Stochastic synthesis of recursive functions made easy with bananas, lenses, envelopes and barbed wire

Jerry Swan¹ · Krzysztof Krawiec² · Zoltan A Kocsis³

Received: 1 November 2018 / Revised: 17 February 2019 / Published online: 14 March 2019
© The Author(s) 2019

Abstract

Stochastic synthesis of recursive functions has historically proved difficult, not least due to issues of non-termination and the often *ad hoc* methods for addressing this. This article presents a general method of implicit recursion which operates via an automatically-derivable decomposition of datatype structure by cases, thereby ensuring well-foundedness. The method is applied to recursive functions of long-standing interest and the results outperform recent work which combines two leading approaches and employs ‘human in the loop’ to define the recursion structure. We show that stochastic synthesis with the proposed method on benchmark functions is effective even with random search, motivating a need for more difficult recursive benchmarks in future.

Keywords Program synthesis · Algebraic Data Types · Recursion schemes · Catamorphisms · Pattern matching

1 Introduction

Synthesis of recursive programs is of long-standing interest in the Genetic Programming (GP) community [18], with numerous novel approaches (e.g. [1,8,26,29,34], plus others discussed in Sect. 4). The majority of them have been concerned with *explicit* recursion, i.e. the recursion expressed directly within the body of the synthe-

✉ Jerry Swan
jerry.swan@york.ac.uk
Krzysztof Krawiec
krawiec@cs.put.poznan.pl
Zoltan A Kocsis
zoltan.kocsis@existence.property

¹ Department of Computer Science, University of York, YO10 5GH York, UK

² Institute of Computing Science, Poznań University of Technology, Poznań 60-965, Poland

³ School of Mathematics, University of Manchester, M13 9PL Manchester, UK

sised code. In contrast, previous work by Yu et al. [38–40] demonstrated the utility of *implicit* recursion, i.e. the control flow of the recursion is orchestrated by a specific template of individually handled *cases* and no recursive calls are made by the synthesized code: they are instead delegated to an external, fixed *combinator*, i.e. a stateless function that factors out the recursion pattern.

The implicit approach has several advantages: firstly, it can ensure that the recursion is well-founded, thereby avoiding the issue of non-termination. Secondly, the search space of the implicit case is likely to be smaller than in the explicit approach, since the code for partitioning the recursion into base and alternative cases is provided by the template. Also, the cases constrain the list of fitness cases (examples) that can be used in GP-based search, thereby reducing the computational expense. Yu's method of implicit recursion was applied to the *List* data-type via its associated *fold* method. The *fold* method of *List* is a higher-order function that takes as argument a callback function used to accumulate information as the *fold* traverses the list. However, though *fold* can express a surprisingly wide range of functions [15], it realises only one specific *recursion scheme*, i.e. it does not implement all possible ways in which recursion may be conducted.

In this article, we describe how a generalisation of this familiar *fold* on lists can be obtained for *all inductively defined datatypes*, and how fold is one of a variety of different recursion schemes (Sect. 2) with different computational expressiveness. We use these recursion schemes as a basis for inducing several widely-studied recursive functions using stochastic heuristic search. The proposed approach (Sect. 3) can (1) automatically derive recursion schemes from the datatype declaration and (2) produce programs that are guaranteed to issue valid recursive calls. When applied to a range of benchmarks (Sect. 6), it (3) robustly produces recursive programs that pass all tests and generalise well, and does so in a smaller number of evaluations than a reference approach.

2 Structural recursion

A recursive function is a computational scheme for constructing a *value* of a certain *type* in a stepwise, compositional way, i.e. via a range of recursive calls. For instance, the factorial of n is composed by gradual accumulation of products of the numbers from 1 to n . As much as this observation seems trivial, it ceases to be such once one starts to express such compositions with the convenient formalism known as Algebraic Data Types (ADTs). With ADTs, each value of a given type is not just an 'anonymous' element in a set, with no obvious relationship to the other elements (e.g. the value 2 in a 'flat' set of integers, meant just as an unstructured 'bag' of elements). Rather, a value is a *combinatorial data structure* that captures the compositional nature of that formal object: e.g. the fact that 2 is the *third* natural number and hence requires exactly *two applications* of a *successor function* to the number zero. Crucially, by considering values as combinatorial entities, the (by definition inductive) structure of ADTs naturally relates to the ways in which such structures can be constructed and processed, i.e. recursive functions. Also, the theory of ADTs reveals that, for most

familiar types, just a handful of elementary compositions suffices to express all values and so facilitate a rich repertoire of ways in which they can be processed.

For these and other reasons, ADTs are ubiquitous in functional programming languages and type theory. The sections that follow introduce the basic concepts of the ADTs and link them to recursion schemes.

2.1 Algebraic Data Types

The most familiar example of an ADT is *List*, which may either be constructed as the empty list (*Nil*), or else as some element to be concatenated with an existing list. However, all familiar datatypes may be represented as ADTs (indeed, these are the only datatypes in the Haskell language, for example).

Formally, there are three fundamental constructions for 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 $S + T$. For those used to the object-oriented (OO) perspective, this corresponds to *inheritance* (specialization) S and T can be considered as specializations of the type $S + T$.
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 . This construction corresponds to *composition* (aggregation) in OO programming: an object of the type $S \times T$ hosts objects of types S and T as its members.
3. *Exponentiation*: the type of functions from S to T , denoted T^S .

Listing 1 shows how ADTs can be represented even in a non-functional language such as Java: an *IntList* is either empty (*Nil*) or else constructed (*Cons*) from the concatenation of an integer value and some pre-existing list. *IntList* is therefore the disjoint union of *Nil* and *Cons*, or more formally, using the above notation: $IntList = Nil + Cons$, while $Cons = int \times IntList$. For simplicity of exposition, we discuss *IntList* rather than the more typical generic notion of *List*, which could be instantiated with an arbitrary element type, for example as a list of integers or a list of strings. Generic recursion schemes over arbitrary data types are discussed in Sect. 8.

In languages such as Haskell or Scala, *IntList* can be represented more succinctly. Listing 2 gives the analogous Scala code for *IntList*, together with a recursive version of the *length* function, implemented via *pattern match* against *cases*. Pattern matching can be seen as a mechanism that is complementary to the above presentation of ADTs as a means of *constructing* elements of a type, in allowing the *deconstruction* of a given object into its constituent components. Values can be matched against atomic values (like the case *Nil()* in Listing 2) or against object ‘templates’, like case *Cons(head,tail)*. Crucially, in the latter case, the constituents of the matched value (*head,tail*) become available as values that can undergo further processing.

```

interface IntList {}

class Nil implements IntList {}

class Cons implements IntList {
    final int    head;
    final IntList tail;

    public Cons(int head, IntList tail){
        this.head = head;
        this.tail = tail;
    }
}

```

Listing 1 Java ADT for a list of integers.

