaves like `T` but logs all method calls and exits, which can be useful for debugging purposes.

```
1  public class Logger(Type T) : @T@
2  {
3      public String Log = new String();
```

```
4
5       @foreach(M in T.GetMethods()) {
6           @const Pars = M.GetParameters();
7
8           public override @M.ReturnType@ @M.Name@
9           (@foreach(P in Pars) { @P.ParameterType@ @P.Name@ })
10          {
11              Log += @new Literal(M.Name)@+" called.\n";
12              base.@M.Name@(@foreach(P in Pars) { @P.Name@ });
13              Log += @new Literal(M.Name)@+" exiting.\n";
14          }
15  }   }
```

In lines 8 and 9 we use generator expressions to generate the signature of each of `T`'s public methods. A list of method parameter declarations is generated by iterating over all the parameters and generating each parameter declaration individually. The same approach is used in line 12 in order to generate the list of arguments for a method call. The `Literal` objects constructed in lines 11 and 13 represent generated string literals, opposed to generated identifiers.

3 Generator Type Safety

When dealing with metaprograms, i.e., programs that process other programs or themselves in some suitable representation, a whole set of new sources of execution errors comes into play. *Generation errors* in generators are those parts of the generator program that can potentially generate malformed code, which in turn may cause execution errors when executed. Of course, we also want our generators to be free of execution errors themselves. In addition to normal type systems, which can only detect potential forbidden errors in the code that is type checked, we need a new kind of type system that can also detect parts in generators that can potentially generate ill-typed code. This requirement leads to a new notion of type safety, which we want to call *generator type safety*. It is the property of a generator not to be able to generate ill-typed code, i.e., code that may cause a forbidden execution error. If a generator is not generator type safe, it contains one or more *generator type errors*, i.e., parts in the generator code that are responsible for the generation of ill-typed code. We call a type system that can detect generator type errors a *generator type system*.

Before we describe the generator type system of Genoupe in the next section, let us look at examples of malformed generators that can potentially generate ill-typed code. The following generator generates a class with a single field:

```
1   class C(Type T)
2   {
3       @T@ x = 1;
4   }
```

The fact that `x` is assigned a numerical value restricts its possible type. The type parameter `T` however is not subject to any such restriction. This is clearly a generator type error that leads to some arguments producing type-correct code and others not.

The next example demonstrates another issue of type compatibility.

```
1   class C(T istype Component)
2   {
3       @T@ x = new Button();
4   }
```

The Genoupe keyword `istype` makes it possible to set a bound for type parameters, i.e., parameters of type `Type`. Line 1 signifies that parameter `T` is a type parameter and that all possible arguments represent types that are either class `Component` itself or one of its subclasses. In the generator body we define a member variable `x` with type `T`, to which we assign a `Button` object. `Button` is a subclass of `Component`, but what if `T` is a subclass of `Component` but not compatible to `Button`, i.e., not either `Button` itself or one of its superclasses? The generated code is type correct iff `T` is `Button` or one of its superclasses.

The following example is a class generator that has a string parameter `ID`. As the name suggests, the string is used to generate the identifier of a local variable in a method.

```
1   class C(String ID)
2   {
3       void m() {
4           int @ID@ = 1;
5           x++;
6       }
7   }
```

In line 5 we increment a variable `x`. Since there are no other variable definitions in the generator, `x` must be defined in the preceding line where the identifier of a variable is generated by a generator expression. If the generator is given the argument `"x"`, the generated code works just fine, otherwise it is ill-typed. This is also known as the problem of *inadvertent capture* [7].

The next generator contains a conditional generation.

```
1   class C(String X)
2   {
3       @if(X.Equals("hello")) {
4           @T@ y = "world";
5       }
6
7       void m() {
8           Console.WriteLine(y);
9       }
10  }
```

The definition of the member variable `y` is only generated when `"hello"` is the string argument in `X`. Again, we have cases where this generates an error and others where it does not.

