Origami: (un)folding the abstraction of recursion schemes for program synthesis

Matheus Fernands, Emilio Francesquini, **Fabrício Olivetti de França**

Federal University of ABC
Center for Mathematics, Computation and Cognition (CMCC)
Heuristics, Analysis and Learning Laboratory (HAL)

01 June 2023

Origami: Program Synthesis by (un)Folding

Generalization: the swiss knife of programmers

- Common patterns observed in different programs: make it generic and reusable!
- Focus on the specificities instead of the repetitive part

Origami Programming

A very general (and unknown) programming paradigm called **Origami Programming** exploits the combination of the recursion schemes of **consuming** (fold) and **producing** (unfold) values to implement algorithms.

Advantages

- The recursion (of the consumer) will always terminate!
- Focus on the important part of the program
- Can be automatically optimized
- Easier to visualize and describe the program flow

Objective

Simplify the program synthesis problem to exploit these abstractions and simplifying the search through two steps:

1. Pick a pattern
2. Evolve specific parts of this pattern

```
data List b a = Nil | Cons b a
                  ^          ^ ^
                  1          2 3
```

1. Null list
2. Current element of the list
3. Current accumulated value of the folding process

Catamorphism - folding right

This is the `fold-right` function: the right associative folding.

Sub-patterns of catamorphism

Considering algorithms involving lists:

- **Reducing:** Type signature `[a] -> b`, consumes all of the elements until only a single value remains.
- **Mapping:** Type signature `[a] -> [b]`, applies a function to every element of our container.
- **Function:** Type signature `[a] -> [a] -> b`, it takes two containers and returns a value of type `b`.
- **Composition:** Composition of any of the previous patterns, e.g., `[a] -> (b, [c])`.

Reducing to a single element

Syllables (A 1) *Given a string containing symbols, spaces, digits, and lowercase letters, count the number of occurrences of vowels (a, e, i, o, u, y) in the string and print that number as X in The number of syllables is X.*

Reducing to a single element

```
syllables :: String -> Int
syllables xs = cata alg xs

alg Nil = e1 -- this can only be a constant
alg (Cons x xs) = e2
```

Reducing to a single element

```
syllables :: String -> Int
syllables xs = cata alg xs

alg Nil = 0
-- e2 :: Char -> Int -> Int
-- x :: Char
-- xs :: Int
alg (Cons x xs) = e2
```

Reducing to a single element

```
syllables :: String -> Int
syllables xs = cata alg xs

alg Nil = 0
alg (Cons x xs) = if isVowel x then xs + 1 else xs
```

Reducing to a single element

In python:

```
def syllables(my_string):
    count = 0
    for c in my_string:
        if isVowel(c):
            count = count + 1
     return count
```

