History of Functional Programming Languages

Zurich FSharp User Meetup
2013-02-27

Stephan Missura (stephan@missura.ch)

Thank you!

- Marc Sigrist
- NetException
- QuantAlea

Grüezi Herr Sigrist

Beim Lesen der Kundenrezension zum Buch "Programming F#" auf Amazon war ich überrascht, dass der Autor (nämlich Sie :-) aus der Schweiz kommt, da ich bisher keinerlei F#-Aktivität in der Schweiz festgestellt habe (nicht einmal bei Microsoft Schweiz...).
Nun nimmt mich Wunder, in wie weit Sie F# planen, produktiv einzusetzen und Ihr Verhältnis zum funktionalen Paradigma im allgemeinen.

Ich freue mich auf Ihre Antworten!

Mit freundlichen Grüssen
Stephan Missura

PS: Ich selber beschäfte mich seit 20 Jahren mit dem funktionalen Paradigma und funktionalen Sprachen, leider aber fast ausschliesslich als Hobby (früher auch akademisch).

Agenda

- Concepts of functional programming
- Timeline of FPL and their contributions
- Lisp
- APL
- Scheme
- SML
- Haskell
- Summary and Q&A
- Appendix: References

Concepts of functional programming (1)

- functions
  - recursion instead of iteration
  - functions as 1st class values
    - higher-order functions
  - applicative vs. compositional style

```
function gcd( a, b )
  while b ≠ 0
     t := b
     b := a mod t
     a := t
  return a
```

```
fun gcd( m, n ) =
  if m = 0 then
    n
  else
    gcd( n mod m, m )
```

```
sum (x:xs) = x + sum xs
sum []     = 0
```

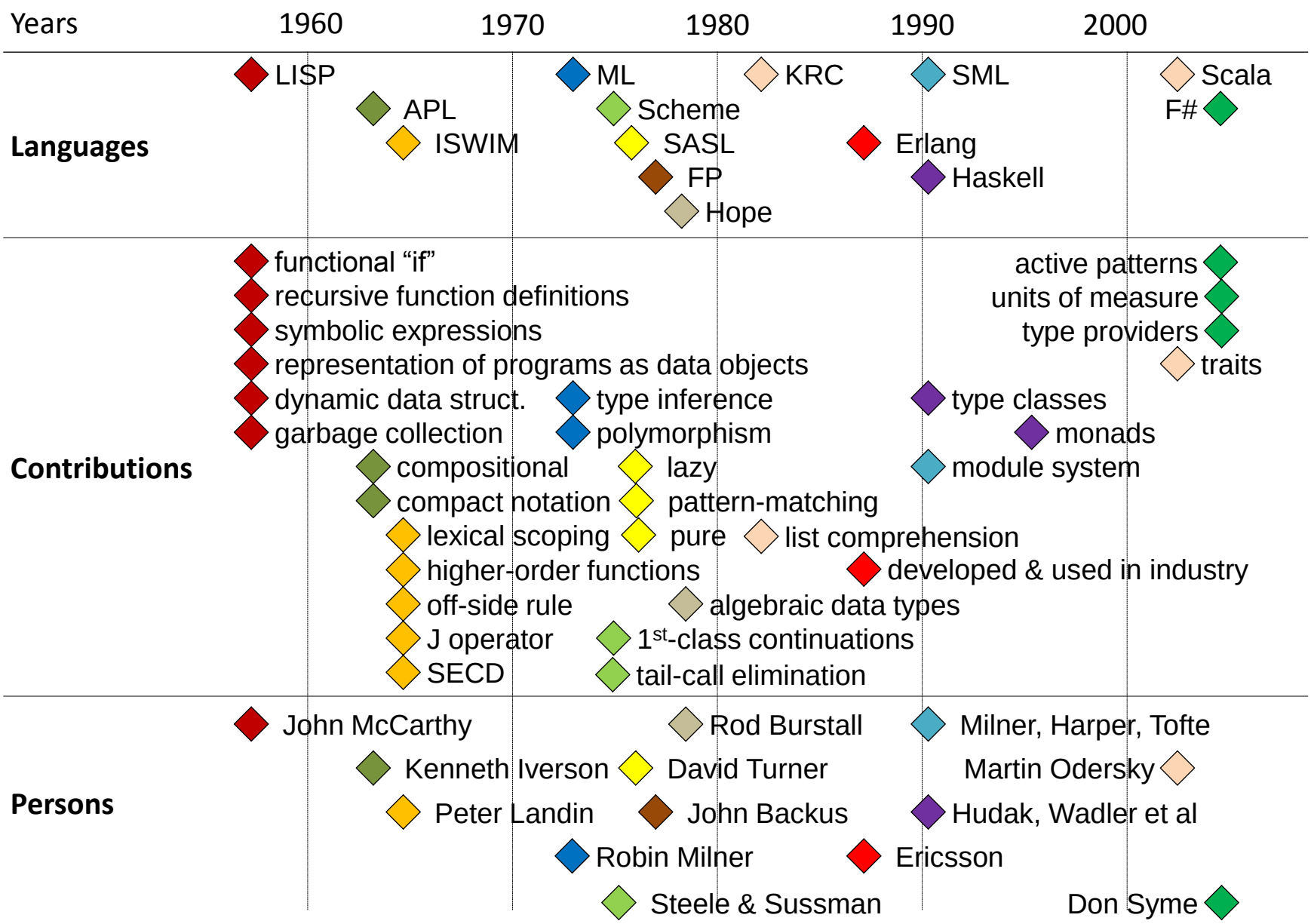
```
sum = foldr (+) 0
```

```
fun gcd( 0, n ) = n
  | gcd( m, n ) = gcd( n mod m, m )
```

