42. Generic Programming with Generic Components

Prof. Dr. Uwe Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de
Version 16-0.1, Juni 4, 2016

1. Full Genericity in BETA
2. Universal Genericity with Slot Markup Languages
3. Semantic Macros
4. Template Metaprogramming
5. Evaluation
Obligatory Reading

- Invasive Software Composition, Chapter 6
Literature

- BETA home page http://www.daimi.au.dk/~beta/
  - Ole Lehmann Madsen. The Mjölnér BETA fragment system. In [BETA-ENV]. See also http://www.daimi.au.dk/~beta/Manuals/latest/yggdrasil
  - The boost C++ library project http://www.boost.org/
42.1 Full Genericity in BETA
**Generic Components**

- A **generic component** is a *template* from which other components can be generated
  - Generic components rely on *bind* operations that bind the template parameter with a value (*parameterization*)
    - The result is called the *extent*
  - A **generic class** is a special case, in which types are parametric
  - Bassett calls generic component **frames**

- A **fully generic language** is a language, in which all language constructs can be generic, i.e., are templates from which concrete constructs can be generated
  - Then, the language need to have a *metamodel*, by which the parameters are typed
Template parameters (slots) must be bound by fragment values

result = bind t.b to f, bind t.d to g

// In Lambda-N:
result = t(b => f, d => g)

// OO notation
result = t.bind(b,f).t.bind(d,g)

Result:
\[
a = 123.4 + c/ e + 4
\]
Generic Programming is a Composition Technique Relying on the Bind Operator (Parameterization)
BETA Fragment Metaprogramming System

- BETA is a modern object-oriented language, developed in the North (Oslo, Copenhagen)
  - BETA is the successor of Simula [BETA]
  - BETA programming environment Mjölner 1994 [BETA-ENV]

Features of BETA

- Classes and methods are unified to patterns (templates)
  - Classes are instantiated statically, methods dynamically
- Programming environment Mjölner is controlled by BETA grammar
  - Extension of the grammar changes all tools
- BETA is a fully generic language: all language constructs can be generic
- BETA metaprogramming system Yggdrasil
  - Separate compilation for all sentential forms of the grammar (all fragments generatable by the grammar)
  - Essentially, a BETA module is a generic fragment of the language
- BETA is a better LISP, supports typed metaprogramming
The Component Model of BETA and Mjölner

- The basic component in the BETA system is a *code fragment (code snippet)*
  - **Plain Fragment (snippet):** Sentential form, a partial sentence derived from a nonterminal
  - **Generic Fragment** (*fragment form, fragment template, snippet template, code frame*): Fragment that still contains nonterminals (*slots, code parameters*)
  - **Fragment Group** (fragment box): Set of fragments

```define fragment-component PersonTemplate = {
   name '/home/assmann/PersonTemplate'
   Person : PatternDecl
   Person : begin
      PersonMembers : begin
         name : @String
         <<EmployerSlot : Attribute>>
      end
   end
}```
A **fragment (snippet)** is a sequence of terminals, derived from a nonterminal in a grammar.

**Grammar example:**

- \( Z ::= \text{Address Salary} . \)
- \( \text{Address ::= FirstName SecondName Street StreetNr Town Country.} \)
- \( \text{Salary ::= int.} \)

Then, the following ones are fragments:

- Uwe Assmann Rudolfstrasse 31 Frankfurt Germany
- 34
- Uwe Assmann

But a complete sentence is

- Uwe Assmann Rudolfstrasse 31 Frankfurt Germany 34

A fragment can be given a **name (named fragment)**

- MyAddress: Uwe Assmann Rudolfstrasse 31 Frankfurt Germany
Generic Fragments

- A **generic fragment** (*fragment form, sentential form, snippet template, frame*) is a sequence of terminals and nonterminals, derived from a nonterminal in a grammar, perhaps named.

- Example:
  - Uwe Assmann <<Strasse>> Frankfurt Germany
  - MyAddress: Uwe Assmann <<Strasse>> Frankfurt Germany

- In BETA, the “left-in” nonterminals are called **slots**.

- A **fragment group** is a set of fragments:
  - { Uwe Assmann Rudolfstrasse 31 Frankfurt Germany 34
    Uwe Assmann }

- A **fragment file** is a file containing a fragment or a fragment group.
  - In BETA metaprogramming environments, all fragments are stored in the file system in fragment files.
Binding a Slot of a Generic Fragment in BETA

Done implicitly by name binding

// Generic Fragment
define fragment-component PersonTemplate = {
    name `'/home/assmann/PersonTemplate'
    Person : PatternDecl
    Person : begin
        PersonMembers : begin
            name : @String
            EmployerSlot : Attribute
        end
        end
    }

// Fragment
define fragment-component PersonFiller = {
    name `'/home/assmann/PersonFiller'
    origin `'/home/assmann/PersonTemplate'
    EmployerSlot: Attribute
    EmployerSlot: begin
        employer: @Employer;
        salary: Integer
    end
}

Person : PatternDecl
Person : begin
    PersonMembers : begin
        name : @String
        employer: @Employer;
        salary: Integer
    end
end
• Binding a slot can be seen as a call to the \texttt{bind} composition operator

\begin{verbatim}
define fragment-component PersonTemplate = {
    name `/home/assmann/PersonTemplate'
    Person : PatternDecl
    Person : begin
        PersonMembers : begin
            name : @String
            <<EmployerSlot : Attribute>>
            end
        end
}
define fragment-component PersonFiller = {
    name `/home/assmann/PersonFiller'
    origin `/home/assmann/PersonTemplate'
    EmployerSlot: Attribute
    EmployerSlot: begin
        employer: @Employer;
        salary: Integer
    end
}
define fragment Person = PersonTemplate.
EmployerSlot.bind(PersonFiller);
\end{verbatim}
Conditional Binding of Generic Statements in BETA Slot Syntax

Component methodComponent = cs.createTemplate("P");
Slot statement = methodComponent.findSlot("MY");

if (StdoutVersion) {
    statement.bind("System.out.println("Hello World");");
} else {
    statement.bind("FileWriter.println("no way");");
}
A **fragment group** is a group of sentential forms, derived from the same nonterminal:

```java
standardLoopIterators = {
    Upwards: for (int i = 0; i < array.<len:Function>>; i++)
    Downwards: for (int i = array.<len:Function>>-1; i >= 0; i--)
}
```

```java
len:Function
```
Implicit Binding also works in BETA Fragment Groups

- Fragments can be combined with others by reference (*implicit* bind operation)
- Given the following fragments:

```
len = { size() }
standardLoopIterators = {
    Upwards: for (int i = 0; i < array.<len:Function>>; i++)
    Downwards: for (int i = array.<len:Function>>-1; i >= 0; i--)
}
LoopIterators = standardLoopIterators, len
```

- The reference binds all used slots to defined fragments. Result:

```
LoopIterators = {
    Upwards: for (int i = 0; i < array.size(); i++)
    Downwards: for (int i = array.size()-1; i >= 0; i--)
}
```
Advantages

- Fine-grained *fragment component model*
  - The slots (code parameters) of a beta fragment form its *composition interface*
  - The BETA compiler can compile all fragments separately
  - Snippets with all kinds of language constructs can be reused
  - Type-safe composition with composition operation *bind-a-fragment*
  - Mjölner metaprogramming environment is one of the most powerful software IDE in the world (even after 15 years)

**Full genericity:** A language is called *fully generic*, if it provides genericity for every language construct.
Inclusion of Fragments into Fragment Groups

- Fragments can be inserted into others by the *include* operator
- Given the above fragments and a new one

\[
\text{whileloopbody} = \text{WHILE } \langle\langle \text{statements:statementList}\rangle\rangle \text{ END;}
\]

- A while loop can be defined using the include operator:

\[
\text{whileloop} = \{ \\
\quad \text{include LoopIterators.Upwards} \\
\quad \text{whileloopbody} \\
\}
\]

- BETA is a fully generic language:
  - Modular reuse of all language constructs
  - Separate compilation: The BETA compiler can compile every fragment separately
  - Much more flexible than ADA or C++ generics!
Evaluating BETA as a Composition System

- BETA's fragment combination facilities use as composition operations:
  - An *implicit bind* operation (fragment referencing by slots)
  - An inclusion operation (concatenation of fragments)
- Hence, BETAs composition language is rather simple, albeit powerful
Generic Components (Templates) Bind at Compile Time

New fragment-component

Generic Fragment (Template)

Value (Fragment)

Value (Generic Fragment)
42.2 Universal Genericity with Slot Markup Languages
**Slots (Declared Hooks)**

*Slots* are declared variation points of fragments.

**Slots (declared hooks)** are declared by the component writer as fragment parameters.
Different Ways to Declare Slots

Slots are denoted by metadata. There are different alternatives:

- **Language extensions with new keywords**
  - SlotDeclaration ::= 'slot' <Construct> <slotName> ';
  - In BETA, angle brackets are used:
    - SlotDeclaration ::= '<<' SlotName ':' Construct '>>'

- **Meta-Data Attributes** are language-specific
  - Java:  @superclass(SC)
  - C#:      [superclass(SC)]

- **Comment Tags** can be used in any language
  - class Set /* @superClass */

- **Markup Tags** in XML can be used for marking up code
  - <superclasshook> SC </superclasshook>^

- **Standardized Names (Hungarian Notation)**
  - class Set extends genericSCSuperClass { }
[Hartmann] showed that any XML language can be enriched by a **slot markup language** to define slots. Slot markup languages use **hedge symbols** to demarcate template and slot (BETA: << >>, XML: < >, Here: <slot >)

[Arnoldus] did the same for textual languages.
Conditional Binding of Generic Modifiers in XML Markup Syntax

- Slot markup languages may contain elements of a composition language, i.e., control flow structures
- A slot program expands the slot to a fragment [Hartmann]

