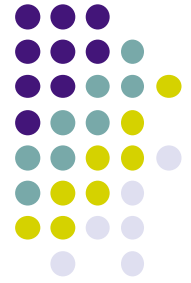
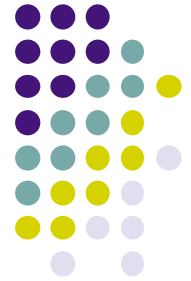


A **for** loop abstraction that collects results



- We show how to use **state (a cell) and higher-order programming together** to build a powerful new abstraction for deterministic dataflow
 - The imperative and functional paradigms are not antagonistic! Using cells can give extra power to dataflow programs.
- Our new abstraction will generalize the declarative **for** loop of Oz to collect results
 - It is a powerful form of **list comprehension**

Declarative for loop



- Oz has a declarative **for** loop

```
for I in [1 2 3] do {Browse I*I} end
```

- This is **exactly the same** as executing the following three statements one after the other:

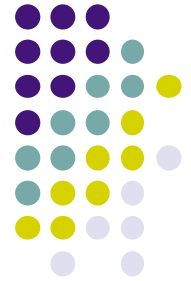
```
local I=1 in {Browse I*I} end
```

```
local I=2 in {Browse I*I} end
```

```
local I=3 in {Browse I*I} end
```

- Each iteration is independent; the identifier I references one element of the list in each iteration

Collecting results in the **for** loop

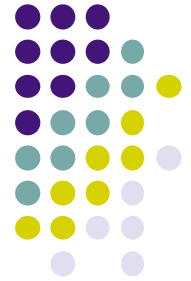


- We would like to extend the declarative **for** loop to accumulate results

R = **for** I **in** [1 2 3] **do** (*accumulate I*I*) **end**

- We would like this to return R=[1 4 9]
- The existing **for** loop cannot do this, but we will define a new abstraction that can

The ForCollect abstraction



- The ForCollect abstraction extends the **for** loop with the ability to accumulate results:

$R = \{\text{ForCollect } [1 \ 2 \ 3] \text{ **proc** } \{\$ \ C \ I\} \text{ <stmt> **end**}\}$

- The loop body is <stmt>
 - I is the loop index
 - C is the « collect procedure »: calling {C X} in the loop body will accumulate X in R

$R = \{\text{ForCollect } [1 \ 2 \ 3] \text{ **proc** } \{\$ \ C \ I\} \{C \ I * I\} \text{ **end**}\}$

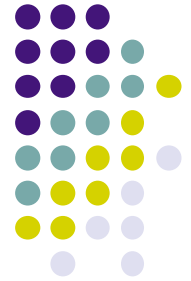
$\Rightarrow R = [1 \ 4 \ 9]$

Defining the collect procedure (1)



- How can we define the collect procedure C?
 - C cannot be written in the functional paradigm because it has **memory**: each time we call {C X} we need to append X to the output list. Each time we call C the output changes.
- C can only be defined using **state**, i.e., a cell
 - The cell is used to **append X to the output list**
- But seen from the outside, ForCollect will still be functional!
 - Let us see how to define the collect procedure...

Defining the collect procedure (2)



- Assume we are building the output list and we have already added three elements to it:

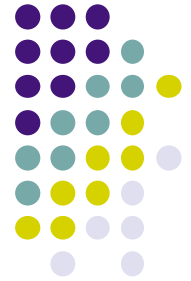
$R = 1|4|9|R1$

- To add another element, we need to bind R1:

$R1=16|R2$

- This makes the new $R = 1|4|9|16|R2$
 - The new end of this list is R2!
 - So the cell always has to store **the end of the list**

Defining the collect procedure (3)



- We can define the collect procedure like this:

Acc={NewCell R} % Cell Acc contains end of the list

proc {C X}

R2 % New end of list

in

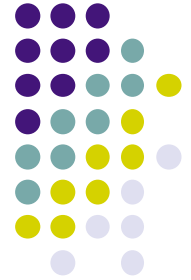
@Acc=X|R2 % Bind old end of list to X|R2

Acc:=R2 % Set C to new end of list R2

end

- This appends X to the output list

Definition of ForCollect



- This gives us the following definition of ForCollect:

```
proc {ForCollect Xs P Ys}  
  Acc={NewCell Ys}  
  proc {C X} R2 in @Acc=X|R2 Acc:=R2 end  
in  
  for X in Xs do {P C X} end  
  @Acc=nil  
end
```

Doing Acc:=nil would be
wrong! Do you see why?

- We need to write ForCollect as a **procedure**, even though we will call it as a function
 - It is because we need to access the output Ys (= initial content of Acc)

Concurrent agent with ForCollect



- We have defined ForCollect on lists, but it can do more!
 - ForCollect also works on streams
- Running ForCollect in a thread **makes a concurrent agent:**

```
Ys=thread {ForCollect Xs  
           proc {$ C X} if X mod 2 == 0 then {C X*X} end end}  
           end
```

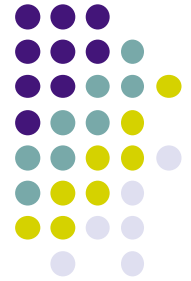
- This agent reads an input stream Xs and returns an output stream Ys that contains the squares of the even elements of Xs

Conclusions of ForCollect



- ForCollect is a powerful abstraction that combines and generalizes both Map and Filter
 - When used with lists, it is called a **list comprehension**
 - Some languages have syntax for this, e.g., Haskell and Python
 - In Oz, **list comprehensions can be concurrent agents**
- ForCollect is defined by **combining cells and higher-order programming**
 - There is no antagonism between the imperative and functional paradigms; they can be used together to the benefit of both
 - Even though ForCollect uses a cell internally, it is completely deterministic when viewed from the outside. This is because we use the cell in a single thread.

Alternative definition of ForCollect



- If the collect procedure C might be used **in more than one thread**, then we need to change its definition to use Exchange:

```
proc {ForCollect Xs P Ys}
  Acc={NewCell Ys}
  proc {C X} R2 in {Exchange Acc X|R2 R2} end
in
  for X in Xs do {P C X} end
  {Exchange Acc nil _}
end
```

- {Exchange Acc Old New} does **two operations atomically**:
 - Old is bound to the old content and New becomes the new content
 - This avoids errors when cells are used by multiple threads: doing @Acc and Acc:=R2 as two separate operations would permit another operation on Acc to be done in between, which is wrong!