Comprehensions in Mozart: Syntax

This paper aims at explaining the complete syntax of comprehensions in Mozart2. There are two kinds of comprehensions implemented for the moment: list and record comprehensions. There are two main differences between the two later. First, list comprehensions return list-s while record comprehensions return record-s. Second, there are more functionalities available with list comprehensions.

1 List comprehensions

A list comprehension allows the creation of lists in an easy and flexible way. In this section, we explain their complete syntax incrementally.

A example of a very simple list comprehension is:

```
[A+1 for A in 1..5] %= [2 3 4 5 6]
```

Let us analyze this comprehension. The first thing that interests us is the `for A in 1..5`. This means that the list comprehensions creates a new variable named `A` and that this variable will take values from 1 to 5. The output is specified by the first part *in extenso* `A+1`. It means that we just add the current value of `A` incremented by 1 to the output list. This expression can be anything.

We will start from this simple syntax and, step by step, add functionalities. All functionalities are compatible with each other, even if examples do not cover all combinations of functionalities.

1.1 Generator, ranger and layer

A *generator* is the specification of a structure on which the comprehension iterates. In our example, `1..5` is a generator, it iterates from 1 to 5.
The *ranger* is the structure that is assigned to the current value of its corresponding generator at each iteration. In our example, `A` is the ranger of the generator we just saw.
A *layer* is the aggregation of a ranger and its generator.

There are five kinds of generators. Here are the descriptions of the first four - examples follow:

Integer generator: `in Low..High ; Step`
Goes from integers `Low` to `High` by integer step of `Step`. The step is optional, default is 1. `Low` must be smaller or equal to `High`.

C-style generator: `in ( Init ; Condition ; Next )`
Initiate its ranger to `Init`, iterates while `Condition` evaluates to true and updates the ranger to `Next` at the end of each iteration. The parenthesis are optional. The condition is optional, default is `true` - this implies that the generator never stops iterating by itself.

List generator: `in List`
`List` can be a variable containing a list or the definition of new list. The generator iterates over all the elements of the list in the same order as they appear in the list. The list can be a stream.

Function generator: `from Function`
`Function` can be a variable containing a function or the definition of a unnamed function. In both cases, the function does not take any arguments. The generator never stops. At each iteration, the current value of the generator is the result of calling `Function`.

```
% Integer generator
 [A for A in 1..5 ; 2] % [1 3 5]
 [A for A in 1..5 ; 1] % [1 2 3 4 5]
 [A for A in 1..5]     % [1 2 3 4 5]
% C-style generator
 [A for A in 1 ; A<6 ; A+1]       % [1 2 3 4 5]
 [A for A in ( 1 ; A<6 ; A+1 )] % [1 2 3 4 5]
 [A for A in 1 ; A+1]             % [1 2 3 4 5 ...] (never stops)
% List generator
L = [1 2 3]
 [A for A in L]           % [1 2 3]
 [A for A in [1 2 3]]     % [1 2 3]
 [A for A in [1 [2] 3]] % [1 [2] 3]
% Function generator
Fun = fun{$} 1 end
 [A for A from Fun]              % [1 1 1 1 ... ] (never stops)
 [A for A from fun{$} 1 end] % [1 1 1 1 ... ] (never stops)
```

The fifth kind of generator, record generators, is a bit more complicated. It is comparable to list generators but record generators are more general. A record generator has the following form:

```
F:V in Record of Function
```

As for lists, `Record` can be a variable containing a record or the definition of a new record. If it is a variable, then we have no way to know in advance what kind of generator it is. That is why we need a way to differentiate them. The ranger contains a feature `F` and a value `V`. This is how we differentiate record from list generators. Note that this is the reason why the function generator uses the `from` keyword and not `in`.

Iterating over a record with a feature and a value means that at every iteration, the feature is assigned to the current feature and the value is assigned to the current value. Such rangers allow having all the field information available.

The iteration goes through all fields of the record in the same order as its arity. When the record contains records, it forms a tree as in figure 1. In that case, only the leaves of the tree are considered in the iterations. The tree is traversed using the depth-first mode.

