Functional Programming in *Mathematica*

A4M36TPJ, WS 15/16, Week 7 - Lecture

Zdeněk Buk, bukz1@fel.cvut.cz

Dept. of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague

Last update: Nov 2015

Introduction

Functional Programming

From Wikipedia, the free encyclopedia: Functional programming is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and **avoids changing-state** and mutable data. It is a **declarative programming paradigm**, which means programming is done with expressions. In functional code, the **output value of a function depends only on the arguments** that are input to the function, so calling a function f twice with the same value for an argument *x* will produce the same result *f(x)* each time. **Eliminating side effects**, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and **predict the behavior of a program**, which is one of the key motivations for the development of functional programming.

- treats computation as the evaluation of mathematical functions
- avoids changing-state
- output value of a function depends only on the arguments
- eliminating side effects can make it much easier to predict the behavior of a program

Functional Programming Concepts

Higher-Order Functions

- functions that can either take other functions as arguments or return them as results

In[1]:=
```
f[g_, x_] := g[x]
```

In[2]:=
```
f[Sqrt, x]
```

Out[2]=
$$\sqrt{x}$$

In[3]:=
```
f[g_, x_] := Function[{y}, g[x + y]]
```

In[4]:=
```
f[Sqrt, 42]
```

Out[4]=
$$\text{Function}\left[\{y\$\},\ \sqrt{42 + y\$}\right]$$

In[5]:=
```
f[Sqrt, 42][z]
```

Out[5]=
$$\sqrt{42 + z}$$

Pure Functions

Purely functional functions have no side effects.

- If the result of a pure expression is not used, it can be removed without affecting other expressions.
- If there is no data dependency between two pure expressions, then their order can be reversed, or they can be performed in parallel.
- If a pure function is called with arguments that cause no side-effects, the result is constant with respect to that argument list.

In[6]:=
```
tutorial/PureFunctions
```

```
In[6]:= Function[{x}, x^2 + 1]
Out[6]= Function[{x}, x^2 + 1]

In[7]:= Function[{x}, x^2 + 1][z]
Out[7]= 1 + z^2

In[8]:= #^2 + 1 &[z]
Out[8]= 1 + z^2

In[9]:= Map[#^2 + 1 &, {1, 2, 3, 4, 5, 6}]
Out[9]= {2, 5, 10, 17, 26, 37}

In[10]:= ParallelMap[#^2 + 1 &, {1, 2, 3, 4, 5, 6}]
Out[10]= {2, 5, 10, 17, 26, 37}
```

Recursion

Iteration (looping) in functional languages is usually accomplished via recursion.

```
In[11]:= Clear[f];
         f[x_] := 1 /; x < 1
         f[x_] := x * f[x - 1]

In[14]:= f[1]
Out[14]= 1

In[15]:= f[4]
Out[15]= 24

In[16]:= f[10]
Out[16]= 3 628 800

In[17]:= Clear[g];
         Nest[g, x, 3]
Out[18]= g[g[g[x]]]

In[19]:= Clear[f, x, a, b, c, d, e]

In[20]:= Fold[f, x, {a, b, c, d}]
Out[20]= f[f[f[f[x, a], b], c], d]
```