Our last example illustrates a generator type error that can occur in iterative generation.

```
1  class C(Type S, Type T)
2  {
3      @foreach(F in S.GetFields()) {
4          @F.FieldType@ @F.FieldName@;
5      }
6
7      void m() {
8          @foreach(F in T.GetFields()) {
9              Console.WriteLine(this.@F.FieldName@);
10         }
11     }
12 }
```

The first generative iteration replicates the field definitions of type parameter `S`. The second one in method `m` generates statements that access and print the values of fields as defined in type parameter `T`. Clearly this can only work if `T` contains fields with identical name for all the field definitions in `T`, which is of course the case when `S` and `T` are bound to the same type.

All these generator type errors also occur in real generators, and usually they occur in a subtler way that makes them much harder to find. Such errors are typically introduced, for example, when applying inconsistent changes: one part of a generator is changed without adjusting other parts accordingly that are affected by that change.

Note that the Genoupe language has another property which makes its generators safer than those in many other languages: if all the methods we use in generator code terminate and we do not use generators recursively, which is usually unnecessary, a generator is guaranteed to terminate. This is because our looping construct, the @foreach, iterates over collections without modifying them, and the collections contain of course only a finite number of elements. In C++ templates, for example, we must use recursion when we want to repeat something arbitrarily often. C++ templates can potentially recourse endlessly, and only a limited recursion-depth prevents this [8]. In other technologies which use a Turing-complete language for metaobject manipulation, like CLOS [9], OpenC++ [10] or Jasper [11], generators potentially do not terminate as well.

3.1 The Genoupe Type System

In order to detect generator type errors, we developed a generator type system which is compatible with and extends the type system of the host language C#. Its notation is similar to the one used in [12]. It consists of rules with judgments about the correctness of certain program parts in their pre- and postconditions,

and only the programs that can be derived by those rules are considered type correct. In some respects, however, our type system deviates from the way in which type systems of object-oriented languages usually work. We use an environment Γ, which keeps track not only of the signatures of declared runtime variables but also of the signatures of generator variables. The signature of a runtime variable can contain generator expressions because its identifier and type may be generated by them. For handling conditional and iterative generation of declarations correctly, definitions that are generated conditionally or iteratively have special signatures, and Γ is also used to store additional facts about the code portion that is being type-checked.

Rather than delivering a complete description of the type system, this paper focuses on explaining the main concepts by looking at some exemplary type rules. These rules can be found in Table 1, and we will go through them one after another. Rule $[Env\ Var]$ describes how the signature of a generated variable can be included into Γ. The two judgments in the precondition state that we need a correct generator expression of type `String` for the variable's identifier, and a correct generator expression of type `Type` for the variable's type. The $::$ symbol associates a generator expression with its type. In the postcondition the new environment is a conjunction of the old Γ and the new signature. The $:$ symbol associates the identifier of a variable with its type. Rule $[Env\ then]$ allows us to register in Γ that a generator expression $Gexpr$ evaluates to true. The generator expression must be of type `Boolean` and the opposite, i.e., that $Gexpr$ evaluates to false, must not be registered in Γ already. As the name of the rule suggests, this rule is used for type-checking in the then-clause of an `@if` construct, where the generator expression describing the condition of the `@if` is known to be true. Analogous to this, rule $[Env\ loop]$ allows us to register in Γ that an iterator variable of a `@foreach` contains an element of a particular collection, which is the collection over which is iterated.

Rule $[Def\ Var]$ describes how a variable definition can be generated with suitable generator expressions and what its signature looks like. The $\therefore$ symbol associates a signature to a definition. A signature is a set of facts that describe a definition. Rule $[Def\ @if]$ describes the conditional generation of definitions. In the second and third line of the precondition, we see that the facts $Gexpr$ and $\neg Gexpr$ are included in the environment when we demand that the declarations D_1 and D_2 have the signatures Sig_1 and Sig_2, respectively. Consequently, the judgment in the postcondition states the correctness of an `@if` with D_1 in the then- and D_2 in the else-clause. The signature of the $@if$, which becomes part of the environment during type-checking, has two parts: one describing the signature of the generated definition in the case that the condition is true and one describing the signature of the generated definition when its not. Rule $[Def\ @foreach]$ describes the iterative generation of definitions. In the second judgment of the precondition we demand that D is a correct definition with signature Sig. The environment states that ID is an iterator variable which contains an element of the collection described by $Gexpr$. The signature of the resulting `@foreach` is again a special one: it signifies that for any generator vari-

able X, with X being an element of some collection described by $Gexpr$, there is a signature that looks like the signature of D, only that each occurrence of ID in that signature is substituted by X.