The above approach is applicable to arbitrary ADTs, not just *List* (for which this sort of construct is more widely known). Moreover, it is possible for compiler to statically determine if a set of cases which pattern-match against an ADT is *exhaustive*. This capability is important, as it not only guarantees that all possibilities are being handled, but, as we will show later, enables us to ensure that recursion is well-founded.

The implementation of the *Cons* case in Listing 2 exemplifies the gradual *accumulation* of the result when execution traverses the recursive structure of the underlying ADT. Such accumulation is present in all recursive functions. The core of the approach presented in this paper is that this *can be formulated in abstraction from a specific implementation of some recursive function*. The arguably simplest and most well-known instance of this observation is the *fold* operation applied to lists. Folding a list of elements of some type (like *Int* in our ongoing example) into a value of some generic *accumulator type A* (to be specified by the caller of the function) requires two components:

- a value of type *A* to be returned when the input list is empty (which is typically a neutral element of type *A*), and
- a function with signature $(List, A) \implies A$ that accumulates the values of type *A* as the computation progresses along the list.

Listing 3 gives an alternative (w.r.t. to Listing 2) implementation of *length* using *fold* for *IntList*. As can be seen, it orchestrates the recursion via a case-specific template *foldList* [37], requiring the user of the function to provide three things: the list *l* to which the *fold* is to be applied; a value *nilCase* of type *A* to be returned when the *Nil* case is encountered, and a binary function *consCase* : *Cons* $\times$ *A* $\rightarrow$ *A* to be applied in all other cases.

Once *foldList* is defined, it can be instantiated with a specific components. For our *length* example, *A*=*Int*, *nilCase*=0 and *consCase* is given by the *lengthConsCase* function that simply increments the accumulator.

```
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)
}
```

Listing 2 Scala ADT for a list of integers and recursive *length* function. *trait* is equivalent to a Java interface; *case class* is one that can be used in a *match* clause for pattern matching; *sealed* ensures that all subclasses of *IntList* are known to the compiler (exhaustiveness).

```
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)
```

Listing 3 Fold for a list of integers and implicitly recursive *length* function. *A* is accumulator type.

2.2 Catamorphisms

As shown in the above examples, by factoring out recursion, *fold* replaces *explicit* recursion with its *implicit* use. In previous GP work by Yu et al., *List* folds were used together with synthesised callbacks represented via lambda functions and applied to the even-parity problem [40], to Fibonacci series, and to determine if a string is a substring of another [39].

However, *fold* on lists is actually a special case of a more general concept known as a *catamorphism*, which can be defined on all algebraic datatypes. The use of the prefix ‘cata’ (from the Greek $\kappa\alpha\tau\alpha$ —“downwards”) refers to the fact that the recursion ‘descends’ through the structure of the object to which it is applied, peeling away a layer of structure at each recursive invocation and applying a specified transformation to the object constructor (in the pattern-matching clause) representing that layer.

For example, the above calculation for *length* on the 2-element list [1, 2], represented by nested constructors *Cons*(1, *Cons*(2, *Nil*())), successively descends through cases *Cons*(1, *Cons*(2, *Nil*())), followed by *Cons*(2, *Nil*()) and then *Nil*.

For brevity, catamorphisms are typically denoted via ‘banana-bracket’ notation [22], of the form

$$\llbracket \text{case}_1, \dots, \text{case}_n \rrbracket, \quad (1)$$

where each case corresponds to a element of the disjoint union expressing the ADT—some of which can be atomic values, while some are functions that determine how

the outcome should be accumulated when the input value is being deconstructed by pattern matching. These elements correspond one-to-one to the consecutive clauses of the pattern matching implementation in our *length* example. The *length* of a *List* is succinctly expressed as

$$\llbracket 0, (l, accumulator) \mapsto 1 + accumulator \rrbracket, \quad (2)$$

corresponding directly to the cases in Listing 3. Notice that this notation implicitly assumes that *l* is being matched against the corresponding ADT constructor (here, *Cons(x, xs)*).

The domain of lists has the didactic advantage of explicitly involving construction/deconstruction of well-known data structures that are widely considered as composite, and so illustrates the underpinnings of ADTs in a down-to-earth manner. However, virtually all familiar datatypes have such a compositional nature and can be thus be conveniently expressed with ADTs—it is just that this fact is commonly ignored, not least because in contemporary hardware architectures, the values of many types (like *Int*) are more familiar in terms of low-level implementations that obscure their underlying compositional nature.

Listing 4 gives a Scala ADT corresponding to the Peano definition of *Nat*, the type of natural numbers, $\mathbb{N}$, viz. that a natural number is either zero or the successor of some natural number. The listing also provides the catamorphism for *Nat*. Familiar functions on *Nat* are readily expressed as catamorphisms: for example, multiplication *mul*(*n*, *m*) is $\llbracket 0, (pred, accumulator) \mapsto pred + m \rrbracket$.

Crucially, catamorphisms on *Nat* suffice to express *every primitive recursive function* [15].¹ The practical implication is that *many commonly used recursive functions defined of Nat can be expressed by calling cataNat with appropriate arguments*, in particular an appropriate accumulator function. For instance, the Fibonacci function is given by the following catamorphism:

$$\llbracket (0, 1), (a, b) \mapsto (b, a + b) \rrbracket. \quad (3)$$

Despite the simplicity of its well-known definition, the factorial function is more readily expressible via the alternative recursion schemes of Sect. 8 than via a catamorphism.

By factoring out recursion, catamorphisms are thus immensely expressive and cover a wide range of primitive recursive functions. No wonder they are considered more and more often as a part of *lingua franca* of functional programming, moving up the level of discourse and facilitating more efficient development of more robust software.

¹ Indeed, by allowing the result type *R* to include function types, even Ackerman's function can be expressed via catamorphisms [28].

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

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

Listing 4 ADT for *Nat* and the corresponding catamorphism

In Sect. 8, we give examples of catamorphisms for ADTs other than *List* and *Nat*.

If the cases provided to a catamorphism collectively define a total function (i.e. there is one case for each element in the disjoint union, and each case is itself total), then termination is guaranteed. Non-termination is a frequent source of difficulty in the synthesis of recursive functions, and has been often addressed with *ad hoc* methods devised over the years. The source of the problem is ill-formed infinite recursive calls, or in the more general sense, viz. that even a minute modification of a recursive function may drastically affect the course of recursion calls and so impair candidate program quality. That brittleness has been pointed to in numerous past works on GP for recursive functions (see, e.g., [4,23,24,40]). The well-foundedness of catamorphisms obviates this problem and addresses it in a principled manner.

3 Program synthesis via recursion schemes

As we showed in the previous section, catamorphisms (1) capture the underlying common recursion scheme for all primitive recursive functions [15], (2) guarantee well-foundedness by ensuring termination, and (3) form an overarching elegant formalism that facilitates abstraction. Given these advantages, it becomes tempting to employ ADTs and recursion schemes for heuristic program synthesis, in the hope of making it both more effective (by eliminating the non-terminating candidate programs) and efficient (by providing the skeleton of the recursion scheme, and so constraining the search space, relative to explicit approaches).

