Functional Approaches to Programming

~ Lisp / Haskell: the diff tutorial ~

Didier Verna

EPITA / LRDE

didier@lrde.epita.fr

lrde/~didier


@didierverna


didier.verna


in/didierverna

Outline

Plan

Syntax and Expressions

Typing and Type Checking

Boolean Logic

Arithmetic

Lists

Corollary: Homoiconicity (Lisp)

sqrt by Newton-Raphson

Syntaxe

- **Layout**
  - **Haskell:** “offside-rule” [Landin, 1966]
    Implicit separator: `';'`, a few reserved words
  - **Lisp:** parentheses (but *cf.* reader-macros), no reserved word
- **Naming**
  - **Haskell:** indentical to C, plus apostrophe
    Upcase first letter for types
  - **Lisp:** anything
    Escape syntax when needed: `|...|`

Operators and Functions

- **Lisp:** no distinction (except “special operators”)
  - `(f arg1 arg2 ...)`
  - Prefix notation (but *cf.* macros)
  - Variadic operators
  - No ambiguity (precedence, associativity, *etc.*)
- **Haskell:** distinction (but two-way correspondance)
  - `f arg1 arg2 ...`
  - `3 + 4` ⇔ `(+) 3 4`
  - `div 3 4` ⇔ `3 `div` 4`
  - Ambiguities (precedence, associativity, *etc.*)
    (`f n+1`, `f -12` *etc.*)
  - New operator definition through prefix notation
    `! # $ % & * + > / < = > ? \ ^ | : - ~`

Variadicity in Lisp

```
(defun mklist (head &rest tail)
  (cons head tail))
;; (mklist 1) => (1)
;; (mklist 1 2) => (1 2)
;; (mklist 1 2 3) => (1 2 3)
;; etc.

(defun msg (str &optional (prefix "error: ") postfix)
  (concatenate 'string prefix str postfix))
;; (msg "hello") => "error: hello"
;; (msg "hello" nil) => "hello"
;; (msg "hello" "me: ") => "me: hello"
;; (msg "hello" "me: " "!") => "me: hello!"

(defun msg* (str &key prefix (postfix "."))
  (concatenate 'string prefix str postfix))
;; (msg* "hello") => "hello."
;; (msg* "hello" :prefix "me: ") => "me: hello."
;; (msg* "hello" :postfix "!" :prefix "me: ") => "me: hello!"
```

- Plus &-combinations
- Remark: `'string` = `(quote string)` ⇒ `string` *(symbol)*

Naming and Assignment

Haskell

```haskell
foo, bar, boo :: Float
foo = 10
bar = sqrt (3 * (6 + 7) - 8)
boo = bar

baz :: Float -> Float -> Float
baz a b = sqrt (3 * (a + 7) - b)
```

Lisp

```lisp
(defvar foo 10)
(setq foo 10)
(setq bar (sqrt (- (* 3 (+ 6 7)) 8)))
(setq boo bar)

(defun baz (a b)
  (sqrt (- (* 3 (+ a 7)) b)))
```

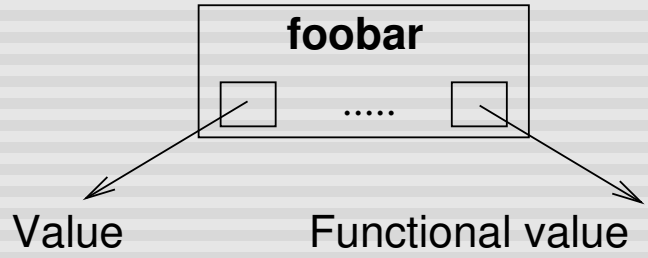
- **Haskell:** *syntactic* abstraction (declarations)
- **Lisp:** *functional* abstraction (expressions)

Naming, Assignment, and Purity

- **Naming** (purely functional): *giving a name to a value*
  A value may hence have different names
- **Assignment** (imperative): *giving a value to a name*
  A name may hence have different values

"Lisp-2"

- **Every symbol has *two* values**
  - an arbitrary value
  - a functional value
  - these values coexist




```
(defvar foobar 3)
(defun foobar (x) (* 2 x))
;; (foobar foobar) => 6
```