Functions in *Mathematica*

Defining Functions

In[21]:=
```
Clear[f, g];
f[x_] := g[x] + 1
```

In[23]:=
```
f[10]
```

Out[23]=
```
1 + g[10]
```

In[24]:=
```
Clear[f];
f = Function[{x}, g[x] + 1]
```

Out[25]=
```
Function[{x}, g[x] + 1]
```

In[26]:=
```
f[10]
```

Out[26]=
```
1 + g[10]
```

In[27]:=
```
Function[{x}, g[x] + 1][10]
```

Out[27]=
```
1 + g[10]
```

In[28]:=
```
g[#] + 1 &[10]
```

Out[28]=
```
1 + g[10]
```

In[29]:=
```
g[#x] + 1 &[⟨|"x" → 10|⟩]
```

Out[29]=
```
1 + g[10]
```

Multiple Arguments

In[30]:=
```
Clear[f]
f[x_, y_] := x^2 + y + 1
```

In[32]:=
```
f[a, b]
```

Out[32]=
$$1 + a^2 + b$$

In[33]:=
```
Function[{x, y}, x^2 + y + 1][a, b]
```

Out[33]=
$$1 + a^2 + b$$

In[34]:=
```
#1^2 + #2 + 1 &[a, b]
```

Out[34]=
$$1 + a^2 + b$$

Calling Function

In[35]:=
```
ClearAll[f]
```

In[36]:=
```
f[x]
```

Out[36]=
```
f[x]
```

In[37]:=
```
f@x
```
Out[37]=
```
f[x]
```

In[38]:=
```
x // f
```
Out[38]=
```
f[x]
```

In[39]:=
```
f[x_] := g[x] + 1
```

In[40]:=
```
x // f
```
Out[40]=
```
1 + g[x]
```

In[41]:=
```
x // g[#] + 1 &
```
Out[41]=
```
1 + g[x]
```

In[42]:=
```
data = RandomInteger[{0, 10}, 10]
```
Out[42]=
```
{5, 6, 3, 8, 8, 2, 3, 7, 5, 9}
```

In[43]:=
```
{Min[data], Max[data], Mean[data]}
```
Out[43]=
$$\left\{2,\ 9,\ \frac{28}{5}\right\}$$

In[44]:=
```
ClearAll[f]
f[x_] := {Min[x], Max[x], Mean[x]}
```

In[46]:=
```
f[data]
```
Out[46]=
$$\left\{2,\ 9,\ \frac{28}{5}\right\}$$

In[47]:=
```
{Min[#], Max[#], Mean[#]} &@data
```
Out[47]=
$$\left\{2,\ 9,\ \frac{28}{5}\right\}$$

In[48]:=
```
data // {Min[#], Max[#], Mean[#]} &
```
Out[48]=
$$\left\{2,\ 9,\ \frac{28}{5}\right\}$$

In[49]:=
```
a + b
```
Out[49]=
```
a + b
```

In[50]:=
```
Plus[a, b]
```
Out[50]=
```
a + b
```

In[51]:=
```
a ~ Plus ~ b
```
Out[51]=
```
a + b
```

In[52]:=
```
{1, 2, 3} ~ Join ~ {4, 5, 6}
```
Out[52]=
```
{1, 2, 3, 4, 5, 6}
```

In[53]:=
```
a ~ (#1 ^ 2 + #2 ^ 3 &) ~ b
```
Out[53]=
$$a^2 + b^3$$

Mapping

In[54]:=
```
ClearAll[f];
f[x_] := g[x] + 1
```

In[56]:=
```
Range[5]
```

Out[56]=
```
{1, 2, 3, 4, 5}
```

In[57]:=
```
f[Range[5]]
```

Out[57]=
```
1 + g[{1, 2, 3, 4, 5}]
```

In[58]:=
```
Map[f, Range[5]]
```

Out[58]=
```
{1 + g[1], 1 + g[2], 1 + g[3], 1 + g[4], 1 + g[5]}
```

In[59]:=
```
f /@ Range[5]
```

Out[59]=
```
{1 + g[1], 1 + g[2], 1 + g[3], 1 + g[4], 1 + g[5]}
```

In[60]:=
```
Attributes[f]
```

Out[60]=
```
{}
```

In[61]:=
```
SetAttributes[f, Listable]
```

In[62]:=
```
f[Range[5]]
```

Out[62]=
```
{1 + g[1], 1 + g[2], 1 + g[3], 1 + g[4], 1 + g[5]}
```

In[63]:=
```
g[x] + 1 & /@ Range[5]
```

Out[63]=
```
{1 + g[x], 1 + g[x], 1 + g[x], 1 + g[x], 1 + g[x]}
```

Mathematica: Core Language

Expressions

```
In[64]:= FullForm[x + y]
Out[64]//FullForm= Plus[x, y]

In[65]:= Head[x + y]
Out[65]= Plus

In[66]:= FullForm[{x, y, z}]
Out[66]//FullForm= List[x, y, z]

In[67]:= Head[{x, y, z}]
Out[67]= List

In[68]:= Apply[head, {x, y, z}]
Out[68]= head[x, y, z]

In[69]:= Apply[Plus, {x, y, z}]
Out[69]= x + y + z

In[70]:= FullForm[Apply[Plus, {x, y, z}]]
Out[70]//FullForm= Plus[x, y, z]

In[71]:= Apply[Sequence, {x, y, z}]
Out[71]= Sequence[x, y, z]

In[72]:= ClearAll[f];
         f[x___] := g[x]

In[74]:= f[1, 2, 3, 4]
Out[74]= g[1, 2, 3, 4]

In[75]:= ClearAll[f];
         f[x___] := List[x]

In[77]:= f[1, 2, 3, 4]
Out[77]= {1, 2, 3, 4}
```