In this study, we propose a heuristic approach to program synthesis that uses ADTs and structural recursion to constrain the space of candidate solutions and so cope with the brittleness of recursion. Similarly to standard GP, the method learns inductively and thus requires *fitness cases* (tests), each of which is an input-output sample from the target function to be synthesized. The design of the method is dictated by the form of catamorphism (or, in general, any recursion scheme—see the discussion in Sect. 8), which is essentially a list of non-recursive functions, each meant to handle one of the pattern-matching cases (c.f. Eq. 1). We thus perform synthesis of the complete catamorphism-based implementation in two phases that follow.

Phase 1: Synthesis of case expressions In the first stage, given the ADT $\mathcal{T}$ of interest, we need to determine the ADT *case expressions* (pattern ‘matchers’, Eq. 1) that will be

used to match against the arguments of the synthesized function. The disjoint union of those expressions needs to be equal to $\mathcal{T}$, so that each value in $\mathcal{T}$ is matched by one and only one pattern matching case. This decomposition can be performed procedurally, as evidenced by, e.g., the relationship between the structure of *IntList* and the corresponding *foldList* cases of Listing 3. The decomposition is generic in the choice of accumulator type *A*. This therefore requires domain specific knowledge to inform the specific accumulator type to be used, e.g. a single *Nat* for the length function (Eq. 2), pairs of *Nats* for the Fibonacci function (Eq. 3), etc, as discussed further in Sect. 8. For *recursive* ADTs (such as the rational function *Expr* in Sect. 8) the procedure requires a somewhat technical Category-Theoretic construction [6, 17], but it is still nonetheless automatable.

Note that Phase 1 does not involve the fitness cases mentioned above.

Phase 2: Synthesis of case callback functions Once an ADT case expressions are available, it is then necessary to synthesize a function for each case. These case-specific *case callback functions* are supplied as arguments to the corresponding recursion scheme, which then represents a candidate solution that is evaluated on the fitness cases.

To synthesize the case callbacks we could in principle engage any type-aware variant of GP, or any other method capable of synthesizing programs from input-output examples. To demonstrate the usefulness of our approach in real-world settings and for a fully-fledged programming language, we use CONTAINANT [16]. CONTAINANT is an online algorithm configurator/optimiser that can optimise any measurable property of code, given a set of components defined by a context-free grammar. Rather than being specified explicitly, *the grammar is automatically derived via reflection from client code*, by analysing the fields/attributes (*val*) and method signatures (*def*) of a user-defined subclass of *containant.Module*.

To search the space of such solutions, CONTAINANT implements a range of strongly-typed metaheuristic search algorithms that guarantee candidate solutions to be consistent with the grammar. In this study, we employ *Ant Programming* [7], a variant of Ant-Colony Optimization (ACO) [9] in which the combinatorial structure traversed by the ‘ants’ is the tree of grammar productions. Typically for ACO, in each iteration solutions are generated from that structure and evaluated, pheromone traces corresponding to specific construction paths are updated and solutions are discarded. As a baseline, CONTAINANT includes also *Random Search*, which draws each solution independently by traversing grammar productions at random [16].

```

<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>

```

Listing 5 Grammar of catamorphism cases for unary function on Nat

We emphasise again that the individual case functions synthesized in Phase 2 are themselves non-recursive, i.e. the entirety of recursion required to solve the synthesis

task in question is captured by the underlying recursion scheme of catamorphism. This allows us to mitigate the ‘brittleness challenge’ mentioned earlier. Also, by virtue of the recursion schemes being derived from the structure of their associated ADTs, each candidate solution is guaranteed to well-behave in execution, i.e. to *always* issue correct recursive calls. For instance in the *Nat* example, calling the case callback *succCase()* for argument *Zero* is impossible by construction.

In the following, we illustrate the above two phases on the *Int* domain introduced earlier in this paper (Listing 4).

Example: Synthesis of successor function We consider the task of synthesizing the successor function on *Nat*: $\text{succ}(n) \mapsto n + 1$ (cf. Listing 4). The optimal (i.e. correct) solution to this problem is represented by the catamorphism $\llbracket 1, n \mapsto n + 1 \rrbracket$, and equivalently by two case callbacks:

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

Assume our set of fitness cases $C = \{(0, 1), (1, 2), (3, 4)\}$.

In Phase 1, the case expressions are automatically derived from the definition of type ADT *Nat* (Listing 4), resulting in two case expressions: *Zero* and *Succ(x)*.

In Phase 2, we first use the case expressions resulting from Phase 1 to partition the available fitness cases into subsets that will be used to synthesize the individual case functions. The necessity of this step should be obvious: clearly, when synthesizing a given case function, there is no point testing it on tests that it will be never applied to. In our particular problem, this step results in partitioning C into $C_0 = \{(0, 1)\}$ (for the *Zero* case) and $C_1 = \{(1, 2), (3, 4)\}$ (for the *Succ* case).

Each of these subsets defines a separate synthesis problem, which is subject to an independent run of *ContainAnt* that uses the grammar shown in Listing 5 and fitness function in Listing 6, which aggregates the errors on individual fitness cases using root mean square error. The case functions synthesized by *ContainAnt* are identical to those shown in the above listing of the correct solution. Together with the case expressions obtained in Phase 1 and with the catamorphism skeleton, they form the desired implementation of the *Succ* function. $\square$

```
def fitnessCataNatUnaryFn(fitnessCases: List[(Nat, Nat)],
    zeroCase: Nat, succCase: Nat  $\Rightarrow$  Nat): Double = {

    def synthesizedRecursiveFn(n: Nat): Nat =
        cataNat(input, zeroCase, succCase)

    val errors = fitnessCases.map {
        case (input, output)  $\Rightarrow$ 
            (output, synthesizedRecursiveFn(input))
    }
    return rmsError(errors)
}
```

Listing 6 Example-based fitness evaluation for a unary function of *Nat* represented by a catamorphism. *rmsError* calculates the root mean square error between the actual and desired output.

4 Related work

The merits of recursion schemes as a form of *implicit recursion* for guiding formal approaches to program transformation/induction on ADTs have been known for some years (e.g. [13,20]). In respect of *stochastic* program synthesis, apart from the seminal study by Yu et al. [40], there are hardly any works that involve implicit recursion. As the work of Yu et al. [40] was already covered earlier in this paper, here we review the *explicit* approaches, where recursive calls can appear directly within the body of synthesised code. In particular, we discuss the methods that are relevant to the approach proposed in this paper, and refer readers interested in a wider review of stochastic synthesis to of recursive functions to a recent survey [3].

