Code Generation from Applicative Terms

Jan Bessai

Technische Universität Dortmund, Chair of Software Engineering,
Otto-Hahn-Str. 14, 44227 Dortmund, Germany
jan.bessai@tu-dortmund.de
http://ls14-www.cs.tu-dortmund.de/index.php/Seminar-AI-Planning-12-13

Abstract. This work analyzes how to generate code from applicative terms, especially in the context of Dependency Injection. Relevant basic concepts are introduced and a threefold correspondence between Dependency Injection style object creation, applicative terms and Hilbert-style proofs is explored. The implementation of Syringe, a tool which can translate λ-expressions to XML fragments for the Spring Framework, is presented together with a demo application to compose GUI components.

1 Introduction

Today abstraction is the most essential way to tame the ever increasing complexity of software systems. Automated translation between the different domains connected with different abstraction layers has become standard. Part of this standard is code generation, which will be studied for the special case of applicative input terms throughout the following text. Section 2 describes the goals and architecture of code generators and the special properties of the chosen source language. An architectural design pattern known as Dependency Injection is introduced in section 3 in order to analyze its structural connection to applicative terms in section 4. This connection is used to implement a code generator in section 5. The gain in abstraction is demonstrated by using it to translate one simple expression into a complete GUI application. Related work is discussed in section 6. In the last part results are summarized and possible future research topics outlined.

2 Code Generation and Applicative Terms

Code generation is part of every compilation process and in fact the same as compilation itself. Therefore its goal is the same: namely to translate different code representations into another while obeying their semantics. Reasons to do this can vary. Maybe the source language is simpler and closer to a user domain (as it is with every high level programming language, which is compiled to assembler). Maybe the source language was only an internal representation, e.g. for a compiler. Or the source language might have been the target language of another code generator, as it is for the final code generation of AI-Planning processes. The vague concept of obeying semantics can be formalized further. Such formalization has been studied at great length and can be captured by the concept of Bisimulation [15]. It consists of several components:

P_S: The set of all source programs
P_T: The set of all target programs
$\mathcal{F}$: A Functor from pograms to processes $\mathcal{F}(P) = (\text{Input}_P \to \text{Output}_P)$
$\mathbf{run}_S$, $\mathbf{run}_T$, $\mathbf{run}_{ST}$: Coalgebras to execute programs and turn them into processes
$R \subseteq P_S \times P_T$: A relation denoting semantic equivalence

$$\begin{tikzcd}
P_S \arrow[d, "\text{run}_S"'] & R \arrow[l, "\pi_S"'] \arrow[r, "\pi_T"] \arrow[d, "\text{run}_{ST}"] & P_T \arrow[d, "\text{run}_T"] \\
\mathcal{F}\, P_S & \mathcal{F}\, R \arrow[l, "\mathcal{F}\,(\pi_S)"] \arrow[r, "\mathcal{F}\,(\pi_T)"'] & \mathcal{F}\, P_T
\end{tikzcd}$$

Fig. 1: A Bisimulation of source and target programs

These components form a Bisimulation if diagram 1 commutes, i.e. if

$$\begin{aligned} & \text{run}_S \circ \pi_S = \mathcal{F}(\pi_S) \circ \text{run}_{ST} \\ \text{and} \quad & \text{run}_T \circ \pi_T = \mathcal{F}(\pi_T) \circ \text{run}_{ST} \end{aligned}$$

hold for all elements of R. In words this means, that if source and target programs are semantically related, their execution has to give rise to processes which are related in the same way. The requirement for the mapping $\mathcal{F}$ to be a Functor is to ensure the upholding of sane mathematical properties (preservation of identity and composition). This is especially needed, if the process relation is to be studied further, as usually relations on processes (defined by their inputs and outputs in certain states) are subject to Bisimulation themselves [15]. Although the abstract notion of bisimilarity can be very helpful to formally verify certain aspects of codegeneration, the practical situation often rather looks like depicted in diagram 2. The lower left part of the diagram is completely grayed out, because there is usually no way to directly execute source programs. Moreover their specification is mostly given in an informal way, by natural language only. The interesting part of the diagram is the upper blue arrow, which symbolizes the morphism from source to target programs, i.e. the code generator. This morphism is equal to treating the relation R as a function, since there should be just one target program per source program instead of many.

While it might be worthwhile to study the mathematical properties of such a function, in computer science it is much more important, to be able to compute

$$\begin{tikzcd}
P_S \arrow[d, dotted, "\mathrm{run}_S"'] \arrow[rr, bend left, "r"] & R \arrow[l, "\pi_S"'] \arrow[r, "\pi_T"] \arrow[d, dotted, "\mathrm{run}_{ST}"] & P_T \arrow[d, "\mathrm{run}_T"] \\
\mathcal{F}\, P_S & \mathcal{F}\, R \arrow[l, dotted, "\mathcal{F}\,(\pi_S)"] \arrow[r, dotted, "\mathcal{F}\,(\pi_T)"'] & \mathcal{F}\, P_T
\end{tikzcd}$$

Fig. 2: Practical situation when constructing codegenerators (unknown parts are grayed out)

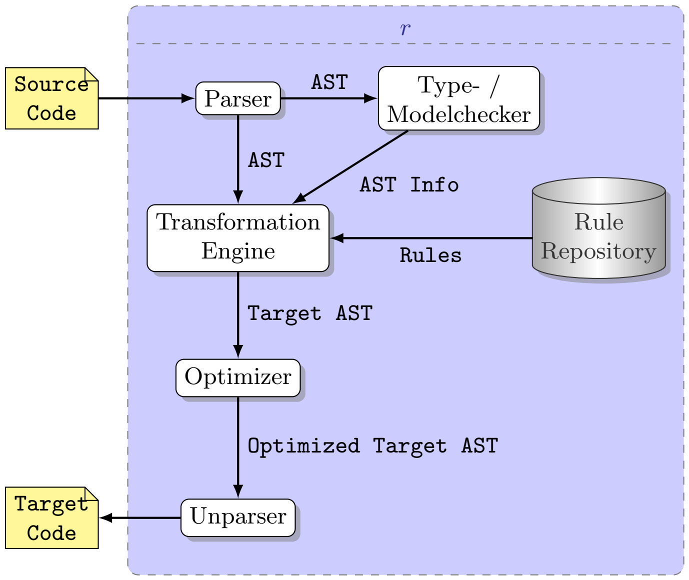


Fig. 3: Common architecture of codegenerators, based on [10]

it. The common architectural features of code generators are explained in great detail in [10]. Figure 3 shows an overview of these features: A parser is used to translate sequential input programs into an internal structure, usually an abstract syntax tree (AST). This structure is passed on and validated by a type or sometimes even model checker. Information from the validation process (e.g. inferred types) and the AST are then translated using rules from some kind of repository. In many generators this repository is woven into the transformation engine and does not appear as a separate component. The output of the transformation engine is an internal structure describing the program in terms of the target language. This structure can be optimized in an additional step and is finally translated into a sequential form again, which is the textual representation of the target program. The final translation step is inverse to parsing and therefore called unparsing. Sometimes a template engine is used to perform the last three steps in one, by directly replacing variables in a target code fragment with preprocessed parts of the input AST [10].

