A Type System for Reflective Program Generators

Christof Lutteroth

[a] *Department of Computer Science*
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand

Dirk Draheim

Central Information Technology Services
University of Innsbruck
Technikerstrasse 23, 6020 Innsbruck, Austria

Gerald Weber [a]

Abstract

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, which enables us to perform static type checks on generator code. This leads to a generalized notion of type safety for generators.

Key words: type safety, generic programming, reflection, model-based generation

1 Introduction

Generators are a cornerstone of today's software engineering, especially in the area of enterprise application development. A large variety of tools exists for

Email addresses: `lutteroth@cs.auckland.ac.nz` (Christof Lutteroth), `draheim@acm.org` (Dirk Draheim), `g.weber@cs.auckland.ac.nz` (Gerald Weber).
URLs: `http://www.cs.auckland.ac.nz/~lutteroth` (Christof Lutteroth), `http://draheim.formcharts.org` (Dirk Draheim), `http://www.cs.auckland.ac.nz/~gerald` (Gerald Weber).

the generation of database interfaces, GUIs and compilers, and even CASE tools can be subsumed under the notion of generators. A generative approach fits naturally with important enterprise computing paradigms [16].

Besides these very specialized examples of code generation technology, many systems have been developed that offer a more generic approach to code generation. Some of these systems allow the user to extend a programming language with new constructs which trigger the generation of customized code. Code generation is a particularly powerful decomposition and reuse mechanism.

The use of custom-made code generators based on classic technologies such as compiler-compilers is a challenging task. Generation typically depends on other code as a parameter, and the traditional data structure involved, a syntax tree, is not trivial. Code generators require not only firm knowledge of the theory of compiler construction, but also familiarity with the technicalities of the particular source and target languages involved. From a software engineering standpoint, this amounts to additional risks concerning the long term maintainability of such code.

Generators introduce new classes of errors. A generator may work well most of the time, but may fail with some rare actual parameters. 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. However a good generator has the potential to provide an economic and solid solution to a common problem. Complexity in the development of code generators leads to generators that are more error-prone.

In this article, we show how the concept of code generators can be made directly accessible to the user 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 rather than part of the compiler while retaining the safety properties of a typed language. No internal knowledge of the compiler would be required, and the generation process should be transparent for the user. Placing generators in the language itself instead of in 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. However, with the new concept of generators, new types of potential execution errors are introduced, namely when code generation produces ill-typed code. Consequently, code generation poses new challenges for type systems.

In Sect. 2, we introduce the Genoupe language, which integrates code generators into the C# language, and give some source code examples. We also

discuss its general applicability to different problems. Section 3 presents the Genoupe type system and discusses some malformed examples of Genoupe code that cannot be given a correct type. It also discusses the limitations of the type system. Section 4 discusses soundness aspects of the type system, in particular in relation to the use of imperative code in the Generators. Section 5 discusses how the integration into a novel code repository can alleviate some of the problems faced in the language definition. Section 6 discusses the relation between code generators and reflection. Section 7 looks at related work, and explains how Genoupe is different from similar approaches. In Sect. 8 we summarize the main findings of our work.

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 [11], 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. As with 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 understand its meaning intuitively.

Natutrally, the type system used to check generator code requires additional features that are not found in ordinary type systems. As we will see in Sects. 3, 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. One of the significant features of the Genoupe language is that calls can be made to ordinary C# code during generation, even if this code contains non-determinism or side effects. An explanation for why Genoupe can allow the execution of arbitrary C# code in the generator without compromising type safety is given in Sect. 4.1.

In the Genoupe language a generator can be embedded into the source code in the same way an ordinary type definition. Generators can have parameters that are accessible in the generator code through *generator variables*. Unlike generic types, these parameters can be used to pass any kind of values into a generator, and not just types. In contrast to runtime variables, generator variables hold objects at generation-time and make them accessible in the generator code. Analogous to the parameters of an ordinary method, each declared generator parameter is a generator variable, which can be used in

generator expressions. A generator expression describes a value 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 if same values have been assigned to the generator variables, two structurally equivalent generator expressions describe the same value. This is guaranteed through the use of memoization during the evaluation of generator expressions, which is explained further in Sect. 4.1.

Generator expressions are often 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 represented by the expression's value is allowed to occur. For example, if we want 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 replace the generator expression by the name of the type represented by the type object during code generation. 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 describe important parts of the Genoupe syntax, and 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 [14,13].

2.1 Syntax

This section describes the syntax of the Genoupe language, using context-free grammar rules. After describing the syntax of generator expressions, the syntax of class generators with method and variable definitions will be given. Finally we discuss the syntax of statements as they appear in generated methods and constructors.

2.1.1 Generator Expressions

Generator expressions are similar to ordinary C# expressions in structure. However, they are evaluated not during runtime but during generation time. A generator expression has the following syntax:

$$ge \; : \; id \mid typeid \mid \texttt{@}gname\texttt{(}(geargs)^{?}\texttt{)@} \mid \texttt{@this@} \mid \texttt{@}gepart\texttt{@}$$

$$geargs : \; gepart \; (\texttt{,}gepart)^{*}$$

A generator expression is either an ordinary C# identifier *id*, an ordinary C# type name *typeid*, a generator call, the generated type `@this@`, or an expression *gepart* enclosed by `@` characters. In the first two cases, we call this a constant generator expression because the result is always the same identifier or type name. In the latter three cases, the generator expression may evaluate to different values depending on the generator parameters. A generator call may invoke any Genoupe generator with name *gname* in any of the source files given to the Genoupe precompiler. `@this@` refers to the type that is being generated. An expression *gepart* enclosed in `@`-signs may evaluate to any C# object. Arguments in a generator expression are comma-separated lists of C# expressions.

Many generator expressions are composed of partial generator expressions *gepart*, which have the following syntax:

$$gepart : \; literal \mid id \mid gepart\texttt{.}id \mid typeid\texttt{.}id$$

$$\mid \; gepart\texttt{.}id\texttt{(}(geargs)^{?}\texttt{)} \mid typeid\texttt{.}id\texttt{(}(geargs)^{?}\texttt{)}$$

$$\mid \; \texttt{new}\; typeid\texttt{(}(geargs)^{?}\texttt{)} \mid \texttt{typeof(}typeid\texttt{)} \mid \texttt{(}typeid\texttt{)}\; gepart$$

The grammar rule *gepart* describes a minimal syntax for C# expressions: an expression is either a literal, an identifier of a generator variable, a field access on an object, a static field access on a class, a method call on an object, a static method call on a class, a constructor call, an application of the `typeof` operator, or a type cast. For simplicity, other operators are not considered here since they can be wrapped in methods.

A literal is any of the standard C# literals, such as `"hello"` for a string, `21` for an integer or `null` for a null reference. An identifier is either the identifier of a generator parameter or an identifier of any other generator variable in the scope. How generator variables enter the scope will be described later on. A field access on an object can be performed if the expression before the dot operator denotes a valid object and the following identifier is a valid field name. A static field access can be performed if there is a valid class identifier before the dot, and the following identifier is a valid field name. A method call on an object can be performed if the expression before the dot operator denotes

a valid object and the following identifier is a valid method name, followed by optional arguments in brackets. A static method call can be performed if there is a valid class identifier before the dot and the following identifier is a valid method name, followed by optional arguments in brackets. A constructor call consists of the `new` operator, a type identifier and optional arguments in brackets. A type cast consists of a C# type name in brackets, followed by an expression.

A generator expression can be used to generate a program part. This is done by using that expression instead of the program part in the generator code. Analogously to an ordinary expression, a generator expression has a type. To generate a particular program part, the generator expression has to have a particular type. The following rules apply for the generation of a program part with a generator expression:

- To generate a type, the generator expression has to be of type `Type` or `GeneratedType`.
- To generate a variable or method identifier, the generator expression has to be of type `String`.
- To generate an integer literal, the generator expression has to be of type `Integer`.
- To generate a string literal, the generator expression has to be of type `StringLiteral`.

All those types, with the exception of `StringLiteral`, are standard C# classes. Type `StringLiteral` is necessary to distinguish string literals from identifiers: if a generator expression is of type string, then an identifier is generated (e.g. `foo`), but if it is of type `StringLiteral` then a string literal will be generated (e.g. `"foo"`). Other literals such as those for `Boolean`s or `Double`s can be generated using a generator expression of the corresponding C# type, e.g. `Boolean` or `Double`.

2.1.2 Generators and Definitions

A class generator has the following syntax:

generator: $(modifier)^*$ `class` *id*`(`$(gpars)^?$`)` $(\texttt{:}supertypes)^?$ { *gbody* }

gpars: *typename id* $(\texttt{,}typename\ id)^*$

supertypes: *ge*$(\texttt{,}ge)^*$

The generator begins with a sequence of modifiers such as `public` and the `class` keyword. The identifier of the generator is given by *id*. For Genoupe identifiers, the same rules apply as for ordinary C# identifiers. The generator parameters are defined in *gpars*, in the same way as in C# methods as a

comma separated list of type names and parameter identifiers. There can be an arbitrary number of generator parameters: the $()^?$ operator in the grammar above indicates that the parameters are optional; the $()^*$ operator indicates that there can be several other parameters, separated by commas, after the first one. For simplicity, we assume here that the types are class types. This is not a limitation as all other types, e.g. primitive types and arrays, can be wrapped in a class. The generator parameters can only be accessed by the generator code inside the generator. Similar to methods, the parameter list can be empty, but the pair of parentheses after the identifier has to remain to indicate that this is a generator and not an ordinary class. Normal brackets `()` are used instead of the angle brackets `<>` that are commonly used for generic types. This stresses that generators can have parameters of arbitrary type and not just type parameters, and keeps the Genoupe syntax separated from C#'s syntax for generic types. Generated classes can have supertypes, and a supertype can be generated with a generator expression *ge*.

In the following we will look at the syntax for the body of the generator *gbody*. This is where most of the generation usually takes place. In general, *gbody* is similar to the body of an ordinary C# class:

$$
\begin{array}{lll}
gbody & : & (vardef \mid methoddef \mid constructordef \mid gconst \mid gif \\
 & & \mid gfor)^* \\
vardef & : & (modifier)^*\ ge\ ge\texttt{;} \\
methoddef & : & (modifier)^*\ ge\ ge\texttt{(}(pars \mid gforpars)^?\texttt{)}\ \ \{\ mbody\ \} \\
constructordef & : & (modifier)^*\ \texttt{@this@(}(pars \mid gforpars)^?\texttt{)}\ \ \{\ mbody\ \} \\
pars & : & ge\ ge\ (\texttt{,}ge\ ge)^* \\
mbody & : & (stat)^*
\end{array}
$$

The generator body can contain an arbitrary number of definitions. Each definition is either a variable definition *vardef*, a method definition *methoddef*, a constructor definition *constructordef*, or a generator construct for defining a constant *gconst*, conditional generation *gif* or iterative generation *gfor*. A variable definition consists of a sequence of modifiers such as `public` or `private`, a generator expression specifying a type and a generator expression specifying an identifier. A method definition consists of a sequence of modifiers, a generator expression for the return type, a generator expression for the method identifier, and an optional specification of parameters enclosed in brackets. A constructor definition consists of a sequence of modifiers, the keyword `@this@`, which refers to the name of the type that is generated, and an optional specification of parameters enclosed in brackets. Parameters are either specified as a comma-separated list of pairs of generator expressions, one for the type and one for the parameter identifier, or by an iterative gener-

ation *gforpars*, which will be described later. A method or constructor body *mbody* consists of an arbitrary number of statements, which will be described in the next section.

We now want to look at the generator constructs that can be used in a generator body *gbody*. First of all, we look at the *gconst* constructs for defining generation-time constants. If a generator expression is used more than once, then it is convenient to define a new generator variable that holds the value of that expression. Instead of repeating the generator expression, the value of the generator variable can be used. Such a generator variable can be defined in the following way:

gconst: `@const` *id* `=` *gepart*`;`

The new generator variable has the identifier *id* and the value given by partial generator expression *gepart*. The `@const` keyword indicates that the generator variable is a constant at generation time. A value is only assigned once after evaluating the expression *gepart*.

If variable and method definitions in a class should only be generated under a particular condition, the *gif* construct for conditional generation is used. It has the following syntax:

gif: `@if(`*gepart*`)` $\{$ $(def)^*$ $\}$ `(else` $\{$ $(def)^*$ $\}$`)`$^?$

The condition *gepart* must be of type `Boolean`. The first sequence of definitions $(def)^*$ is generated if the condition is true, the second sequence, after the `else`, is generated if the condition is false. The else-clause is optional.

The construct for iterative generation of variable or method definitions in a class has the following syntax:

gfor: `@foreach(`*id* `in` *gepart*`)` $\{$ $(def)^*$ $\}$

gepart must evaluate to a collection type that implements the standard C# interface for collections `ICollection`. The standard collections and all arrays do that. *id* is the name for a new generator variable that is assigned each of the elements in the collection successively, and makes the element accessible in the loop body. The definitions $(def)^*$ in the loop body are generated once, for each element in the collection.

As pointed out above, iterative generation can also be used to generate method parameter specifications, and also to generate method arguments:

gforpars: `@foreach(`*id* `in` *gepart*`)` $\{$ *ge ge* $\}$

gforargs: `@foreach(`*id* `in` *gepart*`)` $\{$ *ge* $\}$

As specified in the rules above, *gforpars* can be used in a method definition to generate a list of parameter specifications. Analogous to *gfor*, *gepart* must evaluate to a collection type that implements the standard C# interface for collections `ICollection`. For each element in the collection, one parameter is generated. The first generator expression between the curly brackets generates the parameter type and the second generator expression generates the parameter name. Similarly, *gforargs* can be used in a method call to generate a list of arguments. For each element in the collection, one argument is generated. The arguments are generated by the generator expression between the curly braces. Arguments cannot be iteratively generated for generator expressions, therefore rule *gepart* does not contain *gforargs* in its specification of method arguments. Iteratively generated arguments are made available for generated C# expressions only as they appear in methods. This is described below.

2.1.3 Statements

The generator constructs can also be used in a method body to generate statements. A method body *mbody* has the following syntax:

$$stat :\ me\texttt{;} \mid ge\ \texttt{=}\ me\texttt{;} \mid \texttt{return}\ me\texttt{;} \mid gconst \mid gstatif \mid gstatfor$$

A statement *stat* is either an expression *me* followed by a semicolon, an assignment, a return statement, the definition of a generator constant *gconst*, a sequence of conditionally generated statements *gstatif*, or a sequence of iteratively generated statements *gstatfor*. We only consider some exemplary statement types; other types of statements work similarly to those presented here. An assignment consists of a generator expression for the identifier of the variable to assign the value to, the equal sign, and an expression *me* for the value followed by a semicolon. A return statement returns the value of an expression *me*.

$$\begin{aligned}
me\ &:\ literal \mid ge \mid me\texttt{.}id \mid ge\texttt{.}id \\
&\mid\ me\texttt{.}id\texttt{(}(args)^?\texttt{)} \mid ge\texttt{.}id\texttt{(}(args)^?\texttt{)} \mid \texttt{new}\ ge\texttt{(}(args)^?\texttt{)} \\
&\mid\ ge\texttt{(}gforargs\texttt{)} \mid \texttt{new @this@(}gforargs\texttt{)} \\
&\mid\ \texttt{typeof(}ge\texttt{)} \mid \texttt{(}ge\texttt{)}\ me \mid \texttt{this} \\
args\ &:\ me\ (\texttt{,}me)^*
\end{aligned}$$

An expression *me* is similar to the aforementioned simple C# expressions defined in *gepart*. However, there are a few differences. Identifiers are generated with generator expressions, which may be constant. When accessing a static field of a type, the type identifier is generated with a generator expression, which may be constant. Analogously, types for static method calls, construc-

tor calls, applications of `typeof` and type casts are generated with a generator expression as well. For example, the type of a constructor call can be `@this@`, referring to the generated type itself. The arguments of a method or constructor call can alternatively be generated iteratively with the aforementioned $gforargs$, if the method or constructor was defined in the generated class. The keyword `this` can be used to access the current object in a method, which is not possible in generator expressions as there is no current object during generation time.

The constructs $gstatif$ and $gstatfor$ for conditional and iterative generation in methods are almost identical to gif and $gfor$:

$gstatif$: `@if(`*gepart*`)` { (*stat*)* } (`else` { (*stat*)* })$^?$

$gstatfor$: `@foreach(`*id* `in` *gepart*`)` { (*stat*)* }

The only difference from gif and $gfor$ is that the bodies contain (*stat*)* instead of (*def*)*.

explain scoping of generator variables and parameters

2.2 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  }
```

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.

In order to generate a statically type safe stack containing `String` elements with the `Stack` generator, one would write a generator expression that applies

the generator to type `String`. This generator expression would evaluate to a `Type`, so it can be used in a variable definition:

```
@Stack(typeof(String))@ myStack = new @Stack(typeof(String))@();
```

A similar generator expression of type `Type` is used to generate the type name after the `new` operator.

2.3 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 [28].

The following code snippet shows a generator that takes a class `T` and a list of field names `FNames` for that class. It generates a subclass of `T` that provides a new method `Randomize`. This method assigns random values to those fields of `T` that have their name mentioned in `FNames`. This can be useful, for example, for the generation of test data.

