TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Software and Multimedia Technology, Software Technology Group

SS2018 – Model-driven Software Development in Technical Spaces

Template Meta Programming with Beta

Professor: Prof. Dr. Uwe Aßmann
Tutor: Dr.-Ing. Thomas Kühn

Task 1 Template Meta Programming with Beta

BETA[1] is an experimental language from Mjølner Informatics in Denmark that runs on the `Mjølner` runtime environment [1]. In BETA any component is a fragment that can be composed with other fragments. The execution of a BETA program revolvs around instantiating, matching, and composing patterns. To better understand BETA, read Chapters 3–5 of *"Object-Oriented Programming in the BETA Programming Language"* [1]. Moreover, BETA comes with a powerful fragment language that allows arbitrary parts of a BETA program to be split out into separate files. The connection between such fragments is maintained through slots marking up certain non-terminals and declaring that those need to be filled with concrete code before execution of the program (cf. [1, Chapter 17 and 20]).

a) What is the basic construct of BETA?

Solution: BETA's basic construct is a pattern. It consists of an optional list of attribute declarations and behavior.

```
(#
  (* Attributes *)
  enter ...
  do ...
  exit ...
#)
```

b) How are objects and operations represented using this construct?

Solution: The behavior of Patterns have an optional `enter`-clause, and optional `do`-part, and an optional `exit`-clause. Moreover, inheritance is realized with sub-patterns and instantiated with `@`. An example is depicted in Lst. 1.

c) How does fragment composition work in BETA?

Solution: Any pattern can introduce slots to patterns that are typed holes in its definition. These slots can be filled later with patterns of the correct type.

[1] http://www.daimi.au.dk/~beta

Listing 1: Example of object-orientation in BETA.

```
Person: Record (#
  Name: @text; Sex: @SexType;
  Display::< (# do {Display Name,Sex} ; INNER #)
#);
Employee: Person (#
  Salary: @integer; Position: @PositionType;
  Display::< (#do {Display Salary,Position}; INNER #)
  Work: ...
#);
employee1: @Employee;
8->&employee1.Work;
```

Listing 2: Example of fragments in BETA.

```
Up: DoPart (# do n+7->n #)
Down: DoPart (# do n-5->n #)
Counter: (#
  Up: (# n: @integer enter n <<SLOT Up:DoPart>> #);
  Down: (# n: @integer <<SLOT Down:DoPart>> exit n #)
#)
```

d) Describe BETA's *component model*, *composition technique* and *composition language*.

Solution:

- **Component model**: everything is a pattern with optional slots that can be filled by other patterns.
- **Composition technique**: BETA's composition operations involve an implicit bind operation (fragment referencing by slots) and an inclusion operation (concatenation of fragments).
- **Composition language**: There is a simple composition language based on fragment groups and the composition operations.

Task 2 Programming with BETA

In 1972, David Parnas proposed the following problem [2]:

> *The* Key Word In Context *(KWIC) index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.*

Within this exercise, you will be tasked to implement the main modules of the KWIC components within the **BETA** language, i.e., input, cycle, and output, and use the `Mjølner` runtime environment to run the KWIC application.

a) Download and install the `Mjølner` runtime environment.

b) Work through some of the tutorials. In particular, *MIA 99-40* and *MIA 94-24* (to get a feeling for OO development in BETA and for using the `Mjølner` environment).

c) Design the KWIC index system by implementing each *component* in the `do`-part of a separate pattern. Use `attribute`-definitions and `enter`- and `exit`-statements and assignments to represent the application's architecture.

Solution: A possible solution is shown in Listing 3.

d) Refactor your KWIC implementation, such that the main program fragment should only specify the architecture of your application, while the implementations should be split up into individual fragments' `do`-part.

Solution: A possible solution is shown in Listing 4.

Hand in your solution as archive `*.zip` before the next exercise.