In recent work [5], Alexander and Zacher proposed *Call-Tree-Guided Genetic Programming* (CTGGP). In contrast to conventional GP which normally expects problems to be specified with fitness cases, CTGGP requires the user to provide a *partial call tree* that reflects the structure of recursive calls, the arguments passed in those calls, and the corresponding values returned. CTGGP first datamines the tree, collecting the information on the arity of recursion, the number of base cases, and input-output pairs for individual nodes. This results in two grammars, one defining the arguments of recursive calls, and one describing the main body of the recursive function. The grammars are subsequently used to conduct a two-phase search with a variant of Grammatical Evolution (GE). The method is quite flexible, i.e. the return values do not have to be specified for each node of the call tree, and the tree can be disjoint. Typically a handful of tree nodes is sufficient to specify the task such that the correct program can be found within hundreds of evaluations.

A follow-up work by Alexander et al. [4] combined CTGGP with the *scaffolding* of Moraglio et al. [24]. In essence, scaffolding mitigates infinite recursion by resorting to fitness cases whenever the argument of a recursive call is present among them. For instance, consider a candidate program that calls itself with argument 3: if an input-output fitness case of the form (3, y) is present in the training set, in scaffolding that call will immediately return y rather than actually execute. Otherwise, the call is executed normally. In their follow-up work, Alexander et al. compared ‘vanilla’ GE, GE with scaffolding, CTGGP, and CTGGP with scaffolding. An assessment on several widely-used benchmarks (see Sect. 5) clearly demonstrated that CTGGP with scaffolding was most efficient.

Scaffolding was also found useful in a number of other studies, including the recent one by Moraglio and Krawiec on synthesis of recursive Boolean programs using semantic GP [23]. The authors demonstrated that, by constraining the class of recursive programs to *k-fold functions* and using scaffolding, a limited number of fitness cases (all base cases plus a number of subsequent cases) is sufficient to synthesize programs that are guaranteed to generalize correctly to all possible inputs.

It may be worth noting that the above-mentioned distinction between the base case and the recursive call cases is essential in virtually all methods reviewed in this section. In the context of this study, those cases correspond to individual elements of the disjoint union of the ADT of consideration (Phase 1 of the proposed approach, Sect. 3). As we will argue in Sect. 8.1, ADTs are however more general. In combination with recursion

schemes, they offer a more systematic and universal framework for capturing various types of recursion and guaranteeing its well-foundedness.

5 Benchmarks

The set of problems considered in stochastic synthesis of recursive functions is relatively small, with unary functions on natural numbers receiving the most attention. Factorial has been widely studied (e.g. [4,5,14,29,31,36]), but Fibonacci and its variants have probably attracted even more attention. Koza considered Fibonacci in his inaugural GP volume [18] and it has subsequently been tackled in many other works (e.g. [3–5,14,25,31,36,39]). Lucas, Pell and Fib3 (a.k.a. ‘Tribonacci’) are Fibonacci-like functions, each of which were studied by Alexander et al. [4,5] and the latter by Wilson and Heywood [36]. Lucas is a ‘shifted’ version of Fibonacci, differing only in using 2 and 1 as the initial elements. Pell starts from 0 and 1 like Fibonacci, but each subsequent element is $2f_{n-1} + f_{n-2}$. Fib3 sums the three preceding elements, starting with 0, 0 and 1. Some recursive unary functions (Factorial, Fib2, Power(2, n)) were also considered in works by Spector et al. [31] on autoconstructive evolution (i.e. the co-evolution of genetic operators in tandem with a solution to some base problem). Other works have also considered integer-valued functions such as Sum, Binomial [26], Square, Cube [14], Power(2, n) [31], Log2, and OddEvens [4,5]. The latter returns zeros and ones alternately for odd- and even-depth recursive calls.

6 Experiments

The goal of experiment is to compare the proposed approach (Sec. 3) to a state-of-the-art method, which we consider to be Alexander et al.’s CTGGP [4], described in Sect. 4.

The ‘traditional’ function set used for stochastic synthesis of recursive functions is the rational functions $\{+, -, *, \%\}$. % denotes ‘protected division’—in the event of the denominator being zero, it returns 1 [27]. The CTGGP function set does not include subtraction and (since we wish to perform as direct a comparison as possible) we do not include it in the function set of Table 1 either.

Previous experiments with CTGGP [4] used as benchmarks some of the most commonly-referenced unary functions from the *Nat* domain (as described in Sect. 5): Factorial, Fib2 plus Fib3 (and their variants Lucas and Pell), together with Log2 and OddEvens. As signaled earlier and discussed in greater detail in Sec. 8, Factorial is not readily expressible as a catamorphism [21] and neither is Log2 with the given function set. We therefore omit them from this study.

We therefore compare the methods on Fib2, Fib3, OddEvens, Lucas and Pell benchmarks. We replicate other details of the CTGGP setup [4] and use as the baseline two best of the four configurations reported there: ‘Vanilla’ Grammatical Evolution (referred to as *Plain* in [4]; **GE** in the following) and CTGGP combined with Scaffolding (referred to as *Combined* in [4]; **CTGGP** in the following). Alexander reports also the constituent variants: Scaffolding and CTGGP, but individually they fare worse

Table 1 Function set for program search

Name	Definition
succ	Successor function $m \mapsto m + 1$
Add	Addition
Mul	Multiplication
PDiv	Protected division

than when combined. Following the CTGGP setup, we conduct 50 trials of each configuration and report the number of correct answers/solutions and mean and maximum number of evaluations.

As for our approach, we employ it in two variants introduced in Sec. 3, which vary in the search algorithm used in Phase 2 to synthesize the case functions: Ant Programming (**Cata-AP**) and Random Search (**Cata-RS**). In both cases, we rely on the implementations from CONTAINANT [16].

The grammar defining the search space was the same for each of the search algorithms, and is given in Listing 5. As explained in Sect. 3, the grammar is automatically extracted from source code by CONTAINANT via reflective analysis of the Scala code that defines the corresponding ADT classes (e.g., case class Add(a:Nat, b:Nat) extends NatExpr).

The parameters for each of algorithms (and those common to all) are given in Table 2. The number of fitness cases was set to the minimal value that caused search algorithms find optimal solutions systematically, i.e. 8. In contrast, Alexander et al. used “5 or 6” cases (with an attendant reduction in the number of evaluations they required); however, it is not clear how generality was established there with this smaller number of cases. Maximum program depth determines the maximum depth of the derivation tree that the methods use to construct a candidate program by following grammar rules. The parameter names for AP refer to the corresponding notions in the Min-Max Ant System algorithm [16,33].

The evaluation budget was obtained from Alexander et al.’s use of 2 separate phases, with a population of 1000 for 300 generations. For fair comparison, the maximum possible number of evaluations was set to $2 * 1000 * 300 = 600,000$. As can be seen from Table 3, nothing approaching this value was ever reached for any benchmark except in the three cases where Cata-RS failed on the cube benchmark.