Own Values

```
In[78]:= a = 42
Out[78]= 42

In[79]:= OwnValues[a]
Out[79]= {HoldPattern[a] :↦ 42}
```

Down Values

A *DownValue* is defined when the variable itself does not have a meaning, but can get one when combined with the proper arguments. This is the case for the most function definitions.

In[80]:=
```
f[x_] := x^2
```

This defines a pattern for *f* specifying that each time *f[...]* is encountered, it is to be replaced by ...^2. This pattern is meaningless if there is a lonely *f,*

In[81]:=
```
f
```

Out[81]=
```
f
```

However, when encountered with an argument downwards (i.e. down the internal structure of the command you entered), the pattern applies,

In[82]:=
```
f[y]
```

Out[82]=
```
y^2
```

In[83]:=
```
DownValues[f]
```

Out[83]=
```
{HoldPattern[f[x_]] :> x^2, HoldPattern[f[x___]] :> {x}}
```

Up Values

Sometimes, it's convenient not to associate the rule to the outermost symbol. For example, you may want to have a symbol whose value is *2* when it has a subscript of *1*, for example to define a special case in a sequence. This would be entered as follows:

In[84]:=
```
c /: Subscript[c, 1] := 2
```

If the symbol *c* is encountered, neither of the discussed patterns apply. *c* on its own has no own hence no OwnValue, and looking down the command tree of c when seeing *Subscript[c,1]* yields nothing, since c is already on an outermost branch. An *UpValue* solves this problem: a symbol having an *UpValue* defines a pattern where not only the children, but also the parents are to be investigated, i.e. Mathematica has to look up the command tree to see whether the pattern is to be applied.

In[85]:=
```
UpValues[c]
```

Out[85]=
```
{HoldPattern[c1] :> 2}
```

Sub Values

SubValues is used for definitions of the type

In[86]:=
```
d[e][f] = x;
```

This defines neither an *OwnValue* nor a *DownValue* for *d*, since it does not really define the value for the atomic object *d* itself, but for *d[e]*, which is a composite. Read the definition above as *(d[e])[f]=x*.

In[87]:=
```
SubValues[d]
```

Out[87]=
```
{HoldPattern[d[e][f]] :> x}
```

Examples

Counting

Data in this example are represented as time series of 0's and 1's.

In[88]:=
```
SeedRandom[0]
data = RandomInteger[{0, 1}, {20}]
```

Out[89]=
```
{1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0}
```

Task is to get a sum of 0's and 1's for each time step.

Idea

In[90]:=
```
(* C style *)
n = Length[data];
cnt0 = 0;
cnt1 = 0;
output = Table[0, {n}];
For[i = 1, i <= n, i++,
  If[data[[i]] == 0, cnt0++, cnt1++];
  output[[i]] = {cnt0, cnt1};
 ];
output
```

Out[95]=
```
{{0, 1}, {1, 1}, {1, 2}, {2, 2}, {3, 2}, {3, 3}, {3, 4}, {3, 5}, {4, 5}, {5, 5}, {6, 5},
 {7, 5}, {7, 6}, {8, 6}, {9, 6}, {10, 6}, {11, 6}, {12, 6}, {12, 7}, {13, 7}}
```

Example Solutions

In[96]:=
```
Tally[Take[data, #] ~Join~ {0, 1}][[All, 2]] - {1, 1} & /@ Range[Length[data]]
```

Out[96]=
```
{{1, 0}, {1, 1}, {2, 1}, {2, 2}, {2, 3}, {3, 3}, {4, 3}, {5, 3}, {5, 4}, {5, 5}, {5, 6},
 {5, 7}, {6, 7}, {6, 8}, {6, 9}, {6, 10}, {6, 11}, {6, 12}, {7, 12}, {7, 13}}
```

