Programming Language Concepts, cs2104 Lecture 08 (2003-10-03)

Seif Haridi
Department of Computer Science, NUS

haridi@comp.nus.edu.sg

Reading Suggestions

- Chapter 3
  - Sections 3.1 – 3.4 [careful]
    but skip or browse [3.4.4,3.4.7,3.4.8]
  - Sections 3.6, 3.7 [careful]
- And of course the handouts!

Organizational

- Assignment 3 deadline is extended (again) to October 13 (Monday)
- Consultation at October 11 (Saturday) from 11:00 – 13:00 at PC Lab2
- Check your marks for Assignment 1

Content

- Higher Order Programming
- Abstract Data Types

Higher-Order Programming

Remember

- Functions are procedures
  - Special Syntax, nested syntax, expression syntax
  - They are procedures with one argument as the result arguments
- ```
  fun {F X}
     fun {$ Y} X+Y end
  end
  ```
- A function that returns a function that is specialized on X
  ```

Remember II

- Successive transformations

1.
```
fun {F X}
   fun {$ Y} X+Y end
end
```
2.
```
proc {F X ?R}
   R = fun {$ Y} X+Y end
end
```
3.
```
proc {F X ?R}
   R = proc {$ Y ?R1} R1=X+Y end
end
```


Remember III

3.
```
proc {F X ?R}
   R = proc {$ Y ?R1} R1=X+Y end
end
```

- F is a procedure value
- When F is called its 2nd argument returns a procedure value
- '?' is comment indicating output argument

Remember IV

1. fun {F X}
   fun {$ Y} X+Y end
   end

- You should think directly in terms of functions
- F is a function of one argument
- When F is called it returns a function (Call it G), e.g. the call {F 1}
- This function G when called, {G Y}, it returns 1+Y

Remember IV

1. fun {F X}
   fun {$ Y} X+Y end
   end

- The type of F
- ⟨fun {F ⟨Num⟩}:
  ⟨fun {$ ⟨Num⟩}: ⟨Num⟩⟩
  ⟩

Higher-Order Programming

- Higher-order programming = the set of programming techniques that are possible with procedure values (lexically-scoped closures)
- higher-order programming is the foundation of secure data abstraction component-based programming and object-oriented programming

Higher-Order Programming

- Basic operations
  - Procedural abstraction: the ability to convert any statement into a procedure value
  - Genericity: the ability to pass procedure values as arguments to a procedure call
  - Instantiation: the ability to return procedure values as results from a procedure call
  - Embedding: the ability to put procedure values in data structures

Higher-order programming

- Control abstractions
  - The ability to define control constructs
  - Integer and list loops, accumulator loops, folding a list (left and right)

Genericity

- To make a function generic is to let any specific entity (i.e., any operation or value) in the function body become an argument of the function.
- The entity is abstracted out of the function body.

Genericity

- Replace specific entities (zero 0 and addition +) by function arguments

```
fun {SumList Ls}
    case Ls
    of nil then 0
    []  X|Lr then X+{SumList Lr}
    end
end
```


Genericity

```
fun {SumList L}
    case L
    of nil then 0
    []  X|L2 then X+{SumList L2}
    end
end
```

⇩

```
fun {FoldR L F S}
    case L
    of nil then S
    []  X|L2 then {F X  {FoldR L2 F S}}
    end
end
```

Genericity SumList

```
fun {SumList Ls}
   {FoldR L
    fun {$ X Y} X+Y end 0}
end
```


Genericity ProductList

```
fun {ProductList Ls}
   {FoldR L
    fun {$ X Y} X*Y end 1}
end
```

Genericity Some

```
fun {Some Ls}
   {FoldR L
    fun {$ X Y} X orelse Y end
    false}
end
```


List Mapping

- Mapping
  - each element recursively
  - *calling function for each element*
  - Constructs an output list
- Separate function calling by passing a function as argument

Other generic functions: Map

```
fun {Map Xs F}
   case Xs
   of nil then nil
   [] X|Xr then
      {F X}|{Map Xr F}
   end
end
{Browse {Map [1 2 3]
            fun {$ X} X*X end}}
```


Other generic functions: Filter