To provide an additional reference point, we also attempt to solve the benchmarks in question using PushGP [32] (**PushGP** in the following), the arguably most popular and continuously developed variant of stack-based GP. The runtime environment of PushGP comprises a code stack that stores the program to be executed, and a separate stack for each datatype. To execute a PushGP program, an interpreter repeatedly pops an instruction from the code stack and executes it, until the stack is depleted, upon which program is terminated. When executed, an instruction pops the required arguments from the type-compatible data stacks (e.g., two elements from the integer stack to be compared with the ‘<’ operator), and pushes the result on to the stack that is type-compatible with the output value (Boolean in this case). Upon program termination, the top element of the stack that is type-compatible with the desired output is fetched

Table 2 Parameters for program search

Common	
Number of runs	50
Number of fitness cases	8
Maximum program depth	3
Cata-AP	
Maximum pheromone	10
Evaporation rate	0.4
Minimum fraction	0.1
Number of ants	1
Evaluation budget	600,000
Cata-RS	
Evaluation budget	600,000

as the outcome of program execution. PushGP does not feature explicit recursion nor looping; rather than that, iterative computation is facilitated with combinators.

We conduct the PushGP runs using its most state-of-the-art implementation, Clojush.² For the parameters that can be reconciled with other algorithms compared here (number of runs, number of fitness cases), we set them to the values shown in Table 2. When it comes to the remaining parameters, we rely on the defaults used in Clojush integer regression example suite, more specifically the factorial example. The key parameter settings include: Lexicase selection [10,30] of parent candidates to mate, mutation rate of 0.05, limiting program length in initial population to 100, while allowing them to grow later to up to 500 instructions. A program is allowed to execute up to 1,000 instructions before being terminated. The complete list of parameters can be found under the link given in footnote.³ This includes also the list of 33 instructions used in these benchmarks, comprising, among others integer arithmetic, Boolean expressions, conditional statements, and the S, K and Y combinators.

7 Results

The results, presented in Table 3, are unanimous: our approach (Cata-AP and Cata-RS) not only manages to synthesize optimal recursive programs in each run, but systematically achieves that in a lower number of evaluations than GE, CTGGP, and PushGP. Strikingly, this holds not only for the quite sophisticated AP, but also for Random Search (RS), a memory-less stochastic trial-and-error. This clearly indicates that the case-by-case problem decomposition defined by the ADT catamorphism has reduced the search space such that finding the optimal program is relatively easy. Still,

² <https://github.com/lsector/Clojush>.

³ https://github.com/lsector/Clojush/blob/master/src/clojush/problems/integer_regression/factorial.clj.

Table 3 Experimental results

Benchmark	Number of successful runs			
	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
Benchmark	Mean number of evaluations			
	GE	CTGGP	PushGP	Cata-RS Cata-AP
Fib2	53,168	1081	288,800 ± 10,103	449 ± 122 418 ± 329
Fib3	117,875	10,347	278,140 ± 13,747	10,301 ± 2444 5722 ± 3722
Lucas	105,663	1622	275,780 ± 15,400	1116 ± 287 699 ± 494
OddEvens	539	255	14,480 ± 2754	81 ± 28 26 ± 22
Pell	56,240	1879	300,000 ± 0	1827 ± 401 544 ± 339
Benchmark	Max number of evaluations			
	GE	CTGGP	PushGP	Cata-RS Cata-AP
Fib2	130,923	3617	300,000	1615 1200
Fib3	125,723	40,771	300,000	41,812 14,226
Lucas	123,455	7070	300,000	4607 1958
OddEvens	2069	885	62,000	558 112
Pell	129,823	4904	300,000	6104 1414

Bold values indicate the best performing methods

Mean number of evaluations for *Cata* accompanied with 0.95 confidence intervals

Results for GE and CTGGP quoted from [4] (where the former was referred to as *Plain*, and the latter as *Combined*)

the fact that Ant Programming is somewhat faster on all these benchmarks suggests that driving search with fitness does bring some benefits.

For Fib2, Lucas and OddEvens, the confidence intervals for *Cata* are narrow enough to assume that its performance is significantly better than that of GE and CTGGP. For Fib3 and Pell, this is not the case (though *Cata*-AP seems close to significance for the latter benchmark). Though the intervals could be narrowed by conducting more runs of *Cata* configurations, that has not been done in order to ease side-by-side comparison with Alexander et al. [4] (where 50 runs were used). The Wilcoxon one-sided rank test on the mean number of evaluations applied to CTGGP and *Cata*-RS yields p -value of 0.03125, thereby signalling statistically significant superiority for the latter. For CTGGP and *Cata*-AP, the p -value is the same as above. The number of evaluations used by PushGP is on each benchmark much higher than for CTGGP, implying even stronger statistical evidence.

In terms of worst-case performance (‘Max number of evaluations’), the proposed approach is also usually better or on par with GE and CTGGP, except for the Pell benchmark. The typical runtime of our method is below 0.1 second per run on a desktop PC (JVM, Intel™ i5 CPU 3.4 GHz, 8GB RAM).

It may be worth noting that *Cata*-RS and *Cata*-AP surpass PushGP even though the latter uses Lexicase selection, which was shown to immensely improve search convergence in many studies [10,11,30], while the former relies on conventional tournament selection. The very good performance of PushGP on the OddEvens benchmark stems from the fact that we do not *force* PushGP to synthesize recursive programs, and the synthesizer is free to reach for other means in order to produce the required output. For this benchmark, evolutionary search quickly discovers that the parity of the input can be conveniently tested using the modulo operator, which is available in the considered instruction set (*integer_mod*). Nevertheless, even for that PushGP needs on average one to two orders of magnitude more evaluations than for *Cata*-RS and *Cata*-AP; comparing them on worst-case performance is also favorable for the approach proposed here.

We also empirically corroborated the correctness of all programs synthesized by *Cata*-RS and *Cata*-AP by testing their generalisation capability on an external test set. To this aim, each best-of-run program was applied to arguments ranging from 9 (the next successive case after those in the training set) up to 20 inclusive. All synthesised programs proved to generalise perfectly on that test set, and indeed the source code for each synthesised program was subsequently observed to be correct by inspection.

We do not present the synthesized functions verbatim for the simple reason that, for each of the benchmarks considered, our proposed method yields the same results as the familiar human-described versions of these problems. For example, *Fib2* is expressed as:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}, \quad (4)$$

and the solution obtained is precisely the catamorphism of Sect. 2.2:⁴

```
def zeroCase(): (Nat, Nat) = (Zero, Succ(Zero))
def succCase(a: Nat, b: Nat): (Nat, Nat) = (b, a+b)
```

7.1 Harder Benchmarks

In addition to the unary recursive functions $\mathbb{N}$ considered above by Alexander, we conduct another experiments using other functions of interest mentioned in Section 5, including Sum [26], Square, Cube [14], Power(2,n) [31] and Log2 [4,5]. For these benchmarks, we use the the same settings of Cata-RS and Cata-AP as in the first experiment, and confront our method with PushGP, parameterized as therein.