```
1   public class Randomizeable(Type T, List<String> FNames) : @T@
2   {
3      Random r;
4
5      public void Randomize() {
6         r = new Random();
7         @foreach(F in T.GetFields()) {
8            @if(FNames.Contains(F.Name)) {
9               @if(F.FieldType==typeof(Double)) {
10                 @F.Name@ = r.NextDouble();
11              } else {
12                 @if(F.FieldType==typeof(Boolean)) {
13                    @F.Name@ = (r.NextDouble()>=0.5);
14                 }
15                 // ...handle other data types...
16              }
17        }  }
18  }  }
```

In line 7 we see the `@foreach` construct for iterative generation of statements, which is used to iterate through all the fields of `T`. The iteration variable `F` holds the `FieldInfo` object for the field that is currently processed in the loop. Only the fields mentioned in `FNames` should be assigned random values.

Therefore the `@if` in line 8 checks that the field name of the current field is in `FNames`, and only then a value is assigned. The following `@ifs` check which type the current field has, and generate an assignment to set an appropriate random value. The generator expression `@F.Name@` of type `String` is used to generate a field access to a field inherited from `T`.

2.4 Proxies and Wrappers

A common pattern for modifying the behavior of existing classes or bridging incompatibility is the use of proxies [20] and wrappers. With Genoupe both of these can be generated automatically, which makes it possible to address dynamic crosscutting concerns [28].

The following class generator takes a type parameter `T` and creates a subtype of `T` that overrides `T`'s methods. A class generated by this generator can be used to create mocking objects for `T`, i.e. objects that do not implement the real functionality but are convenient for testing. In this case, the methods only print out all method calls, which can be useful for debugging purposes.

```
1   public class Mock(Type T) : @T@
2   {
3       @foreach(M in T.GetMethods()) {
4           @const Pars = M.GetParameters();
5
6           public override @M.ReturnType@ @M.Name@
7           (@foreach(P in Pars) { @P.ParameterType@ @P.Name@ })
8           {
9               Console.WriteLine("Method "
10                  +@new StringLiteral(M.Name)@+" called.");
11          }
12  }   }
```

In lines 6 and 7, 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 `StringLiteral` object constructed in line 10 represents a generated string literal, opposed to a generated identifier.

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. 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 notion of type safety, which we want to call *generator type safety*. This 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 happens if T is a subclass of Component but is not compatible with 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 satisfactorily, otherwise it is ill-typed. This is also known as the problem of *inadvertent capture* [30].

The next generator contains a conditional generation.

```
1   class C(String X)
2   {
3       @if(X.Equals("hello")) {
4           String 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 an 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, although usually they occur in a more subtle 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 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, 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 recurse endlessly, and only a limited recursion-depth prevents this [10]. In other technologies which use a Turing-complete language for metaobject manipulation, like CLOS [19], OpenC++ [9] or Jasper [36,37], generators potentially do not terminate as well.

3.1 The Genoupe Type System

In order to detect generator type errors, we have 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 [8]. 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 that keeps track not only of the signatures of declared runtime variables but also of the signatures of generator variables, among other things. 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. Γ is also used to store additional facts about the code portion that is being type-checked.

The type system that we present here is provably decidable. The proof is rather straightforward and we give the main argument here. The naming scheme of the type rules indicates the order in which the rules will be applied. The following pseudocode represents the top-level sequence in which the type rules must be applied: the derivation can be denoted in a tree-like fashion, and the levels of the tree correspond to nesting levels of the two block concepts of the generator, namely $@if$ and $@foreach$ blocks. Our nesting depth is the combined depth for both blocks.

1. Apply [*env* ...] rules for primitive expressions
2. For each nesting level in the block structure:
   apply the [def ...] rules and then the [*env* ...] rules using
   those [def ...] rules.
3. Apply the [*generator*] rule.

The type system is decidable because we can bind a size metric of the derivation by a metric of the program: the derivation can be denoted in a tree-like fashion and the maximum depth of the tree is bound by the maximum nesting depth of the $@if$ and $@foreach$ blocks.

A derivation always starts with the rule [*env* $\emptyset$] for creating an empty environment Γ, which does not have any preconditions and can thus be applied first:

$$[env\ \emptyset]\ \frac{}{\emptyset \vdash \diamond}$$

The postcondition states that the empty environment is a well-formed environment. Usually a judgment of the form $\Gamma \vdash X$ states that a program part X in an environment Γ is correct. This rule states only that the environment on the left side of $\vdash$ is well-formed, so the $\diamond$ on the right side is just a placeholder for no program part at all.

3.1.1 Generator Expressions

In this section we look at rules for generator expressions. The syntax of generator expressions is defined by the grammar rules *ge*, *geargs* and *gepart*. Grammar rule *ge* treats identifiers *id* and *typeid* as constant generator expres-

sions. For simplicity, the following type rules will not support this. Without loss of generality, identifiers *id* and *typeid* have to be generated explicitly with constant generator expressions `@"`*id*`"@` and `@typeof(`*typeid*`)@`. Both can be derived with the following rules.

The Genoupe precompiler uses a function *Generators* to keep track of all the generators in the scope:

$$Generators\colon GeneratorIds \rightarrow (Types)^*$$

For each generator in the scope, the function maps the identifier of the generator, which is an element of set *GeneratorIds*, to the generator signature, which is an n-tuple of types. For example, the Randomizeable generator described above corresponds to an element $\texttt{Randomizeable} \mapsto (\texttt{Type}, \texttt{String[]})$. Generator calls are checked with rule [*ge gcall*]:

$$[ge\ gcall]\ \frac{\begin{array}{c} generatorname \in dom(Generators) \\ Generators(generatorname) = (type_1, \ldots, type_n) \\ \Gamma \vdash gepart_1 \colon\colon type_1 \;\ldots\; \Gamma \vdash gepart_n \colon\colon type_n \end{array}}{\Gamma \vdash \texttt{@}gname\texttt{(}gepart_1, \ldots, gepart_n\texttt{)@} \colon\colon \texttt{Type}}$$

The precondition requires that *generatorname* is the name of an accessible generator with the parameter types $type_1, \ldots, type_n$, and that $gepart_1, \ldots,$ $gepart_n$ are correct partial generator expressions of the same types. The $::$ symbol is used to associate a partial generator expression or generator expression with its type. By contrast, ordinary C# expressions are associated with a generator expression that generates their type using the : symbol, as we will see later on. The postcondition contains a generator expression that is a call to generator *generatorname*, using the partial generator expressions as arguments. Since generators generate types, the generator expression is of type `Type`.

Rule [*ge* `@this@`] can be used to check a generator expression that refers to the type that is generated:

$$[ge\ \texttt{@this@}]\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{@this@} \colon\colon \texttt{Type}}$$

If Γ is a well-formed environment, then `@this@` is a correct generator expression of type `Type`.

As specified in the syntax rule *ge*, we can use a partial generator expression *gepart* enclosed by `@`-signs to construct a generator expression:

$$[ge\ \texttt{@}gepart\texttt{@}]\ \frac{\Gamma \vdash gepart \colon\colon t}{\Gamma \vdash \texttt{@}gepart\texttt{@} \colon\colon t}$$

If *gepart* is a correct partial generator expression of type t, it can be enclosed in @-signs and the result is a generator expression of type t.

Similarly to ordinary C# expressions, partial generator expressions can contain literals, which are checked with rule [*gepart literal*]:

$$[\textit{gepart literal}]\ \frac{\Gamma \vdash \diamond \quad x \in \textit{Literals}(t)}{\Gamma \vdash x\!::t}$$

Literals is a function that maps types to their literal values, e.g.

$$\textit{Literals}(\texttt{int}) = \{\ldots, -1, -, 1, \ldots\}.$$

If Γ is a well-formed environment and x is a correct literal of type t, then a partial generator expression x can be formed that has type t.

Rule [*gepart id*] checks a partial generator expression that accesses a generator variable:

$$[\textit{gepart id}]\ \frac{\Gamma \cup \{id\!::t\} \vdash \diamond}{\Gamma \cup \{id\!::t\} \vdash id\!::t}$$

If there is a well-formed environment that contains the signature of a generator variable, then the identifier of the generator variable can be used as a partial generator expression that has the type of the generator variable. The signature of a generator variable consists of the variable identifier, the :: sign, and the variable type.

For all the possible elements in Γ, we need rules that allow us to add them to Γ. In principle, these rules make it possible to add things to the environment that are not correctly part of it. However, the rules that derive larger generator parts, e.g. rule [*generator*] later on, require that their constituents are derived in correct environments. So if we derived a generator part using an incorrect environment, it would no longer be possible to combine that part with other parts to form a complete generator. Rule [*env gvar*] can be used to add the signature of a generator variable, e.g. a generator parameter or the iteration variable of a @*foreach*-loop, to the environment:

$$[\textit{env gvar}]\ \frac{\Gamma \vdash \diamond \quad t \in \textit{Types} \quad (\text{generator variable } ID) \notin \Gamma}{\Gamma \cup \{ID\!::t\} \vdash \diamond}$$

The precondition makes sure that Γ is a well-formed environment, that t is a valid C# type, and that the identifier ID is not already defined as a generator variable in the environment, which would indicate a collision of identifiers.

Rule [*gepart field*] is used to derive a partial generator expression that accesses a field of an object:

$$[\textit{gepart field}]\ \frac{\Gamma \vdash \textit{gepart}\!::s \quad s \text{ has accessible field } id\!:t}{\Gamma \vdash \textit{gepart}.id\!::t}$$

If *gepart* is a correct partial generator expression of type t, and t has an accessible field *id* of type t, then $gepart.id$ is a correct partial generator expression of type t. The rules to determine whether type s has an accessible field *id* are the standard C# rules.

Rule [*gepart sfield*] is used to derive a partial generator expression that accesses a static field of a type:

$$[gepart\ sfield]\ \frac{typeid \in Types \quad typeid \text{ has accessible static field } id\colon t}{\Gamma \vdash typeid.id\colon\colon t}$$

If *typeid* is a correct C# type identifier, and type *typeid* has an accessible static field *id* of type t, then $typeid.id$ is a correct partial generator expression of type t. The rules to determine whether type *typeid* has an accessible field *id* are the standard C# rules.

Rule [*gepart call*] is used to derive method calls in partial generator expressions:

$$[gepart\ call]\ \frac{\begin{array}{c}\Gamma \vdash gepart\colon\colon s \\ s \text{ has accessible method } id\colon t_1 \times \ldots \times t_n \to t \\ \Gamma \vdash gepart_1\colon\colon t_1\ \ldots\ \Gamma \vdash gepart_n\colon\colon t_n\end{array}}{\Gamma \vdash gepart.id(gepart_1, \ldots, gepart_n)\colon\colon t}$$

The precondition requires three things. First, *gpart* is a correct partial generator expression of type s. Second, s has an accessible method *id* taking n parameters of the types $t_1, \ldots, t_n$. This is checked using the standard C# rules. Third, $gepart_1, \ldots, gepart_n$ are n correct generator expressions in the same environment as *gepart* and have the same types as the method parameters $t_1, \ldots, t_n$. With these requirements satisfied, a partial generator expression can be derived consisting of a call to method *id* on *gpart* using $gepart_1, \ldots, gepart_n$ as arguments. This partial generator expression has the return type of method *id*.

Rule [*gepart scall*] is used to derive static method calls in partial generator expressions:

$$[gepart\ scall]\ \frac{\begin{array}{c}typeid \in Types \\ typeid \text{ has accessible static method } id\colon t_1 \times \ldots \times t_n \to t \\ \Gamma \vdash gepart_1\colon\colon t_1\ \ldots\ \Gamma \vdash gepart_n\colon\colon t_n\end{array}}{\Gamma \vdash typeid.id(gepart_1, \ldots, gepart_n)\colon\colon t}$$

If *typeid* refers to a correct C# type with a static method *id* that takes n parameters of the types $t_1, \ldots, t_n$, and there are n correct partial generator

expressions in the same environment with the same types $t_1, \ldots, t_n$, then a partial generator expression can be formed for a static method call on *id*. The availability of an appropriate static method is checked using the standard C# rules. $gepart_1, \ldots, gepart_n$ are used as arguments and the resulting partial generator expression has the return type of the method *id*.

Rule [*gepart* `new`] is used to derive constructor calls in partial generator expressions:

$$[\mathit{gepart}\;\texttt{new}]\;\frac{\begin{array}{c}\mathit{typeid} \in \mathit{Types} \\ \mathit{typeid} \text{ has accessible constructor with parameters } : t_1 \times \ldots \times t_n \\ \Gamma \vdash \mathit{gepart}_1 :: t_1 \;\ldots\; \Gamma \vdash \mathit{gepart}_n :: t_n\end{array}}{\Gamma \vdash \texttt{new}\;\mathit{typeid}\texttt{(}\mathit{gepart}_1, \ldots, \mathit{gepart}_n\texttt{)} :: \mathit{typeid}}$$

If *typeid* refers to a correct C# type with a constructor that takes n parameters of the types $t_1, \ldots, t_n$, and there are n correct partial generator expressions in the same environment with the same types $t_1, \ldots, t_n$, then a partial generator expression can be formed for a constructor call on *id*. Whether *typeid* has an appropriate constructor is checked using the standard C# rules. $gepart_1, \ldots, gepart_n$ are used as arguments and the resulting partial generator expression has type *typeid*.

Rule [*gepart* `typeof`] is used to derive an application of the `typeof` operator in a partial generator expression:

$$[\mathit{gepart}\;\texttt{typeof}]\;\frac{\Gamma \vdash \diamond \quad \mathit{typeid} \in \mathit{Types}}{\Gamma \vdash \texttt{typeof(}\mathit{typeid}\texttt{)} :: \texttt{Type}}$$

If Γ is a well-formed environment and *typeid* is a valid C# type, then we can derive that `typeof(`*typeid*`)` is a correct partial generator expression of type `Type`. The resulting `Type` object is the one representing type *typeid*.

Rule [*gepart cast*] is used to derive an application of the type cast operator in a partial generator expression:

$$[\mathit{gepart\ cast}]\;\frac{\mathit{typeid} \in \mathit{Types} \quad \Gamma \vdash \mathit{gepart} :: t}{\Gamma \vdash \texttt{(}\mathit{typeid}\texttt{)}\,\mathit{gepart} :: \mathit{typeid}}$$

If *typeid* is a valid C# type and *gepart* a correct partial generator expression, then `(`*typeid*`)`*gepart* is also a correct partial generator expression of type *typeid*. Note that by their very nature, type casts may fail during runtime. However, using the standard C# rules, some type casts that are guaranteed to fail can be detected statically.

3.1.2 Generators and Definitions

In this section we describe the type rules for checking the correctness of a generator and the variable and method definitions it generatesd by it. For simplicity, we omit the rules for the `@const` generator construct since it is only syntactic sugar for the generator developer. Using `@const` to define a new generator variable with an expression is equivalent to substituting the expression for the name of the generator variable wherever it is used. For example, `@const T = "x"; int @T@;` is equivalent to `int @"x"@;`. We also do not consider modifiers in the following type rules.

In the following we will describe the rules for the definitions that can be generated in the body of the generator, starting with the generation of variable definitions:

$$[def\ var]\ \frac{\Gamma \vdash ge_{type} :: \mathtt{Type} \quad \Gamma \vdash ge_{id} :: \mathtt{String}}{\Gamma \vdash ge_{type}\ ge_{id}\ \mathtt{;}\ \therefore \{ge_{id} : ge_{type}\}}$$

This rule can be used to derive a local variable definition in the generator body. The precondition requires suitable generator expressions for generating the variable's identifier and type. The $\therefore$ symbol in the postcondition associates a signature with the definition. The signature is a set that contains the information that needs to be added to the environment so that the variable can be used in other parts of the generator. For the above variable definition, the signature contains only the generator expression that generates the variable identifier and the generator expression that generates the variable type, separated by the : symbol.

Rule [*env var*] inserts the signature of a generated variable into the environment, so that it can be used in other parts of the generator:

$$[env\ var]\ \frac{\Gamma \vdash ge_{id} :: \mathtt{String} \quad \Gamma \vdash ge_{type} :: \mathtt{Type} \quad (\text{variable } ge_{id}) \notin \Gamma}{\Gamma \cup \{ge_{id} : ge_{type}\} \vdash \diamond}$$

The precondition states 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. Furthermore, the generator expression ge_{id} must not already be used to generate another variable identifier in Γ to help avoid identifier collisions. Note how the symbol between an expression and its type is used to express whether an expression is an ordinary expression or a generator expression. In the case of an ordinary expression, the symbol between the expression and its type is :; in the case of a generator expression the symbol is ::. In the postcondition the new environment is a conjunction of the old Γ and the new signature. In a signature, the : symbol associates the generator expression that generates the identifier of the variable with the generator expression that generates its type.

