Stochastic Synthesis of Recursive Functions Made Easy with Bananas, Lenses, Envelopes and Barbed Wire

Krzysztof Krawiec[1] and Jerry Swan[2]

Poznan University of Technology, University of York

1. krawiec@cs.put.poznan.pl
2. jerry.swan@york.ac.uk

LambdaDays'19

March 14, 2019

The background

Program synthesis: automatic generation of programs (functions) from

- examples (tests, cases, input-output pairs), or
- formal specifications (e.g., contracts), or
- other forms of *user's intent*.

Success stories:

- FlashFill[1], reinventing existing algorithms, discovering unknown algorithms, new hardware designs ...

[1] Gulwani 2011.

Outline

- The *FP community* has long been interested in *principled methods* of program generation for *property based testing*, e.g. QuickCheck[2] and SmallCheck[3] in Haskell, and the Scala analog ScalaCheck[4].
- Conversely, the *Metaheuristics community* has long used *stochastic search* for generating programs according to a *quality measure*, using e.g. Genetic Programming[5], Ant Programming[6], 'Estimation of Distribution' Programming etc.
- This talk describes a *hybrid approach*, using principled methods to provide a skeleton for metaheuristic search.

[2] Claessen and Hughes 2000.
[3] Runciman, Naylor, and Lindblad 2008.
[4] Nilsson 2014.
[5] Koza 1992.
[6] Roux and Fonlupt 2000.


WILL CODE
HTML For
FOOD

Synthesizing recursive programs

Challenges:

- Testing (executing) an ill-formed recursive program may lead to infinite sequence of nested calls.
- Recursive programs are particularly brittle: a minor modification may impact program's behavior on multiple tests, or worse - render it ill-formed.

Our contribution: Mitigating these problems by structuring/constraining the generate-and-test approach with formalisms known from FP:

- Algebraic Data Types
- Recursion schemes.

Algebraic data types (ADTs)

Defining new data types from existing ones S and T:

1. *Disjoint union*: the type containing *either* an instance of S *or* an instance of T, denoted $S + T$.
2. *Cartesian product*: denoted $S \times T$, the type of pairs (s, t), where s is of type S, and t is of type t.
3. *Exponentiation*: the type of functions from S to T, denoted T^S.

ADT for list of integers

- Haskell ADT:

```
data IntList = Nil | Cons Int List
```

- Scala ADT for a list of integers and recursive *length* function:

```
sealed trait IntList
case class Nil() extends IntList
case class Cons(head: Int, tail: IntList) extends
    IntList

def length(l: IntList): Int = l match {
  case Nil() ⇒ 0
  case Cons(head,tail) ⇒ 1 + length(tail)
}
```

Beyond lists

Lists are well-known data structures that are 'obviously composite'.

- However, virtually all familiar datatypes have such an inductively-definable nature and can be thus be conveniently expressed with ADTs.

Beyond lists

Lists are well-known data structures that are 'obviously composite'.

- However, virtually all familiar datatypes have such an inductively-definable nature and can be thus be conveniently expressed with ADTs.

Example: ADT for *Nat*

```
sealed trait Nat
case class Zero() extends Nat
case class Succ(pred: Nat) extends Nat
```

Recursion schemes

Recursion schemes ‘externalize’ recursion, i.e. replace *explicit* recursion with *implicit* recursion.

Fold for a list of integers and implicitly recursive *length* function:

```
def foldList[A](l: IntList,
    nilCase: A,
    consCase: (Cons,A) ⇒ A):A = l match {
  case Nil()       ⇒ nilCase
  case Cons(x,xs) ⇒
    consCase(Cons(x,Nil()),foldList(xs,nilCase,consCase))
}

def lengthConsCase(c: Cons, acc: Int):Int = 1 + acc

def length(l: IntList): Int =
  foldList(l, 0, lengthConsCase)
```

Catamorphisms

Catamorphism = one of most common recursion schemes.

For brevity, often denoted via 'banana-bracket' notation[7]:

$$(\!| case_1, \ldots, case_n |\!) \tag{1}$$

The *length* of a *List* is succinctly expressed as

$$(\!| 0, (l, accumulator) \mapsto 1 + accumulator |\!), \tag{2}$$

For *Nats*:

```
def cataNat[A](n: Nat,
  zeroCase: A,
  succCase: A ⇒ A): A = n match {
    case Zero()    ⇒ zeroCase
    case Succ(pred) ⇒ succCase(cataNat(pred, zeroCase,
        succCase))
}
```

[7] Meijer, Fokkinga, and Paterson 1991.