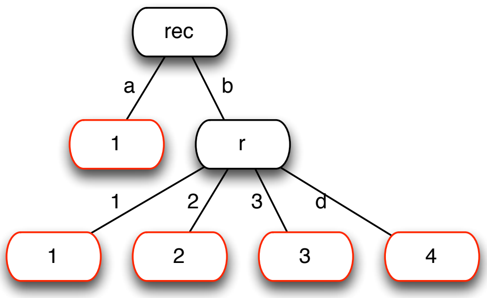

Figure 1: *Representation of the tree formed by the record rec(a:1 b:r(1 2 3 d:4)).*

We did not explained what is `Function` yet. It is a function taking two arguments that returns a boolean. It is called by the list comprehension every time it encounters a nested record inside the given record. The arguments are respectively the feature and the value of the field containing the nested record. When the result is true, the nested record is considered as such and we traverse it in depth-first mode. On the other hand, when the result is false then the nested record is treated as a leaf.

Here are some examples of list comprehensions with record generators:

```
R = r(a:1 b:r(1 2))
Fun = fun{$ F V} F \= b end
[A for _:A in 1#2#3]                                   % [1 2 3]
[A for _:A in R]                                       % [1 1 2]
[F for F:_ in R]                                       % [a 1 2]
[F#A for F:A in R]                                     % [a#1 1#1 2#2]
[F#A for F:A in r(a:1 b:r(1 2)) of fun{$ F V} F \= b end] % [a#1 b#r(1 2)]
[F#A for F:A in R of Fun]                              % [a#1 b#r(1 2)]
```

1.2 Multi layer

Now that all kinds of generators have been specified, we can move on to specifying several layers. One specifies several layers as follows:

```
[... for Layer_1 ... Layer_N]
% Example
[[A B] for A in 1..3 B in [a b]] % [[1 a] [2 b]]
```

Layers are traversed simultaneously. It means that a layer stops iterating when its iteration has reached its end or once at least one of its neighbor layers stops iterating. So the *smallest* generator - with the smallest number of iterations - decides when to stop iterating. In our example above, `B` is the range of the smallest generator and will take the values `a` then `b`. `A` will follow B and take the values 1 then 2, never 3.

Thanks to this functionality, function generators can be stopped as well as C-style generators without a condition.

1.3 Multi level

A *level* is the aggregation of one or more layers traversed simultaneously - *in extenso* inside the same `for` - together with an optional condition. A list comprehension contains at least one level.

Levels are like nested loops. It means that for each element of the main loop, the nested loop will run a complete iteration. Here is an example with two levels:

```
L = [4 7]
[[A 2*B] for A in L for B in 3..5] % [[4 6] [4 8] [7 6] [7 8]]
```

In addition to all its layers, a level can have a condition. The latter is always at the end of the level and is delimited by the keyword `if`. The condition is a boolean expression. When the condition evaluates to true, the iteration is accepted and the next level is called if it exists, otherwise an element is added to the output list. When the condition evaluates to false, the current iteration of the current level is skipped. Here are some examples:

```
L = [1 2 4 7]
[[A 2*B] for A in L if A>6 for B in 3..5] % [[7 6] [7 8] [7 10]]
[[A 2*B] for A in L for B in 3..5 if A>B] % [[4 6] [7 6] [7 8] [7 10]]
[[A 2*B] for A in L for B in 3..5 if A>B andthen B<5] % [[4 6] [7 6] [7 8]]
```

1.4 Laziness

A level can be specified as lazy. It is done by specified a lazy flag as any other layer - the flag is considered as a layer:

```
declare L in
thread L = [A for lazy A in 1..5] end
% L = _
{List.drop L 1 _}
% L = 1|_
{List.drop L 3 _}
% L = 1|2|3|_
```

As levels are specified as lazy, one must choose wisely which level-s must be lazy as in the following examples:

```
declare L in
thread L = [A#B for lazy A in 1..5 for B in [a b]] end
% L = _
{List.drop L 1 _}
% L = 1#a|1#b|_
{List.drop L 2 _}
% L = 1#a|1#b|_
{List.drop L 3 _ }
% L = 1#a|1#b|2#a|2#b|_
```

As the first level is declared as lazy, this level waits for an element to be needed to launch the iteration over its next element. When a value is needed, the first level allows one iteration to execute. This means that the second level can traverse all of its iterations because it is not lazy. That is why in the above example, elements are created by two - the number of iterations of the second level.