Rule [*def method*] can be used to derive a method definition in the body of a generator:

$$[def\ method]\ \frac{\begin{array}{l}\Gamma \vdash ge_{id}\texttt{::String} \quad \Gamma \vdash ge_{ret}\texttt{::Type} \\ \Gamma \vdash ge_{id1}\texttt{::String}, \ldots, \Gamma \vdash ge_{idn}\texttt{::String} \\ \Gamma \vdash ge_{t1}\texttt{::Type}, \ldots, \Gamma \vdash ge_{tn}\texttt{::Type} \\ \Gamma \cup \{ge_{id}\colon ge_{t1} \times \ldots \times ge_{tn} \rightarrow ge_{ret}, \\ \quad ge_{id1}\colon ge_{t1}, \ldots, ge_{idn}\colon ge_{tn}\} \vdash stat_i \quad \text{for } i \in \{1, \ldots, k\}\end{array}}{\begin{array}{l}\Gamma \vdash ge_{ret}\ ge_{id}\texttt{(}ge_{t1}\ ge_{id1}, \ldots, ge_{tn}\ ge_{idn}\texttt{)}\ \ \{stat_1 \ldots stat_k\} \\ \quad \therefore \{ge_{id}\colon ge_{t1} \times \ldots \times ge_{tn} \rightarrow ge_{ret}\}\end{array}}$$

The precondition requires appropriate generator expressions for the method identifier, method return type, the identifiers of the method parameters and the types of the method parameters. Furthermore, it requires k correct statements for the method body. The environment of the statements contains the signature of the method itself so that it can be called recursively from within the method body, as well as the signatures of the method parameters. The postcondition contains a method definition in which all types and identifiers are generated with – possibly constant – generator expressions. The method body contains the k statements. The signature of a method, i.e. the set after the $\therefore$ sign, consists of the generator expression for the method identifier and the method type, separated by the : symbol. The type consists of a cartesian product $\times$ of the generator expressions for the parameter types and the generator expression for the return type, separated by the $\rightarrow$ sign.

Rule [*env method*] inserts the signature of a generated method into the environment, so that it can be used in other parts of the generator:

$$[env\ method]\ \frac{\begin{array}{l}\Gamma \vdash ge_{id}\texttt{::String} \quad \Gamma \vdash ge_{ret}\texttt{::Type} \\ \Gamma \vdash ge_{t1}\texttt{::Type}, \ldots, \Gamma \vdash ge_{tn}\texttt{::Type} \\ (\text{method } ge_{id}\colon ge_{t1} \times \ldots \times ge_{tn}) \notin \Gamma\end{array}}{\Gamma \cup \{ge_{id}\colon ge_{t1} \times \ldots \times ge_{tn} \rightarrow ge_{ret}\} \vdash \diamond}$$

This requires a generator expression ge_{id} for generating the method identifier, generator expressions $ge_{t1}, \ldots, ge_{tn}$ for generating the method parameter types, and a generator expression ge_{ret} for generating the method return type. The generator expression that generates an identifier has to be of type `String`, and the generator expressions generating the types have to be of type `Type`. To help avoid identifier collisions, ge_{id} must not yet be used in Γ to generate an identifier of a method with the same parameter types. The result is a well-formed environment that contains a method signature.

Rule [*def constructor*] is used to derive a constructor definition in the body of the generator:

$$[\mathit{def\ constructor}]\ \frac{\begin{array}{l}\Gamma \vdash ge_{id1}\!::\texttt{String}, \ldots, \Gamma \vdash ge_{idn}\!::\texttt{String} \\ \Gamma \vdash ge_{t1}\!::\texttt{Type}, \ldots, \Gamma \vdash ge_{tn}\!::\texttt{Type} \\ \Gamma \cup \{\texttt{@this@}\!: ge_{t1} \times \ldots \times ge_{tn} \to \texttt{@this@}, \\ \quad ge_{id1}\!: ge_{t1}, \ldots, ge_{idn}\!: ge_{tn}\} \vdash stat_i \quad \text{for } i \in \{1, \ldots, k\}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{@this@(}ge_{t1}\ ge_{id1}, \ldots, ge_{tn}\ ge_{idn}\texttt{)}\ \{stat_1 \ldots stat_k\} \\ \quad \therefore \{\texttt{@this@}\!: ge_{t1}, \ldots, ge_{idn} \to \texttt{@this@}\}\end{array}}$$

Similarly to a method definition, the precondition requires correct generator expressions for the constructor parameter identifiers and types. However, since the constructor has the same identifier and return type as the type it is associated with, no generator expressions for the identifier and return type are required. Apart from that, the rule works analogously to rule [*env method*]. The signature of a constructor for the generated type always has identifier and return type `@this@`, which is the generator expression denoting the generated type itself.

Rule [*env constructor*] inserts the signature of a generated constructor into the environment, so that it can be used in other parts of the generator:

$$[\mathit{env\ constructor}]\ \frac{\Gamma \vdash ge_{t1}\!::\texttt{Type}, \ldots, \Gamma \vdash ge_{tn}\!::\texttt{Type}}{\Gamma \cup \{\texttt{@this@}\!: ge_{t1} \times \ldots \times ge_{tn} \to \texttt{@this@}\} \vdash \diamond}$$

The rule is analogous to rule [*env method*].

Rule [*def* `@if`] is used to derive conditional generation of definitions in a generator body:

$$[\mathit{def}\ \texttt{@if}]\ \frac{\begin{array}{l}\Gamma \vdash gepart\!::\texttt{Boolean} \\ \Gamma \cup sig_1 \cup \ldots \cup sig_m \cup \{gepart\} \vdash def_1 \therefore sig_1, \ldots, \\ \Gamma \cup sig_1 \cup \ldots \cup sig_m \cup \{gepart\} \vdash def_m \therefore sig_m \\ \Gamma \cup sig'_1 \cup \ldots \cup sig'_n \cup \{\neg gepart\} \vdash def'_1 \therefore sig'_1, \ldots, \\ \Gamma \cup sig'_1 \cup \ldots \cup sig'_n \cup \{\neg gepart\} \vdash def'_n \therefore sig'_n\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{@if(}gepart\texttt{)}\ \{\ def_1 \ldots def_m\ \}\ \text{else}\ \{\ def'_1 \ldots def'_n\ \} \\ \therefore \{gepart \to sig_1, \ldots, gepart \to sig_m, \\ \quad \neg gepart \to sig'_1, \ldots, \neg gepart \to sig'_n\}\end{array}}$$

The precondition requires three things. First, *gepart* must be a correct partial generator expression of type `Boolean`. Secondly, the m definitions $def_1, \ldots,$ def_m with signatures $sig_1, \ldots, sig_m$ must be correct in an environment that contains their signatures and the element *gepart*. The signatures are contained in the environment because this enables recursive definitions, e.g. a the definition of a recursive method that calls itself. The element *gepart* in the environment of the definitions $def_1, \ldots, def_m$ signifies that the definitions are generated in the then-clause of a conditional generation where the condition *gepart* is true. This element is used later on to make sure that conditionally generated definitions are only used if the condition under which they were generated is true. Thirdly, the precondition requires that the n definitions $def'_1, \ldots, def'_n$ with signatures $sig'_1, \ldots, sig'_n$ must be correct in an environment that contains their signatures and the element $\not{gepart}$. Analogous to the second requirement, these definitions are generated in the else-clause of a conditional generation, and element $\not{gepart}$ signifies that in the environment the condition is false. The postcondition contains the conditional generation of definitions with a special signature on the right side of the $\therefore$ symbol. The signatures of the definitions $def_1, \ldots, def_m$ are prefixed with $gepart \rightarrow$ to express that the definitions are only accessible when *gepart* is true, i.e. when they were actually generated. Analogously, the signatures of the definitions $def'_1, \ldots, def'_n$ are prefixed with $\neg gepart \rightarrow$.

Rule [*env* `@if`] inserts the signatures of conditionally generated definitions into the environment, so that they can be used in other parts of the generator:

$$[env\ \texttt{@if}]\ \frac{\begin{array}{l}\Gamma \vdash gepart\texttt{::Boolean} \\ \Gamma \vdash def_1 \therefore sig_1, \ldots, \Gamma \vdash def_m \therefore sig_m \\ \Gamma \vdash def'_1 \therefore sig'_1, \ldots, \Gamma \vdash def'_n \therefore sig'_n\end{array}}{\begin{array}{l}\Gamma \cup \{gepart \rightarrow sig_1, \ldots, gepart \rightarrow sig_m, \\ \qquad \neg gepart \rightarrow sig'_1, \ldots, \neg gepart \rightarrow sig'_n\} \vdash \diamond\end{array}}$$

If there are both a partial generator expression *gepart* of type `Boolean` and correct definitions, then the signatures of the definitions can be prefixed with either $gepart \rightarrow$ or $\neg gepart \rightarrow$ and inserted into the environment.

Rule [*env then*] registers in Γ that a partial generator expression *gepart* evaluates to true:

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

The partial generator expression must be of type `Boolean` and the opposite, i.e. that *gepart* evaluates to false, must not be registered in Γ already. The rule is used for type checking in the then-clause of an `@if` construct, where

the partial generator expression describing the condition of the `@if` is known to be true. In such an environment, definitions that were generated under the condition *gepart* are accessible, as we will see later. Analogously, rule [*env else*] registers in Γ that a partial generator expression *gepart* evaluates to false:

$$[\mathit{env\ else}]\ \frac{\Gamma \vdash \mathit{gepart}\!::\!\texttt{Boolean} \quad \mathit{gepart} \notin \Gamma}{\Gamma \cup \{\neg \mathit{gepart}\} \vdash \diamond}$$

The rule is used for type checking in the else-clause of an `@if` construct, where the partial generator expression describing the condition of the `@if` is known to be false. In such an environment, definitions that were generated under the condition $\neg$*gepart* are accessible.

Rule [*def* `@foreach`] derives iterative generation of definitions in a generator body using the `@foreach` construct:

$$[\mathit{def}\ \texttt{@foreach}]\ \frac{\begin{array}{l}\Gamma \vdash \mathit{gepart}\!::\!t \quad t \text{ implements } \texttt{ICollection} \\ \Gamma \cup \mathit{sig}_1 \cup \ldots \cup \mathit{sig}_n \cup \{ID\!::\!t, ID \in \mathit{gepart}\} \\ \quad \vdash \mathit{def}_1 \therefore \mathit{sig}_1, \ldots, \\ \Gamma \cup \mathit{sig}_1 \cup \ldots \cup \mathit{sig}_n \cup \{ID\!::\!t, ID \in \mathit{gepart}\} \\ \quad \vdash \mathit{def}_n \therefore \mathit{sig}_n \end{array}}{\begin{array}{l}\Gamma \vdash \texttt{@foreach(}ID\ \texttt{in}\ \mathit{gepart}\texttt{)}\ \{\ \mathit{def}_1 \ldots \mathit{def}_n\ \} \\ \quad \therefore \{\forall \alpha \in \mathit{gepart}.\mathit{sig}_1[\alpha/ID], \ldots, \\ \qquad \forall \alpha \in \mathit{gepart}.\mathit{sig}_n[\alpha/ID]\}\end{array}}$$

The precondition requires three things. First, *gepart* is a correct partial generator expression with type t. Second, t implements the standard c# interface `ICollection`, which means that it is a collection type that can be iterated over. Third, the definitions $\mathit{def}_1, \ldots, \mathit{def}_n$ with signatures $\mathit{sig}_1, \ldots, \mathit{sig}_n$ are correct in an environment that contains their own signatures, the generator variable ID of type t, and the element $ID \in \mathit{gepart}$. The signatures of the definitions themselves are contained in the environment to enable recursive definitions. The generator variable ID is the iteration variable used in the `@foreach`-loop that generates the definitions, and hence can be used in generator expressions occurring in the definitions. The element $ID \in \mathit{gepart}$ indicates that the environment is located in a `@foreach`-loop that uses an iteration variable with identifier ID to iterate over a collection *gepart*. The postcondition contains a correct iterative generation of definitions, using ID as the iteration variable and *gepart* as the collection to iterate over. The loop body consists of the definitions $\mathit{def}_1, \ldots, \mathit{def}_n$, which are generated for each iteration. The iterative generation of definitions has a special signature: it uses the prefix $\forall \alpha \in \mathit{gepart}.$ to signify that, for each element α of the col-

lection that was iterated over, a definition was generated. After the prefixes, we have the signatures of the definitions in the loop body, $sig_1, \ldots, sig_n$, with the identifier of the iteration variable ID replaced by the special universally quantified variable α. The special symbol α is used instead of ID because the generated definitions do not depend on the name of the iteration variable; they merely depend on the collection that was iterated over. Hence, the signatures are normalized and no longer contain the particular identifier ID. By using the symbol α, which cannot appear syntactically in a generator, clashes with other generator variables in the definitions are avoided.

Rule [*env* `@foreach`] inserts the signature of an iterative generation of definitions into the environment, so that the definitions can be used in other parts of the generator:

$$[\textit{env}\ \texttt{@foreach}]\ \frac{\begin{array}{l}\Gamma \vdash \textit{gepart}::t \quad t \text{ implements } \texttt{ICollection} \\ \Gamma \vdash \textit{def}_1 \therefore \textit{sig}_1, \ldots, \Gamma \vdash \textit{def}_n \therefore \textit{sig}_n\end{array}}{\begin{array}{l}\Gamma \cup \{\forall\alpha \in \textit{gepart}.\textit{sig}_1[\alpha/ID], \ldots, \\ \qquad \forall\alpha \in \textit{gepart}.\textit{sig}_n[\alpha/ID]\} \vdash \diamond\end{array}}$$

If there is a partial generator expression *gepart* of type t, t is a collection type, and $\textit{def}_1, \ldots, \textit{def}_n$ are correct definitions, then after replacing any occurrence of an identifier ID with the symbol α, the signatures can be prefixed with $\alpha \in \textit{gepart}.$ and inserted into the environment.

Rule [*env loop*] registers in Γ that an iterator variable of a `@foreach` contains an element of a particular collection:

$$[\textit{env loop}]\ \frac{\Gamma \vdash \textit{gepart}::t \quad t \text{ implements } \texttt{ICollection} \quad (ID \in \textit{gepart}) \notin \Gamma}{\Gamma \cup \{ID \in \textit{gepart}\} \vdash \diamond}$$

The type t of the partial generator expression must be a collection type and the element $(ID \in \textit{gepart})$ must not already be part of Γ. The rule is used for type checking the body of a `@foreach`-loop, where the iteration variable contains an element of the partial generator expression *gepart* over which is iterated. In such an environment, definitions that were generated while iterating over elements of *gepart* are accessible, as we will see later.

Rule [*def gmethod*] derives a method definition where the method parameters

are generated iteratively:

$$[def\ gmethod]\ \frac{\begin{array}{l}\Gamma \vdash ge_{id}\colon\colon \texttt{String} \quad \Gamma \vdash ge_{ret}\colon\colon \texttt{Type} \\ \Gamma \vdash gepart\colon\colon t \quad t \text{ implements } \texttt{ICollection} \\ \Gamma \vdash ge_{pid}\colon\colon \texttt{String} \quad \Gamma \vdash ge_{pt}\colon\colon \texttt{Type} \\ \Gamma \cup \{ge_{id}\colon (\forall\alpha \in gepart.ge_{pt}[\alpha/ID]) \rightarrow ge_{ret}, \\ \qquad \forall\alpha \in gepart.(ge_{pid}\colon ge_{pt}[\alpha/ID])\} \vdash stat_i \\ \quad \text{for } i \in \{1,\ldots,k\}\end{array}}{\begin{array}{l}\Gamma \vdash ge_{ret}\ ge_{id}\texttt{(@foreach(}ID\ \texttt{in}\ gepart\texttt{)}\ \texttt{\{}\ ge_{pt}\ ge_{pid}\ \texttt{\})} \\ \qquad \{stat_1 \ldots stat_k\} \\ \qquad \therefore \{ge_{id}\colon (\forall\alpha \in gepart.ge_{pt}[\alpha/ID]) \rightarrow ge_{ret}\}\end{array}}$$

The precondition requires the following. First, there must be correct generator expressions for the method identifier and return type. Secondly, there must be a partial generator expression *gepart* of a collection type. Thirdly, there must be correct generator expressions that can be used to generate a parameter identifier and parameter type. Fourthly, $stat_1, \ldots, stat_k$ must be correct statements in an environment that includes the signature of the generated method and the signatures of the generated method parameters. The signature of the generated method, which is described in detail below, is included to enable recursive method calls. The signature of the iteratively generated method parameters is analogous to the signature of the iteratively generated definitions in rule [*def* `@foreach`]. The difference is that the exact nature of the signature after the prefix $\forall\alpha \in gepart.$ is known: the definition is a parameter definition, which has the structure of a variable definition as described in rule [*def var*]. The precondition consists of a method definition in which the parameters are generated by iterating over the elements of collection *gepart*. The signature of the definition is similar to that of an ordinary method definition, as described in rule [*def method*]. However, the parameters are iteratively generated for each element of *gepart*. Therefore after all occurrences of the iteration generator variable ID have been replaced with the special symbol α for normalization, the generator expression for the parameter types ge_{pt} is prefixed with $\forall\alpha \in gepart.$, similarly to rule [*def* `@foreach`].

Rule [*env gmethod*] inserts the signature of a method with iteratively generated parameters into the environment, so that the method can be used in

other parts of the generator:

$$[env\ gmethod]\ \frac{\begin{array}{c}\Gamma \vdash ge_{id}\colon\colon \texttt{String} \quad \Gamma \vdash ge_{ret}\colon\colon \texttt{Type} \\ \Gamma \vdash gepart\colon\colon t \quad t \text{ implements } \texttt{ICollection} \\ \Gamma \vdash ge_{pid}\colon\colon \texttt{String} \quad \Gamma \vdash ge_{pt}\colon\colon \texttt{Type} \\ (\text{method } ge_{id}\colon (\forall\alpha \in gepart.ge_{pt}[\alpha/ID])) \notin \Gamma\end{array}}{\Gamma \cup \{ge_{id}\colon (\forall\alpha \in gepart.ge_{pt}[\alpha/ID]) \rightarrow ge_{ret}\} \vdash \diamond}$$

There must be appropriate generator expressions for the method identifier and return type. The type t of the partial generator expression must be a collection type. There must also be appropriate generator expressions for a parameter identifier and parameter type. There must not be a method signature in the environment already where ge_{id} is used to generate the method identifier and the parameter types are generated with generator expression $ge_{pt}[\alpha/ID]$ by iterating over the elements of $gepart$.