The rules [$Expr\ Var$ 1], [$Expr\ Var$ 2] and [$Expr\ Var$ 3] will hopefully clarify why we need these unusual elements in Γ. They all specify how we can use a generated variable in a generated expression. Rule [$Expr\ Var$ 1] states that if there is a generated variable declared in Γ, we can generate an expression that uses it by generating its identifier with a corresponding generator expression. Rule [$Expr\ Var$ 2] describes under which circumstances a variable can be used that has been generated in the then-clause of a conditional generation: it can be used if Γ states that $Gexpr$, the condition under which the variable was generated, is true. Analogous to this rule, there is also one for using a variable that has been generated in the else-clause of an `@if`. Finally, rule [$Expr\ Var$ 3] handles the usage of variables that have been generated in a `@foreach`. Such a variable can be used if Γ states that the usage of the variable is in the body of a `@foreach` loop that loops over a collection described by the same generator expression as the collection of the loop in which the variable was defined. This means that the collections of the two loops are equivalent. In the loop in which we generate code that uses the variable, the iterator variable may have a different identifier. Therefore we substitute the X in the variable's signature by the ID of this loop's iterator variable.

3.2 Limitations

Like most type systems, the Genoupe type system is restrictive: it forbids not only programs that are obviously incorrect but also many others which do not contain generator type errors. In the rules for the `@if`, for example, we require that a conditionally generated variable must be used in the body of a conditional with equivalent condition. Logically it would be enough, though, to require that the condition of the defining conditional *implies* the condition of the conditional in which the variable is used. Analogously, if variables are generated in a `@foreach`, it would be sufficient to demand that they are used in a loop that iterates over a *subset* of the collection in the defining iteration. Because the underlying problems are undecidable, we did not try to solve them, although it would be possible to address these issues using approaches from logical programming like, for example, constraint solving and model checking. Note that this is a popular way for type systems to deal with issues that restrict the way a language is used but do not really limit its applicability: C# and Java, for example, do not really check whether a method with a non-void return type returns a value; they merely check if a superset of possible execution paths returns a value.

The possibility to generate arbitrary identifiers with generator expressions brings about lexical problems: a generated identifier might be malformed, e.g., it might clash with a keyword, or might not be unique. Both these problems could only be solved if we restricted the way identifiers can be generated. But if we did that, we would lose flexibility and potentially the ability to produce clear human-readable names, and the language would become more complicated.

Table 1. Exemplary type rules of the Genoupe generator type system

$$[Env\ Var] \quad \frac{\Gamma \vdash Gexpr_1 :: \texttt{String} \quad \Gamma \vdash Gexpr_2 :: \texttt{Type} \quad Gexpr_1 \notin \mathrm{Dom}(\Gamma)}{\Gamma \cup \{Gexpr_1 : Gexpr_2\} \vdash \diamond}$$

$$[Env\ then] \quad \frac{\Gamma \vdash Gexpr :: \texttt{Boolean} \quad (\neg Gexpr) \notin \Gamma}{\Gamma \cup \{Gexpr\} \vdash \diamond}$$

$$[Env\ loop] \quad \frac{\Gamma \vdash Gexpr :: \texttt{ICollection}}{\Gamma \cup \{ID \in Gexpr\} \vdash \diamond}$$

$$[Def\ Var] \quad \frac{\Gamma \vdash Gexpr_1 :: \texttt{Type} \quad \Gamma \vdash Gexpr_2 :: \texttt{String}}{\Gamma \vdash @Gexpr_1@\ @Gexpr_2@; \therefore \{Gexpr_2 : Gexpr_1\}}$$