- **Access**
  - `foobar` ⇒ *valeur*
  - `#'foobar` = `(function foobar)` ⇒ *functional value*
- **Remark**
  - `'(foobar foobar)` = `(quote (foobar foobar))`
    ⇒ `(foobar foobar)`

Interactive Development

- **"Read-eval-print"** loop (REPL)
  - **Read:** input an *expression*
  - **Eval:** compute ("evaluate") its value
  - **Print:** output the result *in printable format*
- **Remarks**
  - **Haskell:** limited REPL (expressions *vs.* declarations)
  - **Lisp:** interpretation and/or byte- [JIT] compilation

Plan

Typing and Type Checking

- **Remark:** problem orthogonal to functional programming
- **Static Typing** (Haskell)
  - Typing of *variables / functions*
  - *Compile-time* type checking
- **Dynamic Typing** (Lisp)
  - Typing of *values*
  - *Run-time* type checking

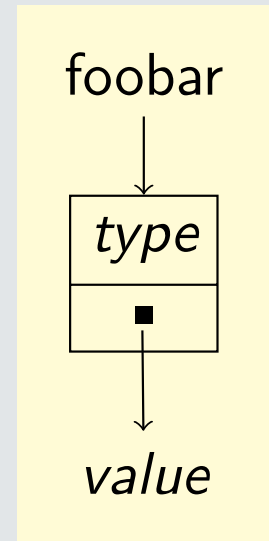
Static Typing (Haskell)

- Type information *useless* at run-time
  Type checking done at compile-time
- Potential dynamic access (introspection, REPL)
  E.g. `:type` in GHCi
- **Beware the vocabulary**
  - *Static* type checking: at compile-time
  - *Explicit* or *implicit* typing: Inference / Polymorphism
  - *Strong* typing: all type errors are caught (at compile-time)

Dynamic Typing (Lisp)

- Type information *necessary* at run-time
  Type checking done at run-time
- Boxing: typing of *values*
- Dynamic access in the language (functional)

```
(type-of this)
(typep that 'integer)
```



- **Beware the vocabulary**
  - *Dynamic* type checking: at run-time
  - *Implicit* typing (at least in the code)
  - *Strong* typing (always): all type errors are (always) caught (but at run-time)

Static Typing and Polymorphism

Lisp

```
(defun len (l)
  (if (null l) 0 (1+ (len (cdr l)))))
```

Haskell

```
len :: [a] -> Int
len [] = 0
len (x:xs) = 1 + len xs
```

- **Type of `length` in Haskell**
  - `[Int] -> Int` ? `[String] -> Int` ? …?
- **Parametric polymorphism**
  - Type variable: `length :: [a] -> Int`
  - But lists remain homogeneous
  - `:type` returns the *most general* type
- **Polymorphism *vs.* Overloading**
  - Polymorphism: *unique* definition ∀ type
  - Overloading: ≠ definitions according to the type

Static *vs.* Dynamic Typing

- **Dynamic**
  - More expressive (*de-facto* polymorphism)
  - Impact on performance and safety
- **Static**
  - More restrictive (parametric polymorphism)
  - *De-facto* safety
- **Static / dynamic mix** (hot!)
  - Libraries (*e.g.* Shen, static *and* strong typing)
  - Types

    C# `dynamic`: short-circuits static checks

    Haskell `Dynamic` + `toDyn` + `fromDynamic`
  - Common Lisp static annotations (optimization / weak typing)
  - Gradual typing [Siek, 2006]

Plan

Booleans

- **Haskell**
  - `Bool` type
  - `True` and `False` values
  - Operators: `&&`, `||`, `not`
  - Functions: `and, or :: [Bool] -> Bool`
- **Lisp**
  - No specific type
  - True: `t` and everything except `nil`
  - False: `nil` and `()`
  - Special operators (variadic): `and`, `or`
  - `not`

Conditionals

- `if then else`

Haskell

```haskell
max3 :: Int -> Int -> Int -> Int
max3 m n p = if (m >= n) && (m >= p) then m
             else if (n >= m) && (n >= p)
                  then n else p
```

Lisp

```lisp
(defun max3 (m n p)
  (if (and (>= m n) (>= m p))
      m
    (if (and (>= n m) (>= n p))
        n
      p)))
```