Rule $[env\ gargs]$ inserts the signature of iteratively generated method or constructor parameters into the environment, so that the method or constructor arguments can be accessed in the method or constructor body:

$$[env\ gargs]\ \frac{\begin{array}{c}\Gamma \vdash gepart\colon\colon t \quad t \text{ implements } \texttt{ICollection} \\ \Gamma \vdash ge_{pid}\colon\colon \texttt{String} \quad \Gamma \vdash ge_{pt}\colon\colon \texttt{Type} \\ (\forall\alpha \in gepart.(ge_{pid}\colon ge_{pt})[\alpha/ID]) \notin \Gamma\end{array}}{\Gamma \cup \{\forall\alpha \in gepart.(ge_{pid}\colon ge_{pt})[\alpha/ID]\} \vdash \diamond}$$

The type t of the partial generator expression must be a collection type. There must also be appropriate generator expressions for a parameter identifier and parameter type. There must not be a parameter signature in the environment already where $ge_{pid}[\alpha/ID]$ and $ge_{pt}[\alpha/ID]$ are used to generate parameters by iterating over the elements of $gepart$.

Analogously to rule $[def\ gmethod]$, rule $[def\ gconstructor]$ derives a construc-

tor definition where the constructor parameters are generated iteratively:

$$[def\ gconstructor]\ \frac{\begin{array}{l}\Gamma \vdash gepart\colon\colon t \quad t \text{ implements } \texttt{ICollection} \\ \Gamma \vdash ge_{pid}\colon\colon \texttt{String} \quad \Gamma \vdash ge_{pt}\colon\colon \texttt{Type} \\ \Gamma \cup \{\texttt{@this@}\colon (\forall \alpha \in gepart.ge_{pt}[\alpha/ID]) \rightarrow \texttt{@this@}, \\ \quad \forall \alpha \in gepart.(ge_{pid}\colon ge_{pt}[\alpha/ID])\} \vdash stat_i \\ \text{for } i \in \{1, \ldots, k\}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{@this@(@foreach(}ID \texttt{ in } gepart\texttt{) \{ } ge_{pt}\ ge_{pid} \texttt{ \})} \\ \quad \{stat_1 \ldots stat_k\} \\ \quad \therefore \{\texttt{@this@}\colon (\forall \alpha \in gepart.ge_{pt}[\alpha/ID]) \rightarrow \texttt{@this@}\}\end{array}}$$

The difference to rule [*def gmethod*] is that the identifier as well as the return type of generated constructors is `@this@`.

Analogously to rule [*env gmethod*], rule [*env gconstructor*] inserts the signature of a constructor with iteratively generated parameters into the environment, so that the constructor can be used in other parts of the generator:

$$[env\ gconstructor]\ \frac{\begin{array}{c}\Gamma \vdash gepart\colon\colon t \quad t \text{ implements } \texttt{ICollection} \\ \Gamma \vdash ge_{pid}\colon\colon \texttt{String} \quad \Gamma \vdash ge_{pt}\colon\colon \texttt{Type} \\ (\texttt{@this@}\colon (\forall \alpha \in gepart.ge_{pt}[\alpha/ID]) \rightarrow \texttt{@this@}) \notin \Gamma\end{array}}{\Gamma \cup \{\texttt{@this@}\colon (\forall \alpha \in gepart.ge_{pt}[\alpha/ID]) \rightarrow \texttt{@this@}\} \vdash \diamond}$$

The last rule that is applied in a type derivation is [*generator*], which is defined

as follows:

$$
[generator]\ \frac{\begin{array}{l}
\{id_1{:}{:}t_1, \ldots, id_m{:}{:}t_m\} \vdash \texttt{@}gepart_{t1}\texttt{@}{:}{:}\texttt{Type} \\
\{id_1{:}{:}t_1, \ldots, id_m{:}{:}t_m\} \vdash ge_{t2}{:}{:}\texttt{Type}, \ldots, \\
\{id_1{:}{:}t_1, \ldots, id_m{:}{:}t_m\} \vdash ge_{tk}{:}{:}\texttt{Type} \\
ge_{t2} \in Interfaces, \ldots, ge_{tk} \in Interfaces \\
\{id_1{:}{:}t_1, \ldots, id_m{:}{:}t_m\} \cup sig_1 \cup \ldots \cup sig_n \cup \\
\quad \{\forall \alpha \in (gepart_{t1}\texttt{.GetFields()}).(\texttt{@}\alpha\texttt{.Name@}{:}\,\texttt{@}\alpha\texttt{.FieldType@}), \\
\quad\ \forall \alpha \in (gepart_{t1}\texttt{.GetMethods()}).(\texttt{@}\alpha\texttt{.Name@}{:} \\
\qquad (\forall \alpha' \in (\alpha\texttt{.GetParameters()}).\texttt{@}\alpha'\texttt{.ParameterType@}) \\
\qquad \rightarrow \texttt{@}\alpha\texttt{.ReturnType@})\} \vdash def_i \therefore sig_i \quad \text{for } i \in \{1, \ldots, n\}
\end{array}}{\begin{array}{l}
\emptyset \vdash \texttt{class}\ id\ (t_1\ id_1, \ldots, t_m\ id_m)\ \texttt{:}\ \texttt{@}gepart_{t1}\texttt{@}, ge_{t2}, \ldots, ge_{tk} \\
\{def_1 \ldots def_n\}
\end{array}}
$$

This rule derives a class generator with the name id, m generator parameters $id_1, \ldots, id_m$ with types $t_1, \ldots, t_m$, k supertypes generated with generator expressions $\texttt{@}gepart_{t1}\texttt{@}$, $ge_{t2}, \ldots, ge_{tk}$, and n generated member definitions $def_1, \ldots, def_n$ with signatures $sig_1, \ldots, sig_n$. The precondition of this rule requires three things. First, the generator expressions used to generate the supertypes must be valid and of type `Type` in an environment that contains only the generator parameters. Secondly, all but the first generator expression must generate an interface. This is a requirement of the C# language, and it means that all but the first generator expression are constant so that it is evident whether an interface is generated or not. Without loss of generality, the first generator expression is written as a partial generator expression enclosed in `@`-signs. This is because we need to refer to the partial generator expression that it is made up of later on in the precondition. Thirdly, all the definitions def_i, $i \in \{1, \ldots, m\}$ that will be placed into the body of the generator must be correct in an environment that contains the following signatures: the generator's parameters, all the definitions' signatures and signatures for the fields and methods that are inherited from the superclass. The environment in which the definitions must be derived contains exactly those elements that are in the scope of the generator. If the precondition holds, then the generator is correct.

The environment contains the signature sig_i of def_i of each definition so that a definition may use itself recursively, e.g. a recursive method. The fields and methods of the superclass are included in the environment by pretending that they are generated iteratively in the generator body, using appropriate collections. This makes it possible to derive accesses to superclass fields and methods correctly by iterating over the same collections, even though the su-

perclass is not known statically. According to the first signature after sig_n, the superclass' fields are generated in the generator body by iterating over the collection $gepart_{t1}$`.GetFields()`, with $gepart_{t1}$ being the partial generator expression describing the superclass' `Type` object. `GetFields()` returns a collection of `FieldInfo` objects describing the fields of the superclass. According to the signature, the identifier of each field is generated using the `Name` field of the respective `FieldInfo` object, and the type using the `FieldType` field. A similar approach is taken for the signatures of the superclass' methods. According to the following signature, the superclass' methods are generated by iterating over the collection $gepart_{t1}$`.GetMethods()`. `GetMethods()` returns a collection of `MethodInfo` objects describing the methods of the superclass. According to the signature, the identifier of each method is generated using the `Name` field of the respective `MethodInfo` object, and the return type using the `ReturnType` field. The parameter types of each method are generated by iterating over the `ParameterInfo` objects of each method, as given in a collection by the `GetParameters` method. The parameter is extracted from each `ParameterInfo` object by accessing the `ParameterType` field. By using @foreach constructs with the same partial generator expressions as in these signatures, the superclass' fields and methods are accessible, as will be described in the next section.

3.1.3 Statements

In this section we discuss type rules for deriving statements as described in the syntax rule *stat*. This section contains the rules for accessing variables and methods that were generated using generator expressions and iterative or conditional generator constructs. The general idea is that definitions that were generated in an `@if` or `@foreach` can only be used in an equivalent `@if` or `@foreach`.

In a method or constructor, a correct expression *me* can be used as a statement:

$$[stat\ me]\ \frac{\Gamma \vdash me\colon ge_t}{\Gamma \vdash me\texttt{;}}$$

While expressions are associated with a generator expression ge_t that generates type using the : symbol, statements are not explicitly associated with a type. A judgment of the form $\Gamma \vdash stat$ means that the statement *stat* is correct in the given environment.

Assignments are derived with rule [*stat assign*]:

$$[stat\ assign]\ \frac{\Gamma \vdash ge_{id}\colon ge_s \quad \Gamma \vdash me\colon ge_t \quad ge_s \text{ compatible with } ge_t}{\Gamma \vdash ge_{id}\ \texttt{=}\ me\texttt{;}}$$

Since all types in the generated class are generated, ge_s and ge_t are generator

expressions. If they are constant generator expressions such as `typeof(int)` or `typeof(Object)`, then type compatibility can be checked using the existing C# type rules. Otherwise, they are only considered compatible if the generator expressions ge_s and ge_t used for generating the types are equal, i.e. structurally the same.

Return statements are derived with rule [*stat return*]:

$$[\textit{stat return}]\ \frac{\Gamma \vdash me\colon ge_t \quad \text{return type of method compatible with } ge_t}{\Gamma \vdash \texttt{return}\ me\texttt{;}}$$

If the expression *me* has a type generated by ge_t that is compatible with the generator expression describing the return type of the method it should be returned from, then the return statement can be derived. Analogously to rule [*stat assign*], if ge_t and the return type of the method are constant generator expressions, then type compatibility can be checked using the existing C# type rules. Otherwise, they are only considered compatible if the generator expressions used for generating them are equal.

Conditionally generated statements are derived with rule [*stat* `@if`]:

$$[\textit{stat}\ \texttt{@if}]\ \frac{\begin{array}{l}\Gamma \vdash gepart\colon\colon\texttt{Boolean}\\ \Gamma \cup \{gepart\} \vdash stat_1, \ldots, \Gamma \cup \{gepart\} \vdash stat_m\\ \Gamma \cup \{\neg gepart\} \vdash stat'_1, \ldots, \Gamma \cup \{\neg gepart\} \vdash stat'_n\end{array}}{\Gamma \vdash \texttt{@if(}gepart\texttt{)}\ \{\ stat_1 \ldots stat_m\ \}\ \text{else}\ \{\ stat'_1 \ldots stat'_n\ \}}$$

The precondition requires the following. First, the partial generator expression *gepart* to be used as condition must be of type `Boolean`. Secondly, the statements $stat_1, \ldots, stat_m$ to be generated in the then-clause of the conditional must be correct in an environment that contains element *gepart*. As described in rule [*def* `@if`], this element signifies that the condition *gepart* is true in the then-clause. It makes it possible to access variables or methods that were generated in the then-clause of a conditional with the same condition *gepart*. These variables or methods will have a signature of the form $gepart \rightarrow sig$. Thirdly, if there is an else-clause, the statements $stat'_1, \ldots, stat'_n$ to be generated in the else-clause of the conditional must be correct in an environment that contains element $\neg gepart$. This element signifies that the condition *gepart* is false in the else-clause. It makes it possible to access variables or methods that were generated in the else-clause of a conditional with the same condition *gepart*. These variables or methods will have a signature of the form $\neg gepart \rightarrow sig$. If all the requirements are met, then the conditional generation of statements can be derived.

Rule [*stat* `@foreach`] derives iteratively generated statements:

$$[\mathit{stat}\ \texttt{@foreach}]\ \frac{\Gamma \vdash \mathit{gepart}\!::t \quad t \text{ implements } \texttt{ICollection} \qquad \Gamma \cup \{ID \in \mathit{gepart}\} \vdash \mathit{stat}_1, \ldots, \Gamma \cup \{ID \in \mathit{gepart}\} \vdash \mathit{stat}_n}{\Gamma \vdash \texttt{@foreach(}ID\ \texttt{in}\ \mathit{gepart}\texttt{)}\ \{\ \mathit{stat}_1 \ldots \mathit{stat}_n\ \}}$$

The precondition requires the following. First, the partial generator expression *gepart* to be used as collection over which to iterate must be of a type that implements a collection. Second, the statements $stat_1, \ldots, stat_m$ to be generated in the then-clause of the conditional must be correct in an environment that contains element $ID \in gepart$. As described in rule [*def* `@foreach`], this element signifies that ID is the iteration variable of a `@foreach`-loop that iterates over the element of collection *gepart*. Such an environment occurs only in the loop body. It makes it possible to access variables or methods that were generated in the body of a `@foreach`-loop iterating over the same collection *gepart*. These variables or methods will have a signature of the form $\forall \alpha \in gepart.sig$. The derived iterative statement generation can be used, for example, to execute some code on variables that were iteratively generated previously. The code could print out the values of the variables, or change their values.

An expression *me* can be derived with the following rules. Literal expressions are derived with rule [*me literal*]:

$$[\mathit{me\ literal}]\ \frac{\Gamma \vdash \diamond \quad x \in \mathit{Literals}(t)}{\Gamma \vdash x\colon \texttt{@typeof(}t\texttt{)@}}$$

If Γ is well-formed and x is a value of type t that can be represented as a literal, then x is a correct expression with a type generated by the constant generator expression `@typeof(`t`)@`.

Rule [*me gliteral*] derives a literal expression that was generated with a generator expression:

$$[\mathit{me\ gliteral}]\ \frac{\Gamma \vdash \mathit{ge}\!::t \quad \text{values of } t \text{ can be represented as literals}}{\Gamma \vdash \mathit{ge}\colon \texttt{@typeof(}t\texttt{)@}}$$

If *ge* is a generator expression of an appropriate type, then it can be used to generate a literal of that type. For example, a generator expression of type `Integer` generates an integer literal, and a generator expression of type `StringLiteral` generates a `String` literal. Note that generator expressions of type `String` are already used to generate identifiers and therefore can not generate `String` literals.

The following rules check expressions that consist only of a variable identifier, i.e. an access to a variable that was defined in the current generator. De-

pending on how the variable definition was generated, a different rule will be used. Rule [*me var*] derives an access to a variable that was neither generated conditionally nor iteratively:

$$[\mathit{me\ var}]\ \frac{\Gamma \vdash \diamond \quad (ge_{id}\colon ge_{type}) \in \Gamma}{\Gamma \vdash ge_{id}\colon ge_{type}}$$

If Γ is a well-formed environment that contains the signature of a generated variable, as described in rule [*def var*], then the generator expression ge_{id} that was used to generate the variable identifier can be used to generate an access to the variable. The resulting expression has as its type the generator expression ge_{type} that was used to generate the variable type.

Rule [*me var* `@if`] derives an access to a variable that was generated conditionally:

$$[\mathit{me\ var}\ \texttt{@if}]\ \frac{\Gamma \vdash \diamond \quad \{gepart,\ gepart \rightarrow ge_{id}\colon ge_{type}\} \subseteq \Gamma}{\Gamma \vdash ge_{id}\colon ge_{type}}$$

The requirements are as follows: Γ must be a well-formed environment that contains an element *gepart* (indicating that the environment is in the then-clause of a conditional generation) and the signature of a conditionally generated variable, as described in rule [*def* `@if`]. The condition *gepart* in the environment and the condition in the signature of the conditionally generated variable are the same. Therefore we can be sure that the variable definition was generated. Hence, the generator expression ge_{id} that was used to generate the variable identifier can be used to generate an access to the variable, which has as its type the generator expression ge_{type} that was used to generate the variable type. Analogously, the rule can be formulated with $\neg gepart$ instead of *gepart* to access a variable that was generated in the else-clause of an `@if`.

Rule [*me var* `@foreach`] derives an access to a variable that was generated iteratively:

$$[\mathit{me\ var}\ \texttt{@foreach}]\ \frac{\begin{array}{c}\Gamma \vdash \diamond \quad \{ID \in gepart,\ \forall\alpha \in gepart.(ge'_{id}\colon ge'_{type})\} \subseteq \Gamma \\ (ge'_{id}\colon ge'_{type})[ID/\alpha] = (ge_{id}\colon ge_{type})\end{array}}{\Gamma \vdash ge_{id}\colon ge_{type}}$$

The rule requires the following. Γ must be a well-formed environment that contains an element $ID \in gepart$ signifying that the environment is in the body of a `@foreach`-loop iterating over collection *gepart*. Furthermore, Γ must contain the signature of a variable that was generated in a `@foreach`-loop iterating over the same collection. If all occurrences of the placeholder α in this signature are replaced by the name of the iteration variable ID in the environment, then we get a generator expression ge_{id} that can be used to

generate the identifier of the variable and a generator expression ge_{type} for the variable type.