One of the input languages can be applicative terms [24]. These are comprised of a very simple grammar, consisting of only two rules

$$\Lambda \rightarrow V \mid (\Lambda)\,\Lambda \ ,$$

where V denotes members of a set of variable identifiers. If full the lambda calculus is to be considered, a third rule has to be added:

$$\Lambda \rightarrow \lambda V \,.\, \Lambda$$

Restrictive syntactical properties are advantageous not only in parser construction, but also for reasoning about programs. Hence the lambda calculus is perhaps the best studied form of writing down programs with observations going back to the early 1930s [9]. These observations especially include well understood type systems. Some type systems, for example simple types ($\lambda^{\rightarrow}$), or intersection types ($\lambda_{\cap}$), even allow to assert termination properties due to the fact that they guarantee strong normalization [24, 13]. They are of special interest to AI-Planning, since terms can be automatically generated for types by means of inhabitation [22]. By the Curry-Howard-Isomorphism [24] simply typed lambda terms are isomorphic to intuitionistic logic. Untyped lambda terms are even turing-complete [24]. Therefore they are, despite their simple structure, expressive enough to encode complex programs. An important part of being a well suited input language for code generation are the requirements imposed on output languages. These are exceptionally low for applicative terms. Basically only a template for the translation of term application to n-ary function calls is needed. This template can be exemplified for two arguments as follows:

$$((f)x)y \Rightarrow_r \$F(\$X_1, \$X_2)[\$F := f, \$X_1 := x, \$X_2 := y]$$

So within the text $\$F(\$X_1, \$X_2)$ the variables F, X_1 and X_2 are substituted by the free variables f, x and y from the applicative term. Of course the resulting target program has to define a binding to those variables in which f is a function taking 2 arguments and x and y have types corresponding to the argument types. New function definitions are not required, unless code generation is to be done for the full lambda calculus. If this should be the case nevertheless, the target language has to provide means for creating closures and passing them as parameters (usually in form of function pointers stored with arguments on a stack of continuations [19]). Further the typesystem of the target language has to be expressive enough to allow all terms of the input language. This is not always the case as can be best seen for the translation from $\lambda_\cap$ to $\lambda^\to$, which has to fail because

$$\vdash_\cap \lambda x\,.(x)x \,:\, (\sigma \cap (\sigma \to \tau)) \to \tau$$

and

$$\nvdash \lambda x\,.(x)x \,:\, \gamma\ ,$$

meaning that $\lambda x.(x)x$ is typeable in $\lambda_\cap$, but not in $\lambda^\to$ [13].

3 Dependency Injection

This section gives a short introduction to Dependency Injection, which serves as basis for an output language in section 4. Dependency Injection is an architectural pattern and as such best explained by example (this was also the method of choice in one of the earliest articles about it [12]). Listing 1.1 shows

```java
public class MissileLaunchpad {
  public void fireMissile() {
    Missile m = new NuclearMissile();
    m.launch();
  }
}
```

Listing 1.1: Every day Java code

an unusual Java class, `MissileLaunchpad`, that has a method to fire a missile. Calling this method creates a new `NuclearMissile` and fires it. Easy as it may seem, there are a few problems with this implementation: it introduces a mixture of concerns, since `MissileLaunchpad` should not be responsible for the creation of new missiles. While the newly created `NuclearMissile` is assigned to an object of the supertype (or interface) `Missile`, the advantages of polymorphism are instantly lost, because no different type of missile can be created without a change in code. The aforementioned problems are solved in the implementation presented in listing 1.2. The `Missile` object `MissileLaunchpad` depends on is injected during the constructor call, which is the reason to call this style of programming Dependency Injection. The aspect of Object creation can now be treated in a completely separate place, making it possible for users

```java
public class MissileLaunchpad {
  private final Missile m;

  MissileLaunchpad(Missile m) {
    this.m = m;
  }

  public void fireMissile() {
    m.launch();
  }
}
```

Listing 1.2: Dependency Injection version of listing 1.1

of `MissileLaunchpad` to fire different types of missiles. There are two popular variations on the presented pattern. Listing 1.3 shows a setter-based version of

```java
public class MissileLaunchpad {
  private Missile m;

  public setMissile(Missile m) {
    this.m = m;
  }

  public void fireMissile() {
    m.launch();
  }
}
```

Listing 1.3: Setter Dependency Injection

Dependency Injection. While this version can cause problems, if the setter is not called before each execution of `fireMissile`, it is sometimes necessary to allow objects to be initialized after their construction. Possible scenarios for this include cyclic dependencies and conformance requirements to standards relying on default consturctors (e.g. JSR-316 for managed Java Beans [23]).

In the previous versions a one to one correspondence between `Missile-Launchpad` and its associated `Missile` objects was implicitly created. This does not reflect reality and was not the case in the original code. The coupling is disbanded in listing 1.4 where factory based Dependency Injection is used. Now new missiles are provided by a factory object (line 9) of the (possibly abstract) type `MissileFactory`. This object can be configured to provide arbitrary instances of `Missile` and does so every time `fireMissile` is called. In contrast to listing 1.3, there is no risk of accidentally omitting the creation of new missiles.

```java
public class MissileLaunchpad {
  private final MissileFactory missileFactory;

  MissileLaunchpad(MissileFactory f) {
    this.missileFactory = f;
  }

  public void fireMissile() {
    Missile m = missileFactory.createMissile();
    m.launch();
  }
}
```

Listing 1.4: Factory Dependency Injection

The advantages mentioned above especially pay of, if a program is to be composed from multiple Dependency Injection style modules with unmodifiable source-code (e.g. third party libraries) [12]. In addition testability of such components is greatly improved [25], because Mock-Objects can be used to emulate complex behavior possibly causing unwanted side-effects (in the given example these would be the start of nuclear missiles on every test run). Further dependencies are explicitly stated, keeping them as precise and local as possible. Therefore all parts of a program only share essential parts of their state. This isolates failures and increases maintainability, because changes by error or design only affect subcomponents and not the program as a whole. To help with the final wiring of objects and their lifecycle a plethora of frameworks exist for almost all object oriented languages [4, 2, 7, 6, 5]. They commonly provide ways to create standardized factory code, support the notion of scopes and can in some places automatically fill in missing dependencies. The popularity of Dependency Injection has even led to affords of standardizing frameworks [18].