The success rates of methods are summarized in Table 4. Cata-RS and Cata-AP excel as they used to Table 3, with the only exception of the former failing to find the correct program in three out of 50 runs for the Cube problem. Similarly to the previous experiment, the superior performance of PushGP on the Square and Cube benchmarks is caused by PushGP not being restricted to recursive functions only. Therefore, for these problems, a perfect program can be easily synthesized by constructing a trivial arithmetic expression. However, for Sum, where a non-recursive expression exists but is only slightly more complex, PushGP achieves success in only 56 percent of runs, and for Power(2,n) it never manages to produce a correct program.

These observations are confirmed when examining the mean and maximum number of evaluations in Table 4. For the benchmarks that cannot be easily solved without help of recursion, the proposed approach finds the correct programs at the lowest computational expense. This is particularly impressive for the Power(2,n), where no run of Cata-RS and Cata-AP requires more than 30 evaluations, i.e. four orders of magnitude less than for the (unsuccessful in this case) PushGP.

7.2 Summary of experimental outcomes

The empirical evidence presented above clearly indicates that structuring the synthesis of recursive functions with catamorphisms helps to decompose the problem and guarantee well-foundedness of candidate solutions, which in combination leads to superior effectiveness of the proposed method. Of the comparisons provided above, we find the superiority to CTGGP particularly important, as that method has been designed specifically with recursive programs in mind and, as argued earlier, is a fair representative of the state-of-the art in metaheuristic synthesis of recursive programs from examples. On the other hand, ‘vanilla’ GE and PushGP are generic

⁴ For the convenience of readers unfamiliar with Scala, we simplified the syntax for the *succCase* function; as indicated earlier, the type of the accumulator here is not a single *Nat*, but a *pair* of *Nats*, so the actual implementation is:

```
def succCase(acc:(Nat, Nat)): (Nat,Nat) = (acc._2, acc._1+acc._2) ,
```

where the underscore symbol is the idiomatic Scala syntax for extracting elements from tuples.

Table 4 Experimental results on harder benchmarks

Benchmark	Number of successful runs		
	PushGP	Cata-RS	Cata-AP
Sum	28	50	50
Square	50	50	50
Cube	50	47	50
Power(2,n)	0	50	50
Benchmark	Mean number of evaluations		
	PushGP	Cata-RS	Cata-AP
Sum	190,180 $\pm$ 33,460	5305 $\pm$ 5230	6404 $\pm$ 5581
Square	1000 $\pm$ 0	6637 $\pm$ 5367	2850 $\pm$ 1973
Cube	2040 $\pm$ 402	94,191 $\pm$ 71,503	14,227 $\pm$ 10,525
Power(2,n)	300,000 $\pm$ 0	13 $\pm$ 6	12 $\pm$ 6
Benchmark	Max number of evaluations		
	PushGP	Cata-RS	Cata-AP
Sum	300,000	24,186	21,554
Square	1000	21,452	7616
Cube	11,000	237,204	42,194
Power(2,n)	300,000	27	24

Bold values indicate the best performing methods

Mean number of evaluations for *Cata* accompanied with 0.95 confidence intervals

frameworks, not meant to address this aspect natively. Arguably, extending them with additional features (e.g. more sophisticated safeguards against infinite recursion), combined with thorough tuning, could improve their scores. Nevertheless, we anticipate their success rates to improve mostly incrementally, as the challenges posed by recursion need to be handled in a principled manner, which may be hard to reconcile with the rather unconstrained ways in which those methods perform search.

The observed differences aside, our intent here, rather than pointing out how challenging recursion is for contemporary GP/GE systems, was to bring forward the benefits of formal mechanisms that originate in category theory, algebraic datatypes and modern functional programming languages. The main conceptual upshot of our findings is that the space of recursive programs of practical relevance turns out to be much smaller than widely assumed *when harnessed/constrained in a principled way*—to the extent that makes even random search sufficient to synthesize them efficiently. In a broader context, it is entirely likely that making more intense use of the conceptual framework offered by those formalisms may help addressing other challenges inherent to program synthesis.

8 Discussion

In the current form, the proposed approach is admittedly not entirely black-box, i.e. does not rely only on the provided fitness cases. It is also necessary to provide the *type* of the accumulator, which for current experiments is restricted to $\mathbb{N}^m$, with m being the anticipated dependency of the target function on its recursive call history (e.g. 2 in the case of Fib2, 3 for Fib3 etc). However, state-of-the-art CTGGP is even more reliant on additional knowledge by requiring the partial call tree, which provides not only input-output pairs on several levels of the recursion tree, but also the structure of the recursion tree itself. It is clear that our method can be trivially generalised by invoking the algorithm described in Section 3 for different values of m , ordered in decreasing likelihood of anticipated success.

8.1 Application to other Algebraic Data Types

The catamorphism for *Nat* (Listing 4) looks very much like that for *List*.⁵ This may give the impression that catamorphisms may only be defined for ADTs that have an ‘obvious’ notion of descent ordering for the recursion to follow. However, it would be more accurate to consider that the nested constructors of an ADT object *provide a grammar* that the recursion follows down to its leaves. For example, catamorphisms can be defined for ADTs typically of interest in evolutionary program synthesis, e.g. polynomials, rational functions, binary trees etc. Given the ubiquity of these constructs in programming (and also in GP context, e.g. rational functions for symbolic regression), it would be interesting to see how the corresponding catamorphism performs relative to previous work on evolving recursive functions for such problems [26].

To illustrate that such extensions are entirely plausible, in Listings 7 and 8 we present minimal implementations of ADTs for binary trees and rational functions, together with the corresponding (generalised) catamorphisms, where the return type (marked by *R*) can be different for the accumulator type.

8.2 Other recursion schemes

A catamorphism is essentially a category-theoretic construct [19], and as such has a *dual* construct: the *anamorphism*. Just as a catamorphism corresponds to the layer-wise *deconstruction* of an existing ADT to obtain a value (of some potentially different type), so an anamorphism for an ADT is the *construction* of an instance of that ADT from (1) an initial seed value and (2) a state transition that takes the current state, optionally returning the next state. If no next state is returned (indicated by ‘None’), the anamorphism terminates.