$$[Def\ @if] \quad \frac{\begin{array}{c}\Gamma \vdash Gexpr :: \texttt{Boolean} \\ \Gamma \cup Sig_1 \cup \{Gexpr\} \vdash D_1 \therefore Sig_1 \\ \Gamma \cup Sig_2 \cup \{\neg Gexpr\} \vdash D_2 \therefore Sig_2\end{array}}{\begin{array}{l}\Gamma \vdash @\mathrm{if}(Gexpr)\ \{\ D_1\ \}\ \mathrm{else}\ \{\ D_2\ \} \\ \therefore \{Gexpr \rightarrow Sig_1,\ \neg Gexpr \rightarrow Sig_2\}\end{array}}$$

$$[Def\ @foreach] \quad \frac{\Gamma \vdash Gexpr :: \texttt{ICollection} \quad \Gamma \cup Sig \cup \{ID \in Gexpr\} \vdash D \therefore Sig}{\Gamma \vdash @\mathrm{foreach}(ID\ in\ Gexpr)\ \{\ D\ \} \therefore \{\forall X \in Gexpr.Sig[X/ID]\}}$$

$$[Expr\ Var\ 1] \quad \frac{(Gexpr_1 : Gexpr_2) \in \Gamma}{\Gamma \vdash @Gexpr_1@ : Gexpr_2}$$

$$[Expr\ Var\ 2] \quad \frac{\{Gexpr,\ Gexpr \rightarrow Gexpr_1 : Gexpr_2\} \subseteq \Gamma}{\Gamma \vdash @Gexpr_1@ : Gexpr_2}$$

$$[Expr\ Var\ 3] \quad \frac{\begin{array}{c}\{ID \in Gexpr,\ \forall X \in Gexpr.(Gexpr'_1 : Gexpr'_2)\} \subseteq \Gamma \\ (Gexpr'_1 : Gexpr'_2)[ID/X] = (Gexpr_1 : Gexpr_2)\end{array}}{\Gamma \vdash @Gexpr_1@ : Gexpr_2}$$

The more freedom we allow for the generation of identifiers, the more complex a collision detection scheme would have to be in order to avoid this problem. We decided not to implement any such restriction or detection scheme and take the risk of lexical collisions, which is inherent when working with a textual source code representation. The responsibility for handling the generation of identifiers carefully lies with the programmer of a generator, for whom this is usually unproblematic.

4 Related Work

Genoupe is an extension of *genericity* or parametric polymorphism found, for example, in ADA or Java [13,14]. With parametric polymorphism it is possible to program components that are uniformly reusable for many types. However, these generic type parameterization mechanisms are at the same time type abstraction mechanisms: the construction of the type cannot be exploited in the

parameterized software component – at most it can be exploited up to a bound, known as bounded parametric polymorphism. Therefore it is useful for container libraries, e.g., C++ Standard Template Libraries, but it is not as powerful as Genoupe.

The original *C++ template mechanism* does not allow for the enforcement of properties for actual type parameters as, for example, supported by the notion of bounded parametric polymorphism [12,15]. Ad-hoc solutions to provide some level of concept checking for C++ templates, like specialized macros [16] and static interfaces [17], has been generalized by the introspection library approach in [18]. This approach targets user-customized checks for both compile-time adaptation and diagnostics.

The *new C++ templates* standard allows in principle Turing-complete metaprogramming with static and dynamic reflection in C++ [19], sufficient, e.g., for an interface generator for a relational database [20]. It is still less powerful than Genoupe; for example, it is not possible to generate function names dependent on a parameter. It does not support any static notion of generator type safety; type-checks are done with the ordinary C++ type system. Furthermore, a template metaprogram may not terminate. The Turing-completeness makes it impossible to analyze the generating templates exhaustively.

Aspect oriented programming aims at handling of crosscutting concerns in programs. AspectJ [5] is a Java extension for aspect oriented programming, which offers two approaches: dynamic and static crosscutting. Crosscutting does not help us with type-dependent generative problems, e.g., the implementation of a transparent data-access layer. Static crosscutting allows to extend the signature of classes and interfaces, but not in an adaptive manner: we can add a new method to a class from within an aspect – so-called member introduction – but still the method has to be specified literally and cannot be made dependent on some parameter. The generative approach to aspect-oriented programming in [21] characterizes certain uniform patterns that arise in using the aspect oriented style of inverting functional decomposition as amenable to be handled by the incremental computation approach. Based on this insight the approach establishes a behavioral semantics for generative aspect-oriented features that are oriented towards finite differencing [22].