Additional Information

- `Mjølner` runtime environment[2]
- **BETA** language reference manual[3]
- Tutorial on the use of `Mjølner` (MIA 99-40)[4]
- Tutorial on fragment composition (MIA 94-24)[5]

[2] `http://daimi.au.dk/~beta/mjolner_system`
[3] `http://www.cs.au.dk/~beta/Manuals/latest/beta-intro/index.html`
[4] `http://www.cs.au.dk/~beta/Manuals/latest/mjolner-tut/index.html`
[5] `http://www.cs.au.dk/~beta/Manuals/latest/tutorial/index.html`

References

[1] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented programming in the BETA programming language*. Addison-Wesley, 1993. URL http://www.daimi.au.dk/~beta/Books/index.html#betabook.download. Reprinted by Mjølner Informatics with permission from Addison-Wesley.

[2] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972. ISSN 0001-0782. doi: 10.1145/361598.361623. URL http://doi.acm.org/10.1145/361598.361623.

Listing 3: Possible solution for task 3c in BETA.

```
ORIGIN '~beta/basiclib/betaenv';
-- program: Descriptor --
(#
   inp: @
     (# sentence: ^Text;
     do 'Please enter a sentence: '->puttext; getline->sentence[];

     exit sentence
     #);

   cycle: @
     (#
        sentence: @Text;
        cycleList: [1] @Text;
        cycleEndList: [1] @Text;
        currentAtom: ^Text;

     enter sentence
     do
        0->sentence.pos;
        produceCycle:
          (#
          do
             sentence.getAtom->currentAtom[];
             (if currentAtom.length
              // 0 then leave produceCycle
              else
             (* First remember everything that came so far. This must be
                 appended
                to the current cycle result later on. *)
              cycleList[1]->cycleEndList[cycleEndList.range];
              (* Now append the current atom to all elements of the
                  cycle list.
                 This will also create the correct new entry. *)
              (for i: cycleList.range repeat
                (if cycleList[i].length > 0 then
                    ' '->cycleList[i].append;
                if);
                currentAtom[]->cycleList[i].append;

              for);
              (* Finally, extend lists. *)
              1->cycleEndList.extend;
              1->cycleList.extend;
              restart produceCycle
             if)
          #);
        (for i: cycleList.range repeat
          ' '->cycleList[i].append;
          cycleEndList[i][]->cycleList[i].append
        for)

     exit cycleList
     #);

   out: @
     (# textList: [1] @Text;
     enter textList

     do (for i: textList.range repeat textList[i][]->putText; newline;
         for)
     #);
do inp->cycle->out;
#)
```

Listing 4: Possible solution for task 3d in BETA.

```
ORIGIN '~beta/basiclib/betaenv';
-- DoInput: DoPart --
do 'Please enter a sentence: '->puttext; getline->sentence[];

-- DoCycle: DoPart --
do
     (# cycleEndList: [1] @Text; currentAtom: ^Text;
     do
        0->sentence.pos;
        produceCycle:
          (#
          do
             sentence.getAtom->currentAtom[];
             (if currentAtom.length
              // 0 then leave produceCycle
              else
             (* First remember everything that came so far. This must be
                 appended
                to the current cycle result later on. *)
              cycleList[1]->cycleEndList[cycleEndList.range];
              (* Now append the current atom to all elements of the
                  cycle list.
                 This will also create the correct new entry. *)
              (for i: cycleList.range repeat
                (if cycleList[i].length > 0 then
                    ' '->cycleList[i].append;
                if);
                currentAtom[]->cycleList[i].append;

              for);
              (* Finally, extend lists. *)
              1->cycleEndList.extend;
              1->cycleList.extend;
              restart produceCycle
             if)
          #);
        (for i: cycleList.range repeat
          ' '->cycleList[i].append;
           cycleEndList[i][]->cycleList[i].append
        for)
     #)

-- DoOutput: DoPart --
do (for i: cycleList.range repeat cycleList[i][]->puttext; newline for)

-- program: Descriptor --
(#
   input: @ (# sentence: ^Text;  <<SLOT DoInput:DoPart>> exit sentence
       #);

   cycle: @
     (# sentence: @Text; cycleList: [1] @Text;
     enter sentence
     <<SLOT DoCycle:DoPart>>
     exit cycleList
     #);

   output: @
     (# cycleList: [1] @Text;
     enter cycleList
     <<SLOT DoOutput:DoPart>>
     #);
do input->cycle->output;
#)
```