11) Generic Programming with Generic Components

Prof. Dr. Uwe Aßmann
Florian Heidenreich
Technische Universität Dresden
Institut für Software- und Multime diatechnik
http://st.inf.tu-dresden.de
Version 10-0.2, June 22, 2010

1. Full Genericity in BETA
2. Semantic Macros
3. Template Metaprogramming
4. Evaluation
Obligatory Reading

- Invasive Software Composition, Chapter 6
Literature

- BETA home page http://www.daimi.au.dk/~beta/
- Ole Lehrmann Madsen. The Mjölner 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/
11.1 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

- **Fully generic language**: A language, in which all language constructs can be generic
  - Then, the language need to have a *metamodel*, by which the parameters are typed
Composition Technique: Bind Operator (Parameterization)

Composition Level

Composition Language
- Component Model of Composition Language
- Composition Technique for Composition Language
- Composition Language for Composition Language

Composition System
- Component Model
- Composition Technique <<bind>>
- Composition Language

Prof. U. Aßmann, CBSE
Binding Templates As Sequence of Compositions

- Template with parameters
  - Bind
    - Value
    - Value

Value

Bind

Template with parameters
**BETA Fragment Metaprogramming System**

- BETA is a modern object-oriented language, developed in the North
  - BETA definition [BETA]
  - BETA programming environment Mjölner 1994 [BETA-ENV]

- Features
  - Single inheritance
  - Classes and methods are unified to *patterns*
  - Fully generic language
  - Environment is controlled by BETA grammar
    - Extension of the grammar changes all tools
  - 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 Fragments

- A fragment is a sequence of terminals, derived from a nonterminal in a grammar

Example:

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

Then, the following ones are fragments:

- Uwe Assmann Rudolfstrasse 31 Frankfurt Germany
- 34

But a complete sentence is:

- Uwe Assmann Rudolfstrasse 31 Frankfurt Germany 34

A fragment can be given a name

- MyAddress: Uwe Assmann Rudolfstrasse 31 Frankfurt Germany
Generic Fragments

- A **generic fragment** (*fragment form, sentential form*) is a sequence of terminals and nonterminals, derived from a nonterminal in a grammar

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

- In BETA, the “left-in” nonterminals are called *slots*
Slots (Declared Hooks) are declared by the component writer as code 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 '>>'`

- **Markup Tags** in XML:
  - `<superclasshook> X </superclasshook>`

- **Standardized Names (Hungarian Notation)**
  - `class Set extends genericXSuperClass {  }`

- **Comment Tags**
  - `class Set /* @superClass */`

- **Meta-Data Attributes**
  - Java: `@superclass(X)`
  - C#: `[superclass(X)]`
Defining Generic Types with XML Markup

```java
<< ClassTemplate >>
GenericSimpleList

class SimpleList {
    <slot name="T" type="Type"/>
    SimpleList next;
    <slot name="T" type="Type"/>
    getNext() {
        return next.elem;
    }
}

<< Class >>
SimpleList

class SimpleList {
    WorkPiece elem;
    SimpleList next;
    WorkPiece getNext() {
        return next.elem;
    }
}
```
Generic Modifiers in XML Markup Syntax

Component methodComponent = cs.createGenericComponent();
Hook modif = methodComponent.findSlot("M");
if (parallelVersion) {
    modif.bind("synchronized");
} else {
    modif.bind(" ");
}

.slot(name="M", type="Modifier")

public print() {
    System.out.println("Hello World");
}

synchronized public print () {
    System.out.println("Hello World");
}

public print () {
    System.out.println("Hello World");
}
Generic Statements in BETA Syntax

Component methodComponent = cs.createGenericComponent();
Hook statement = methodComponent.findSlot("MY");
if (StdoutVersion) {
    statement.bind("System.out.println("Hello World");");
} else {
    statement.bind("FileWriter.println("no way");");
}

public print() {
    <<MY:Statement>>;
}

public print () {
    System.out.println("Hello World");
}

public print () {
    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:Funktion
```

```java
standardLoopIterators : {
    Upwards: for (int i = 0; i < array.<len:Function>>; i++)
    Downwards: for (int i = array.<len:Function>>-1; i >= 0; i--)
}
```
**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.size(); i++)
    Downwards: for (int i = array.size()-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--)
  }
Inclusion of Fragments into Fragment Groups

- Fragments can be inserted into others by `include`.
- Given the above fragments and a new one:

  ```
  whileloopbody : WHILE <<statements:statementList>> END;
  ```

- A while loop can be defined as follows:

  ```
  whileloop:
    include LoopIterators.Upwards
    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)
11.2 Semantic Macros
Semantic Macros (Hygenic Macros)

- Macros usually are string-replacement functions (lambdas)
- Macro arguments can be typed by nonterminals (as in BETA; builds on the typed lambda calculus)

```plaintext
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);
}
i:int = eval(incr(2));
// result: i == 3;
k:int = eval(sqr(10));
// result k == 100;
```
11.3 Template Metaprogramming and Layered Template Metaprogramming
Template Metaprogramming

► Template Metaprogramming [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
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< T1< T2< T3>, T4<T5> >
```

all Ti can be exchanged independent of each other, i.e., configured! (static composition)
Embodiment View

- GenVoca components are parameterizable in layers. A layer has a nesting depth
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”
11.4 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

Prof. U. Aßmann, CBSE
The End