Concepts of functional programming (2)

- sophisticated type systems
  - parametric polymorphism
  - dependent types
- algebraic data types
  - recursive types
  - pattern matching
- lazy evaluation
  - infinite data structures
- no implicit state
  - no variables & no assignments
  - free of side-effects or controlled side-effects
  - pure functions (= mathematical functions)

Timeline of FPL and their contributions



LISP – contributions (1)

- functional “if”
- recursive function definitions

$$n! = (n = 0 \rightarrow 1, T \rightarrow n \cdot (n-1)!)$$

$$gcd(m,n) = (m > n \rightarrow gcd(n,m), rem(n,m) = 0 \rightarrow m, T \rightarrow gcd(rem(n,m),m))$$

LISP – contributions (2)

- symbolic expressions
- lists

1. Atomic symbols are S-expressions.
2. If e_1 and e_2 are S-expressions, so is $(e_1 \cdot e_2)$.

Examples of S-expressions are

$$AB$$
$$(A \cdot B)$$
$$((AB \cdot C) \cdot D)$$

The list

$$(m_1, m_2, \cdots, m_n)$$

is represented by the S-expression

$$(m_1 \cdot (m_2 \cdot (\cdots (m_n \cdot NIL) \cdots)))$$

Here NIL is an atomic symbol used to terminate lists.

LISP – contributions (3)

- representation of programs as data objects

```
(define (scale-list items factor)
  (if (null? items)
      nil
      (cons (* (car items) factor)
            (scale-list (cdr items)
                        factor))))
(scale-list (list 1 2 3 4 5) 10)
(10 20 30 40 50)
```

APL – contributions

- compact notation
- compositional style

$$+/\iota 100$$

$$(\sim R \in R \circ . \times R)/R \leftarrow 1 \downarrow \iota R$$

```
     ∇ Z←FFT X;C;D;E;J;K;LL;M;N;O
[1]    LL←⌊2*-O-⍳M←⌊2⊛N,0⍴E←1-2×∼O←⍳1.J←⍳L
       ←0,0⍴K←⍳N←¯1↑⍴X
[2]    →(M>L←L+1)/1+⍴⍴J←J,N⍴ 0 1 ∘.=(
       2*L)⍴1
[3]    Z←X[;(L←0)+(⌽LL)+.×J←(M,N)⍴J]
[4]    X← 2 1 ∘.○○(-O-K)÷¯1↑LL
[5]    Z←Z[;K-,LL[L]×J[L;]]+(⍴Z)⍴(-⌿X[;D]×
       Z[;C]),+⌿X[;D←O+N⍴LL[E+M-L]×-O-⍳
       2×LL[L]]×⊖Z[;C←K+,LL[L]×0=J[L;]]
[6]    →((M+O)>L←L+1)/5
     ∇
```

Scheme – contributions

- tail-call elimination
- 1st-class continuations

```
(define (f return)
  (return 2)
  3)

(display (f (lambda (x) x))) ; displays 3

(display (call-with-current-continuation f)) ; displays 2
```

```
(let* ((yin
        ((lambda (cc) (display "@") cc) (call-with-current-continuation (lambda (c) c))))
       (yang
        ((lambda (cc) (display "*") cc) (call-with-current-continuation (lambda (c) c)))))
  (yin yang))
```

SML – module system: signatures (1)

```
signature FIFO =
sig
  type 'a fifo

  exception Dequeue

  val empty    : 'a fifo
  val isEmpty  : 'a fifo -> bool
  val enqueue  : 'a fifo * 'a -> 'a fifo
  val dequeue  : 'a fifo -> 'a fifo * 'a
  val delete   : ('a fifo * ('a -> bool)) -> 'a fifo
  val head     : 'a fifo -> 'a
  val peek     : 'a fifo -> 'a option
  val length   : 'a fifo -> int
  val contents : 'a fifo -> 'a list
  val app      : ('a -> unit) -> 'a fifo -> unit
  val map      : ('a -> 'b) -> 'a fifo -> 'b fifo
  val foldl    : ('a * 'b -> 'b) -> 'b -> 'a fifo -> 'b
  val foldr    : ('a * 'b -> 'b) -> 'b -> 'a fifo -> 'b
end (* FIFO *)
```