4 Translating λ-Terms to Injection Code

This section will explore the structural relation between object instantiation of Dependency Injection style code and applicative terms. Listing 1.5 shows a

```java
class X {
  public X(Y y, Z z) { ... }
}
```

Listing 1.5: Class with a binary constructor

Java class `X` with a binary constructor taking arguments of type `Y` and `Z`. It will serve as a placeholder for arbitrary constructor based Dependency Injection Code throughout the following text. Treating binary constructors is suffice, since the generalization to n-ary constructors follows from the isomorphism

$$\prod_{i=1\ldots k} x_i \cong (x_1, \prod_{i=2\ldots k} x_i))$$

and unary constructors or constructors without arguments need no special preparation. A new variable can be introduced for the constructor function and its type denoted as

$$\mathtt{X} : (\mathtt{Y}, \mathtt{Z}) \to \mathtt{X}$$

meaning that variable `X` has the function type $(\mathtt{Y}, \mathtt{Z}) \to \mathtt{X}$ taking a tuple of $(\mathtt{Y}, \mathtt{Z})$ as arguments and producing a value of type `X`. The tuple can be eliminated by a technique called currying [24]:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{X}:(\mathtt{Y},\mathtt{Z})\to\mathtt{X} \vdash \mathtt{X}:(\mathtt{Y},\mathtt{Z})\to\mathtt{X}} \quad \dfrac{\dfrac{}{\mathtt{y}:\mathtt{Y}\vdash\mathtt{y}:\mathtt{Y}} \quad \dfrac{}{\mathtt{z}:\mathtt{Z}\vdash\mathtt{z}:\mathtt{Z}}}{\mathtt{y} : \mathtt{Y},\, \mathtt{z} : \mathtt{Z} \vdash (\mathtt{y}, \mathtt{z}) : (\mathtt{Y}, \mathtt{Z})}\,(\wedge I)}{\mathtt{X} : (\mathtt{Y}, \mathtt{Z}) \to \mathtt{X},\, \mathtt{y} : \mathtt{Y},\, \mathtt{z} : \mathtt{Z} \vdash \mathtt{X}(\mathtt{y}, \mathtt{z}) : \mathtt{X}}\,(\to E)}{\mathtt{X} : (\mathtt{Y}, \mathtt{Z}) \to \mathtt{X} \vdash \lambda \mathtt{yz}.\mathtt{X}(\mathtt{y}, \mathtt{z}) : \mathtt{Y} \to \mathtt{Z} \to \mathtt{X}}\,(2{\times}{\to}I)$$

The proof above assumes properly typed arguments `x` and `y` exist, applies them to `X` and introduces two separate abstractions to eliminate the assumptions. To be able to reason about purely applicative terms the newly won curried constructor should be replaced by a fresh variable `x`, subject to substitution later. This is allowed, which can be shown by first using the free variables lemma [24]

$$\dfrac{\mathtt{x} : \mathtt{Y} \to \mathtt{Z} \to \mathtt{X} \vdash M : \sigma}{\mathtt{x} : \mathtt{Y} \to \mathtt{Z} \to \mathtt{X},\, \mathtt{X} : (\mathtt{Y}, \mathtt{Z}) \to \mathtt{X} \vdash M : \sigma}$$

to add the original constructor function to a reasoning environment only involving `x` and then applying the substitution lemma [24]

$$\dfrac{\dfrac{}{\mathtt{X}:(\mathtt{Y},\mathtt{Z})\to\mathtt{X}\vdash\lambda\mathtt{yz}.\mathtt{X}(\mathtt{y},\mathtt{z}):\mathtt{Y}\to\mathtt{Z}\to\mathtt{X}} \quad \mathtt{x} : \mathtt{Y} \to \mathtt{Z} \to \mathtt{X},\, \mathtt{X} : (\mathtt{Y}, \mathtt{Z}) \to \mathtt{X} \vdash M : \sigma}{\mathtt{X} : (\mathtt{Y}, \mathtt{Z}) \to \mathtt{X} \vdash M[\mathtt{x} := \lambda \mathtt{yz}.\mathtt{X}(\mathtt{y}, \mathtt{z})] : \sigma}$$

to substitute `x` by the curried form of `X`. The resulting context is purely applicative. It assigns free variables for every constructor to types either elementary, if the constructor takes no arguments, or higher order, if it takes one or more arguments. Application of those free variables corresponds to object construction in the target language. To allow this in a type-safe manner, two main typing rules are needed [24]:

$$\dfrac{}{\varGamma, x : \sigma \vdash x : \sigma} \qquad \dfrac{\varGamma \vdash M : \sigma \to \tau \;\; \varGamma \vdash N : \sigma}{\varGamma \vdash (M)N : \tau}$$

The first rule states that free variables are always typeable and the second rule states that application is typeable, if a matching argument type is supplied. By the Curry-Howard-Isomorphism the second rule corresponds to modus ponens [24]:

$$\frac{\sigma \to \tau,\ \sigma}{\tau}$$

In this logical interpretation the first rule just states that assumptions are always valid. If the target language allows to implement equivalents for the combinators $K = \lambda\, x\, y.\, x$ and $S = \lambda\, x\, y\, z.\, x\, z\, (y\, z)$, the axioms

$$\phi \to \psi \to \phi \tag{1}$$

$$(\phi \to \psi \to \vartheta) \to (\phi \to \psi) \to \phi \to \vartheta \tag{2}$$

may be added to the logical interpretation. Then the first rule can even be omitted by proof of $\phi \to \phi$ and application of Herbrand's theorem [24]. Listing

```java
public class Combinators {
  public static interface Function<S, T> {
    public T apply(S x);
  }
  public static <S, T> S k(S x, T y) { return  x; }
  public static <R, S, T> R s(Function<S, Function<T, R>> x,
      Function<S, T> y, S z) {
    return x.apply(z).apply(y.apply(z));
  }
}
```

Listing 1.6: Combinators in Java

1.6 shows a possible (uncurried) implementation of the S and K combinator in Java. Here generics are used to abstract from concrete types in the same way that type variables do. Function passing is facilitated by the generic interface `Function`. If the target language lacks support for generics and interfaces, a new version of the combinators could be created per supplied type. The possibility of describing object construction in an environment only dependent on the functional equivalent of modus ponens suggests that Dependency Injection style object construction corresponds to Hilbert-style proofs on a logical level. Table 1 sums up the correspondence.