In[97]:=
```
FoldList[#1 + (#2 /. {0 → {1, 0}, 1 → {0, 1}}) &, {0, 0}, data]
```

Out[97]=
```
{{0, 0}, {0, 1}, {1, 1}, {1, 2}, {2, 2}, {3, 2}, {3, 3}, {3, 4}, {3, 5}, {4, 5}, {5, 5},
 {6, 5}, {7, 5}, {7, 6}, {8, 6}, {9, 6}, {10, 6}, {11, 6}, {12, 6}, {12, 7}, {13, 7}}
```

In[98]:=
```
Accumulate[data /. {0 → {1, 0}, 1 → {0, 1}}]
```

Out[98]=
```
{{0, 1}, {1, 1}, {1, 2}, {2, 2}, {3, 2}, {3, 3}, {3, 4}, {3, 5}, {4, 5}, {5, 5}, {6, 5},
 {7, 5}, {7, 6}, {8, 6}, {9, 6}, {10, 6}, {11, 6}, {12, 6}, {12, 7}, {13, 7}}
```

Recursive Fibonacci Number

In[99]:=
```
fib[0] := 0;
fib[1] := 1;
fib[n_] := fib[n - 1] + fib[n - 2] /; n > 1
```

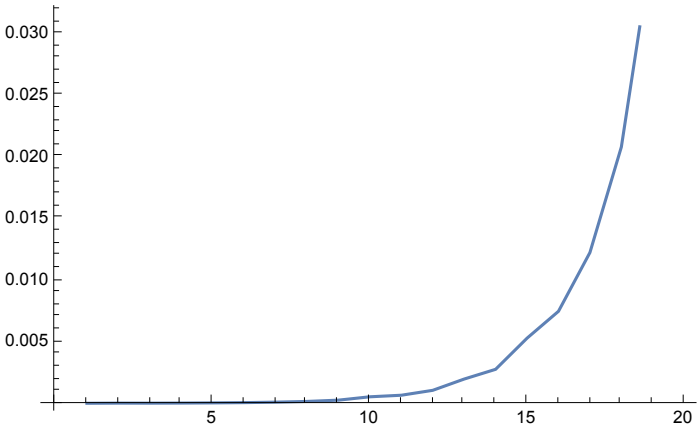
In[102]:=
```
fib[5]
```

Out[102]=
```
5
```

In[103]:=
```
AbsoluteTiming[fib[27]]
```
Out[103]= {1.58688, 196 418}

In[104]:=
```
AbsoluteTiming[fib[28]]
```
Out[104]= {2.77744, 317 811}

In[105]:=
```
ListLinePlot[(AbsoluteTiming[fib[#]] & /@ Range[1, 20])[[All, 1]]]
```
Out[105]=



In[106]:=
```
ClearAll[fib]
```

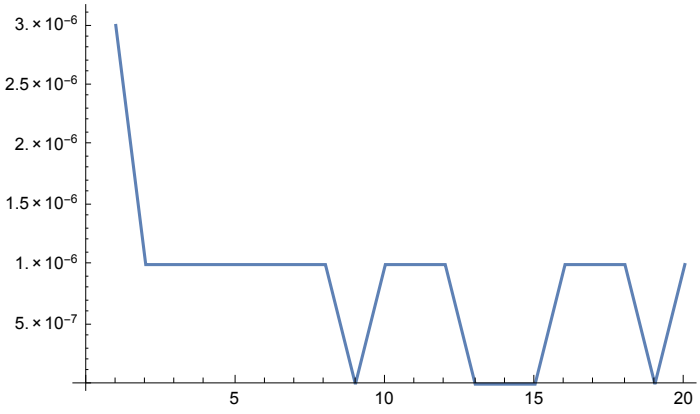
In[107]:=
```
fib[0] := 0;
fib[1] := 1;
fib[n_] := (fib[n] = fib[n - 1] + fib[n - 2]) /; n > 1
```

In[110]:=
```
fib[5]
```
Out[110]= 5

In[111]:=
```
AbsoluteTiming[fib[27]]
```
Out[111]= {0.000285, 196 418}

In[112]:=
```
AbsoluteTiming[fib[28]]
```
Out[112]= {0.000026, 317 811}

In[113]:=
```
ListLinePlot[(AbsoluteTiming[fib[#]] & /@ Range[1, 20])[[All, 1]]]
```
Out[113]=