Program Synthesis with Recursion Schemes

Idea: combine ADTs with catamorphisms in a method for synthesizing recursive functions, in hope for:

- Improved effectiveness (by eliminating the non-terminating candidate programs)
- Improved efficiency (by providing the skeleton of the recursion scheme, and so constraining the search space)

Two phases:

1. Synthesis of case expressions
2. Synthesis of case callback functions

Phase 1: Synthesis of case expressions

- Requires domain-specific knowledge to inform the specific accumulator type to be used, e.g.
  - a single *Nat* for the length function,
  - pairs of *Nat*s for the Fibonacci function,
  - etc,
- For *recursive* ADTs the procedure requires a Category-Theoretic construction[8], but it is still automatable.

[8] Kocsis and Swan 2017b; Bird and Moor 1997.

Phase 2: Synthesis of case callback functions

- Synthesizing a callback function for each case independently.
- The candidate programs for each case *are non-recursive.*
- Search can be performed with any algorithm, e.g.,
  - systematic exact search,
  - heuristic search (stochastic or not).
- We engage our grammatical optimization tool ContainAnt[9], an algorithm configurator/optimiser:
  1. Derives the grammar of the 'DSL' from client code, via reflection, by analysing the fields/attributes (*val*) and method signatures (*def*)
  2. Performs search in the space of solutions defined by the grammar.

[9] Kocsis and Swan 2017a.

The grammar

Grammar of catamorphism cases for unary functions on *Nat*:

```
<CataNat>   ::= <CaseZero> <CaseSucc>
<CaseZero>  ::= <Nat>
<Nat>       ::= Zero | Succ <Nat>
<CaseSucc>  ::= <NatExpr>
<NatExpr>   ::= Const <Nat>
              | Var <Nat>
              | Add <NatExpr> <NatExpr>
              | Mul <NatExpr> <NatExpr>
              | PDiv <NatExpr> <NatExpr>
```

Toy Example: Synthesis of successor function

Solution sought: $(\!|1, n \mapsto n+1|\!)$, or equivalently in Scala:

```
def zeroCase(): Nat = Succ(Zero)
def succCase(n: Nat): Nat = Succ(n)
```

Set of examples $C = \{(0,1),(1,2),(3,4)\}$.

Toy Example: Synthesis of successor function

Solution sought: $(\!|1, n \mapsto n+1|\!)$, or equivalently in Scala:

```
def zeroCase(): Nat = Succ(Zero)
def succCase(n: Nat): Nat = Succ(n)
```

Set of examples $C = \{(0,1),(1,2),(3,4)\}$.

Phase 1:

1. Automatically derive case expressions from the definition of ADT *Nat*: *Zero* and *Succ(x)*.

Phase 2:

1. Partition C into:
   - $C_0 = \{(0,1)\}$, for the *Zero* case,
   - $C_1 = \{(1,2),(3,4)\}$, for the *Succ* case.
2. Apply *ContainAnt* to each of above problems independently.

Experiment

Benchmarks:

- Fib2: Fibonacci function
- Lucas: starts with 2 and 1 as the initial elements
- Pell: starts like Fibonacci, but $f_n = 2f_{n-1} + f_{n-2}$
- Fib3: starts with 0, 0 and 1 and sums *three* preceeding elements
- OddEvens: returns zeros and ones alternately for odd- and even-depth recursive calls

Function (operator) set for program search:

`Succ`	Successor function $m \mapsto m + 1$
`Add`	Addition
`Mul`	Multiplication
`PDiv`	Protected division

Results

Benchmark	Number of successful runs (out of 50)				
	GE	CTGGP	PushGP	Cata-RS	Cata-AP
Fib2	40	50	7	50	50
Fib3	3	50	13	50	50
Lucas	8	50	13	50	50
OddEvens	50	50	50	50	50
Pell	41	50	0	50	50

- Similar performance on: Sum, Square, Cube, Power(2,n)
- Cata-RS and Cata-AP visit fewer candidate solutions on average (lower computational effort)
- Statistically significant differences

To appear in *'Genetic Programming and Evolvable Machines'*: `https://link.springer.com/journal/10710`

Conclusions

ADTs + Recursion Schemes = Effectiveness *and* efficiency of synthesis.

- Problems decomposed and 'structurized' to the extent that makes them solvable with random search.

Prospects:

- Other ADTs.
- Other recursion schemes.
- Optimization of non-functional properties of program execution.