Subtype polymorphism is an important feature in all object oriented languages [10]. Nevertheless the previous correspondence does not yet represent it. One possibility to do so is to extend the type system by introducing a subtyping relation $\leq$ and type intersection $\cap$ allowing for implementation of more than one interface or multiple inheritance respectively. Additional rules for a consistent intersection type system are given in [13]. The subtyping relation is chosen to be transitive and reflexive, exactly like the `instanceof`-relation in Java [14]:

$$\sigma \leq \sigma$$

$$\sigma \leq \tau \wedge \tau \leq \rho \Rightarrow \sigma \leq \rho$$

Dependency Injection	**Lambda Calculus**	**Logic**
Constructors	Free variables	Assumptions
Object construction	Application	Modus ponens
Objects	Primitive typed terms	Proven proposition variables
Static functions returning objects	Combinators	Axioms

Table 1: Correspondence of Dependency Injection, Lambda Calculus and Logic

Classes implementing more than one interface (or inheriting from more than one base) can be cast to all of these, therefore

$$\sigma \cap \tau \leq \sigma \wedge \sigma \cap \tau \leq \tau$$

which holds for constructor functions as well:

$$(\sigma \rightarrow \tau) \cap (\sigma \rightarrow \rho) \leq \sigma \rightarrow (\tau \cap \rho)$$

Nonelementary typed terms are expected to be contravariant in their arguments and covariant in their results, which is the same for functions in most object oriented languages:

$$\sigma \leq \sigma' \wedge \tau \leq \tau' \Rightarrow \sigma' \rightarrow \tau \leq \sigma \rightarrow \tau'$$

To incorporate subtyping and intersection properly three new rules are added [13]. First type-safe downcasting is obviously allowed:

$$\frac{\Gamma \vdash M : \sigma \; \sigma \leq \tau}{\Gamma \vdash M : \tau}$$

Second the intersection of several types (which cannot be directly denoted in Java [14]) can be cast to all of its components

$$\frac{\Gamma \vdash M : \sigma \cap \tau}{\Gamma \vdash M : \sigma} \; \frac{\Gamma \vdash M : \sigma \cap \tau}{\Gamma : M : \tau}$$

and third restored from them:

$$\frac{\Gamma \vdash M : \sigma \; \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau}$$

While the correspondence now covers all important parts of object oriented languages some corner cases remain. These are usually language specific. A common example are primitive types (e.g. `int`, `char`, ...), which are not captured by the inheritance system. Generalized proper inclusion of such types is almost impossible, because of their language and sometimes even platform specific semantic. Nevertheless they can usually be wrapped into objects, as is automatically the case for Java [14].

5 Implementation

In the previous section applicative terms were related to Dependency Injection style object construction. This section aims to describe the implementation of a code generator, named Syringe[1], which computes the translation from terms to code, realizing the morphism r from section 2 diagram 2. Preliminary to this some elementary design choices have to be made. A target language and an implementation language have to be selected. Since a mapping from every (curried) constructor function to a free variables will have to be created, it is desirable to chose an implementation language capable of analyzing constructors of the target language. Further to improve maintainability and integration in existing projects the use of a Dependency Injection framework is advisable. Some Dependency Injection frameworks, like Google Guice [2], do not support addressing of more than one object per type. In the input language context this limits each type to be inhabitated by only one term. Such a restriction is theoretically possible using linear logic and the corresponding **BCI** propositional calculus [24], but imposes a loss of generality. The Spring Framework [4] for Java can address arbitrarily many objects per type. Further it supports easy to create XML artifacts to describe object instantiation. In combination with the reflection API [1] to analyze existing constructors, Java and Spring XML fragments provide a suitable implementation and target language. Next the architecture presented in section 2 figure 3 has to be adapted. The resulting architecture is shown in figure 4. Instead

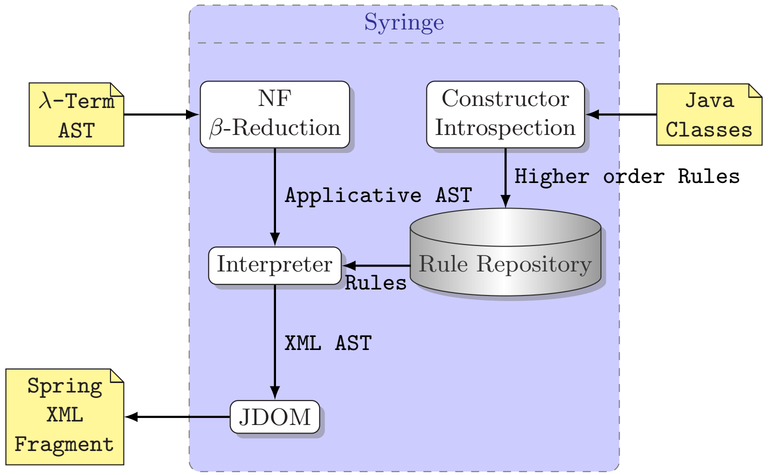


Fig. 4: Architecture of Syringe

of only supporting applicative terms, full lambda calculus was chosen as input

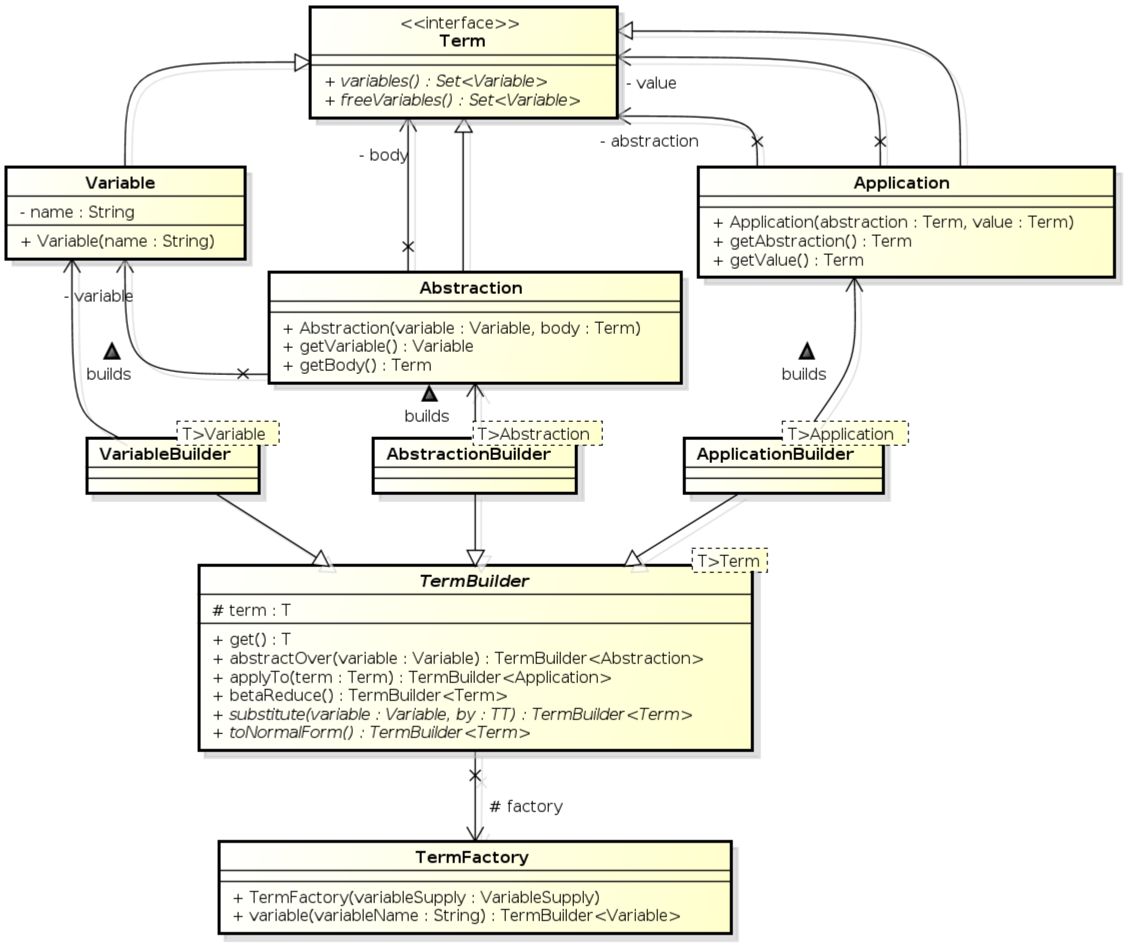
[1] The full source is available on Github: `https://github.com/JanBessai/Syringe`

