Programming Languages through the Lens of Program Generation

Jacques Carette
joint work with Bianca Curutan

McMaster University

WG 2.11 – June 2013

Quicksort – in C

```
void quickSort( int a[], int l, int r) {
   int j;
   if( l < r ) {
       j = partition( a, l, r);
       quickSort( a, l, j-1);
       quickSort( a, j+1, r);
   }
}
int partition( int a[], int l, int r) {
   int pivot, i, j, t;
   pivot = a[l];
   i = l; j = r+1;
   while( 1 ) {
       do ++i; while( a[i] <= pivot && i <= r );
       do --j; while( a[j] > pivot );
       if( i >= j ) break;
       t = a[i]; a[i] = a[j]; a[j] = t;
   }
   t = a[l]; a[l] = a[j]; a[j] = t;
   return j;
}
```

Quicksort – in Haskell

```
import Data.List (partition)

quicksort [] = []
quicksort (x:xs) = quicksort less ++ (x:quicksort rest)
    where (less, rest) = partition (< x) xs
```

Quicksort – in λ-Prolog

```
quicksort([], []).

quicksort([H | T], S) :-
  partition(>(H), T, L, R),
  quicksort(L, SL),
  quicksort(R, SR),
  append(SL, [H | SR], S).
```

Quicksort – in Scheme

```
(define pivot (lambda (l)
  (cond ((null? l) 'done)
        ((null? (cdr l)) 'done)
        ((<= (car l) (cadr l)) (pivot (cdr l)))
        (#t (car l)))))

(define partition (lambda (piv l p1 p2)
  (if (null? l) (list p1 p2)
     (if (< (car l) piv) (partition piv (cdr l) (cons (car l) p1) p2)
         (partition piv (cdr l) p1 (cons (car l) p2))))))

(define (quicksort l)
 (let ((piv (pivot l)))
   (if (equal? piv 'done) l
     (let ((parts (partition piv l () ())))
       (append (quicksort (car parts))
               (quicksort (cadr parts)))))))
```

Quicksort – in Scala I

```
class QuickSort[A <% Ordered[A]] extends SortingAlgorithm[A] {
    def sort(l:List[A]): List[A] = {
      if(l.size <= 1) return l
      val pivot = l.head
      val (lt,gt) = l.tail.partition( _ <= pivot)
      sort(lt) ::: List(pivot) ::: sort(gt)
    }
}
```

Quicksort – in Scala II

```
class QuickSort[A <% Ordered[A]] {
  def sort(xs: Array[A]): Array[A] {
    def swap(i: Int, j: Int) {
      val t = xs(i); xs(i) = xs(j); xs(j) = t
    }
    def sort1(l: Int, r: Int) {
      val pivot = xs((l + r) / 2) var i = l; var j = r
      while (i <= j) {
        while (xs(i) < pivot) i += 1
        while (xs(j) > pivot) j -= 1
        if (i <= j) {
          swap(i, j)
          i += 1
          j -= 1
        }
      }
      if (l < j) sort1(l, j)
      if (j < r) sort1(i, r)
    }
    sort1(0, xs.length - 1)
  }
}
```

Quicksort – in Haskell II

```
partition arr left right pivotIndex = do
    pivotValue <- readArray arr pivotIndex
    swap arr pivotIndex right
    storeIndex <- foreachWith [left..right-1] left (\i storeIndex -> do
        val <- readArray arr i
        if (val <= pivotValue)
            then do
                swap arr i storeIndex
                return (storeIndex + 1)
            else do return storeIndex )
    swap arr storeIndex right
    return storeIndex
qsort arr left right = when (right > left) $ do
    let pivotIndex = left + ((right-left) `div` 2)
    newPivot <- partition arr left right pivotIndex
    qsort arr left (newPivot - 1)
    qsort arr (newPivot + 1) right
swap arr left right = do
    leftVal <- readArray arr left
    rightVal <- readArray arr right
    writeArray arr left rightVal
    writeArray arr right leftVal
```

And others too

- Hello World
- max/min
- powering
- fibonacci
- factorial
- gcd/lcm
- palindrome
- Gaussian Elimination
- λ-calculus interpreter

Will the real Quicksort please stand?