The following rules check expressions that consist only of a call to a method that was defined in the current generator. Depending on how the method definition was generated, a different rule will be used. Rule [*me method*] derives a call to a method that was neither generated conditionally nor iteratively:

$$[\mathit{me\ method}]\ \frac{(ge_{id}\colon ge_{t1} \times \ldots \times ge_{tn} \rightarrow ge_{ret}) \in \Gamma \qquad \Gamma \vdash me_1\colon ge_{t1}\ \ldots\ \Gamma \vdash me_n\colon ge_{tn}}{\Gamma \vdash ge_{id}\,\texttt{(}me_1, \ldots, me_n\texttt{)}\colon ge_{ret}}$$

If Γ is a well-formed environment that contains the signature of a generated variable, as described in rule [*def method*], then the generator expression ge_{id} that was used to generate the method identifier can be used to refer to that method in a method call. For the generation of the method call parameters, there have to be correct expressions with appropriate types. The resulting expression will have as its type the generator expression ge_{ret} that was used to generate the method return type.

Rule [*me method* `@if`] derives a call to a method that was generated conditionally:

$$[\mathit{me\ method}\ \texttt{@if}]\ \frac{\{gepart,\ gepart \rightarrow (ge_{id}\colon ge_{t1} \times \ldots \times ge_{tn} \rightarrow ge_{ret})\} \subseteq \Gamma \qquad \Gamma \vdash me_1\colon ge_{t1}\ \ldots\ \Gamma \vdash me_n\colon ge_{tn}}{\Gamma \vdash ge_{id}\,\texttt{(}me_1, \ldots, me_n\texttt{)}\colon ge_{ret}}$$

The requirements are as follows: Γ must be a well-formed environment that contains an element *gepart* (indicating that the environment is in the then-clause of a conditional generation) and the signature of a conditionally generated method, as described in the rules [*def* `@if`] and [*def method*]. The condition *gepart* in the environment and the condition in the signature of the conditionally generated variable are the same. Therefore we can be sure that the variable definition was generated. Hence, the generator expression ge_{id} that was used to generate the method identifier can be used to refer to the generated method in a method call. For the method call parameters there have to be correct expressions with appropriate types. The resulting expression has as its type the generator expression ge_{ret} that was used to generate the method return type. Analogously, the rule can be formulated with $\neg gepart$ instead of *gepart* to call a method that was generated in the else-clause of an `@if`.

Rule [*me method* `@foreach`] derives a call to a method that was generated

iteratively:

$$[me\ method\ \texttt{@foreach}]\ \frac{\begin{array}{l}\{ID \in gepart, \\ \ \forall\alpha \in gepart.(ge'_{id}\colon ge'_{t1} \times \ldots \times ge'_{tn} \rightarrow ge'_{ret})\} \subseteq \Gamma \\ (ge'_{id}\colon ge'_{t1} \times \ldots \times ge'_{tn} \rightarrow ge'_{ret})[ID/\alpha] \\ \quad = (ge_{id}\colon ge_{t1} \times \ldots \times ge_{tn} \rightarrow ge_{ret}) \\ \Gamma \vdash me_1\colon ge_{t1}\ \ldots\ \Gamma \vdash me_n\colon ge_{tn}\end{array}}{\Gamma \vdash ge_{id}\texttt{(}me_1, \ldots, me_n\texttt{)}\colon ge_{ret}}$$

The rule requires the following. Γ is a well-formed environment that contains an element $ID \in gepart$ signifying that the environment is in the body of a `@foreach`-loop iterating over collection *gepart*. Furthermore, Γ contains the signature of a method that was generated in a `@foreach`-loop iterating over the same collection. If all occurrences of the placeholder α in this signature are replaced by the name of the iteration variable ID in the environment, then we get a method signature that can be referred to in a method call. With appropriate expressions for the method parameters, a method call expression can be formed that has type ge_{ret}, the return type of the method.

The other rules for expressions are very similar to the rules for partial generator expressions that were described in Section 3.1.1. The following rules derive expressions with accesses to ordinary C# fields and methods, i.e. fields and methods that are not generated. Rules [*me field*] and [*me sfield*] derive non-static and static field accesses:

$$[me\ field]\ \frac{\Gamma \vdash me\colon ge \quad ge \text{ has accessible field } id\colon t}{\Gamma \vdash me\texttt{.}id\colon \texttt{@typeof(}t\texttt{)@}}$$

$$[me\ sfield]\ \frac{\Gamma \vdash ge\colon\colon\texttt{Type} \quad ge \text{ has accessible static field } id\colon t}{\Gamma \vdash ge\texttt{.}id\colon \texttt{@typeof(}t\texttt{)@}}$$

In both the above rules, *ge* needs to be a constant generator expression because otherwise it is generally impossible to determine what fields the type generated by *ge* contains.

Rules [*me call*], [*me scall*] and [me `new`] derive non-static and static method calls and constructor calls:

$$[me\ call]\ \frac{\begin{array}{c}\Gamma \vdash me\colon ge \\ ge \text{ has accessible method } id\colon t_1 \times \ldots \times t_n \rightarrow t \\ \Gamma \vdash me_1\colon \texttt{@typeof(}t_1\texttt{)@}\ \ldots\ \Gamma \vdash me_n\colon \texttt{@typeof(}t_n\texttt{)@}\end{array}}{\Gamma \vdash me\texttt{.}id\texttt{(}me_1, \ldots, me_n\texttt{)}\colon \texttt{@typeof(}t\texttt{)@}}$$

$$[\mathit{me\ scall}]\ \frac{\Gamma \vdash ge{:}{:}\texttt{Type} \qquad ge \text{ has accessible static method } id{:}\, t_1 \times \ldots \times t_n \to t \qquad \Gamma \vdash me_1{:}\,\texttt{@typeof(}t_1\texttt{)@} \ \ldots\ \Gamma \vdash me_n{:}\,\texttt{@typeof(}t_n\texttt{)@}}{\Gamma \vdash ge\,\texttt{.}\,id\texttt{(}me_1, \ldots, me_n\texttt{)}{:}\,\texttt{@typeof(}t\texttt{)@}}$$

$$[\mathit{me}\ \texttt{new}]\ \frac{\Gamma \vdash ge{:}{:}\texttt{Type} \qquad ge \text{ has accessible constructor with parameters } : t_1 \times \ldots \times t_n \qquad \Gamma \vdash me_1{:}\,\texttt{@typeof(}t_1\texttt{)@} \ \ldots\ \Gamma \vdash me_n{:}\,\texttt{@typeof(}t_n\texttt{)@}}{\Gamma \vdash \texttt{new}\ ge\texttt{(}me_1, \ldots, me_n\texttt{)}{:}\, ge}$$

ge needs to be a constant generator expression because otherwise it is generally impossible to tell what methods or constructors the type generated by *ge* contains.

Rule [*me gcall*] derives a call to a method in the generated class with parameters that were generated iteratively:

$$[\mathit{me\ gcall}]\ \frac{\Gamma \vdash \diamond \qquad \{ge_{id}{:}\,(\forall \alpha \in gepart.ge_{pt}) \to ge_{ret},\ \forall \alpha \in gepart.(ge_{pid}{:}\, ge_{pt})\} \subseteq \Gamma}{\Gamma \vdash ge_{id}\texttt{(@foreach(}ID\ \texttt{in}\ gepart\texttt{)}\ \{\ ge_{pid}[ID/\alpha]\ \}\texttt{)}{:}\, ge_{ret}}$$

As a precondition, the environment must be well-formed and contain the signature of a method with iteratively generated parameters as well as a signature for the generated parameters, as described previously for rule [*def gmethod*]. There is a method parameter for each element in the collection *gepart* that can be used as an argument for the method call. In the resulting method call, the placeholder α is replaced by the identifier of the iteration variable ID. In this way it is possible to call a method with iteratively generated parameters from within a method with iteratively generated parameters, as long as the collection over which was iterated is the same and the two methods use equivalent generator expression ge_{pt} for the generation of the parameter types.

Similarly to the previous rule, rule [*me g*`new`] derives a call to a constructor in the generated class with parameters that were generated iteratively:

$$[\mathit{me\ g}\texttt{new}]\ \frac{\Gamma \vdash \diamond \qquad \{\texttt{@this@}{:}\,(\forall \alpha \in gepart.ge_{pt}) \to \texttt{@this@},\ \forall \alpha \in gepart.(ge_{pid}{:}\, ge_{pt})\} \subseteq \Gamma}{\Gamma \vdash \texttt{@this@(@foreach(}ID\ \texttt{in}\ gepart\texttt{)}\ \{\ ge_{pid}[ID/\alpha]\ \}\texttt{)}{:}\,\texttt{@this@}}$$

The rule works analogously to [*me gcall*]. The identifier and return type of

the constructor are `@this@` because the constructor is in the generated class, as described in rule [*def gconstructor*].

The rules [*me* `typeof`] and [*me cast*] for applications of the `typeof` operator and type casts are analogous to the rules for partial generator expressions:

$$[me\ \texttt{typeof}]\ \frac{\Gamma \vdash ge\!::\!\texttt{Type}}{\Gamma \vdash \texttt{typeof(}ge\texttt{)}\!:\texttt{@typeof(Type)@}}$$

$$[me\ cast]\ \frac{\Gamma \vdash ge\!::\!\texttt{Type} \quad \Gamma \vdash me\!: t}{\Gamma \vdash \texttt{(}ge\texttt{)}me\!: ge}$$

Instead of permitting only type identifiers, the types used as arguments can be generated with generator expressions. Note that type casts, by their very nature, might fail during runtime.

Rule [*me* `this`] derives an expression with the keyword `this` referring to the current object of the generated class:

$$[me\ \texttt{this}]\ \frac{\Gamma \vdash \diamond \quad \text{method is non-static}}{\Gamma \vdash \texttt{this}\!:\texttt{@this@}}$$

If the environment is well-formed and the method to place the statement into is non-static, the keyword `this` can be used to access the current object of the generated type `@this@`.

3.2 Type Derivation Example 1

In the following we type check a simple generator in order to demonstrate how the type rules are used. All of the used rules were described in the previous sections of this article. The example we want to type check has a `String` parameter `ID` and generates a class. The generated class contains an `int` member variable with the name of the given string, and a method `m` that assigns the value 1 to the variable. For simplicity, the constant types `int` and `void` and the method identifier `m` are created explicitly with generator expressions, so that the previously described type rules can be applied. In particular, this example demonstrates how generator expressions are used as part of the environment Γ.

```
1   class Example1(String ID)
2   {
3       @typeof(int)@ @ID@;
4
5       @typeof(void)@ @"m"@() {
6           @ID@ = 1;
7       }
```

```
8   }
```

First, an appropriate environment Γ needs to be created. An empty environment can be created using rule [*env* $\emptyset$], which has no precondition and thus can always be used. The generator variable for the `String` parameter of the generator is added to Γ with rule [*env gvar*]:

$$[\textit{env gvar}]\ \frac{[\textit{env}\ \emptyset]\ \frac{}{\emptyset \vdash \diamond} \quad \texttt{String} \in \textit{Types} \quad (\text{generator variable } \texttt{ID}) \notin \emptyset}{\{\texttt{ID}\!::\!\texttt{String}\} \vdash \diamond}$$

$$\Gamma_1 =_{def} \{\texttt{ID}\!::\!\texttt{String}\}$$

As described previously, in the environment the :: sign associates a generator variable with its type. In this case, the generator variable is the generator parameter `ID` of type `String`. The new environment is named Γ_1.

Next, the signature of the member variable is added to the environment. Rules [*gepart id*] and [*ge* `@`*gepart*`@`] allow us to use the generator variable in the environment as a generator expression:

$$[\textit{ge}\ \texttt{@}\textit{gepart}\texttt{@}]\ \frac{[\textit{gepart id}]\ \frac{\Gamma_1 \vdash \diamond}{\Gamma_1 \vdash \texttt{ID}\!::\!\texttt{String}}}{\Gamma \vdash \texttt{@ID@}\!::\!\texttt{String}}$$

Then, a generator expression for type `int` is derived using rule [*gepart* `typeof`] and rule [*ge* `@`*gepart*`@`]:

$$[\textit{ge}\ \texttt{@}\textit{gepart}\texttt{@}]\ \frac{[\textit{gepart}\ \texttt{typeof}]\ \frac{\Gamma_1 \vdash \diamond \quad \texttt{int} \in \textit{Types}}{\Gamma_1 \vdash \texttt{typeof(int)}\!::\!\texttt{Type}}}{\Gamma_1 \vdash \texttt{@typeof(int)@}\!::\!\texttt{Type}}$$

At this point all the preconditions have been derived that are needed to derive the signature of the member variable with rule [*env var*]:

$$[\textit{env var}]\ \frac{\Gamma_1 \vdash \texttt{@ID@}\!::\!\texttt{String} \quad \Gamma_1 \vdash \texttt{@typeof(int)@}\!::\!\texttt{Type} \quad (\text{variable } \texttt{ID}) \notin \Gamma}{\Gamma_1 \cup \{\texttt{@ID@}\!:\!\texttt{@typeof(int)@}\} \vdash \diamond}$$

$$\Gamma_2 =_{def} \{\texttt{ID}\!::\!\texttt{String}, \texttt{@ID@}\!:\!\texttt{@typeof(int)@}\}$$

There is already a generator variable with the name `ID` in Γ_1, but there is no ordinary symbol that has a name generated by the generator expression @`ID`@. Therefore this is not a collision of identifiers. The new environment is called Γ_2.

After deriving the identifier of method `m` with the rules [*gepart literal*] and [*ge* `@`*gepart*`@`], and deriving its return type with rule [*gepart* `typeof`] and rule [*ge* `@`*gepart*`@`], method `m`'s signature is added to the environment using rule

[*env method*]:

$$[ge\ \texttt{@}gepart\texttt{@}]\ \frac{[gepart\ literal]\ \frac{\Gamma_2 \vdash \diamond \quad \texttt{"m"} \in Literals(\texttt{String})}{\Gamma_2 \vdash \texttt{"m"} :: \texttt{String}}}{\Gamma_2 \vdash \texttt{@"m"@} :: \texttt{String}}$$

$$[ge\ \texttt{@}gepart\texttt{@}]\ \frac{[gepart\ \texttt{typeof}]\ \frac{\Gamma_2 \vdash \diamond \quad \texttt{void} \in Types}{\Gamma_2 \vdash \texttt{typeof(void)} :: \texttt{Type}}}{\Gamma_2 \vdash \texttt{@typeof(void)@} :: \texttt{Type}}$$

$$[env\ method]\ \frac{\Gamma_2 \vdash \texttt{@"m"@} :: \texttt{String} \quad \Gamma_2 \vdash \texttt{@typeof(void)@} :: \texttt{Type} \quad (\text{method } \texttt{@"m"@} : \to \texttt{@typeof(void)@}) \notin \Gamma_2}{\Gamma_2 \cup \{\texttt{@"m"@} : \to \texttt{@typeof(void)@}\} \vdash \diamond}$$

$$\Gamma_3 =_{def} \{\texttt{ID} :: \texttt{String}, \texttt{@ID@} : \texttt{@typeof(int)@}, \texttt{@"m"@} : \to \texttt{@typeof(void)@}\}$$

The new environment is called Γ_3.

The generator expressions for the variable type and identifier of the variable definition are derived using Γ_3 and rules [*gepart* `typeof`], [*gepart id*] and [*ge* `@`*gepart*`@`]. Then the variable definition is derived using rule [*def var*]:

$$[ge\ \texttt{@}gepart\texttt{@}]\ \frac{[gepart\ \texttt{typeof}]\ \frac{\Gamma_3 \vdash \diamond \quad \texttt{int} \in Types}{\Gamma_3 \vdash \texttt{typeof(int)} :: \texttt{Type}}}{\Gamma_3 \vdash \texttt{@typeof(int)@} :: \texttt{Type}}$$

$$[ge\ \texttt{@}gepart\texttt{@}]\ \frac{[gepart\ id]\ \frac{\Gamma_3 \vdash \diamond}{\Gamma_3 \vdash \texttt{ID} :: \texttt{String}}}{\Gamma_3 \vdash \texttt{@ID@} :: \texttt{String}}$$

$$[def\ var]\ \frac{\Gamma_3 \vdash \texttt{@typeof(int)@} :: \texttt{Type} \quad \Gamma_3 \vdash \texttt{@ID@} :: \texttt{String}}{\Gamma_3 \vdash \texttt{@typeof(int)@ @ID@;} \therefore (\texttt{@ID@} : \texttt{@typeof(int)@})}$$

To derive the assignment in the method `m`, we first derive its left side and then its right side. The variable on the left side is derived with rule [*me var*]:

$$[me\ var]\ \frac{\Gamma_3 \vdash \diamond \quad (\texttt{@ID@} : \texttt{@typeof(int)@}) \in \Gamma_3}{\Gamma_3 \vdash \texttt{@ID@} : \texttt{@typeof(int)@}}$$

The right side of the assignment is derived using rule [*me literal*], and rule [*stat assign*] then derives the whole assignment:

$$[me\ literal]\ \frac{\Gamma_3 \vdash \diamond \quad \texttt{1} \in Literals(\texttt{int})}{\Gamma_3 \vdash \texttt{1} : \texttt{@typeof(int)@}}$$

$$[stat\ assign]\ \frac{\Gamma_3 \vdash \texttt{@ID@} : \texttt{@typeof(int)@} \quad \Gamma_3 \vdash \texttt{1} : \texttt{@typeof(int)@} \quad \texttt{@typeof(int)@} \text{ compatible with } \texttt{@typeof(int)@}}{\Gamma_3 \vdash \texttt{@ID@ = 1;@}}$$

Next, the definition of method `m` is derived with rule [*def method*], after first deriving its identifier and return type using Γ_3:

$$[\mathit{ge}\ \texttt{@}\mathit{gepart}\texttt{@}]\ \frac{[\mathit{gepart\ literal}]\ \frac{\Gamma_3 \vdash \diamond \quad \texttt{"m"} \in \mathit{Literals}(\texttt{String})}{\Gamma_3 \vdash \texttt{"m"} :: \texttt{String}}}{\Gamma_3 \vdash \texttt{@"m"@} :: \texttt{String}}$$

$$[\mathit{ge}\ \texttt{@}\mathit{gepart}\texttt{@}]\ \frac{[\mathit{gepart}\ \texttt{typeof}]\ \frac{\Gamma_3 \vdash \diamond \quad \texttt{void} \in \mathit{Types}}{\Gamma_3 \vdash \texttt{typeof(void)} :: \texttt{Type}}}{\Gamma_3 \vdash \texttt{@typeof(void)@} :: \texttt{Type}}$$

$$[\mathit{def\ method}]\ \frac{\begin{array}{c}\Gamma_3 \vdash \texttt{@"m"@} :: \texttt{String} \quad \Gamma_3 \vdash \texttt{@typeof(void)@} :: \texttt{Type} \\ \Gamma_3 \vdash \texttt{@ID@ = 1;@}\end{array}}{\begin{array}{c}\Gamma_3 \vdash \texttt{@typeof(void)@ @"m"@() \{ @ID@ = 1; \}} \\ \therefore (\texttt{@"m"@} : \rightarrow \texttt{@typeof(void)@})\end{array}}$$

The signature of the method after the $\therefore$ sign specifies that the identifier of the method is generated by the constant generator expression `@"m"@`, that the method has no parameters, and that the return type is generated by `@typeof(void)@`.