language. This is advantageous, if custom combinators are to be introduced in order to shorten input. However the first stage of Syringe performs β-reduction to a normal form which may not contain any leftover abstractions. The parser stage is omitted in figure 4 because it is not essential to the core architecture. It can be completely skipped, if a code generator is configured to directly output ASTs as Java objects. Nevertheless a simple parser was implemented using ANTLR [21]. Its grammar, which is adapted to allow top down parsing with highest precedence of abstraction, is shown in listing 1.7. After β-reduction to an applicative

```
grammar Lambda;

lambda : abstractionLevel;

variableLevel : ID                          # variable
              | '(' abstractionLevel ')' # parenthesis
              ;

applicationLevel :
  variableLevel variableLevel* # application
  ;

abstractionLevel :
    '\\' ID+ '->' applicationLevel # abstraction
  | applicationLevel               # passDown
  ;

ID : [a-zA-Z]([a-zA-Z0-9]*);
WHITESPACE : [ \t\n\r] -> skip;
```

Listing 1.7: ANTLR Grammar for the lambda calculus

normal form an interpreter uses rules form a repository, in order to map remaining free variables to curried constructor functions and rewrite their application to an XML-AST. Typechecking is not yet implemented, because it is beyond the scope of this work, but it is planed for possible future versions of Syringe. The final AST is currently unparsed by JDOM [3], but the interpreter was designed independently from the output format of the rules it draws from its repository. Therefore, if the repository is initialized with rules, which build target code other than XML (e.g. source code for Java or even other languages), only the unparser stage is required to change. Figure 5 shows the class diagram for the term model and the β-reduction engine. Terms are modeled closely to their definition: the classes `Variable`, `Abstraction` and `Application` match the role of their abstract counterparts. They are generalized by the interface `Term`, providing methods to get all identifiers of occurring and free variables. Term construction and the β-reduction engine are implemented using the builder pattern. An appropri-



<<interface>>
Term
+ variables() : Set<Variable>
+ freeVariables() : Set<Variable>
Variable
- name : String
+ Variable(name : String)
Abstraction
+ Abstraction(variable : Variable, body : Term)
+ getVariable() : Variable
+ getBody() : Term
Application
+ Application(abstraction : Term, value : Term)
+ getAbstraction() : Term
+ getValue() : Term
- body
- value
- abstraction
- variable
builds
builds
builds
VariableBuilder
T>Variable
AbstractionBuilder
T>Abstraction
ApplicationBuilder
T>Application
TermBuilder
T>Term
# term : T
+ get() : T
+ abstractOver(variable : Variable) : TermBuilder<Abstraction>
+ applyTo(term : Term) : TermBuilder<Application>
+ betaReduce() : TermBuilder<Term>
+ substitute(variable : Variable, by : TT) : TermBuilder<Term>
+ toNormalForm() : TermBuilder<Term>
# factory
TermFactory
+ TermFactory(variableSupply : VariableSupply)
+ variable(variableName : String) : TermBuilder<Variable>

Fig. 5: UML class diagram for the Syringe AST and β-reduction engine

ate instance of a subtype of `TermBuilder` controls how the currently underlying term is to be manipulated (e.g. by substitution). This way manipulation logic is separated from the AST implementation itself. The class `TermFactory` provides methods to get a fresh `TermBuilder` starting with a single variable. It depends on an auxiliary `VariableSupply` object, which supplies fresh variable names (the class `VariableSupply` is not depicted to omit clutter in the diagram). Figure