```
fun {Filter Xs P}
   case Xs
   of nil then nil
   [] X|Xr andthen {P X} then
      X|{Filter Xr P}
   [] X|Xr then {Filter Xr P}
   end
End
{Browse {Filter [1 2 3] IsOdd}}
```

Instantiation

- Instantiation: the ability to return procedure values as results from a procedure call
- A factory of specialized functions

```
fun {Add X}
   fun {$ Y} X+Y end
end

Inc = {Add 1}
{Browse {Inc 5}} % shows 6
```


Instantiation: An application (protected values)

- Given a value (e.g. [1 2 3]) we would like to seal it for any other client except for a person that has a key
- A procedure that has the key can access the list
- No other procedures can access the value

Instantiation: An application (protected values)

[1 2 3] Wrap $\longrightarrow$ $\boxed{[1\ 2\ 3]}$

$\boxed{[1\ 2\ 3]}$ UnWrap $\longrightarrow$ [1 2 3]

Instantiation: An application (protected values)

New basic data type is names
A name is unforgeable atom:

- Cannot be displayed
- The only possible operation is comparison
- This a fundamental addition to the model
- No longer declarative

X = {NewName}
Y = {NewName}

- X and Y are different

Wrappers

- proc {NewWrapper Wrap Unwrap}
     Key={NewName}
  in
     fun {Wrap X} … end
     fun {Unwrap W} {W Key} end
  end

Wrappers

- proc {NewWrapper ?Wrap ?Unwrap}
     Key={NewName}
  in
     fun {Wrap X} … end
     fun {Unwrap W} {W Key} end
  end
- ? Indicate that Wrap and UnWrap is output arguments (just a comment)

Wrappers

- **declare W UW**
  {NewWrapper W UW}
  RL = {W [1 2 3]}
  {Browse RL} % cannot see
  {Browse {UW RL}} % shows [1 2 3]

- NewWrapper is a factory that creates two related procedures

Instantiation: An application (protected values)

{Wrap [1 2 3]} → [1 2 3]

{UnWrap [1 2 3] } → [1 2 3]

Wrappers

- ```
  proc {NewWrapper ?Wrap ?Unwrap}
        Key={NewName}
  in
    fun {Wrap X}
        fun{$ K} if K==Key then X end
    end
    fun {Unwrap W} {W Key} end
  end
  ```


Instantiation: An application (protected values)

[1 2 3] Wrap → $\boxed{\text{[1 2 3]}}$

$\boxed{\text{[1 2 3]}}$ UnWrap → [1 2 3]
  ```

Embedding

- Embedding is when procedure values are put in data structures
- Embedding has many uses:
  - Modules: a module is a record that groups together a set of related operations
  - Software components: a software component is a generic function that takes a set of modules as input (imported modules) and returns a new module. It can be seen as specifying a module in terms of the modules it needs.

Control Abstractions

```
proc {For I J P}
   if I > J then skip
   else {P I} {For I+1 J P}
   end
end

{For 1 10 Browse}

for I in 1..10 do {Browse I} end
```

Control Abstractions

```
proc {ForAll Xs P}
  case Xs
  of nil then skip
  [] X|Xr then
    {P X} {ForAll Xr P}
  end
End

{ForAll [a b c d]
  proc{$ I}
     {System.showInfo "the item is: " # I}
  end}
```


Control Abstractions

```
proc {ForAll Xs P}
  case Xs
  of nil then skip
  [] X|Xr then
    {P X} {ForAll Xr P}
  end
end

for I in [a b c d] do
  {System.showInfo "the item is: " # I}
end
```

Left-Folding

- Two values define "folding"
  - initial value S
  - binary function F

- Left-folding `{FoldL [`$x_1 \ldots x_n$`] F S}`

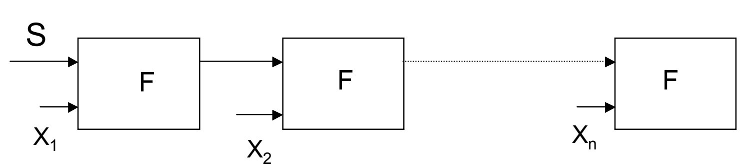
  `{F … {F {F S `x_1`} `x_2`} … `x_n`}`

  or

  `(…((S `$\otimes_F$ x_1`) `$\otimes_F$ x_2`) … `$\otimes_F$ x_n`)`

Left-Folding

- Left-folding `{FoldL [`$x_1 \ldots x_n$`] F S}`



Left-Folding

- Two values define “folding”
  - initial value S
  - binary function F
- Left-folding `{FoldL`

left is here!