ADTs and catamorphism for (generic) binary trees:

```
sealed trait Tree[A]
case class Leaf[A](value: A) extends Tree[A]
case class Node[A](l: Tree[A], r: Tree[A]) extends Tree[A]

def cataTree[A,R](arg: Tree[A], leafCase: A ⇒ R, nodeCase:
    (R, R) ⇒ R): R = arg match {
  case Leaf(value) ⇒ leafCase(value)
  case Node(l, r) ⇒ nodeCase(
    cataTree(l,leafCase,nodeCase),
    cataTree(r,leafCase,nodeCase) )
}
```

Applications to other recursion schemes

- Anamorphisms: constructing an instance of ADT from a value (the opposite to catamorphisms)
  - Example: *downfrom*, $f(n) = [n, n-1, \ldots, 1]$
  - In 'lens brackets' notation:

$$n, m \mapsto \text{if m is 0 then None else } (m, m-1)$$

- Hylomorphisms: an anamorphism followed by a catamorphism
  - Example: factorial.
  - In 'envelope brackets' notation:

$$[\![(1, *), \text{downfrom}]\!]$$

- Paramorphisms: similar to catamorphisms, but have access to entire substructures on which the recursive call is made.
  - Convenient for expressing factorial:

$$1, (n, m) \mapsto (1+n) * m$$

- Zygomorphisms, futumorphisms, chronomorphisms, Elgot (co)algebras ...[10]

[10] Hinze, Wu, and Gibbons 2013.

References I

[1] A. E. I. Brownlee, N. Burles, and J. Swan. "Search-Based Energy Optimization of Some Ubiquitous Algorithms". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 1.3 (2017), pp. 188–201. ISSN: 2471-285X. DOI: 10.1109/TETCI.2017.2699193.

[2] Zoltan A. Kocsis and Jerry Swan. *Dependency Injection for Programming by Optimization*. 2017. URL: http://arxiv.org/abs/1707.04016.

[3] Zoltan A. Kocsis and Jerry Swan. "Genetic Programming + Proof Search = Automatic Improvement". In: *Journal of Automated Reasoning* (2017). ISSN: 1573-0670. DOI: 10.1007/s10817-017-9409-5.

[4] Zoltan A. Kocsis, John H. Drake, Douglas Carson, and Jerry Swan. "Automatic Improvement of Apache Spark Queries Using Semantics-preserving Program Reduction". In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. GECCO '16 Companion. Denver, Colorado, USA: ACM, 2016, pp. 1141–1146. ISBN: 978-1-4503-4323-7. DOI: 10.1145/2908961.2931692. URL: http://doi.acm.org/10.1145/2908961.2931692.

[5] Nathan Burles, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, Jerry Swan, and Nadarajen Veerapen. "Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guava". In: *Search-Based Software Engineering*. Ed. by Márcio Barros and Yvan Labiche. Cham: Springer International Publishing, 2015, pp. 255–261. ISBN: 978-3-319-22183-0.

[6] R. Nilsson. *ScalaCheck: The Definitive Guide*. IT Pro. Artima Press, 2014. ISBN: 9780981531694.

[7] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. "Unifying Structured Recursion Schemes". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: ACM, 2013, pp. 209–220. ISBN: 978-1-4503-2326-0. DOI: 10.1145/2500365.2500578. URL: http://doi.acm.org/10.1145/2500365.2500578.

[8] Sumit Gulwani. "Automating string processing in spreadsheets using input-output examples". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 317–330. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926423. URL: http://doi.acm.org/10.1145/1926385.1926423.

[9] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. "Smallcheck and lazy smallcheck: automatic exhaustive testing for small values". In: *Haskell*. ACM, 2008, pp. 37–48.

References IV

[10] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *ICFP*. ACM, 2000, pp. 268–279.

[11] Oliver Roux and Cyril Fonlupt. "Ant Programming: Or How to Use Ants for Automatic Programming". In: *ANTS'2000 From Ant Colonies to Artificial Ants: 2nd International Workshop on Ant Algorithms*. Ed. by Marco Dorigo. 2000.

[12] Richard Bird and Oege de Moor. *Algebra of Programming*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997. ISBN: 0-13-507245-X.

[13] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[14] Erik Meijer, Maarten Fokkinga, and Ross Paterson. “Functional programming with bananas, lenses, envelopes and barbed wire”. In: *Functional Programming Languages and Computer Architecture: 5th ACM Conference Cambridge, MA, USA, August 26–30, 1991 Proceedings*. Ed. by John Hughes. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 124–144. ISBN: 978-3-540-47599-6. DOI: 10.1007/3540543961_7.