Composition

In[114]:=
```
model[cfg_List][x_] := Plus @@ MapIndexed[#1 x^(First@#2 - 1) &, cfg]
```

In[115]:=
```
model[{10, 20, 30, 40}][x]
```

Out[115]= $10 + 20\,x + 30\,x^2 + 40\,x^3$

In[116]:=
```
model[{10, 20, 30, 40}][4]
```

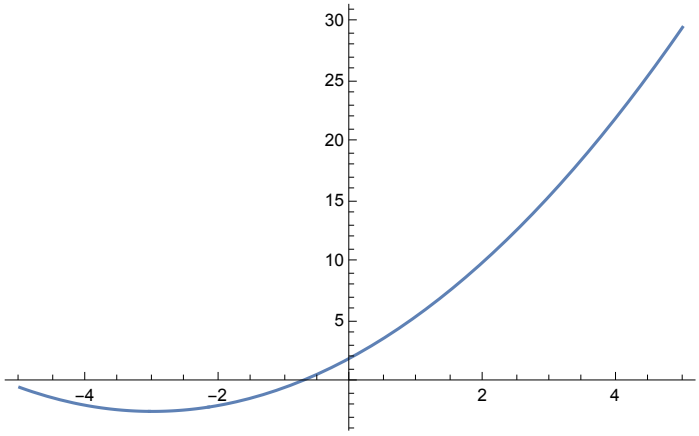
Out[116]= 3130

In[117]:=
```
m = model[{2, 3, 0.5}]
```

Out[117]=
```
model[{2, 3, 0.5}]
```

In[118]:=
```
Plot[m[x], {x, -5, 5}]
```

Out[118]=

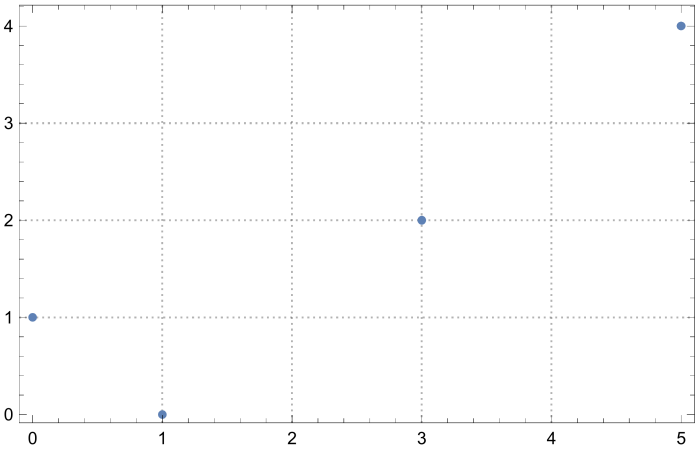


Example: LinearModelFit

In[119]:=
```
data = {{0, 1}, {1, 0}, {3, 2}, {5, 4}};
```

In[120]:=
```
ListPlot[data, PlotTheme → "Detailed"]
```

Out[120]=

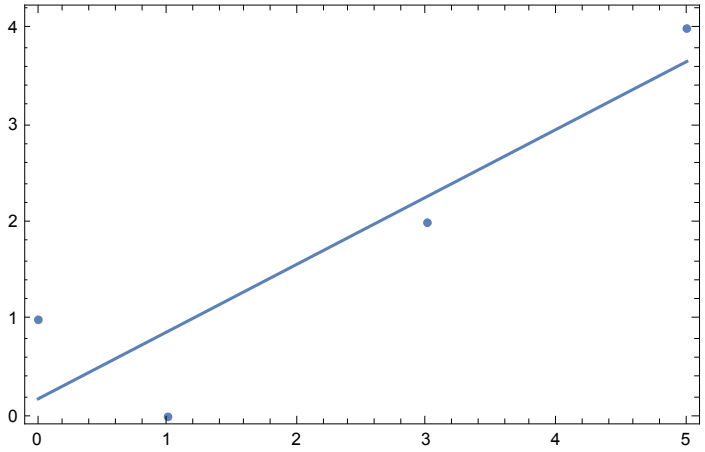


In[121]:=
```
lm = LinearModelFit[data, x, x]
```

Out[121]=
```
FittedModel[ 0.186441+0.694915x ]
```

In[122]:=
```
Show[ListPlot[data], Plot[lm[x], {x, 0, 5}], Frame → True]
```

Out[122]=



In[123]:=
```
FullForm[lm]
```

Out[123]//FullForm=
```
FittedModel[List["Linear", List[0.1864406779661015`, 0.6949152542372882`],
  List[List[x], List[1, x]], List[0, 0]], List[List[1.`, 1.`, 1.`, 1.`]],
 List[List[0, 1], List[1, 0], List[3, 2], List[5, 4]],
 List[List[1.`, 0.`], List[1.`, 1.`], List[1.`, 3.`], List[1.`, 5.`]],
 Function[Null, Internal`LocalizedBlock[List[x], Slot[1]], List[HoldAll]]]
```

In[124]:=
```
lm["Properties"]
```

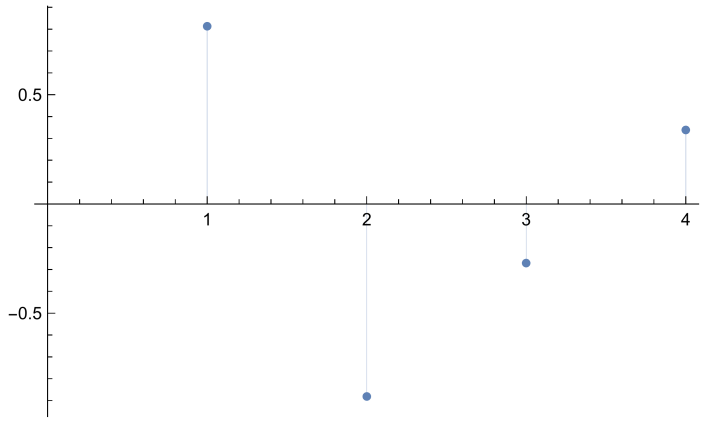
Out[124]=
```
{AdjustedRSquared, AIC, AICc, ANOVATable, ANOVATableDegreesOfFreedom,
 ANOVATableEntries, ANOVATableFStatistics, ANOVATableMeanSquares,
 ANOVATablePValues, ANOVATableSumsOfSquares, BasisFunctions,
 BetaDifferences, BestFit, BestFitParameters, BIC, CatcherMatrix,
 CoefficientOfVariation, CookDistances, CorrelationMatrix,