$$\{F \ldots \{F\ \{F\ S\ x_1\}\ x_2\} \ldots x_n\}$$

or

$$(\ldots((S \otimes_F x_1) \otimes_F x_2) \ldots \otimes_F x_n)$$

FoldL

```
fun {FoldL Xs F S}
   case Xs
   of nil   then S
   [] X|Xr then {FoldL Xr F {F S X}}
   end
end
```

Properties of `FoldL`

- Tail recursive (iterative computation)
- First element of list if folded first…

Right-Folding

- Two values define “folding”
  - initial value S
  - binary function F
- Right-folding `{FoldR` [$x_1 \ldots x_n$] `F S}`

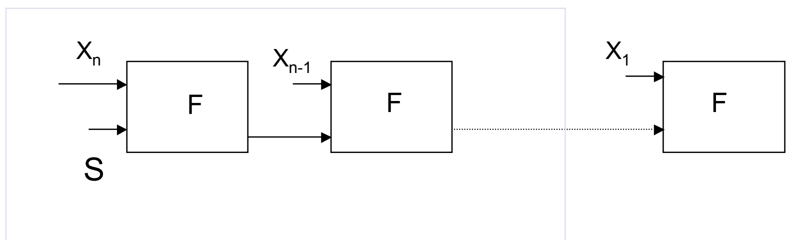
  `{F` x_1 `{F` x_2 … `{F` x_n `S}` …`}}`

  or

  $x_1 \otimes_{\mathtt{F}} (x_2 \otimes_{\mathtt{F}} (\ldots (x_n \otimes_{\mathtt{F}} \mathtt{S}) \ldots))$

Right-Folding

- Two values define "folding"
  - initial value S
  - binary function F



Right-Folding

- Two values define "folding"
  - initial value
  - binary function
- Right-folding `{FoldR [x` **right** is here!

  `{F` x_1 `{F` x_2 … `{F` x_n

  or

  $x_1 \otimes_{\mathtt{F}} (x_2 \otimes_{\mathtt{F}} (\ldots (x_n \otimes_{\mathtt{F}} \mathtt{S}) \ldots))$

FoldR

```
fun {FoldR Xs F S}
    case Xs
    of nil   then S
    [] X|Xr then {F X {FoldR Xr F S}}
    end
end
```


Properties of FoldR

- Not tail-recursive (recursive computation)
- Elements folded in order

FoldL or FoldR?

- FoldL and FoldR compute same value if:
  - {F S x_j} = {F x_i S}, for all i, ($1 \leq i \leq n$)
  - {F {F x_i x_j} x_k} = {F x_i {F x_j x_k}}
- If the condition holds use FoldL
  - FoldL tail-recursive
  - Not always true (see next example)
- Otherwise: choose FoldL or FoldR
  - depending on required order of result

Example: Appending Lists

- Given: list of lists

```
[[a b] [1 2] [e] [g]]
```

- Task: compute all elements in one list in order
- We can use Fold* with (* is either L or R)
  - Append as F
  - nil as S

Example: Appending Lists

- Given: list of lists

  `[[a b] [1 2] [e] [g]]`
- We can use Fold* with
  - Append as F
  - nil as S
- {Append nil L} = {Append L nil}
- {Append L1 {Append L2 L3}} = {Append {Append L1 L2} L3}

Example: Appending Lists

- Given: list of lists

  [[a b] [1 2] [e] [g]]
- Task: compute all elements in one list in order
- Solution:

```
fun {AppAll Xs}
   {FoldR Xs Append nil}
end
```

- Question: What would happen with FoldL?