- Guards and `cond`

Haskell

```haskell
max3 :: Int -> Int -> Int -> Int
max3 m n p
    | (m >= n) && (m >= p) = m
    | (n >= m) && (n >= p) = n
    | otherwise = p
```

Lisp

```lisp
(defun max3 (m n p)
  (cond ((and (>= m n) (>= m p))
         m)
        ((and (>= n m) (>= n p))
         n)
        (t p)))
```

Conditionals *vs.* Equations (Haskell)

```
ssq :: Int -> Int
ssq n = if (n == 1) then 1
        else n*n + ssq (n-1)
```

```
ssq :: Int -> Int
ssq n
    | (n == 1) = 1
    | otherwise = n*n + ssq (n-1)
```

- Equation style preferable

```
ssq :: Int -> Int
ssq 1 = 1
ssq n = n*n + ssq (n-1)
```

- Conditionals & Pattern matching

```
lst :: [a] -> a
lst x = case (reverse x) of
  [] -> error "Empty list"
  (r:rs) -> r
```

Other Conditionals (Lisp)

- **On objects**

```
(defun month-length (month)
  (case month
    ((jan mar may jul aug oct dec) 31)
    ((apr jun sept nov) 30)
    (feb 28) ;; Y2K bug !!!
    (otherwise (error "Unknown month !"))))
```

- **On types**

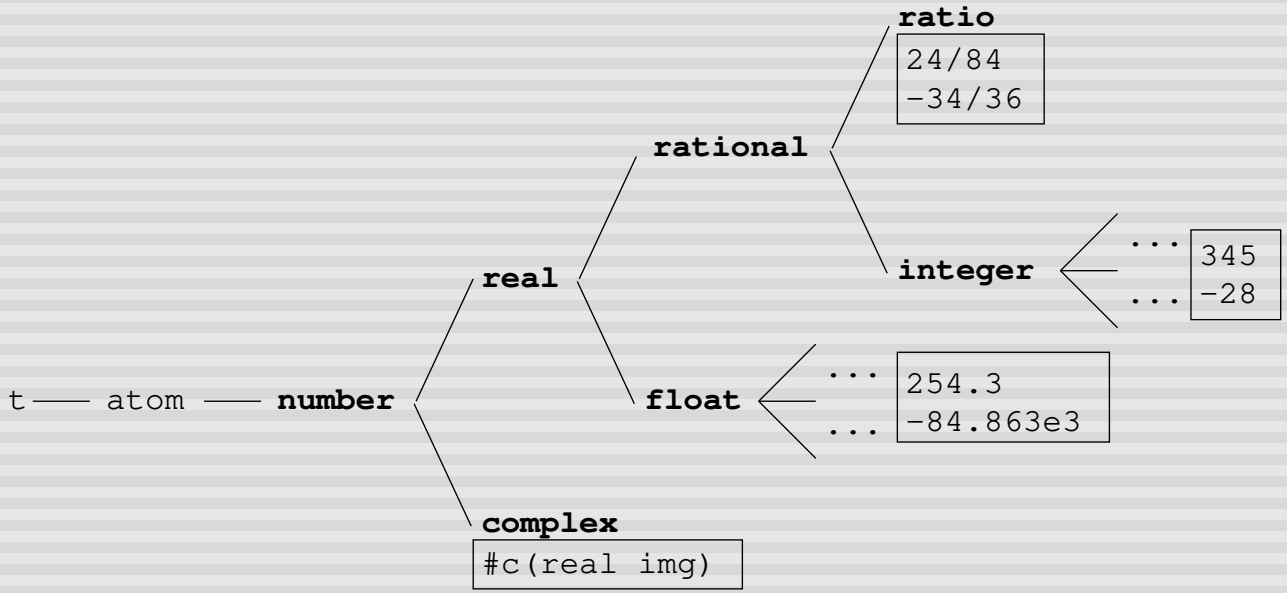
```
(defun +func (x)
  (typecase x
    (number #'+)
    (list #'append)))

(defun my+ (&rest args)
  (apply (+func (car args)) args))
```

- **Other:** `when`, `unless`, *etc.*

Plan

Numeric Types (Lisp)



