Code Generation from Applicative Terms

Seminar on AI-Planning

Jan Bessai

LEHRSTUHL 14
SOFTWARE ENGINEERING

Outline

Goal of a Code Generator

Same as for compilers:

- Translate code representations while obeying semantics

Because the input language is:

- more simple and closer to the user domain
- only an intermediate representation (e.g. within compilers)
- output language of another generator

Abstract Description (Bisimulation)

Given:

- Source programs P_S, target programs P_T
- Functor $\mathcal{F}$: Programs $\to$ Processes
    - $\mathcal{F}(P) = (\mathrm{Input}_P \to \mathrm{Output}_P)$
- Execution coalgebras run_S, run_T and run_{ST}
- Relation $R \subseteq P_S \times P_T$ denoting semantic equivalence

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

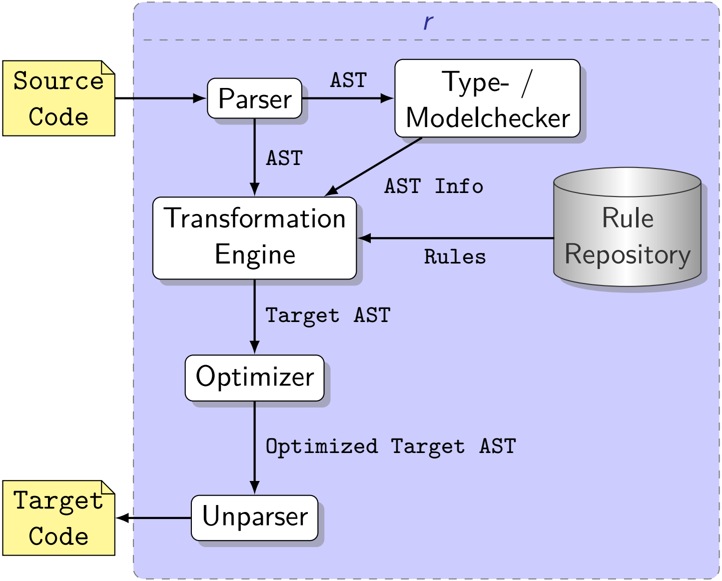
Compute R, such that the above diagram commutes
(π_S and π_T are projections)

In Practice

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

- run_s usually only given as an informal (textual) description
  - No way to execute source programs directly
- Relation R is seen as a function $r : P_S \to P_T$
- We are interested in a program computing r

Architecture of a Code Generator



Based on [1]

Applicative Terms as an Input Language

Applicative terms are a very convenient input language:

- Simple grammar
  - $\Lambda \rightarrow V \mid (\Lambda)\,\Lambda$
  - $\Lambda \rightarrow \lambda\, V.\, \Lambda$
- Well understood type systems
- Provable termination properties if properly typed (due to strong normalization [5, 2])
- Expressive enough for first order logic (Curry-Howard Isomorphism [5])
- Efficient reduction schemes (e.g. via DAGs [4])
- Automatic generation via **type inhabitation** [3]

Arbitrary Programming Languages as Output

There are very limited requirements on output languages

- A template for some form of n-ary function calling is needed:
    - $((f)x)y \Rightarrow \$F(\$X_1, \$X_2)[\$F := f, \$X_1 := x, \$X_2 := y]$
- The typesystem of the output language must not be too restrictive
    - e.g. there is no translation from $\lambda\cap$ to $\lambda \rightarrow$ (since $\vdash_\cap \lambda x.(x)x : (\sigma \cap (\sigma \rightarrow \tau)) \rightarrow \tau$)
- For abstractions new functions have to be declared
    - Some way to return higher order functions is required (e.g. via function pointers)

Dependency Injection

Every Day Code (Java)

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

Problems:

- Interface `Missile` used, but advantages of subtype polymorphism ignored
- Always fires nuclear missiles
- Mixture of concerns: launch pads should not produce missiles

Version using Dependency Injection

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

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

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

- Now the user can decide, which missile type is fired

Possible Variations (Setter Dependency Injection)

```
public class MissileLaunchpad {
  private Missile m;

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

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

- Adheres to standard for Managed Java Beans (JSR-316), which requires a 0-argument constructor
- Problematic if setup code fails to call setter before usage
- Undefined behavior if same missile is fired twice

Possible Variations (Factory Dependency Injection)

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

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

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

Even better approach:

- `fireMissile` can be called more than once

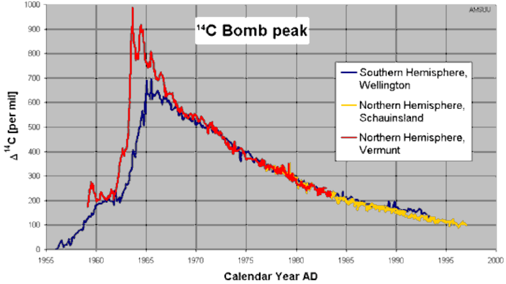
Advantages (Composition & Reuse)