Finally rule [*generator*] is applied to derive the whole generator. Our example generator does not have generated supertypes. Therefore the rule becomes a bit simpler:

$$[\mathit{generator}]\ \frac{\begin{array}{l}\Gamma_3 \vdash \texttt{@typeof(int)@ @ID@;} \therefore (\texttt{@ID@} : \texttt{@typeof(int)@}) \\ \Gamma_3 \vdash \texttt{@typeof(void)@ @"m"@() \{ @ID@ = 1; \}} \\ \quad \therefore (\texttt{@"m"@} : \rightarrow \texttt{@typeof(void)@})\end{array}}{\begin{array}{l}\emptyset \vdash \texttt{class Example1(String ID) \{} \\ \quad \texttt{@typeof(int)@ @ID@;} \\ \quad \texttt{@typeof(void)@ @"m"@() \{ @ID@ = 1; \} \}}\end{array}}$$

This step concludes the derivation.

3.3 Type Derivation Example 2

In the following a more sophisticated generator is type-checked, which uses the `@foreach` construct. It gets a type `T` as its parameter and iteratively replicates the public fields of that type. Furthermore, it generates a getter method for each of the replicated fields. This example demonstrates how the type system makes sure that only those definitions that have actually been generated can

be accessed. Note that the two `@foreach` loops could have been merged; they were separated to emphasize that they could potentially occur in different locations within the generator.

```
1   class Example2(Type T)
2   {
3       @foreach(F in T.GetFields()) {
4           @F.FieldType@ @F.Name@;
5       }
6
7       @foreach(F in T.GetFields()) {
8         @F.FieldType@ @"Get"+F.Name@() {
9           return @F.Name@;
10        }
11      }
12  }
```

In this example the derivations of the environments and the smaller generator parts are omitted, as they have been illustrated fully in the previous example. The well-formed environment Γ is the environment in the body of the generator. It is created using [*env* $\emptyset$] as the initial rule, [*env gvar*] to derive the elements for the generator parameter `T`, and [*gepart literal*], [*gepart id*], [*gepart field*], [*gepart call*], [*ge* `@`*gepart*`@`] and [*env* `@foreach`] to derive the signatures of the `@foreach` loops. The application of the `+` operator, which is used to add the string prefix `"Get"` to the name of the getter method, is treated as a normal method application in a partial generator expression, using rule [*gepart call*].

$$\begin{aligned}\Gamma =_{def} \{\ & \texttt{T}{::}\,\texttt{Type},\\ & \forall\alpha \in (\texttt{T.GetFields()}).(\texttt{@}\alpha\texttt{.Name@}{:}\ \texttt{@}\alpha\texttt{.FieldType@}),\\ & \forall\alpha \in (\texttt{T.GetFields()}).(\texttt{@"Get"+}\alpha\texttt{.Name@}{:} \rightarrow \texttt{@}\alpha\texttt{.FieldType@})\}\end{aligned}$$

Using this environment and the rules [*gepart id*], [*gepart field*], [*ge* `@`*gepart*`@`] and [*def var*], the variable definition in the first loop body can be derived.

The next step is to derive the first `@foreach` loop, using rule [*def* `@foreach`]:

$$[def\ \texttt{@foreach}]\ \frac{\begin{array}{l}\Gamma \vdash \texttt{T.GetFields()}::\texttt{FieldInfo[]}\\ \texttt{FieldInfo[]}\ \text{implements}\ \texttt{ICollection}\\ \Gamma \cup \{F::\texttt{FieldInfo}, F \in \texttt{T.GetFields()},\\ \quad \texttt{@F.Name@}: \texttt{@F.FieldType@}\}\\ \quad \vdash \texttt{@F.FieldType@ @F.Name@;}\\ \quad \therefore \{\texttt{@F.Name@}: \texttt{@F.FieldType@}\}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{@foreach(F in T.GetFields())}\ \{\\ \qquad \texttt{@F.FieldType@ @F.Name@;}\ \}\\ \quad \therefore \{\forall \alpha \in (\texttt{T.GetFields()}).(\texttt{@}\alpha\texttt{.Name@}: \texttt{@}\alpha\texttt{.FieldType@})\}\end{array}}$$

The additional elements in the environment of the definition `@F.FieldType@ @F.Name@;` in the precondition are derived using the rule [*env gvar*] for the iteration variable `F`, [*gepart id*], [*gepart field*] and [*env loop*] for the element specifying the collection over which `F` iterates, and [*gepart id*], [*gepart field*], [*ge* `@`*gepart*`@`] and [*env var*] for the signatures of the definition in the loop body.

Now the method definition in the second `@foreach` loop is derived. The iteratively generated variables are accessed in the expression in the return statement of the method body. The derivation of this expression illustrates how the type system ensures that variables generated in a `@foreach` loop can only be accessed from within a similar `@foreach` loop. This means that only those variables that were actually generated can be accessed. Additional elements are added to the environment to express the following: in the scope of the derived expression, F is an iteration variable of type `FieldInfo` iterating over a collection given by the partial generator expression `T.GetFields()`. The signature of the getter method in the loop body is also registered. This results in a new environment Γ':

$$\begin{array}{l}\Gamma' =_{def} \Gamma \cup \{\ F::\texttt{FieldInfo}, F \in \texttt{T.GetFields()},\\ \qquad\qquad \texttt{@"Get"+}\alpha\texttt{.Name@}: \rightarrow \texttt{@}\alpha\texttt{.FieldType@}\}\end{array}$$

Γ' is used to derive the expression in the return statement:

$$[\textit{me var } \texttt{@foreach}]\ \frac{\begin{array}{l}\Gamma' \vdash \diamond \\ \{F \in \texttt{T.GetFields()}, \\ \quad \forall \alpha \in (\texttt{T.GetFields()}). \\ \qquad (\texttt{@}\alpha\texttt{.Name@}: \texttt{@}\alpha\texttt{.FieldType@})\} \subseteq \Gamma' \\ (\texttt{@}\alpha\texttt{.Name@}: \texttt{@}\alpha\texttt{.FieldType@})[\texttt{F}/\alpha] \\ \quad = (\texttt{@F.Name@}: \texttt{@F.FieldType@})\end{array}}{\Gamma' \vdash \texttt{@F.Name@}: \texttt{@F.FieldType@}}$$

This rule ensures that the current scope is that of a `@foreach` loop analogous to the one in which the derived variable is defined. That is, the partial generator expression specifying the collection to iterate over is the same, and the generator expression for the variable identifier is the same except for the name of the iteration variable. The derived expression can be used in the return statement:

$$[\textit{stat return}]\ \frac{\begin{array}{l}\Gamma' \vdash \texttt{@F.Name@}: \texttt{@F.FieldType@} \\ \text{return type of method compatible with } \texttt{@F.FieldType@}\end{array}}{\Gamma \vdash \texttt{return @F.Name@;}}$$

Now the method definition in the body of the second `@foreach` loop is derived:

$$[\textit{def method}]\ \frac{\begin{array}{c}\Gamma' \vdash \texttt{@"Get"+F.Name@}:: \texttt{String} \quad \Gamma' \vdash \texttt{@F.FieldType@}:: \texttt{Type} \\ \Gamma' \vdash \texttt{return @F.Name@;}\end{array}}{\begin{array}{c}\Gamma' \vdash \texttt{@F.FieldType@ @"Get"+F.Name@() } \{ \\ \texttt{return @F.Name@; } \} \\ \therefore (\texttt{@"Get"+F.Name@}: \rightarrow \texttt{@F.FieldType@})\end{array}}$$

The second `@foreach` loop is derived as follows:

$$[def\ \texttt{@foreach}]\ \frac{\begin{array}{l}\Gamma' \vdash \texttt{T.GetFields()::FieldInfo[]} \\ \texttt{FieldInfo[]}\ \text{implements}\ \texttt{ICollection} \\ \Gamma' \vdash \texttt{@F.FieldType@ @"Get"+F.Name@() \{} \\ \quad \texttt{return @F.Name@; \}} \\ \quad \therefore (\texttt{@"Get"+F.Name@}\colon \rightarrow \texttt{@F.FieldType@})\end{array}}{\begin{array}{l}\Gamma' \vdash \texttt{@foreach(F in T.GetFields()) \{} \\ \quad \texttt{@F.FieldType@ @"Get"+F.Name@() \{} \\ \quad \texttt{return @F.Name@; \} \}} \\ \therefore \{\forall \alpha \in (\texttt{T.GetFields()}). \\ \quad (\texttt{@"Get"+}\alpha\texttt{.Name@}\colon \rightarrow \texttt{@}\alpha\texttt{.FieldType@})\}\end{array}}$$

Finally, the whole generator can be derived. In the following rule application, the two `@foreach` loops as they were derived above are abbreviated as $foreach_1$ and $foreach_2$.

$$[generator]\ \frac{\begin{array}{l}\Gamma \vdash foreach_1 \\ \quad \therefore \{\forall \alpha \in (\texttt{T.GetFields()}).(\texttt{@}\alpha\texttt{.Name@}\colon \texttt{@}\alpha\texttt{.FieldType@})\} \\ \Gamma \vdash foreach_2 \\ \quad \therefore \{\forall \alpha \in (\texttt{T.GetFields()}). \\ \qquad (\texttt{@"Get"+}\alpha\texttt{.Name@}\colon \rightarrow \texttt{@}\alpha\texttt{.FieldType@})\}\end{array}}{\emptyset \vdash \texttt{class Example2(Type T)}\ \{foreach_1\ foreach_2\}}$$

This step concludes the derivation.

3.4 Type Derivation Example 3

This example is similar to the previous one: the generator gets a type `T` as its parameter and generates getters for the public fields in `T`. However, the fields of `T` are not replicated iteratively, but are inherited because the generated class is a subclass of `T`. This example demonstrates how inheritance can be used and inherited fields accessed in a type-safe manner.

```
1   class Example3(Type T) : @T@
```

```
2  {
3      @foreach(F in T.GetFields()) {
4        @F.FieldType@ @"Get"+F.Name@() {
5          return @F.Name@;
6        }
7      }
8  }
```

The environment Γ in the generator body contains the signature of the generator parameter `T`, the signatures of the fields and methods of the superclass as specified in rule [*generator*], and the signature of the `@foreach` that generates the getters:

$$
\begin{aligned}
\Gamma =_{def} \{\; & \texttt{T::Type}, \\
& \forall\alpha \in (\texttt{T.GetFields()}).(\texttt{@}\alpha\texttt{.Name@}\colon \texttt{@}\alpha\texttt{.FieldType@}), \\
& \forall\alpha \in (\texttt{T.GetMethods()}).(\texttt{@}\alpha\texttt{.Name@}\colon \\
& \quad (\forall\alpha' \in (\alpha\texttt{.GetParameters()}).\texttt{@}\alpha'\texttt{.ParameterType@}) \\
& \quad \rightarrow \texttt{@}\alpha\texttt{.ReturnType@}), \\
& \forall\alpha \in (\texttt{T.GetFields()}).(\texttt{@"Get"+}\alpha\texttt{.Name@}\colon \rightarrow \texttt{@}\alpha\texttt{.FieldType@})\}
\end{aligned}
$$

The method definition in the `@foreach` loop can be derived similarly to the previous example. The difference is that in the expression in the return statement of the method body, the inherited variables are accessed instead of iteratively generated variables. The derivation of this expression illustrates how variables defined in the superclass can only be accessed from within a `@foreach` loop that iterates over the variables of the superclass as obtained by the reflection method `GetFields()`. This means that only those variables that were actually defined in the superclass can be accessed. Analogously to the previous example, the environment Γ' in the context of the `@foreach` loop body contains additional elements. They express that F is an iteration variable of type `FieldInfo` iterating over a collection given by the partial generator expression `T.GetFields()`. The signature of the getter method in the loop body is also registered.

$$
\begin{aligned}
\Gamma' =_{def} \Gamma \cup \{\; & F\texttt{::FieldInfo}, F \in \texttt{T.GetFields()}, \\
& \texttt{@"Get"+}\alpha\texttt{.Name@}\colon \rightarrow \texttt{@}\alpha\texttt{.FieldType@}\}
\end{aligned}
$$

Now the expression in the return statement can be derived in the same manner as in the previous example:

$$[\mathit{me\ var}\ \texttt{@foreach}]\ \frac{\begin{array}{l}\Gamma' \vdash \diamond \\ \{F \in \texttt{T.GetFields()}, \\ \quad \forall\alpha \in (\texttt{T.GetFields()}). \\ \quad\quad (\texttt{@}\alpha\texttt{.Name@:}\,\texttt{@}\alpha\texttt{.FieldType@})\} \subseteq \Gamma' \\ (\texttt{@}\alpha\texttt{.Name@:}\,\texttt{@}\alpha\texttt{.FieldType@})[\texttt{F}/\alpha] \\ \quad = (\texttt{@F.Name@:}\,\texttt{@F.FieldType@})\end{array}}{\Gamma' \vdash \texttt{@F.Name@:}\,\texttt{@F.FieldType@}}$$

The method definition and the `@foreach` loop are also derived in the same manner as in the previous example.

Finally, the whole generator can be derived. In the following rule application the `@foreach` loop is abbreviated as *foreach*.

$$[\mathit{generator}]\ \frac{\begin{array}{l}\{\texttt{T::Type}\} \vdash \texttt{@}T\texttt{@::Type} \\ \Gamma \vdash \mathit{foreach} \\ \quad \therefore \{\forall\alpha \in \texttt{T.GetFields()}. \\ \quad\quad (\texttt{@"Get"+}\alpha\texttt{.Name@:} \to \texttt{@}\alpha\texttt{.FieldType@})\}\end{array}}{\emptyset \vdash \texttt{class Example3(Type T) : @T@}\ \{\mathit{foreach}\}}$$

This step concludes the derivation.

4 Soundness Aspects of Genoupe

Like many other type systems, the Genoupe type system is restrictive: it forbids not only generators that lead to incorrect generated code but also those that violate the type rules, but might still always produce correct generated code. 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 an 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. It is typical for type systems to deal with issues by restricting the way a language can be used, without

limiting the applicability of the language: C# and Java, for example, do not really check whether a method with a non-void return type always returns a value; they merely check whether a superset of possible execution paths returns a value. This is so because in all these cases the class of correct programs can be undecidable. But the type system should be decidable and therefore has to reject a certain superset of all incorrect artifacts. It would be possible to mitigate these issues further using approaches from logical programming such as, for example, constraint solving and model checking.

Soundness of a generator type safety notion means that the type system should exclude generators that lead to non-typesafe code at the time of generator execution. We focus here on two soundness questions with regard to the use of Genoupe expressions in the Genoupe type system. The Genoupe system assumes that expressions that are statically identical in the view of the type system will also evaluate to the same values, e.g. generate the same identifiers. This assumption is essential and seems daring, but can be solved rather straightforwardly with a memoization approach, as explained in the next section. To increase Genoupe's applicability and versatility, it should be possible to generate identifiers freely. However, Genoupe should prevent clashes between generated identifiers. This is discussed in Section 4.2.