- **Predicates:** `floatp`, `integerp`, *etc.*
- **Implicit Transtyping:** ratios to integers, complexes to reals
- **Operational Transtyping**
  - Automatic: to floats or complexes
  - Explicit: `float`, `coerce`

Numeric Types (Haskell)

- Usual notation
- Similar hierarchy under the `Num` class (`Int`, `Float` *etc.*)
  (Inspired from Scheme (inspired from Common Lisp))

- **Overloading of litterals**
  ```
  42 :: Num p => p
  ```
- **Polymorphic operators**
  ```
  (+) :: Num a => a -> a -> a
  (==) :: Eq a => a -> a -> Bool
  ```
- **No automatic operational transtyping !**
  ```
  fromIntegral :: (Integral a, Num b) => a -> b
  ```

Plan

Lists

- Built-in type in all functional languages
- **Syntax**
  - Haskell: `[e1, e2, e3, ...]`
  - Lisp: `(e1 e2 e3 ...)`
- **Type**
  - Haskell: homogeneous lists
    $\forall$ `t`, $\exists$ `[t]` (lists of elements of type `t`)
  - Lisp: heterogeneous lists, `listp` predicate
- **Empty list**
  - Haskell: `[]` (polymorphic, type `[a]` $\forall$ `a`)
  - Lisp: `()` $\Leftrightarrow$ `nil` (from Latin *nihil*)

Construction

- **Canonical Form**
  - Head + Tail
  - Subsequent constructions on top of the empty list
- **Haskell:** `(:) :: a -> [a] -> [a]`
- **Lisp:** `cons` (`consp` predicate)
- **Remark:** unicity of the "construction" ≠ "generation" (++, append, *etc.*)

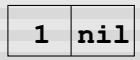
Enumerators (Haskell)

```
-- Form 1: [n .. m]
[2 .. 7]
[3.1 .. 7.0] -- [3.1,4.1,5.1,6.1,7.1]

-- Form 2: [n,p .. m]
[7,6 .. 3]
[0.0,0.3 .. 1.0] -- [0.0,0.3,0.6,0.8999999999999999]
```

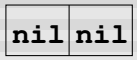
Examples


```
(1)  (cons 1 nil)
[1]  1:[]
```

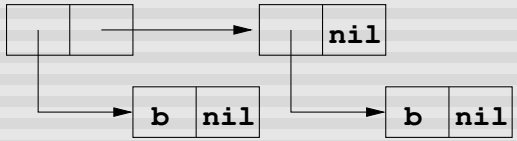



```
(1 2)  (cons 1 (cons 2 nil))
[1, 2] 1:(2:[])
```

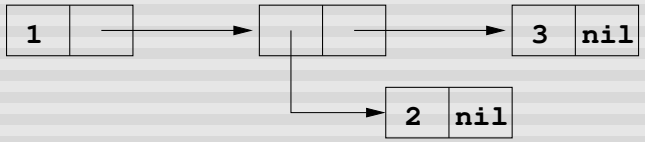

```
(nil)  (cons nil nil)
[[]]   []:[]
```




```
((1) (2))  (cons (cons 1 nil) (cons (cons 2 nil) nil))
[[1], [2]] (1:[]):((2:[]):[])
```




```
(1 (2) 3)  (cons 1 (cons (cons 2 nil) (cons 3 nil)))
```



- **Remark:** purity *vs.* concept(s) of equality

Remarks on the Constructor (Haskell)

- **Associativity:** right

  `[3,4] = 3:[4] = 3:(4:[]) = 3:4:[]`

- **Precedence:** application always prevails

  $\Rightarrow$ *always parenthesize !*

```
tail :: [a] -> [a]
tail [] = []
tail (x:xs) = xs
```

- **Caution:** variables local to pattern matching

```
elt :: a -> [a] -> Bool
elt x []     = False
elt x (x:xs) = True       -- Barf !!
elt x (y:ys) = elt x ys
```