⁵ Indeed, it is well-known that the *Nat* ADT is isomorphic to that of a List over a singleton type.

```
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) )
}
```

Listing 7 ADTs and catamorphism for binary trees

```
sealed trait Expr[A]
case class Const[A](value: A) extends Expr[A]
case class Add[A](a: Expr[A], b: Expr[A]) extends Expr[A]
case class Sub[A](a: Expr[A], b: Expr[A]) extends Expr[A]
case class Mul[A](a: Expr[A], b: Expr[A]) extends Expr[A]
case class PDiv[A](a: Expr[A], b: Expr[A]) extends Expr[A]

case class ExprCases[A,R](constCase: A ⇒ R,
  addCase: (R,R) ⇒ R, subCase: (R,R) ⇒ R,
  mulCase: (R,R) ⇒ R, divCase: (R,R) ⇒ R)

def cataExpr[A,R](arg: Expr[A],
  cases: ExprCases[A,R]): R = arg match {
  case Const(value) ⇒ cases.constCase(value)
  case Add(l, r) ⇒ cases.addCase(
    cataExpr(l, cases), cataExpr(r, cases) )
  case Sub(l, r) ⇒ cases.subCase(
    cataExpr(l, cases), cataExpr(r, cases) )
  case Mul(l, r) ⇒ cases.mulCase(
    cataExpr(l, cases), cataExpr(r, cases) )
  case PDiv(l, r) ⇒ cases.divCase(
    cataExpr(l, cases), cataExpr(r, cases) )
}
```

Listing 8 ADTs and catamorphism for rational functions

An example anamorphism is *downfrom*, which constructs a list of *Nats* from some *Nat* n , consisting of n and each of its predecessors in succession. Listing 9 gives the anamorphism for *List*, together with the implementation of *downfrom*: the type *Option*[*T*] expresses that a value of type *T* may also optionally be *None*, and can be considered as a statically-checked analog of the (mal)practice of using *null* to indicate some singular value. Anamorphisms are denoted by ‘lens brackets’, with e.g. *downfrom*(n) being given by:

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

In contrast to catamorphisms, whether or not an anamorphism terminates depends on whether the state transition ever returns *None*.

```

def anaList[A](initialCase: A,
  nextCase: A ⇒ Option[(Int, A)]): IntList =
  nextCase(initialCase) match {
    case None ⇒ Nil()
    case Some((a, y)) ⇒ Cons(a, anaList(y, nextCase))
  }

def downfrom(n: Nat): IntList = {
  def caseNext(m: Nat): Option[(Nat, Nat)] =
    if (m == 0) None else Some((m, m - 1))
  anaList(n, caseNext)
}

```

Listing 9 Anamorphism for *IntList* and implicitly recursive *downfrom*.

Hylomorphisms are schemes that express functions that can be defined via linear recursion [22]: they can be considered to consist of an anamorphism followed by a catamorphism. A motivating example of a hylomorphism is the factorial function: the anamorphism is the *downfrom* function described above, yielding a list of numbers from n to 1, and the catamorphism is $(1, *)$ (with ‘ $*$ ’ denoting the familiar multiplication operation), which then forms the product of the list elements. Hylomorphisms are denoted by ‘envelope brackets’, of the form $\llbracket \text{cata}, \text{ana} \rrbracket$ with factorial then being:

$$\llbracket (1, *), \text{downfrom} \rrbracket$$

The factorial function is an example of a function much more readily expressed as a hylomorphism than directly as a catamorphism. To address such cases, an alternative recursion scheme termed a *paramorphism* was proposed by Meertens [21]. Paramorphisms are a variant of catamorphisms (and thereby also express primitive recursion) but have access not only to the results of recursive calls, but also to the substructures on which these calls are made. Since this additional information can greatly reduce the case complexity relative to catamorphisms, paramorphisms are of great potential interest for induction of recursive functions. They are denoted by ‘barbed wire brackets’, with factorial then being $\{1, (n, m) \mapsto (1 + n) * m\}$.

In summary, recursion schemes fall into three categories: catamorphisms (‘folding’), anamorphisms (‘unfolding’) and hylomorphisms (‘refolding’). A menagerie of roughly 20 (not by any means exhaustive) variants on these basic patterns have been identified, including zygomorphisms, futumorphisms, chronomorphisms, and Elgot (co)algebras [12]. Some of these variants offer the potential for induced recursive functions which are ‘efficient by construction’. For example Elgot algebras as refolds that allow short-circuited evaluation. We next discuss some possible applications.

8.3 Synthesis of efficient algorithms

As already mentioned at the end of Sect. 7, for each of the benchmarks considered, our proposed method yields the same results as the familiar human-described versions of these problems, like the one for *Fib2* shown in Eq. 4. However, program representation

via recursion schemes provides us with a toolbox of *semantics-preserving transformations* that can be applied to existing programs in order to obtain their behavioral equivalents.

Memoization A *histomorphism* is a memoizing variant of a catamorphism [12], making all previously-computed values available. This recursion scheme could be applied to the naive implementation of *Fib2* in Eq. 4, which has the time-complexity $\Theta(\phi^n)$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio, leading so to a linear-time algorithm.⁶ In general, this opens the door to stochastic synthesis of *efficient* algorithms (or stochastic transformation of existing implementations), which historically has not been given a great deal of attention.

Fusion Previous work in GP has evolved recursive functions for statistics [2], e.g. the mean and length of a sequence. The naïve implementation of mean traverses its input sequence twice: once to compute the sum and once for the length. Both *length* and *sum* are clearly expressible as catamorphisms.

The so-called ‘banana-split’ law [6] allows any pair of catamorphisms to be expressed as a single catamorphism, thereby transforming multi-pass algorithms into single-pass.

8.4 Hybridizing recursion schemes

Allowing the accumulator type to be different from the result type (the latter denoted by *R* in Listing 8) allows functions to be synthesised that employ intermediate datatypes in their morphisms. For example, base-2 logarithm on integers can be expressed via a hylomorphism that first uses an anamorphism from *Nat* to *List* to obtain the corresponding sequence of binary digits, then a catamorphism that returns the index of the highest nonzero element. Such an extension is not currently possible in CONTAINANT due to technical limitations of its reflection mechanism, which allows only for statically-typed grammars. However, there are no conceptual obstacles to dynamic typing (e.g. via subtype polymorphism), which would allow automatic adaptation of accumulator and result types to the context. Such synthesis of type conversions has previously been achieved deterministically using proof search [17].

9 Conclusion

This paper provides evidence of the power and practical usefulness of Algebraic Data Types (ADTs) and recursion schemes for synthesizing recursive functions. The structural dependencies conveyed by ADTs eliminate a large number of spurious paths in the search space, thereby facilitating an optimal solution. Also, the domain- and problem-specific knowledge they convey make it very likely that the synthesized program generalises well. Atop of that, pattern matching guided by synthesized cases provides natural form of problem decomposition. As pointed out in Section 8, this may open the door to effective synthesis of programs that not only process arbitrary

⁶ By exploiting Fibonacci-specific domain knowledge one can actually obtain a logarithmic algorithm in this case, but the memoizing transformation is general.

variable-size data structures, but simultaneously optimise non-functional properties of their execution.

The proposed approach allows the recursive functions most commonly-studied in evolutionary computation to be induced easily, even by random search. In tandem with recent proposals on benchmarking for stochastic program synthesis [35], this suggests the necessity to consider more challenging recursive functions in future.

In a broader perspective, this paper demonstrates how ADTs and recursion schemes can be used to constrain search spaces. Conversely, it also points to the vast amount of computation wasted when these mechanisms are not taken into account. This implies that there is a range of ways in which heuristic program synthesis (including stochastic approaches like GP) can benefit from borrowing from better-grounded, more structured and principled approaches.