```
new MissileLaunchpad(new ISSModuleCarrierRocket())
```

Components written in Dependency Injection style

- rely on composition by design
- can be reused differently in different contexts
- only ask for dependencies they really need
  - less global state
  - no more traditional use of singletons

Advantages (Testing)



Source: `http://testweb.science.uu.nl/AMS/Radiocarbon.htm`

Mock objects can be passed into constructors for testing

- Testing the `MissileLaunchpad` class can be done without havoc

Frameworks

Dependency Injection frameworks help to wire things up:

- provide standardized ways to create factory code
- automatically search for ways to fulfill dependencies
- manage repositories of accessible objects
- control life cycles (singleton, request scope,...)

Frameworks exist for many language environments (Java, .NET, C++, Python ...)

- even standardized to some extend (e.g. JSR-330)

λ-Terms to Injection Code

Structural connection (1)

Given any class:

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

- We create a constructor term

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

- Which we can curry:

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

Structural connection (2)

We may reason about an applicative term M and later replace a free variable with our curried constructor:

- By the free variables lemma:

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

- Substitution lemma:

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

Structural connection (3)

Now we can reason with our constructor placeholders as variables and application as the only rule:

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

Thus using the Curry-Howard-Isomorphism:

- Dependency Injection style object creation $\cong$ Hilbert-Style proofs using constructor types as axiom schemes :-)!

A note on types

For inheritance and interfaces we add:

- if `X instanceof Y` then $\mathtt{X} \leq \mathtt{Y}$
- if `X implements Y, Z` then $\mathtt{X} \leq \mathtt{Y} \cap \mathtt{Z}$
- if `X instanceof XParent, Y instanceof YParent` then $\mathtt{XParent} \to \mathtt{Y} \leq \mathtt{X} \to \mathtt{YParent}$

Further we allow typesafe (up)casting by adding:

- $\dfrac{\Gamma \vdash M : \sigma \cap \tau}{\Gamma \vdash M : \sigma} \qquad \dfrac{\Gamma \vdash M : \sigma \cap \tau}{\Gamma \vdash M : \tau}(\cap E)$
- $\dfrac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau}(\cap I)$
- $\dfrac{\Gamma \vdash M : \sigma \quad \sigma \leq \tau}{\Gamma \vdash M : \tau}(\leq)$

Now we have an applicative fragment of $\lambda_\cap$ [2]

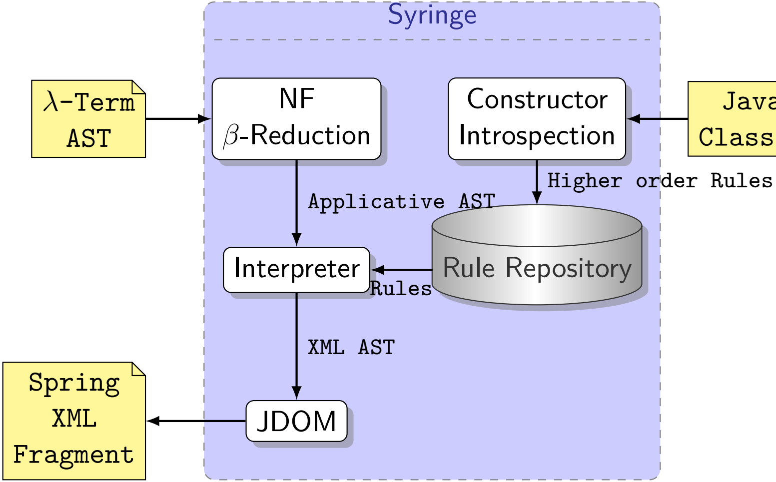
- Our bisimilarity relation R relates object construction with equally typed λ-terms :-)!

Implementation

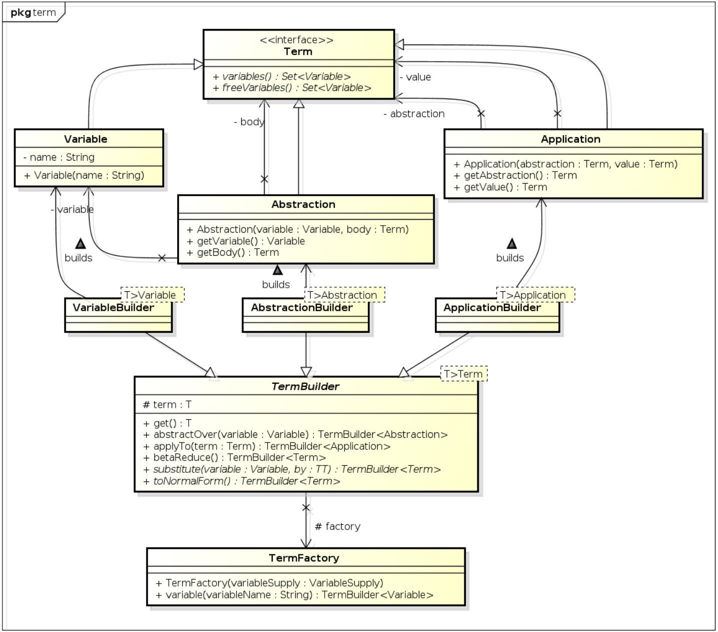
Things to consider during implementation:

- Target language
    - Automatic rule repository generation by constructor introspection
- Use of a dependency injection framework
    - Generated code should control a framework to create objects
    - Framework should support identifiers for multiple values of the same type
- I have chosen Java and the Spring framework to create a tool called Syringe

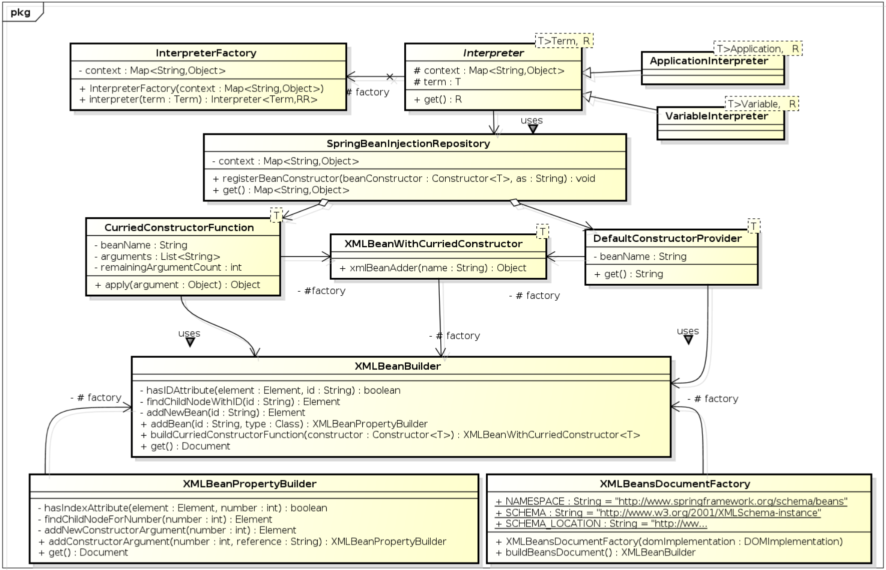
Syringe (Overview)



Software Architecture (Term model)



Software Architecture (Interpreter)



pkg

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

Demo

What we have seen today

- Goals and common structure of code generators
  - Bisimulation
- Properties of applicative terms as input languages
- Dependency injection in relation to applicative terms
  - Hilbert style object construction
  - Bisimilarity relation on types
- Implementation of a code generator

Things I'd like to to

- Add a typechecker to Syringe
- Add support for setter injection
- Implement better reduction strategies
- Real proof of bisimilarty properties of R

Feedback or Questions?

- Thank you :-) !

References

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

[2] Ghilezan, S.: Strong normalization and typability with intersection types. Notre Dame Journal of Formal Logic 37(1), 44–52 (1996)

[3] Rehof, J., Urzyczyn, P.: Finite combinatory logic with intersection types (extended version). Tech. Rep. 834, Technische Universität Dortmund, Department of Computer Science, Dortmund, Germany (February 2011)

[4] Shivers, O., Wand, M.: Bottom-up β-reduction: Uplinks and λ-dags. Fundamenta Informaticae 103(1–4), 247–287 (2010)

[5] 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`