4.1 Type-Safe Generation with Imperative Generator Languages

Intuitively, many people assume that using a language with side effects and/or non-determinism for the specification of the generator expressions necessarily leads to unsoundness. In the case of Genoupe, this suspicion is tied to the question of whether structurally identical expressions in the generator language will evaluate to the same value at generation time. Consider the following example:

```
1   public class RandomTypes()
2   {
3       @MyClass.randomType()@ x;
4       @MyClass.randomType()@ y;
5
6       public void m() {
7           x = y;
8       }
9   }
```

Clearly, the assignment in line 7 is only safe if the types of `x` and `y` are compatible. The types are generated by two generator expressions that are structurally identical: `@MyClass.randomType()@`. However, in an imperative

language such as C# we expect that the generator expressions may have side effects or show non-deterministic behavior, and therefore potentially produce a different value each time they are evaluated. In this example particularly, which uses the static method `randomType`, we expect the generator expressions to return two random types that are different from each other.

It is an important finding, and perhaps one of the particularly wide-ranging messages of this article, that we can circumvent this soundness issue for any imperative generator language by evaluating all generator expressions using *memoization* [34]. This means that during execution, once the variables in a generator expression are assigned with actual values, each structural identical generator expression is evaluated only once and the result is memoized. Memoization works with a look-up table for generator expressions in which all variables have been replaced by concrete values. Such generator expressions form the key of the look-up table, and the result of their first and only evaluation are the look-up values. Hence whenever a structurally identical generator expression is encountered with the same assigned variable values, its value is retrieved from the look-up table rather than evaluating the generator expression again. The lifetime of the look-up table can be limited to generation time. Memoization is a well-known technique in compiler construction [38,18].

Coming back to the example above, using memoization the generator expression `@MyClass.randomType()@` will be evaluated once only. The second time this expression is encountered during generation, the same value will be used. As a result, we can guarantee that `x` and `y` have the same types. Similarly, memoization solves the problem for other structurally equivalent generator expressions with nondeterministic behavior. Even if the result of a generator expression is influenced by earlier side effects, it is guaranteed that all following structurally equivalent generator expressions using the same variable values will yield the same result.

Note that the capability of C# to create non-determinism and side effects is not the motivation for choosing C# as the generator language. The motivation for choosing C# is the principle of economy, since it is a natural choice to have homogeneity between the generator and the host language. With memoization, generator expressions that are structurally identical after their variables have been replaced by values, yield the same value. As a consequence, it is possible to use C# also as the generator language.

4.2 Integration with a Host Language and its Costs

During the development of Genoupe certain design decisions were made in order to maximize its actual usefulness, in addition to its theoretical usefulness

for software development. One of the central questions for a language extension such as Genoupe is its compatibility with existing code in the host language (C# in the case of Genoupe). The developer often might want to use source code files written in the host language together with source code files using the language extension. We refer to the former as *host language sources* and to the latter as *language extension sources*. Here we discuss here four approaches that make the design and implementation of a language extension increasingly more difficult, but also increase the applicability of the language extension. For Genoupe we have deliberately chosen the last, most difficult, yet most applicable and versatile approach.

Source-code incompatibility means that host language sources need to be changed into language extension sources if they are to be used together with other language extension sources. For example, each source code file may need additional code in the header. Such an approach puts a significant burden onto programmers who want to use the language extension in existing projects.

Source-code compatibility means that host language sources can be used unchanged in the same project as language extension sources, but the host language sources need to be recompiled with the language extension compiler. For example, a pre-compiler of the language extension may need to be run over all source code to prevent name clashes by adding prefixes to identifiers.

Upstream compatibility means that the compilation products of the host language sources, e.g. standard libraries, can be used together with language extension sources, without changes or recompilation. However, all code *downstream* must be compiled with the compiler of the language extension. Downstream means that the code depends on language extension sources, i.e. the code directly or indirectly uses code that was generated by the language extension. For example, considering C++ as a language extension of C, C++ is upstream compatible in most implementations because C++ code can use compiled C code, but the converse is not possible. Note that this option is based on the notion of dependency, which may differ between languages.

Indirect downstream compatibility means that only host language sources that directly depend on language extension sources need to be recompiled with the compiler of the language extension. This is a typical case for language extensions that affect the signatures of language entities such as types. For example, code that uses Genoupe-generated signatures indirectly, e.g. by calling a Genoupe-generated method indirectly, does not have to be compiled with Genoupe. However, code that uses Genoupe-generated signatures directly should be compiled with Genoupe because only Genoupe can type check the accesses to these signatures.

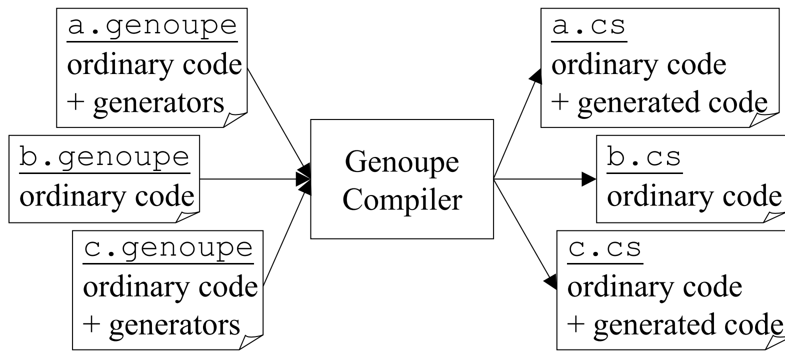

Fig. 1. The Genoupe compilation process.

Direct downstream compatibility means that host language sources can depend on language extension sources directly and still be compiled with the standard host language compiler. For example with Genoupe, the host language sources can use the signatures that were generated with Genoupe and still be compiled with the standard C# compiler. For this the generated signatures need to have at least human-readable names, and at best names that fulfill all the naming conventions that may apply for the generated types.

On the detailed technical level, the Genoupe solution of direct downstream compatibility is implemented as follows: source code files written in Genoupe have the name suffix `.genoupe` and are compiled to ordinary C# source files with the same name but with a `.cs` suffix (see Fig. 1). Each time a generator is applied with new arguments, new types are created, with unique names. If a generator is applied more than once with the same arguments in a compilation run, the corresponding code is generated only once.

One of the application scenarios of Genoupe is the generation of APIs, e.g. for persistence frameworks and patterns. Naming conventions are very important for these applications. For example, a persistence framework may have a naming convention for the properties of data access layer classes that represent database table columns. Therefore we have chosen direct downstream compatibility as the approach for the implementation and for the type system. However, direct downstream compatibility means that the generator has to produce string identifiers, and cannot make generated identifiers a priori distinct from user-defined identifiers.

The possibility of generating arbitrary identifiers with generator expressions brings about lexical problems. A generated identifier might not be unique, e.g. it might clash with another identifier of a different definition in the same scope, or with a keyword of the language. Furthermore, an identifier might be malformed, i.e. not conform to the syntax of the language. These problems can be avoided if we restrict the way identifiers are generated. An *id generation scheme* is a function that is applied during generation-time whenever an identifier is generated.

For example, a simple id generation scheme that is often used to distinguish identifiers in libraries is a *prefix id generation scheme*. Whenever an identifier is generated, this scheme checks whether the identifier clashes with any other identifiers that are in scope so far. That is, it checks clashes with identifiers of all inherited members, as well as with all identifiers that have already been generated. Because id generation schemes operate during generation time, superclasses that were unknown during generator definition time ("mixins") are known and can also be considered. If a clash is detected, the prefix id generation scheme adds a prefix to the generated identifier that makes it unique. Clashes with keywords and malformed identifiers can be avoided in a similar manner.

We have not elaborated on this aspect for several reasons. Enforcing a single id generation scheme is an unacceptable restriction for Genoupe for two reasons. First, by not enforcing a single id scheme we will still allow the generator designer to produce all the generator names necessary in the particular application domain. An important application of Genoupe is to build generators that fulfill the naming conventions of particular frameworks. To give an example that illustrates the language-independence of these questions, in a Genoupe-style extension for Java we could generate Enterprise Java Beans from a given class. The Enterprise Java Beans framework has naming conventions that can be expressed as statically checkable rules. These rules require the breakdown of identifiers, and this is the case with many naming conventions. For example, the classic getter/setter naming conventions require a breakdown of identifiers as well and therefore cannot be expressed on the parser-level of language grammars; rather they go down to the level of lexical analysis.

Secondly, there could be a plethora of possible naming conventions, and Genoupe is intended to be able to cater for all of them. Therefore there can be no single id generation scheme that gives us the flexibility to potentially produce all clear human-readable names. Another reason that such an id scheme is uninteresting is that the more elegant solution would be to circumvent the generation of textual source code altogether. Hence the generated names would have an abstract, inherently unambiguous syntax. This approach is discussed in the following section.

5 Integrating Genoupe into the AP1 System

To be able to use a model-based representation, one needs tools that support i) model-based storage and retrieval and ii) model-based editing of data. The AP1 system [32] offers a model-based repository, which is based on the PD model, and a generic editor, which can be used to edit any data in a model-

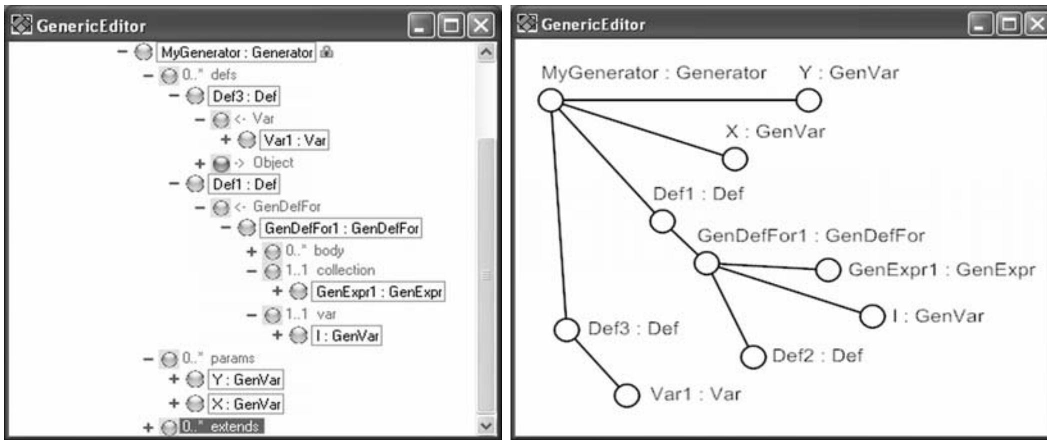

Fig. 2. Screenshots of the generic editor of AP1, showing the tree view (left) and the graph view (right).

based fashion. This repository supports additional functionality such as event notification and management, and the generic editor provides a means for distributed synchronous collaborative work using multiple views. Figure 2 shows two screenshots of the generic editor. On the left side, a tree view is shown, which represents instances and roles as tree nodes. On the right side, a graph view is shown, which represents the same data as the tree view. Using the generic editor, users can edit data collaboratively, using different views. Whenever a data element is changed in one of the views, corresponding changes occur in all the other views. The generic editor supports the invocation of typed operations on the data.

Integrating the Genoupe concepts into the AP1 system is not difficult. In fact, it simplifies the implementation of Genoupe due to AP1's structured repository and its notion of operations. The implementation of Genoupe as a textual stand-alone precompiler and its integration into the AP1 system are illustrated in Fig. 3. In this figure, data artifacts are represented as document shapes, with a folded bottom right corner, and processing components as boxes.

In the precompiler implementation, the initial artifact is textual Genoupe source code, as described in the previous sections. Before processing the generator code, the source code has to be scanned by a lexer and parsed into a Genoupe abstract syntax tree (AST). This is a routine compiler construction task [1], but there are pitfalls such as potential syntactical ambiguities that have to be handled. The grammars of modern languages such as C# can be quite voluminous. Consequently, the construction of a good lexer and parser can consume a significant amount of time. The actual generation work is done by the AST transformation component. This is essentially a tree parser which takes the Genoupe AST as input, eliminates the Genoupe-specific tree nodes, and adds appropriate C# tree nodes to the AST instead. It is the heart of Genoupe, and specified on the relatively high level of typed, abstract syntax.

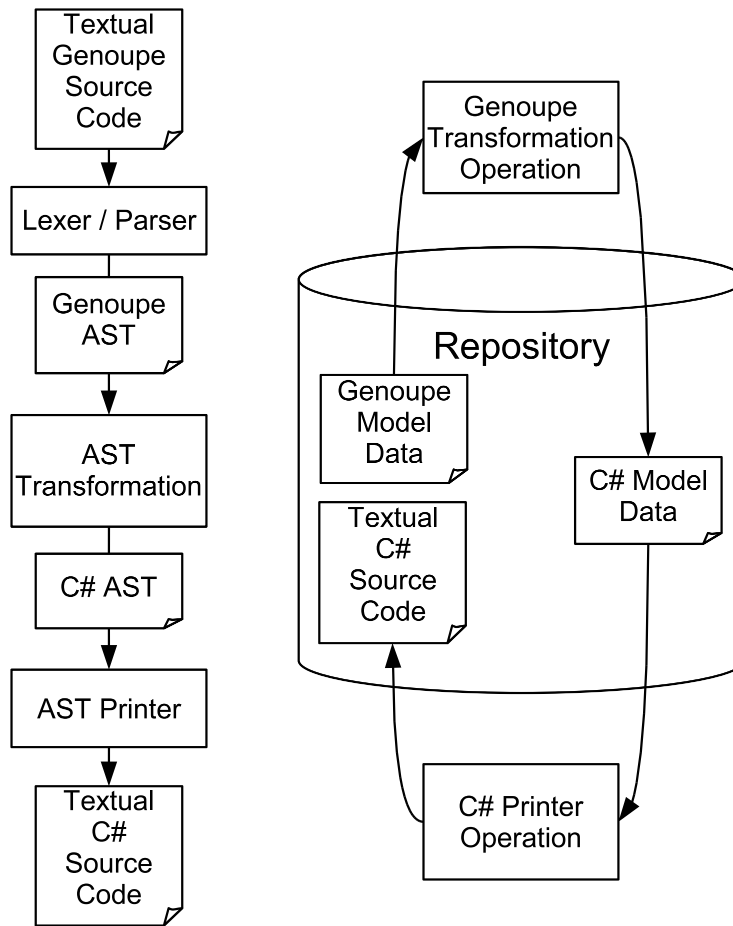

Fig. 3. Implementation of Genoupe as a textual stand-alone tool (left) and integration with the AP1 system (right).

Consequently, the transformation steps can be formulated quite concisely. Finally, the C# AST has to be serialized by another tree parser, i.e. printed back to a textual C# source code representation.

The integration of Genoupe into the AP1 system, as shown on the right side of Fig. 3, eliminates the need for a Genoupe lexer and parser. As a platform for model-based software development it is designed to deal with structured data, such as ASTs, directly. The structure of a Genoupe AST can be reformulated directly as a PD model, which can be managed in the repository. Programming with Genoupe is done in a structured way, by editing instances of that PD model, with tools such as the generic editor. This has advantages: model-based editing is more robust than working with a textual representation because many invalid modifications can be ruled out on the level of the user interface [15]. Furthermore, model-based editing benefits from typed operations, such as typed copy&paste or search and replace.

Generation is implemented with an operation that performs the transformation between the Genoupe source code model and a C# source code model.

This operation is essentially the same as the transformation component of the stand-alone tool, with the difference that it is based on a PD model representation of the involved data. Just like Genoupe source code, C# source code can be specified as data of an appropriate PD model. Another operation, which is essentially the C# AST printer of the stand-alone tool, can be used to transform the C# model data into a string containing corresponding textual C# source code. As a result, C# code can also be managed using the repository, developed with the help of structured tools such as the generic editor, and exported for usage with textual tools such as compilers.

6 Language Independent Lessons Learned: Generators and Reflection

The motivation for the research presented here is not only to provide a workable powerful generation mechanism for the concrete language at hand, here C#. We indeed use our generation mechanism to elucidate the interrelation of two concepts, code generators on the one hand and reflection on the other hand. The deeper conclusions of the research presented here are language independent and the implementation presented here is a mature proof of concept for these language-independent features.

Reflection is a language feature that allows a program to inspect code (introspection), and in the most elaborate case to create new code (intercession). From its definition it is obvious that reflection is relevant to the concept of generators, especially for generators that are parametrized with code, i.e. typically take code and produce new code. On the other hand, although reflection is a fascinating concept, in our view one of the important lessons that the programming language community has learnt since the heydays of language innovations is that we cannot simply use novelty as an argument for a new language feature. Therefore it is justified to ask: what is reflection really needed for, what is the best practice in using it for such purposes, and how can we convince ourselves that the usage in these scenarios is safe and leads to understandable code? Reflection, more than perhaps many other technologies, should immediately raise the spectre of obfuscation in the shape of meta-confusion. Therefore this argument is very important. Naturally, the answers to such questions cannot always be apodictic, but it is important to share arguments within the community. This research, as much as it is a novel take on generators, is also an attempt to characterize good practices for reflection. In a way, for us this research is a basis for exploring the following hypothesis: reflection should be used as a static mechanism for the generation of new code. The usage of reflection that we will present here will indeed be, on the one hand, very powerful in that it allows more than is easily doable with reflection in many languages. On the other hand, the usage will be very controlled in

that reflection only happens at a certain point in time, namely at generation time of code.

Often reflection is primarily understood as the capability of reflection at runtime. In the Genoupe framework, however, we consider reflection only at development time at first. We use the term development time as a generalization for compile time, to emphasize that the general concept behind Genoupe does not rely on the use of compiler languages. As an aside: one of the achievements of the increased interest in good software engineering practices is a consensus that good testing is indispensable. Hence, a discrete point in time, the shipment time after the test phase, has been established, independent from the question of whether the language is a compiler or interpreter language. As a consequence, there is a clearly defined development time, being the time before shipment.