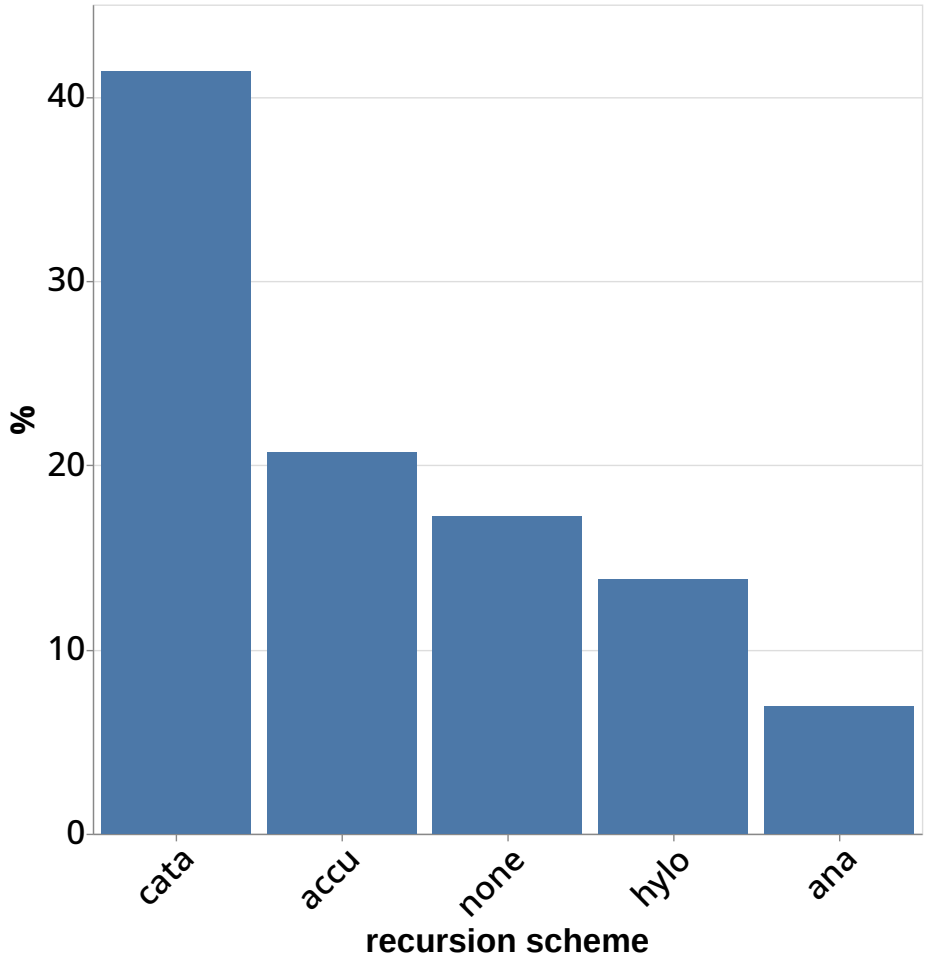

Figura 1: Distribution of recursion schemes used to solve the GPSB problems.

HOTGP

We adapted HOTGP to evolve just the main function of **reducing**, **mapping**, **composition** patterns in catamorphism. For this experiment we provided the choice of pattern and initial values.

	Origami	HOTGP	DSLS	PushGP	G3P	CBGP	G3P+
count-odds	**95**	50	11	8	12	4	0
double-letters	**91**	0	50	6	0	0	–
negative-to-zero	**100**	**100**	82	45	63	24	99
replace-space-with-newline	63	38	**100**	51	0	29	0
scrabble-score	**100**	–	31	2	2	1	–
string-lengths-backwards	94	89	**95**	66	68	20	–
syllables	**66**	0	64	18	0	53	–

Recursion Schemes in this paper

- **Catamorphism**:
  - Reducing
  - Mapping
  - Function
  - Composition
- **Anamorphism**
- **Hylomorphism** (cata followed by ana)
- **Accumorphism** (fold-left):
  - Indexed traversal
  - Catamorphisms with post-processing

Other beasts

- **Para / Apo:** folding and unfolding with access to downward elements of the structure.
- **Mutu / Comutu:** fold with mutual recursion (even/odd) or unfolding a seed into two or more structures.
- **Histo / Dyna / Futu:** folding with access to sub-results, dynamic programming, and generating multiple layers.
- **Indexed catamorphism:** support to nested data structures and type-safe structures
- **Monadic morphisms:** same morphisms with support to monads (e.g., state handling, IO, etc.)

Future steps

- Keep solving benchmark problems using recursion schemes
- Implement a simple breadth-search that searches all patterns in parallel
- Add support to different recursion schemes and base structure

Questions

Spare slides

Generating a Function

Super Anagrams (P 7.3) Given strings x and y of lowercase letters, return true if y is a super anagram of x, which is the case if every character in x is in y. To be true, y may contain extra characters, but must have at least as many copies of each character as x does. \end{displayquote}

Generating a Function

```haskell
superAnagram :: String -> String -> Bool
superAnagram = cata alg . fromList
  where
    alg NilF ys = True
    alg (ConsF x xs) ys = (not.null) ys && elem x ys && xs (dele
```

In Python:

```python
def superAnagram(xs, ys):
    for x in xs:
        if not null(ys) and x in ys:
            ys.delete(x)
        else:
            return False
    return True
```

When you started off with nothing: anamorphism

This pattern represents the unfolding:

```
program :: a -> g b
program = fromFix . ana coalg

coalg :: a -> f b
coalg arg = case e0 arg of
              EndCase -> Pattern1
              Case1   -> Pattern2 e1_1 e1_2 ... e1_m
              ...
              CaseN   -> PatternN eN_1 eN_2 ... eN_m
```

Anamorphism

Digits (A 6) *Given an integer, print that integer's digits each on their own line starting with the least significant digit. A negative integer should have the negative sign printed before the most significant digit.*

Anamorphism

```
digits :: Int -> [Int]
digits x = toList $ ana coalg x
  where
    coalg x =
      case x == 0 of
        True -> NilF
        False -> ConsF (if abs x < 10
                        then (x `rem` 10)
                        else abs (x `rem` 10))
                       (x `quot` 10)
```