1.5 Bounded buffers

When one uses a list generator, a new functionality appears. It is called the bounded buffer[1]. It specifies the length of the bounded buffer to keep for the corresponding list. The syntax is the list followed by the size of the buffer separated by a colon. Here is a complete example:

```
declare Xs Ys Gen in
fun lazy {Gen I N}
   if I =< N then I|{Gen I+1 N}
   else nil
   end
end
thread Xs = {Gen 1 10} end
thread Ys = [A for lazy A in Xs:3] end
% Xs = 1|2|3|_          Ys = _
{List.drop Ys 1 _}
% Xs = 1|2|3|4|_        Ys = 1|_
{List.drop Ys 2 _}
% Xs = 1|2|3|4|5|_      Ys = 1|2|_
```

[1]For more information on why using bounded buffers, please consult Van Roy, P., & Haridi, S., *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004. We can also consult https://github.com/francoisfonteyn/thesis_public/blob/master/Thesis.pdf

A representation of the bounded buffer in the example above is in figure 2.

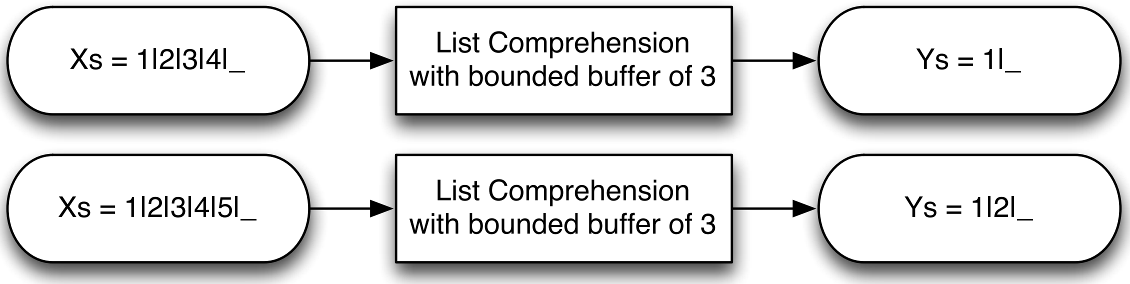
Xs = 1|2|3|4|_
List Comprehension with bounded buffer of 3
Ys = 1|_
Xs = 1|2|3|4|5|_
List Comprehension with bounded buffer of 3
Ys = 1|2|_

Figure 2: *Representation of a bounded buffer.*

The existence of bounded buffers is one of the reasons to differentiate list and record generators - recall that list are special records.

1.6 Multi output

Up to now, we limited ourselves to one output list. List comprehensions allow the result to have more than one output lists. All these output lists are put inside a record. One just specifies one expression by output list as follows:

```
[a b for _ in 1..2] % [a a]#[b b]
```

When features are not specified, they are implicitly created to be the smallest unused integers from one. The label is always `'#'` so that the record can also be equivalent to a tuple declared using its syntactic sugar.

To specify the feature, one just needs to use a colon to separate the feature from the expression. Here are two examples:

```
[a:a b:b for _ in 1..2] % '#'(a:[a a] b:[b b])
[a 1:b for _ in 1..2]   % [b b]#[a a]
```

In addition, each output can have a condition called the *output condition*. This condition allows filtering the elements of the corresponding list. Thanks to this functionality, output lists can have different sizes. Here is an example:

```
[smallerEquals:A if A =< 3 bigger:A if A > 3 for A in [3 4 2 8 9 5 7 1 6]]
% '#'(smallerEquals:[3 2 1] bigger:[4 8 9 5 7 6])
```