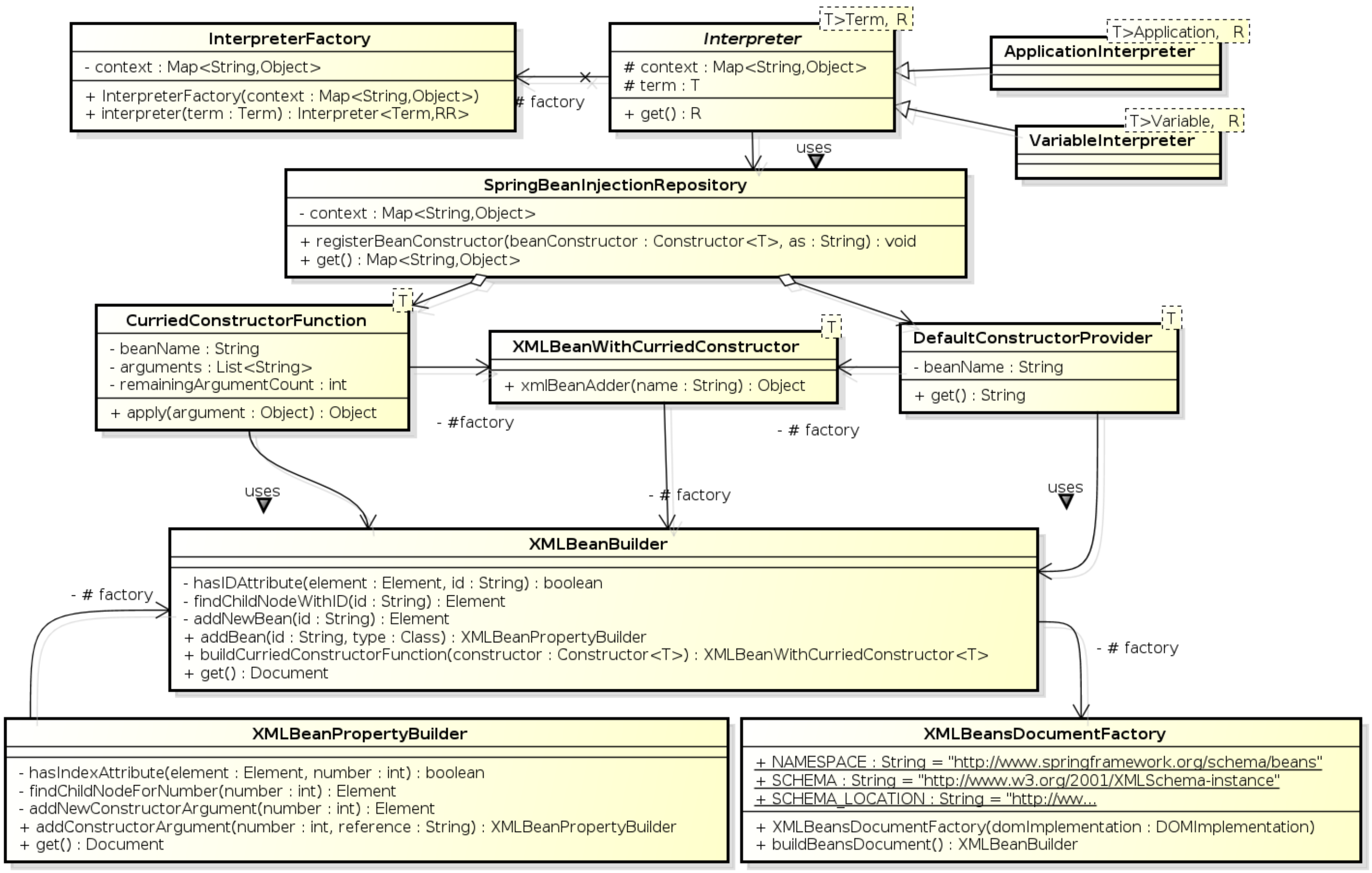

InterpreterFactory
- context : Map<String,Object>
+ InterpreterFactory(context : Map<String,Object>)
+ interpreter(term : Term) : Interpreter<Term,RR>
# factory
Interpreter
T>Term, R
# context : Map<String,Object>
# term : T
+ get() : R
ApplicationInterpreter
T>Application, R
VariableInterpreter
T>Variable, R
uses
SpringBeanInjectionRepository
- context : Map<String,Object>
+ registerBeanConstructor(beanConstructor : Constructor<T>, as : String) : void
+ get() : Map<String,Object>
CurriedConstructorFunction
T
- beanName : String
- arguments : List<String>
- remainingArgumentCount : int
+ apply(argument : Object) : Object
XMLBeanWithCurriedConstructor
T
+ xmlBeanAdder(name : String) : Object
DefaultConstructorProvider
T
- beanName : String
+ get() : String
- #factory
- # factory
uses
- # factory
uses
XMLBeanBuilder
- hasIDAttribute(element : Element, id : String) : boolean
- findChildNodeWithID(id : String) : Element
- addNewBean(id : String) : Element
+ addBean(id : String, type : Class) : XMLBeanPropertyBuilder
+ buildCurriedConstructorFunction(constructor : Constructor<T>) : XMLBeanWithCurriedConstructor<T>
+ get() : Document
- # factory
- # factory
XMLBeanPropertyBuilder
- hasIndexAttribute(element : Element, number : int) : boolean
- findChildNodeForNumber(number : int) : Element
- addNewConstructorArgument(number : int) : Element
+ addConstructorArgument(number : int, reference : String) : XMLBeanPropertyBuilder
+ get() : Document
XMLBeansDocumentFactory
+ NAMESPACE : String = "http://www.springframework.org/schema/beans"
+ SCHEMA : String = "http://www.w3.org/2001/XMLSchema-instance"
+ SCHEMA_LOCATION : String = "http://ww...
+ XMLBeansDocumentFactory(domImplementation : DOMImplementation)
+ buildBeansDocument() : XMLBeanBuilder

Fig. 6: Interpreter and rule repository

6 shows the class diagram for the interpreter and rule repository components. The interpreter is created by a factory which decides for a term in applicative normal form, which specialization to use (`ApplicationInterpereter` or `VariableInterpreter`). Interpreters draw their rules from a mapping between

variable identifiers (strings) and objects. Depending on its currently underlying term the interpreter instance will cast the objects found by lookup in this repository: either the `Provider` interface is chosen and its `get` method called, if no further arguments are to be passed, or the `Function` interface is chosen and its `apply` method is called for an interpreted argument. The mapping is established and supplied by the class `SprinBeanInjectionRepository`. Its method `registerBeanConstructor` takes an identifier and a `Constructor` object as parameter. The `Constructor` instance can be obtained by introspection for any Java class. It is used to create either a `DefaultConstructorProvider` or `CurriedConstructorFunction` object as lookup target. Which of them to choose is again determined by a builder (`XMLBeanBuilder`) which first produces a factory (`XMLBeanWithCurriedConstructor`) to build its results. The `XML BeanBuilder` is also used by the `get` and `apply` methods of `Default-ConstructorProvider` and `CurriedConstructorFunction` in order to append new nodes to the resulting XML-AST. Instantiation of the builder is handled by the `buildBeansDocument` method of `XMLBeansDocumentFactory`.

Figure 7 shows the class diagram for a small GUI-Framework based on Dependency Injection. It is comprised of a panel, which can show a label or a button. Both show a text with a custom font style. If the button is clicked a message box pops up, which again shows a text. This framework serves as an example for code generation with Syringe. An interactive demo allows to manually test the code generator. Figure 8 shows a screenshot in which the term

$$(\lambda\, x\, y\, .\, panel\, (button\, y\, (messageBox\, x)))\, hello\, ciao$$

was used as input. The text boxes above and below the input area on the left show the repository of available constructor functions and the generated output in text form. The input was successfully resolved to the normal form

$$(panel\, ((button\, ciao)\, (messageBox\, hello)))$$

which was translated to the human readable Spring XML fragment from listing 1.8. The right-hand side of figure 8 shows the instantiated GUI objects after the button, labeled “ciao”, was clicked and its message box appeared.