- Same:
  1. Data traversal
  2. Divide and conquer strategy
  3. End result (a sorted container of data)

Will the real Quicksort please stand?

- Same:
  1. Data traversal
  2. Divide and conquer strategy
  3. End result (a sorted container of data)
- Different:
  1. Language
  2. Data Structure
  3. Pivot choice (first, middle, . . .)
  4. Partition: In-place or Copy
  5. Partition: recursive or iterative
  6. Use of library functions
  7. Representation of current 'focus' (implicit/explicit)

(and the differences are not independent)

What we have

A ((pretty) source) code generator,

- written in Haskell,
- for: quicksort (plus partition, swap, concat), Hello World, max/min, powering, fibonacci, factorial, gcd/lcm, palindrome.
- producing: C, C♯, Java, Lua, Lisp, Haskell, LambdaProlog,
- parametrized by design decisions.

What we have

A ((pretty) source) code generator,

- written in Haskell,
- for: quicksort (plus partition, swap, concat), Hello World, max/min, powering, fibonacci, factorial, gcd/lcm, palindrome.
- producing: C, C♯, Java, Lua, Lisp, Haskell, LambdaProlog,
- parametrized by design decisions.

Where the "program" input is roughly

```
quicksort(items)
  if items.length < 1 return items
  choose index
  pivot := items[index]
  partition pivot less rest
  return quicksort(less) ++ (pivot ++ quicksort(rest))
```

Full disclosure: current Haskell output

Choices: Pivot = head, DS = list, lib = [Part, Concat]

```
module QuicksortProg where
import Data.List (partition)
quicksort items len =
  if (<) len 1
    then items
    else quicksortRec items 0 (len-1)
quicksortRec items leftIndex rightIndex =
  if (>=) leftIndex rightIndex
    then items
    else
      let pivot = (head items) in
      let (less,rest) = partition (< pivot) (tail items) in
      quicksortRec less 0 ((-) (length less) 1)
        ++ (pivot:quicksortRec rest 0 ((-) (length rest) 1))
```

Generalizing

$$\text{program family} = \text{concepts} + \text{design choices} + \text{staging}$$

Generalizing

$$\text{program family} = \text{concepts} + \text{design choices} + \text{staging}$$

Concepts present in implementation and theory of quicksort:

- (indexable) container
- ordering / comparison
- partition, permutation, pivot
- choice (of an index)
- data traversal
- memory access (for both read and write)
- (baked in) divide and conquer strategy

Generalizing

$$\text{program family} = \text{concepts} + \text{design choices} + \text{staging}$$

Design choices:

- Implementation Language
- Data Structure (list ,array)
- Pivot choice (first, middle, . . .)
- Partition: recursive or iterative
- Use of library functions for *partition* and/or *concat*

Generalizing

$$\text{program family} = \text{concepts} + \text{design choices} + \text{staging}$$

Staging

- Hard to describe – no good theory of heterogeneous staging
- 5 "stages": meta-theory, concept, design, code, run-time.
- Most important part - the weaving instructions:

```
quicksort(items)
  if items.length < 1 return items
  choose index
  pivot := items[index]
  partition pivot less rest
  return quicksort(less) ++ (pivot ++ quicksort(rest))
```

What's wrong with current PLs?

Some are too low level

- must worry about memory use and traversal
- cannot write generic code
- rigid syntax
- no access to (convenient) high-level concepts

Some are too high level

- cannot worry about memory use and traversal
- hard to write case-efficient code
- so much overloading that code is impossible to read
- high-level concepts cost too much

What's wrong with current PLs?

Some are too low level

- must worry about memory use and traversal
- cannot write generic code
- rigid syntax
- no access to (convenient) high-level concepts

Some are too high level

- cannot worry about memory use and traversal
- hard to write case-efficient code
- so much overloading that code is impossible to read
- high-level concepts cost too much

Opinion: largely caused by:

- no proper understanding of staging
- too high a focus on general purpose

Opportunities

Your Language is my Assembler

Opportunities

Your Language is my Assembler

Concept language and library

Opportunities

Your Language is my Assembler

Concept language and library

Explicit Language of Design

Opportunities

Your Language is my Assembler

Concept language and library

Explicit Language of Design

Proper language of Algorithms