The concept of *runtime reflection* dates back to Lisp [23] and has been subject of major interest in the functional programming community. The combination of parametric polymorphism with reflective features in Generic Haskell [24,25] benefits from the theoretical well-understood type-system of the host language. In the context of the object-oriented functional programming language CLOS [26,9], a mature metaobject protocol has been elaborated. In [27] CLOS is used to prove the value of metaprogramming by embedding representations of common object-oriented design patterns [6] into programs. Multistage programming [28,29] is an approach that focuses on runtime program generation and execution. The programmer is supported by constructs for partial evaluation and program specialization, whereas several properties of runtime generation can already be ensured statically. An implementation of the multistage programming approach is pro-

vided on top of the object-based functional programming language O'Caml [30]. The language Metaphor [31] results from extending the subset of an object-oriented language like C# or Java by the multistage constructs of the functional programming language MetaML [28,29], i.e., a construct for building representations of expressions, a construct for splicing code and a construct for running staged evaluated code. With its multi-staged language design Metaphor achieves type-safe generation of code that makes use of the reflection system of the base language.

Jasper [11] is a reflective syntax processor for Java. It provides mechanisms for *static reflection*. It does not follow the template approach; instead it allows for metaprogramming through the extension/modification of the syntax processor itself [32] – an architecture that is known as *open compiler*. It supports universal metaprogramming and is as such more powerful, but less understood.

5 Conclusion

Genoupe implements a concept for generative programming that integrates reflection by means of a metalanguage into a template mechanism reminiscent of genericity. It can be used to solve common problems of generative programming and offers advantages compared to other languages with respect to the degree of integration of the runtime and the metalanguage and safety:

- Genoupe places the concept of generators into the language instead of relying on an external tool driven approach, thus minimizing the interface to the user and avoiding potential errors.
- It integrates well with an object-oriented host language and can be seen as a generalization of genericity. It uses similar syntax for runtime and generator code, which makes it easy to use and understand.
- A wide range of common applications of generative programming can be addressed.
- Genoupe offers an particular high degree of static safety for reflection by means of a type system that is able to detect generator type errors.

More information about Genoupe and implementations of the Genoupe system can be found on our project web site, `http://www.genoupe.formcharts.org/`.

References

1. Draheim, D., Weber, G.: Form-Oriented Analysis - A New Methodology to Model Form-Based Applications. Springer (2004)
2. Draheim, D., Lutteroth, C., Weber, G.: Factory: Statically Type-Safe Integration of Genericity and Reflection. In: Proceedings of the 4th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, ACIS (2003)
3. Draheim, D., Lutteroth, C., Weber, G.: Integrating Code Generators into the C# Language. In: Proceedings of ICITA 2005: The 3rd International Conference on Information Technology and Applications, IEEE Press (2005) to appear.