In a language, runtime reflection can be offered to varying degrees. The classic concept of code generators is interesting for the concept of reflection because of the following fact: the possibility of generators shows that for all open languages, complete development-time static reflection is possible, through the notion of the parse-tree. This is a language-independent concept. The term "open language" implies a subtle but unfortunately not completely theoretical restriction: if the language is vendor-specific and can only be edited with proprietary tools in practice, the above fact, that development-time reflection is universally possible, is limited in its applicability. Examples of such languages can be found in the area of desktop databases or other office applications as well as in some modeling tool suites that do not follow open standards. This is typically the case if artifacts of the language are not fully specified, or the specification is not strictly implemented. This problem became quite relevant with the proliferation of graphical tools. With the new trend to open XML standards it might again be alleviated to some extent.

If we consider runtime reflection in a given language that has its own standard reflection API, then it might well be that both kinds of reflection, introspection as well as intercession, apply to only some of the code elements of a language. In Java, for instance, the introspection through the `java.lang.reflect` package is limited roughly to the interface concepts in the language (which is more than just the interface keyword – in fact, classes provide interfaces as well). Our language extension serves the purpose of exploring a new programming paradigm that uses reflection, but limits intercession to template-based generation. For introspection, interface introspection is sufficient for most of our considerations.

A further thought that is important to us is that both of the following are possible: development-time execution of generators as well as runtime execution of generators. The latter can be used to shed light on runtime reflection.

Runtime reflection is chiefly necessary if a system supports hot deployment, that is code is loaded at runtime that was not known and not available for static analysis at development time (or at least at startup time). Another term often used is mobile code. In fact both terms are equivalent from the viewpoint taken here, but we prefer the term "hot deployment" because it emphasizes the risks involved in loading code at runtime. We want to call the hosting program the *container*. We consider each act of hot deployment as an atomic act, and consider this a third type of time: hot-deployment time. Batch development tools are amenable to automatization and can be used at hot-deployment time. In UNIX, for instance, a make file could be executed at hot-deployment time.

Hot deployment is actually always runtime intercession, even if it may look different on a technology level, as in the case of the Java class loader. Conversely, we want to view runtime intercession always as hot-deployment. Since runtime execution of generators is possible, and we deem generators a conceptually preferable form of intercession, it suggests itself that we should capture applications of runtime reflection as invocations of runtime generators, and that these invocations happen at hot-deployment time. In these cases, the same intercession could have taken place at development time if the hot-deployed code had been available. First, this idea offers the possibility of using our Genoupe generator type system at hot-deployment time, which is why we deem this chain of thought important. Secondly, this idea sheds light on the question of why runtime intercession should be useful at all, apart from perhaps performance arguments, and why a paradigm based on the eval-function is not sufficient. In fact this shows a further application of generator type safety: in our approach, containers that use hot deployment can be checked at compile time, since this is the generator definition time for the hot-deployment capabilities of the containers. The discussion of different types of time is reminiscent of multi-level specialization [21]. However, our focus here is rather on limiting the number of different points in time reflection can occur, instead of providing an arbitrary number of them.

7 Related Work

Genoupe is an extension of *genericity* or parametric polymorphism found, for example, in ADA or Java [5,6]. 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 such as, for example, those supported by the notion of bounded parametric polymorphism [8,42]. Ad-hoc solutions for providing some level of concept checking for C++ templates, like specialized macros [44] and static interfaces [33], have been generalized by the introspection library approach in [51]. 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++ [2], sufficient, e.g. for an interface generator for a relational database [3]. It is still less powerful than Genoupe; for example, it cannot 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. The distinction between compile-time reflection and run-time reflection has been made in linguistic reflection [47].

Aspect oriented programming aims at handling crosscutting concerns in programs. AspectJ [28] 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 us 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 the method still has to be specified literally and cannot be made dependent on some parameter. The generative approach to aspect-oriented programming in [46] characterizes certain uniform patterns that arise in using the aspect oriented style of inverting functional decomposition as amenable to being 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 [41].

The concept of *runtime reflection* dates back to Lisp [45] and has been the subject of major interest in the functional programming community. The combination of parametric polymorphism with reflective features in Generic Haskell [23,22] benefits from the theoretically well-understood type-system of the host language. In the context of the object-oriented functional programming language CLOS [29,19], a mature metaobject protocol has been elaborated. In [50] CLOS is used to prove the value of metaprogramming by embedding representations of common object-oriented design patterns [20] into programs.

Multistage programming [48,49,7] is an approach that focuses on runtime program generation and execution. It is one approach in the general field of metaprogramming [43]. It is a programming extension that allows the explicit timing of the execution of expressions. The programmer is supported by constructs for partial evaluation and program specialization, whereas several properties of runtime generation can already be ensured statically. Multistage programming is thus a realization of non-transparent partial evaluation. An implementation of the multistage programming approach is provided on top of the object-based functional programming language O'Caml [31]. The language Metaphor [35] results from extending the subset of an object-oriented language like C# or Java by the multistage constructs of the functional programming language MetaML [48,49], 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. Multistage programming is primarily targeted at a different concern than Genoupe, namely optimization of program execution. There is a limited overlap with Genoupe, in that Genoupe allows some partial evaluation as well. One of the main motivations of Genoupe is however the support of generators that simplify the development of consistent libaries through its direct downstream compatibility.

Jasper [36,37] 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 [12] – an architecture that is known as *open compiler*. It supports universal metaprogramming and as such is more powerful, but less understood.

Some approaches use *model checking* in order to determine whether program code is well-typed. SafeGen [24] uses a restricted language based on predicate logic in order to generate program code, and checks safety assumptions such as uniqueness of identifiers using a standard model-checking tool. The fundamental idea of Genoupe to apply reflection at compile time has been taken up in the community. CTR [17] proposes a different solution for addressing well-formedness aspects through effectively imposing more restrictions on parameters. CJ [26] focuses on code generation that is dependent on a condition. MorphJ [25] uses control structures in the generator code similar to Genoupe, but for the selection of subsets of collections, MorphJ uses a pattern matching approach, while Genoupe also allows the use of expressions. Cayenne [4] is a functional language that supports the concept of term-dependent types. This makes it possible to describe types with Turing-complete, parametrized terms, with parameter values that may change during runtime and are statically unknown. Cayenne's type system tries to determine whether the terms that describe types will always result in types that are valid for the given program. A drawback of model checking approaches is that they usually deal

with problems that are generally undecidable, such as equivalence of program terms. This means that the type checker potentially never terminates. Even if it does, the complexity of model-checking is generally exponential, which rules out efficient use with very large programs.

There are other approaches for model-based generation of source code and other artifacts. For example, Extensible Stylesheet Language Transformations (XSLT) [27] makes it possible to define a translation from an XML source schema to a target XML schema, or a different textual document type. XSLT can perform context-free pattern matching, similar to tree parsers or template processors, as well as more complex transformations; it is Turing-complete. Query/View/Transformation [39] makes it possible to define transformations between XML Metadata Interchange (XMI) [40] documents, which are commonly used to encode UML model data. Similar to XSLT, QVT specifies pattern matching mechanisms as well as a Turing-complete imperative language. It is possible to use XSLT or QVT in order to generate source code, but they do not provide any support for generator type safety. Both specifications are large and complex, so it would not be easy to formulate appropriate type rules for them. Furthermore, their Turing-completeness would render type checking undecidable, unless sensible restrictions are applied.

8 Conclusion

We have presented a concept for generative programming that integrates reflection by means of a metalanguage into a template mechanism reminiscent of genericity. Genoupe is our implementation of this concept for the host language C#. It can be used to solve some 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 fosters a restricted and safe use of both generation and reflection, by limiting reflection to compile-time reflection, and limiting generation to generator-level code blocks that are integrated with the code structure of the host language.
- 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 easier to use and understand.
- A wide range of common applications of generative programming can be addressed.
- Genoupe provides a much stronger motivation for parameterized types than

parametric polymorphism. Parametric polymorphism can often be replaced by type inference. Genoupe-style parameterized types can change the type signature depending on the parameters, and can provide direct downstream compatibility. This is useful, for example, for the generation of APIs.
- Genoupe offers a particular high degree of static safety for reflection by means of a type system that is able to detect generator type errors.

The Genoupe type system supports a particularly unrestricted generator language: arbitrary C# methods can be used in a generator. To ensure generator type safety, Genoupe uses a memoization approach, making sure that structurally equivalent generator expressions yield the same value. Furthermore, Genoupe uses id schemes to cope with clashes of generated identifiers. The type system introduces new kinds of elements into the environment to describe important properties of generator variables and generated definitions. By accessing generated types with generator expressions, the type system is able to check whether a generated type is accessed correctly.

Using reflection in generators introduces an interesting shift from generators based on classic compiler-compiler techniques. The latter often use a formal-language approach in every generator to parse generator input. In contrast, by using reflection metaobjects can directly be used as generator parameters. Hence using reflection represents a shift from a language-based approach to a model-based approach to generative programming.

References

[1] A. Aho, R. Sethi, J. Ullman, Compiler: Principles, Techniques and Tools, Addison-Wesley, 1986.

[2] G. Attardi, A. Cisternino, Reflection Support by Means of Template Metaprogramming, in: GCSE '01: Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering, LNCS 2186, Springer, 2001.

[3] G. Attardi, A. Cisternino, Template Metaprogramming an Object Interface to Relational Tables, in: REFLECTION '01: Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, LNCS 2192, Springer, 2001.

[4] L. Augustsson, Cayenne - a language with dependent types, ICFP'98: Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (1998) 239–250.

[5] G. Bracha, N. Cohen, C. Kemper, S. Marx, M. Odersky, S.-E. Panitz, D. Stoutamire, K. Thorup, P. Wadler, Adding Generics to the Java

Programming Language: Participant Draft Specification, Tech. rep., SUN Microsystems (April 2001).

[6] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler, Making the future safe for the past: adding genericity to the Java programming language, in: OOPSLA '98: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, SIGPLAN Notices, ACM Press, 1998.

[7] C. Calcagno, W. Taha, L. Huang, X. Leroy, Implementing multi-stage languages using asts, gensym, and reflection, in: F. Pfenning, Y. Smaragdakis (eds.), GPCE, vol. 2830 of Lecture Notes in Computer Science, Springer, 2003.

[8] L. Cardelli, Type Systems, in: Handbook of Computer Science and Engineering, chap. 103, CRC Press, 1997.

[9] S. Chiba, A Metaobject Protocol for C++, in: OOPSLA '95: Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, 1995.

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

[11] D. Draheim, C. Lutteroth, G. Weber, 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.

[12] D. Draheim, C. Lutteroth, G. Weber, An analytical comparison of generative programming technologies, Tech. Rep. B-04-02, Institute of Computer Science, Freie Universität Berlin (January 2004).
URL
http://www.inf.fu-berlin.de/inst/pubs/tr-b-04-02.abstract.html

[13] D. Draheim, C. Lutteroth, G. Weber, Generative Programming for C#, ACM SIGPLAN Notices 40 (8).

[14] D. Draheim, C. Lutteroth, G. Weber, 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.

[15] D. Draheim, C. Lutteroth, G. Weber, Robust content creation with form-oriented user interfaces, in: Proceedings of CHINZ 2005 – 6th International Conference of the ACM's Special Interest Group on Computer-Human Interaction, ACM Press, 2005.

[16] D. Draheim, G. Weber, Form-Oriented Analysis - A New Methodology to Model Form-Based Applications, Springer, 2004.

[17] M. Fähndrich, M. Carbin, J. R. Larus, Reflective program generation with patterns, in: GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering, ACM, New York, NY, USA, 2006.

[18] R. A. Frost, Using memorization to achieve polynomial complexity of purely functional executable specifications of non-deterministic top-down parsers, SIGPLAN Not. 29 (4) (1994) 23–30.

[19] R. G. Gabriel, D. G. Bobrow, J. L. White, Object Oriented Programming - The CLOS perspective, MIT Press, 1993.

[20] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[21] R. Gl
"uck, J. Jørgensen, Multi-level specialization, Partial Evaluation (1999) 326–337.

[22] R. Hinze, J. Jeuring, Generic Haskell: Applications, in: R. Backhouse, J. Gibbons (eds.), Generic Programming: Advanced Lectures, LNCS 2793, Springer, 2003.

[23] R. Hinze, J. Jeuring, Generic Haskell: Practice and Theory, in: R. Backhouse, J. Gibbons (eds.), Generic Programming: Advanced Lectures, LNCS 2793, Springer, 2003.

[24] S. Huang, D. Zook, Y. Smaragdakis, Statically safe program generation with SafeGen, GPCE'05: Proceedings of the 4th International Conference on Generative Programming and Component Engineering (2005) 309–326.

[25] S. S. Huang, Y. Smaragdakis, Expressive and safe static reflection with morphj, in: PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, ACM, New York, NY, USA, 2008.

[26] S. S. Huang, D. Zook, Y. Smaragdakis, cj: enhancing java with safe type conditions, in: AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development, ACM, New York, NY, USA, 2007.

[27] M. Kay, XSL Transformations (XSLT) Version 2.0, Candidate recommendation feedback, W3C (November 2005).

[28] G. Kiczales, An overview of AspectJ, in: ECOOP '01: Proceedings of the European Conference on Object-Oriented Programming, LNCS 2072, Budapest, Hungary, 2001.

[29] G. Kiczales, J. des Rivières, The Art of the Metaobject Protocol, MIT Press, 1991.

[30] E. Kohlbecker, D. P. Friedman, M. Felleisen, B. Duba, Hygienic macro expansion, in: LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, ACM Press, 1986.

[31] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, J. Vouillon, The Objective Caml System Release 3.08 - Documentation and User's Manual, Tech. rep., Institut National de Recherche en Informatique et en Automatique (July 2004).

[32] C. Lutteroth, AP1: A platform for model-based software engineering, in: TEAA '06: Proceedings of the 2nd International Conference on Trends in Enterprise Application Architecture, Springer, 2006.

[33] B. McNamara, Y. Smaragdakis, Static interfaces in C++, in: Proceedings of the First Workshop on C++ Template Programming, NetObjectDays 2000, 2000.

[34] D. Michie, Memo functions and machine learning., Nature 218 (1968) 19–22.

[35] G. Neverov, P. Roe, Metaphor: A Multi-stage, Object-Oriented Programming Language, in: GPCE '04: Proceedings of the 3rd International Conference on Generative Programming and Component Engineering, LNCS 3286, Springer, 2004.

[36] D. Nizhegorodov, Jasper: Type-Safe MOP-Based Language Extensions and Reflective Template Processing in Java, in: Proceedings of the ECOOP'2000 Workshop on Reflection and Metalevel Architectures: State of the Art, and Future Trends, ACM Press, 2000.

[37] D. Nizhegorodov, Code-Generation Aspects of Jasper, a Reflective Meta-Programming and Source Transformations Processor, in: Proceedings of the ECOOP '02 Workshop on Generative Programming, ACM Press, 2002.

[38] P. Norvig, Techniques for automatic memoization with applications to context-free parsing, Comput. Linguist. 17 (1) (1991) 91–98.

[39] Object Management Group, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (November 2005).

[40] Object Management Group, MOF 2.0/XMI Mapping Specification Version 2.1 (September 2005).

[41] R. Paige, S. Koenig, Finite Differencing of Computable Expressions, ACM Trans. Program. Lang. Syst 4 (3) (1982) 402–454.

[42] B. C. Pierce, Types and Programming Languages, MIT Press, 2002.

[43] T. Sheard, Accomplishments and research challenges in meta-programming, in: SAIG'01: Proceedings of the 2nd international conference on Semantics, applications, and implementation of program generation, Springer-Verlag, Berlin, Heidelberg, 2001.

[44] J. Siek, A. Lumsdaine, Concept checking: Binding parametric polymorphism in C++, in: Proceedings of the First Workshop on C++ Template Programming, NetObjectDays 2000, 2000.

[45] B. C. Smith, Reflection and semantics in LISP, in: POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM Press, 1984.

[46] D. R. Smith, A Generative Approach to Aspect-Oriented Programming, in: GPCE '04: Proceedings of the 3rd International Conference on Generative Programming and Component Engineering, LNCS 3286, Springer, 2004.

[47] D. Stemple, R. Stanton, T. Sheard, P. Philbrow, R. Morrison, G. Kirby, L. Fegaras, R. Cooper, R. Connor, M. Atkinson, et al., Type-safe linguistic reflection: a generator technology.

[48] W. Taha, T. Sheard, Multi-Stage Programming with Explicit Annotations, in: PEPM '97: Partial Evaluation and Semantics-Based Program Manipulation, SIPLAN Notices, ACM Press, 1997.

[49] W. Taha, T. Sheard, MetaML and Multi-stage Programming with Explicit Annotations, Theoretical Computer Science 248 (1-2) (2000) 211–242.

[50] D. von Dincklage, Making Patterns Explicit with Metaprogramming, in: F. Pfenning, Y. Smaragdakis (eds.), GPCE '03: Proceedings of the 2nd International Conference Generative Programming and Component Engineering, LNCS 2830, Springer, 2003.

[51] I. Zólyomi, Z. Porkoláb, Towards a General Template Introspection Library, in: GPCE '04: Proceedings of the 3rd International Conference on Generative Programming and Component Engineering, LNCS 3286, Springer, 2004.