SML – module system: structures

```
structure Fifo : FIFO =
struct
  datatype 'a fifo = Q of { front: 'a list, rear: 'a list }

  exception Dequeue

  val empty = Q{front=[], rear=[]}

  fun isEmpty (Q{front=[], rear=[]}) = true
    | isEmpty _                      = false

  fun enqueue (Q{front, rear},x) = Q{front=front, rear=(x::rear)}

  fun dequeue (Q{front=(hd::tl), rear}) = (Q{front=tl, rear=rear}, hd)
    | dequeue (Q{rear=[],...})          = raise Dequeue
    | dequeue (Q{rear,...})             = dequeue( Q{front=rev rear,
                                                     rear=[]} )
...
end (* Fifo *)
```

SML – module system: signatures (2)

```
signature ORD_KEY =
sig
  type ord_key
  val compare : ord_key * ord_key -> order
end (* ORD_KEY *)

signature ORD_SET =
sig
  structure Key : ORD_KEY

  type item = Key.ord_key
  type set

  val empty     : set
  val singleton : item -> set
  val add       : set * item -> set
  val delete    : set * item -> set
  val member    : set * item -> bool
  val equal     : (set * set) -> bool
  …
```

```
datatype order = LESS | EQUAL | GREATER
```

SML – module system: functors

```
functor ListSetFn (K : ORD_KEY) : ORD_SET =
struct
  structure Key = K

  type item = Key.ord_key
  type set  = item list

  val empty          = []
  fun singleton x    = [x]
  fun add( l, item ) = let
         fun f []         = [item]
           | f( elem::r ) = ( case Key.compare( item, elem )
                of LESS    => item :: elem :: r
                 | EQUAL   => item :: r
                 | GREATER => elem :: (f r) )
         in
           f l
         end
  fun delete (l, elem) = …
  …
```

Haskell – contributions (1)

- type classes

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

instance Eq Int where
  i1 == i2 = eqInt i1 i2
  i1 /= i2 = not (i1 == i2)

instance (Eq a) => Eq [a] where
  []     == []     = True
  (x:xs) == (y:ys) = (x == y) && (xs == ys)
  xs /= ys = not (xs == ys)

member :: Eq a => a -> [a] -> Bool
member x []                  = False
member x (y:ys) | x==y       = True
                | otherwise = member x ys
```

Haskell – contributions (2)

- monads
  - dealing with computations (possessing potential side-effects) in a clean way by associating special types to them

A monad consists of a type constructor `M` and a pair of functions, `return` and `>>=` (sometimes pronounced "bind"). Here are their types:

```
return :: a -> M a
(>>=)  :: M a -> (a -> M b) -> M b
```

```
main :: IO ()
main = do appendChan stdout "enter filename\n"
          userInput <- readChan stdin
          let (name : _) = lines userInput
          appendChan stdout name
          catch (do contents <- readFile name
                    appendChan stdout contents)
                (appendChan stdout "can't open file")
```

Summary

- main functional-programming concepts
- timeline of FPLs, including their contributions
- several contributions in various functional languages

Many thanks…

…for your interest! ☺

Any questions?

Appendix – References (1)

Author(s)	Title	Year	Topics
Abelson & Sussman	Structure and Interpretation of Computer Programs (2nd edition)	1996	Complete introduction to the foundations of computer programming (procedural and data abstractions, modularity, objects, streams, interpreters, register machines, compilers, …) using Scheme. I would say a "must have read"! Meanwhile freely downloadable: http://mitpress.mit.edu/sicp http://sicpebook.wordpress.com/ebook
Armstrong	Programming Erlang – Software for a Concurrent World	2007	Introduction to Erlang.
Backus	Can programming be liberated from the von Neumann style?	1978	John Backus – the leader of the Fortran development team and father of the "Backus-Naur" grammar description – strongly advocates the functional-programming style in his Turing-award lecture.
Beeson	Foundations of Constructive Mathematics	1984	As the title says... Tough!
Engeler & Läuchli	Berechnungstheorie für Informatiker	1988	Among others: Recursion and LISP (german).
Hudak, Hughes, Peyton Jones	A History of Haskell: Being Lazy With Class	2007	History of Haskell, including an overview and references of functional programming history in general.

Appendix – References (2)

Author(s)	Title	Year	Topics
Landin	The Next 700 Programming Languages	1966	ISWIM. See http://www.iro.umontreal.ca/~feeley/cours/ift6232/doc/the-next-700-programming-languages.pdf .
McCarthy	Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I	1960	The original paper introducing LISP. See http://www-formal.stanford.edu/jmc/recursive.pdf (typographically revised version).
Milner et al	The Definition of Standard ML (Revised)	1997	Operational semantics of Standard ML.
Paulson	ML for the Working Programmer	1991	Introduction to functional programming using SML.
Pepper & Hofstedt	Funktionale Programmierung	2006	Extensive introduction to functional programming (german).
Pierce (editor)	Advanced Topics in Types and Programming Languages	2005	Various applications of type systems (based on [Pierce, 2005]).
Pierce	Types and Programming Languages	2002	Excellent comprehensive introduction to type systems. I would say a “must have” in the area of static type systems.
Smith	Programming F# 3.0	2012	A comprehensive guide to learn F#.
Turner	Constructive Foundations for Functional Languages	1991	Introduction to constructive logic and type theories as foundations of functional programming languages.