The output is a list only when there is one output expression without a feature, otherwise, the result is a record of list-s. So the two following examples are different:

```
[1:A if A < 3 for A in 1..5] % '#'(1:[1 2])
[A   if A < 3 for A in 1..5] % [1 2]
```

1.7 Body

As a list comprehensions has an arbitrary number of levels, it is comparable to nested loops. This comparison has driven us to allow list comprehensions to have a body. The *body* of a list comprehensions is a statement that will be executed - just - before each time the comprehensions adds an element to each output - or when the comprehension tries appending, even if output conditions are false. In the comparison with nested loops, the body is the statement inside the *deepest* loop of the nesting. Here is an example:

```
[A for A in 1..5 do {Delay 1000}] % [1 2 3 4 5] after 5 seconds
```

1.8 Collectors

The last functionality of list comprehensions is collectors. As in for loops, a *collector* is a procedure that appends its only argument to the end of its corresponding list. So when specified, a collector is assigned to a procedure that appends an element that a list. When one wants to use a collector, he must specify it as an output with a mandatory feature, the flag `collect` and the variable that will be assigned to the procedure, all separated by colons. Here are some examples:

```
[c:collect:C for A in 1..5 do {C A}] % '#'(c:[1 2 3 4 5])
[c:collect:C for A in 1..5 do {C A}{C A+1}] % '#'(c:[1 2 2 3 3 4 4 5 5 6])
[1:collect:C1 2:collect:C2 for A in 1..5 do {C1 A-1}{C2 A+1}]
% [0 1 2 3 4]#[2 3 4 5 6]
```

Collectors can also be passed as arguments of external procedures. Note that the lists corresponding to collectors are closed at the end of the list comprehension.

2 Record comprehensions

The principle is basically the same as for list comprehensions except that instead of returning a list - or several ones - record comprehensions return a record - or several ones. The idea is to keep the same shape, the same arity as the input record. For this reason, record comprehensions only take one record as input. In other words, we could say that record comprehensions are restricted to one layer and one level. On the other hand we still keep the multi output but without individual conditions, only the level condition can be specified. In list comprehensions, one can specify a boolean function with two arguments to discriminate leaves when traversing a record. With record comprehensions, the same thing is possible but we decided to put the condition directly instead of putting in a function. This is because we think it is more unified with the - unique - level condition not to use a function.

Because record comprehensions output a record - or records - similar to the input, we decided not to allow several levels. If one wants to use several layers then it would imply that all input records have similar - nested - arities. For these reasons, we decided to restrict record comprehensions to one level and one layer. Collectors are specific to lists so they are not implemented in record comprehensions.

As record comprehensions use a syntax very similar to the one of list comprehensions, we need to differentiate them. This is done by using parenthesis instead of square brackets to delimit the comprehension.

Collectors are specific to list comprehensions but bodies are not.

Here are some examples:

```
(A    for A   in r(a:2 b:3 1))                     % r(1:1 a:2 b:3)
(A+1  for A   in r(a:2 b:3 1))                     % r(1:2 a:3 b:4)
(@C   for A   in r(a:2 b:3 1) do C := A)           % r(1:2 a:3 b:4)
(F    for F:A in r(a b c))                         % r(1 2 3)
(F+1  for F:A in r(a b c))                         % r(2 3 4)
(F#A  for F:A in r(a:1 b:r(1 2)))                  % r(a:a#1 b:r(1#1 2#2))
(F#A  for F:A in r(a:1 b:r(1 2)) of F == a)        % r(a:a#1 b:b#r(1 2))
(F#A  for F:A in r(a:1 b:r(1 2)) if {IsRecord A} orelse A>1)
    % r(b:r(2:2#2))
(F#A  for F:A in r(a:1 b:r(1 2)) if {IsRecord A} orelse A>1 of F == a)
    % r(b:b#r(1 2))
```

A more complex example where we handle a binary tree follows:

```
declare Rec = obtree(key:1 left:leaf right:obtree(key:2 left:leaf right:
    leaf))
{Browse (if F==key then N+1 else N end for F:N in Rec of F==left orelse F==
    right)}
% browses
obtree(key:2 left:leaf right:obtree(key:3 left:leaf right:leaf))
```