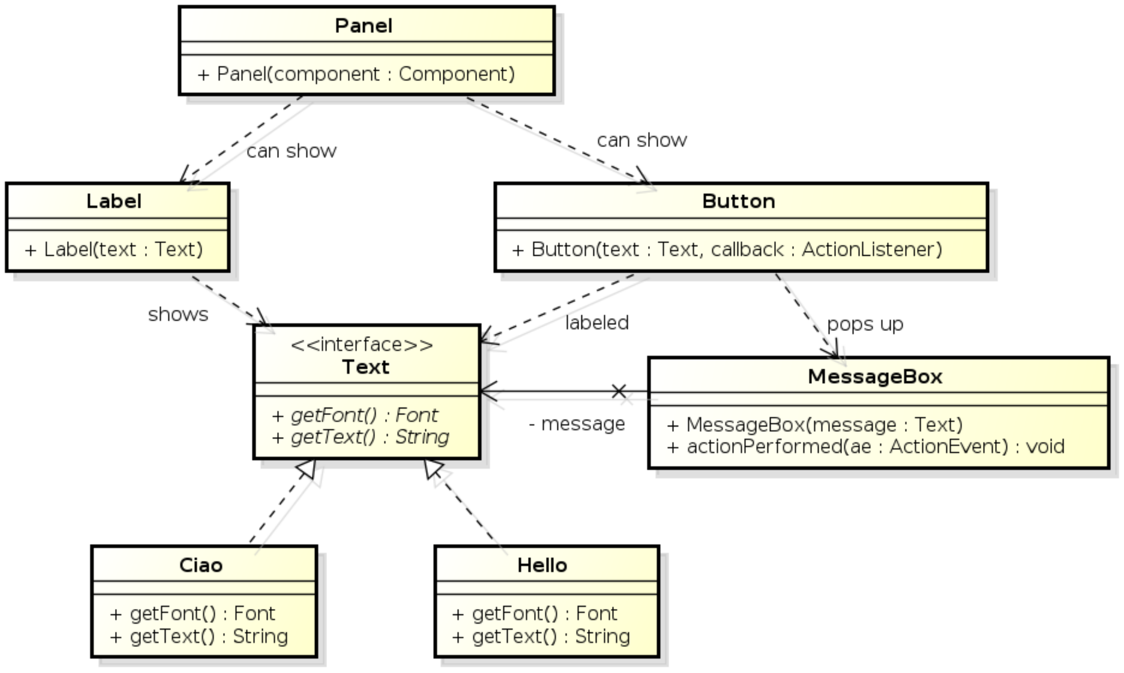
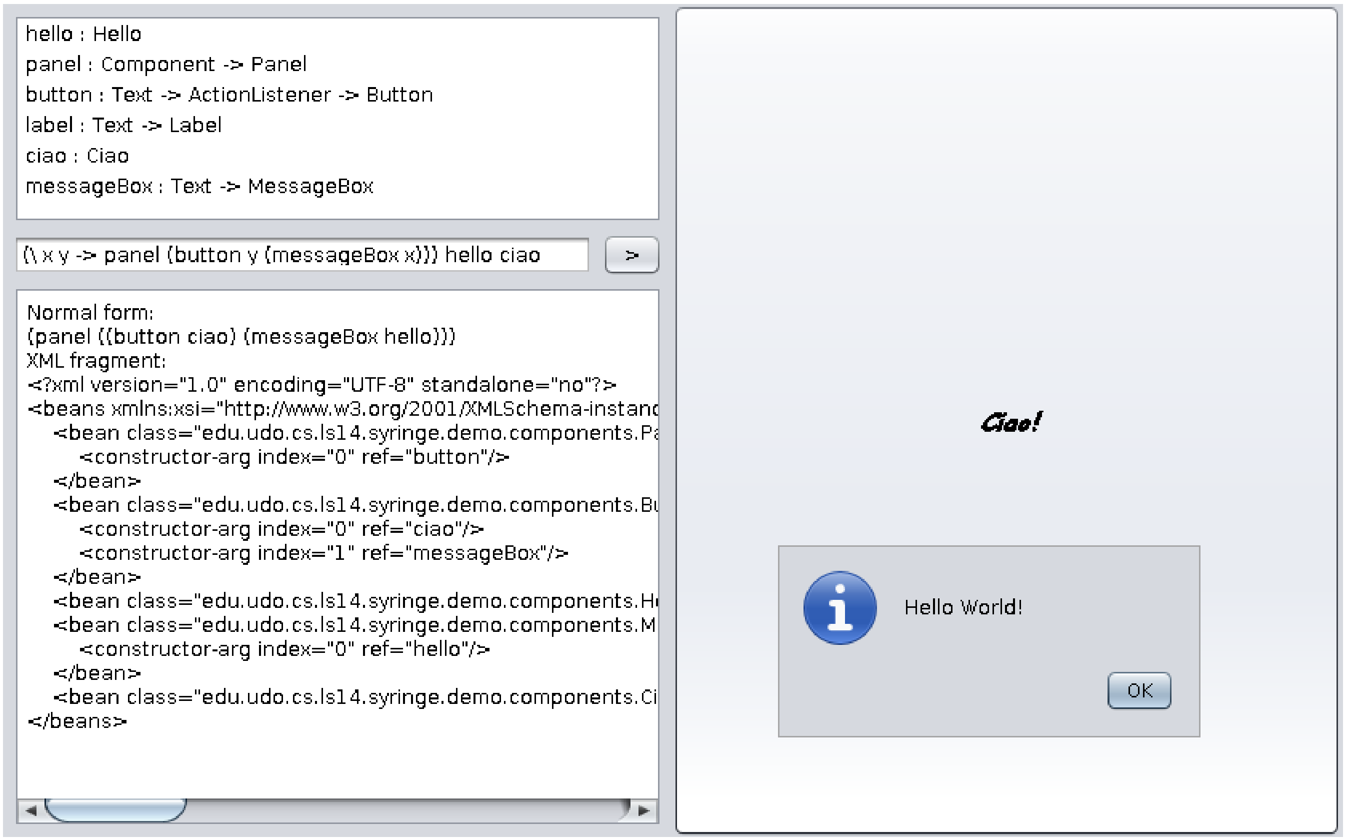


Fig. 7: Dependency Injection style GUI Framework



```
hello : Hello
panel : Component -> Panel
button : Text -> ActionListener -> Button
label : Text -> Label
ciao : Ciao
messageBox : Text -> MessageBox
```

```
(\ x y -> panel (button y (messageBox x))) hello ciao
```

```
Normal form:
(panel ((button ciao) (messageBox hello)))
XML fragment:
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc
    <bean class="edu.udo.cs.ls14.syringe.demo.components.Pa
        <constructor-arg index="0" ref="button"/>
    </bean>
    <bean class="edu.udo.cs.ls14.syringe.demo.components.Bu
        <constructor-arg index="0" ref="ciao"/>
        <constructor-arg index="1" ref="messageBox"/>
    </bean>
    <bean class="edu.udo.cs.ls14.syringe.demo.components.He
    <bean class="edu.udo.cs.ls14.syringe.demo.components.M
        <constructor-arg index="0" ref="hello"/>
    </bean>
    <bean class="edu.udo.cs.ls14.syringe.demo.components.Ci
</beans>
```