4. Draheim, D., Lutteroth, C., Weber, G.: Generative Programming for C#. ACM SIGPLAN Notices (2005) to appear.
5. Kiczales, G.: An overview of AspectJ. In: Proceedings of the European Conference on Object-Oriented Programming. LNCS 2072, Budapest, Hungary (2001) 18–22
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley (1995)
7. Kohlbecker, E., Friedman, D., Felleisen, M., Duba, B.: Hygienic Macro Expansion. In Gabriel, R., ed.: Proceedings of the ACM SIGPLAN Conference on LISP and Functional Programming, ACM Press (1986) 151–181
8. Czarnecki, K., Eisenecker, U.: Generative Programming - Methods, Tools, and Applications. Addison-Wesley (2000)
9. R. G. Gabriel, D. G. Bobrow, J.L.W.: Object Oriented Programming - The CLOS perspective. The MIT Press, Cambridge, MA (1993)
10. Chiba, S.: A Metaobject Protocol for C++. In: OOPSLA 1995 - Proceedings of the 10 th Conference on Object-Oriented Programming Systems, Languages and Programming. SIGPLAN Notices, ACM Press (1995) 285–299
11. Nizhegorodov, D.: Jasper: Type-Safe MOP-Based Language Extensions and Reflective Template Processing in Java. In: Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures: State of the Art, and Future Trends, ACM Press (2000)
12. Cardelli, L.: Type Systems. In: Handbook of Computer Science and Engineering. CRC Press (1997)
13. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In: OOPSLA 1998 - Conference on Object-Oriented Programming Systems, Languages, and Applications. SIGPLAN Notices, ACM Press (1998) 183–200
14. Bracha, G., Cohen, N., Kemper, C., Marx, S., Odersky, M., Panitz, S.E., Stoutamire, D., Thorup, K., Wadler, P.: Adding Generics to the Java Programming Language: Participant Draft Specification. Technical report, SUN Microsystems Ltd. (2001)
15. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
16. Siek, J., Lumsdaine, A.: Concept Checking: Binding Parametric Polymorphism in C++. In: First Workshop on C++ Template Programming, NETOBJECTDAYS 2000. (2000)
17. McNamara, B., Smaragdakis, Y.: Static Interfaces in C++. In: First Workshop on C++ Template Programming, NETOBJECTDAYS 2000. (2000)
18. Zólyomi, I., Porkoláb, Z.: Towards a General Template Introspection Library. In: GPCE 2004 - The 3rd International Conference on Generative Programming and Component Engineering. LNCS 3286, Springer (2004) 266–282
19. Attardi, G., Cisternino, A.: Reflection Support by Means of Template Metaprogramming. In: 3rd International Conference on Generative and Component-Based Software Engineering. LNCS 2186 (2001)
20. Attardi, G., Cisternino, A.: Template Metaprogramming an Object Interface to Relational Tables. In: REFLECTION 2001 - The 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns. LNCS 2192 (2001)
21. Smith, D.R.: A Generative Approach to Aspect-Oriented Programming. In: GPCE 2004 - The 3rd International Conference on Generative Programming and Component Engineering. LNCS 3286, Springer (2004) 39–54
22. Paige, R., Koenig, S.: Finite Differencing of Computable Expressions. ACM Trans. Program. Lang. Syst **4** (1982) 402–454

23. Smith, B.C.: Reflection and semantics in LISP. In: POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press (1984) 23–35
24. Hinze, R., Jeuring, J.: Generic Haskell: Practice and Theory. In Backhouse, R., Gibbons, J., eds.: Generic Programming: Advanced Lectures. LNCS 2793, Springer (2003) 1–56
25. Hinze, R., Jeuring, J.: Generic Haskell: Applications. In Backhouse, R., Gibbons, J., eds.: Generic Programming: Advanced Lectures. LNCS 2793, Springer (2003) 57–97
26. Kiczales, G., des Rivières, J.: The Art of the Metaobject Protocol. MIT Press (1991)
27. von Dincklage, D.: Making Patterns Explicit with Metaprogramming. In Pfenning, F., Smaragdakis, Y., eds.: GPCE 2003 - The 2nd International Conference Generative Programming and Component Engineering. LNCS 2830, Springer (2003) 287–306
28. Taha, W., Sheard, T.: Multi-Stage Programming with Explicit Annotations. In: PEPM 1997 - Partial Evaluation and Semantics-Based Program Manipulation. SIPLAN Notices, ACM Press (1997) 203–217
29. Taha, W., Sheard, T.: MetaML and Multi-stage Programming with Explicit Annotations. Theoretical Computer Science **248** (2000) 211–242
30. Leroy, X., Doligez, D., Garrigue, J., Rmy, D., Vouillon, J.: The Objective Caml system release 3.08 - Documentation and user's manual. Technical report, Institut National de Recherche en Informatique et en Automatique (2004)
31. Neverov, G., Roe, P.: Metaphor: A Multi-stage, Object-Oriented Programming Language. In: Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE'04). LNCS 3286, Springer (2004) 168–185
32. Draheim, D., Lutteroth, C., Weber, G.: An Analytical Comparison of Generative Programming Technologies. Technical Report B-04-02, Institute of Computer Science, Freie Universität Berlin (2004)