Tuples and Records…

- Techniques for lists explored here of course applicable…
  - …see tutorial

Abstract Data Types

Data Types

- Data type
  - set of values
  - operations on these values
- Primitive data types
  - records
  - numbers
  - …
- Abstract data types
  - completely defined by its operations (interface)
  - implementation can be changed without changing use

Example: Lab Assignment 3

- Abstract data type for Huffman-trees
- Different implementations
- Same interface

Motivation

- Sufficient to understand interface only
- Software components can be developed independent of use
  - as long as only interface is used
- Developers need not to know implementation details

Outlook

- How to define abstract data types
- How to organize abstract data types
- How to use abstract data types

Abstract data types (ADTs)

- ADTs: it is possible to change the implementation of an ADT without changing its use
- The ADT is described by a set of procedures
  - Including how to create a value of the ADT
- These operations are the only thing that a user of the abstraction can assume

Example: stack

- Assume we want to define a new data type ⟨stack T⟩ whose elements are of any type T
- We define the following operations (with type definitions)

⟨fun {NewStack}: ⟨stack T⟩⟩
⟨fun {Push ⟨stack T⟩ ⟨T⟩ }: ⟨stack T⟩⟩
⟨proc {Pop ⟨stack T⟩ ?⟨T⟩ ?⟨stack T⟩}⟩
⟨fun {IsEmpty ⟨stack T⟩} : ⟨Bool⟩⟩

Example: stack (algebraic properties)

- Algebraic properties are logical relations between ADT's operations
- These operations normally satisfy certain laws (properties)
- {IsEmpty {NewStack}} = true
- For any stack S, {IsEmpty {Push S}} = false
- For any *E* and *T*, {Pop {Push *S E*} *E S*} holds
- For any stack S, {Pop {NewStack} S} raises error

Stack (implementation I) using lists

```
fun {NewStack} nil end
fun {Push S E} E|S end
proc
  {Pop E|S ?E1 ?S1} E1 = E S1 = S
end
fun {IsEmpty S} S==nil end
```

Stack (implementation II) using tuples

```
fun {NewStack} emptyStack end
fun {Push S E} stack(E S) end
proc {Pop stack(E S) E1 S1}
   E1 = E S1 = S
end
fun {IsEmpty S} S==emptyStack end
```


Example: Dictionaries

- Designing the interface
  `{MakeDict}`
  returns new dictionary
  `{DictMember D F}`
  tests whether feature `F` is member of dictionary `D`
  `{DictAccess D F}`
  return value of feature `F` in `D`
  `{DictAdjoin D F X}`
  return dictionary with value `X` at feature `F` adjoined
- Interface depends on purpose, could be richer (for example, `DictCondSelect`)

Using the `Dict` ADT

- Now we can write programs using the ADT without even having an implementation for it
- Implementation can be provided later
- Eases software development in large teams

Implementing the `Dict` ADT

- Now we can decide on a possible implementation for the `Dict` ADT
  - based on pairlists
  - based on records
- Regardless on what we decide, programs using the ADT will work!
  - the interface is a ***contract*** between use and implementation

Dict: Pairlists

```
fun {MakeDict}
  nil
end
fun {DictMember D F}
  case D
  of nil then false
  [] G#X|Dr then
    if G==F then true
    else {DictMember Dr F}
    end
  end
end
…
```


Dict: Records

```
fun {MakeDict}
  mt
end
fun {DictMember D F}
  {HasFeature D F}
end
fun {DictAccess D F}
  D.F
end
fun {DictAdjoin D F X}
  {AdjoinAt D F X}
end
```

Example Frequency Counting

```
local
  fun {Inc D X}
    if {DictMember D X} then
      {DictAdjoin D X {DictAccess D X}+1}
    else {DictAdjoin D X 1}
    end
  end
in
  fun {Cnt Xs}
    % returns dictionary
    {FoldL Xs Inc {MakeDict}}
  end
end
```


Example Frequency Counting

```
local
   fun {Inc D X}
      if {DictMember
         {DictAdjoin
      else {DictAdjoi
      end
   end
in
   fun {Cnt Xs}
      % returns dictionary
      {FoldL Xs Inc {MakeDict}}
   end
end
```

homework:
understand and try
this example!

Evolution of ADTs

- Important aspect of developing ADTs
  - start with simple (possibly inefficient) implementation
  - refine to better (more efficient) implementation
  - refine to carefully chosen implementation
    - hash table
    - search tree
- All of evolution is local to ADT
  - no change of programs using ADT is needed!

Have a Nice Weekend