Fig. 8: Screenshot: Syringe demo application

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd"
    xmlns="http://www.springframework.org/schema/beans">
    <bean
        class="edu.udo.cs.ls14.syringe.demo.components.Panel"
        id="panel">
        <constructor-arg index="0" ref="button"/>
    </bean>
    <bean
        class="edu.udo.cs.ls14.syringe.demo.components.Button"
        id="button">
        <constructor-arg index="0" ref="ciao"/>
        <constructor-arg index="1" ref="messageBox"/>
    </bean>
    <bean
        class="edu.udo.cs.ls14.syringe.demo.components.Hello"
        id="hello"/>
    <bean
        class="edu.udo.cs.ls14.syringe.demo.components.MessageBox"
        id="messageBox">
        <constructor-arg index="0" ref="hello"/>
    </bean>
    <bean
        class="edu.udo.cs.ls14.syringe.demo.components.Ciao"
        id="ciao"/>
</beans>
```

Listing 1.8: Generated XML fragment

6 Related work

One of the first papers on the generation of programs by composition was written by Parnas in 1976 [20]. Since then a branch of programming, called Generative Programming, has been developed, which exclusively deals with generating programs. The book by Czarnecki and Eisenecker [10] studies this field in great detail. Almost arbitrary input and output language combinations can be chosen for a code generator. In [17] Lämmel even presents a framework to transform any context free grammars into one another. Applicative terms have been used as a generation source mostly in compilers for functional languages. The book by Peyton-Jones [16] gives a good introduction into compiling such languages in general and the newer paper [19] describes an efficient real world approach with a low level target language. Some modern functional programming languages like idris [8] are reduced to a mostly applicative term model during compilation. Finally in the context of AI-Planning the paper by Düdder, Grabe, Martens, Rehof and Urzyczyn [11] describes how to automatically synthesize intersection typed terms and to build GUIs from them. Further results on the complexity of such a synthesis are given in [22]. A very complete overview on the properties of the lambda calculus and type systems in general is given by Sørensen and Urzyczyn [24] and on intersection types in special by Ghilezan in [13].

7 Summary and Outlook

Code generation from applicative terms was studied with a special focus on Dependency Injection. In section 2 code generation was identified as a process translating code while obeying its semantic. The aspect of semantic equality was described on an abstract level using the concept of bisimilarity. A common program architecture to compute the translation function of code generators was presented. Applicative terms were identified as a well suited input language with very few requirements on potential output languages. In section 3 the architectural design pattern Dependency Injection was introduced. Its variations and advantages were discussed. The threefold correspondence between construction of Dependency Injection style objects, applicative terms and Hilbert-style proofs was analyzed in section 4. Next the results were put to a practical use. The implementation and design choices of a complete code generator, called Syringe, were discussed in section 5. An application to create GUIs was demonstrated, showing the potential of the technique. The last part gave an overview of related work.

A lot of interesting further research topics came up: The correspondence between dependency injection, applicative terms and logic could be analyzed, if setter and factory Dependency Injection are taken into account. Formal proof of its Bisimulation properties should be pursued. Further study of the role of linear logic for frameworks like Google Guice could provide interesting insights. Finally the promising results of the Syringe demo application definitely justify further development of the tool.

Bibliography

[1] Java reflection API (2012), http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html, Checked: Feb 22 2013

[2] Guice Framework for Java (2013), http://code.google.com/p/google-guice/, Checked: Feb 18 2013

[3] JDOM XML processing engine (2013), www.jdom.org, Checked: Feb 22 2013

[4] Spring Framework for Java (2013), http://www.springsource.org/spring-framework, Checked: Feb 18 2013

[5] Spring Python Framework (2013), http://springpython.webfactional.com/, Checked: Feb 18 2013

[6] Typhoon Framework for ObjectiveC (2013), http://www.typhoonframework.org/, Checked: Feb 18 2013

[7] Unity Container for .NET (2013), http://msdn.microsoft.com/en-us/library/dd203101.aspx, Checked: Feb 18 2013

[8] Brady, E.: The Idris programming language (2013), http://idris-lang.org/, Checked: Feb 22 2013

[9] Church, A.: A Set of Postulates for the Foundation of Logic. Annals of Mathematics 33(2), pp. 346–366 (1932), http://www.jstor.org/stable/1968337, Checked: Feb 18 2013

[10] Czarnecki, K., Eisenecker, U.: Generative Programming Methods, Tools, and Applications. Addison-Wesley, Amsterdam, Netherlands (2000)

[11] Düdder, B., Garbe, O., Martens, M., Rehof, J., Urzyczyn, P.: Using Inhabitation in Bounded Combinatory Logic with Intersection Types for GUI Synthesis. ITRS Intersection Types and Related Systems (2012)

[12] Fowler, M.: Inversion of Control Containers and the Dependency Injection pattern (2004), http://martinfowler.com/articles/injection.html, Checked: Feb 13 2013

[13] Ghilezan, S.: Strong Normalization and Typability with Intersection Types. Notre Dame Journal of Formal Logic 37(1), 44–52 (1996)

[14] Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: Java Language and Virtual Machine Specifications. Tech. rep., Oracle America, Inc. (2012), http://docs.oracle.com/javase/specs/jls/se7/html/index.html, Checked: Feb 22 2013

[15] Jacobs, B., Rutten, J.: A Tutorial on (Co)Algebras and (Co)Induction. EATCS Bulletin 62, 62–222 (1997)

[16] Jones, S.: The Implementation of Functional Programming Languages. Prentice-hall International Series in Computer Science, Prentice-Hall (1987), http://books.google.co.in/books?id=fZdQAAAAMAAJ, Checked: Feb 22 2013

[17] Lämmel, R.: Grammar Adaptation. In: Proc. Formal Methods Europe (FME) 2001. LNCS, vol. 2021, pp. 550–570. Springer-Verlag (2001)

[18] Lee, B., Johnson, R., et al.: JSR 330: Dependency Injection for Java. Tech. rep., Java Community Process (2009), http://www.jcp.org/en/jsr/detail?id=330, Checked: Feb 18 2013

[19] Marlow, S., Peyton Jones, S.L.: Making a fast curry: push/enter vs. eval/apply for higher-order languages. Journal of Functional Programming 16(4-5), 415–449 (2006)

[20] Parnas, D.L.: On the Design and Development of Program Families. IEEE Trans. Software Eng. 2(1), 1–9 (1976)

[21] Parr, T.: ANTLR parser generator (2013), `www.antlr.org`, Checked: Feb 22 2013

[22] Rehof, J., Urzyczyn, P.: Finite Combinatory Logic with Intersection Types (Extended Version). Tech. Rep. 834, Technische Universität Dortmund, Department of Computer Science, Dortmund, Germany (Feb 2011)

[23] Shannon, B., Chinnici, R., et al.: JSR 316: JavaTM Platform, Enterprise Edition 6 (Java EE 6) Specification. Tech. rep., Java Community Process (2009), `http://www.jcp.org/en/jsr/detail?id=316`, Checked: Feb 15 2013

[24] Sørensen, M.H.B., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism (1998), `http://ls14-www.cs.tu-dortmund.de/images/d/db/Curry-howard.pdf`, Checked: Jan 13 2013

[25] Wolter, J., Ruffer, R., Hevery, M.: Guide: Writing Testable Code (2008), `http://misko.hevery.com/attachments/Guide-Writing%20Testable%20Code.pdf`, Checked: Feb 15 2013