Accessors

- **Canonical**
  - Lisp: `car` / `first`, `cdr` / `rest`
    Cf. IBM-704: **C**ontents of **A**ddress / **D**ecrement **R**egister)
  - Haskell: `head`, `tail`
    Pattern matching: `(x:xs)`
- **Positional**
  - Lisp: `first ... tenth`, `(nth n lst)`
  - Haskell: `last`, `(!!) :: [a] -> Int -> a`
- **Structural**
  - Lisp: `(nthcdr n lst)`, `last` ($\neq$ Haskell !)
  - Haskell: `drop :: Int -> [a] -> [a]`

Plan

Syntax and Expressions

Typing and Type Checking

Boolean Logic

Arithmetic

Lists

Corollary: Homoiconicity (Lisp)

sqrt by Newton-Raphson

Corollary: Homoiconicity (Lisp)

- **Syntax**
  - S-Expression: atom or list
  - Atome: litteral (`1.5`, `"foo"`, *etc.*) or symbol
  - Symbol: `foobar`
  - List: `(a "foo" (bar +) 2.5)`, *etc.*
    In particular: `(+ 1 2)`, `(defvar foo 1)`, *etc.*
- **Homoiconicity:** “same representation”
- **Crucial operator:** `quote`

> *Lisp is the greatest single programming language ever designed.*
> *– Alan Kay [Kay, 2017]*

Power

- **To the meta-level**
  - Tell me your name `(prin1 your-name)`
  - John Doe `=> "John Doe"`
  - Tell me "your name" `(prin1 'your-name)`
  - Your name `=> your-name`
- **Meta-programming** (made trivial)
  - Homoiconicity (code $\Longleftrightarrow$ data) + `quote`
  - `(+ 1 2)` *vs.* `'(+ 1 2)`
  - Complete reflexivity: introspection + intercession

Twists

- **Propagation of equality:**
  – Three equals two plus one ?
  `(= 3 (+ 2 1)) => T`
  – “Three” equals “two plus one” ?
  `(= '3 '(+ 2 1)) => nil`
- **Inference on predicates:**
  “Jazzmen are excellent musicians.”
  – John Scofield is a jazzman.
  ⇒ John Scofield is an excellent musician.
  – Lucy knows that John Scofield is a jazzman.
  ⇒ Does Lucy know that John Scofield is an excellent musician?

Plan

`sqrt` by Newton-Raphson

The Newton-Raphson Method

$$\sqrt{x} = \lim_{n \to +\infty} y_n \quad | \quad y_n = \frac{y_{n-1} + \frac{x}{y_{n-1}}}{2}$$

- Iterate the series y_n until the approximation is satisfactory

nrsqrt (Lisp)

Lisp

```
(defun nrsqrt (x delta)
  (nrfind x delta 1.0))

(defun nrfind (x delta yn)
  (if (nrfound x delta yn)
      yn
      (nrfind x delta (nrnext x yn))))

(defun nrfound (x delta yn)
  (<= (abs (- x (* yn yn))) delta))

(defun nrnext (x yn)
  (/ (+ yn (/ x yn)) 2))
```

nrsqrt (Haskell)

Haskell

```haskell
nrsqrt :: Float -> Float -> Float
nrsqrt x delta = nrfind x delta 1.0

nrfind :: Float -> Float -> Float -> Float
nrfind x delta yn
  | nrfound x delta yn = yn
  | otherwise = nrfind x delta (nrnext x yn)

nrfound :: Float -> Float -> Float -> Bool
nrfound x delta yn = abs (x - yn * yn) <= delta

nrnext :: Float -> Float -> Float
nrnext x yn = (yn + x / yn) / 2
```

Imperative `nrsqrt` (Lisp)

```
(defun nrsqrt (x delta)
  (loop :for y = 1.0 :then (/ (+ y (/ x y)) 2.0)
        :until (<= (abs (- x (* y y))) delta)
        :finally (return y)))
```

- **Reminder:** "what to do" *vs.* "how to do it"
- **Question:** does an algorithm contain its own expression paradigm ?

Your mileage may vary...

Plan

Bibliography

Bibliography

- Peter J. Landin.
  The Next 700 Programming Languages.
  Communications of the ACM 9(3): 157–166.
- Jeremy Siek and Walid Taha.
  Gradual Typing for Functional Languages.
  Scheme and Functional Programming Workshop.
- Alan Kay.
  What did Alan Kay mean by, “Lisp is the greatest single programming language ever designed”?
  Quora.