 CovarianceMatrix, CovarianceRatios, Data, DesignMatrix,
 DurbinWatsonD, EigenstructureTable, EigenstructureTableEigenvalues,
 EigenstructureTableEntries, EigenstructureTableIndexes,
 EigenstructureTablePartitions, EstimatedVariance, FitDifferences,
 FitResiduals, Function, FVarianceRatios, HatDiagonal, MeanPredictionBands,
 MeanPredictionConfidenceIntervals, MeanPredictionConfidenceIntervalTable,
 MeanPredictionConfidenceIntervalTableEntries, MeanPredictionErrors,
 ParameterConfidenceIntervals, ParameterConfidenceIntervalTable,
 ParameterConfidenceIntervalTableEntries, ParameterConfidenceRegion,
 ParameterErrors, ParameterPValues, ParameterTable, ParameterTableEntries,
 ParameterTStatistics, PartialSumOfSquares, PredictedResponse,
 Properties, Response, RSquared, SequentialSumOfSquares,
 SingleDeletionVariances, SinglePredictionBands,
 SinglePredictionConfidenceIntervals, SinglePredictionConfidenceIntervalTable,
 SinglePredictionConfidenceIntervalTableEntries, SinglePredictionErrors,
 StandardizedResiduals, StudentizedResiduals, VarianceInflationFactors}
```

In[125]:=
```
lm["FitResiduals"]
```

Out[125]=
```
{0.813559, -0.881356, -0.271186, 0.338983}
```

In[126]:=
```
ListPlot[lm["FitResiduals"], Filling → Axis]
```

Out[126]=



References

- http://mathematica.stackexchange.com/questions/96/what-is-the-distinction-between-downvalues-upvalues-subvalues-and-ownvalues
- Hudak, Paul: *Conception, evolution, and application of functional programming languages*. September 1989, ACM Computing Surveys 21 (3): 359–411. doi:10.1145/72551.72554; http://www.dbnet.ece.ntua.gr/~adamo/languages/books/p359-hudak.pdf