```xml
<slot name="M" type="Modifier" />
public print() {
    System.out.println("Hello World");
}
```

```java
Component methodComponent = cs.createTemplate();
Slot modif = methodComponent.findSlot("M");
if (parallelVersion) {
    modif.bind("synchronized");
} else {
    modif.bind(" ");
}
```
Universal Genericity with Slot Markup Languages

• Do not use string template engines, they render development error-prone
• Use slot markup languages to exploit their typing
• With appropriate hedge symbols, a slot markup language can be combined with a base language [Hartmann]

Principle of universal genericity:
With slot markup separated by appropriate hedge symbols, any language may have typed generic components, as well as full genericity.
Macros based on metamodels or grammars
**Semantic Macros (Hygenic Macros)**

- Usually, macros are string-replacement functions (lambdas)
- **Semantic macros** are macros whose arguments are typed by nonterminals (as in BETA; builds on the typed lambda calculus)
- Example:

```javascript
function makeExpression(Left:Expression, Op:Operator, Right:Expression):Expression {
    return Left ++ Op ++ Right;  // ++ is AST concatenation
}
function incr(a:Expression):Expression {
    return makeExpression(1,+,a); }
function sqr(a:Expression):Expression {
    return makeExpression(a,*,a); }
Code result = incr(2);
i:int = eval(result);
// result: i == 3;
k:int = eval(sqr(10));
// result k == 100;
```
Comparing Semantic Macros and Slot Markup Languages

- Semantic Macros use the functional application symbols (\(\)\) as hedge symbols, i.e., are better integrated with the host language
  - Like slot programs they expand in-place
- Semantic Macros are better reusable, because they have a name
  - Slot programs are anonymous lambdas
42.4 Template Metaprogramming and Layered Template Meta-programming

The poor man‘s generic programming
Template Metaprogramming (TMP)

- Template Metaprogramming (TMP) is programming with generic fragments
- TMP in C++ [CE00] is an attempt to realize the generic programming facilities of BETA in C++
  - C++ has templates, i.e., parameterized expressions over types, but is not a fully generic language
  - C++ template expressions are Turing-complete and are evaluated at compile time
  - C++ uses class parameterization for composition
- Disadvantage: leads to unreadable programs, since the template concept is being over-used
- Advantage: uses standard tools
- Widely used in the
  - C++ Standard Template Library STL
  - boost library [www.boost.org](http://www.boost.org)
- Should be replaced by full genericity (generic fragments) or semantic macros
Template Metaprogramming in C++

template <int N>
struct fact {
    enum { value = N * fact<N-1>::value };
};

template <>
struct fact<1> {
    enum { value = 1 };
};

std::cout << "5! = " << fact<5>::value << std::endl;

More advanced examples in [CE00]
Generic Classes (Class Templates) Bind At Compile Time

New Class

Template class

Hook class

Hook class
Layered Template Metaprogramming with GenVoca

- GenVoca: Composition by Nesting of Generic Classes [Batory]
- Use nesting of templates parameters to parameterise multiply
  - Every nesting level is called a *layer*
  - Every layer describes a configuration/composition dimension

Template $T<T_1<T_2<T_3>, T_4<T_5>>$

- All $T_i$ can be exchanged independent of each other, i.e., configured! (static composition)
GenVoca

Applications
- Parameterizing implementations of data structures
- Synchronization code layers

Interesting parameterization concept
- Not that restricted as C++ templates: nested templates are a simpler form of GenVoca
- Maps to context-free grammars. A single configuration is a word in a context-free language
- Many tools around the technique

However: parameterization is the only composition operator, there is no full composition language

more in “Design Patterns and Frameworks”
42.5 Evaluating BETA Fragments, TMP, GenVoca as Composition Systems

**Component model**
- Source *and* binary components
- Generic components
- *Composition interfaces* with declared slots

**Composition technique**
- Composition operators:
  - bind (parameterize)
  - include
  - nest: nest a template into a slot

**Composition language**
- Simple combination of the composition operators
- Slot markup languages
- Semantic macros
The End

- Do not use string template engines, they render development error-prone
- Use slot markup languages and semantic macros to exploit their typing
- Look out for languages with full genericity