Acknowledgements Krzysztof Krawiec acknowledges support from the Polish National Science Centre, Grant 2014/15/B/ST6/05205.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. A. Agapitos, S.M. Lucas, *Learning Recursive Functions with Object Oriented Genetic Programming* (Springer, Berlin, 2006), pp. 166–177
2. A. Agapitos, S.M. Lucas, *Evolving a Statistics Class Using Object Oriented Evolutionary Programming* (Springer, Berlin, 2007), pp. 291–300
3. A. Agapitos, M. O'Neill, A. Kattan, S.M. Lucas, Recursion in tree-based genetic programming. *Genet. Program. Evolvable Mach.* **18**(2), 149–183 (2017)
4. B. Alexander, C. Pyromalllis, G. Lorenzetti, B. Zacher, *Using Scaffolding with Partial Call-Trees to Improve Search* (Springer, Cham, 2016), pp. 324–334
5. B. Alexander, B. Zacher, in *Parallel Problem Solving from Nature (PPSN) XIII: Conference Proceedings*. Chapter Boosting Search for Recursive Functions Using Partial Call-Trees (Springer, Ljubljana, Slovenia, 2014), pp. 384–393
6. R.S. Bird, O. de Moor, *Algebra of Programming*, Prentice Hall International series in computer science (Prentice Hall, Upper Saddle River, 1997)
7. M. Boryczka, *Ant Colony Programming for Approximation Problems* (Physica-Verlag HD, Heidelberg, 2002), pp. 147–156
8. C. Clack, T. Yu, *Performance enhanced genetic programming* (Springer, Berlin, 1997), pp. 85–100
9. M. Dorigo, T. Stützle, *Ant Colony Optimization* (Bradford Company, Scituate, 2004)
10. T. Helmuth, L. Spector, J. Matheson, Solving uncompromising problems with lexibase selection. *IEEE Trans. Evol. Comput.* **19**(5), 630–643 (2015)
11. T.M. Helmuth, General Program Synthesis from Examples Using Genetic Programming with Parent Selection Based on Random Lexicographic Orderings of Test Cases. PhD thesis, College of Information and Computer Sciences, University of Massachusetts Amherst, USA, (September 2015)
12. R. Hinze, N. Wu, J. Gibbons, *Unifying structured recursion schemes*, in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP '13* (ACM, New York, NY, USA, 2013), pp. 209–220
13. M. Hofmann, U. Schmid, Data-driven detection of recursive program schemes, in *Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, (IOS Press, Amsterdam, The Netherlands, The Netherlands, 2010), pp. 1063–1064

14. L. Huelsbergen, Learning recursive sequences via evolution of machine-language programs, in *Genetic Programming 1997*, ed. by J.R. Koza, K. Deb (Morgan Kaufmann, Burlington, 1997), pp. 186–194
15. G. Hutton, A tutorial on the universality and expressiveness of fold. *J. Funct. Program.* **9**(4), 355–372 (1999)
16. Z.A. Kocsis, J. Swan, Dependency injection for programming by optimization. *ArXiv e-prints*, (July 2017)
17. Z.A. Kocsis, J. Swan, Genetic programming + proof search = automatic improvement. *J. Autom. Reason.* **60**(2), 157–176 (2018)
18. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (MIT Press, Cambridge, MA, 1992)
19. F.W. Lawvere, S.H. Schanuel, *Conceptual Mathematics: A First Introduction to Categories* (Buffalo Workshop Press, Buffalo, NY, 1991)
20. G. Malcolm, Data structures and program transformation. *Sci. Comput. Program.* **14**(2–3), 255–279 (1990)
21. L. Meertens, Paramorphisms. *Form. Aspects Comput.* **4**(5), 413–424 (1992)
22. E. Meijer, M. Fokkinga, R. Paterson, *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire* (Springer, Berlin, Heidelberg, 1991), pp. 124–144
23. A. Moraglio, K. Krawiec, Geometric semantic genetic programming for recursive boolean programs, in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '17*, Berlin, Germany, 15–19 (ACM, July 2017), pp. 993–1000
24. A. Moraglio, F. Otero, C. Johnson, S. Thompson, A. Freitas, Evolving recursive programs using non-recursive scaffolding, in *Proceedings of the 2012 IEEE Congress on Evolutionary Computation*, ed. by L. Xiaodong (Brisbane, Australia, 10–15, 2012), pp. 2242–2249
25. Masato Nishiguchi, Yoshiji Fujimoto. Evolutions of recursive programs with multi-niche genetic programming (mngp). in *Proceedings of the 1998 IEEE World Congress on Computational Intelligence* (IEEE Press, Anchorage, Alaska, USA, 5–9 May, 1998). pp. 247–252
26. T. Phillips, M. Zhang, B. Xue, Genetic programming for solving common and domain-independent generic recursive problems, in *2017 IEEE Congress on Evolutionary Computation (CEC)*, ed. by J.A. Lozano (Donostia, San Sebastian, Spain, 5–8, 2017), pp. 1279–1286. IEEE
27. R. Poli, W.B. Langdon, N.F. McPhee, *A Field Guide to Genetic Programming* (Lulu Enterprises, UK Ltd, Essex, 2008)
28. J.C. Reynolds, *Three Approaches to Type Structure* (Springer, Berlin, 1985), pp. 97–138
29. S. Shirakawa, T. Nagao, *Graph Structured Program Evolution: Evolution of Loop Structures* (Springer, Boston, 2010), pp. 177–194
30. L. Spector, Assessment of problem modality by differential performance of lexicase selection in genetic programming: A preliminary report, in *1st workshop on Understanding Problems (GECCO-UP)*, ed. by K. McClymont, E. Keedwell (ACM, Philadelphia, Pennsylvania, USA, 7–11, 2012), pp. 401–408
31. L. Spector, J. Klein, M. Keijzer, The push3 execution stack and the evolution of control. in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, GECCO '05* (ACM, New York, 2005), pp. 1689–1696
32. L. Spector, A. Robinson, Genetic programming and autoconstructive evolution with the push programming language. *Genet. Program. Evolv. Mach.* **3**(1), 7–40 (2002)
33. T. Stützle, H.H. Hoos, Max-min ant system. *Fut. Gen. Comput. Syst.* **16**(9), 889–914 (2000)
34. P.A. Whigham, R.I. McKay, Genetic approaches to learning recursive relations, in *Progress in Evolutionary Computation*, vol. 956, Lecture Notes in Artificial Intelligence, ed. by X. Yao (Springer, Manager, 1995), pp. 17–24
35. D.R. White, J. McDermott, M. Castelli, L. Manzoni, B.W. Goldman, G. Kronberger, W. Jaskowski, U.-M. O'Reilly, S. Luke, Better gp benchmarks: community survey results and proposals. *Genet. Program. Evolv. Mach.* **14**, 3–29 (2013)
36. G. Wilson, M. Heywood, Learning recursive programs with cooperative coevolution of genetic code mapping and genotype, in *GECCO '07: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, vol. 1, ed. by D. Thierens, H.-G. Beyer, et al. (ACM Press, London, 2007), pp. 1053–1061
37. J.R. Woodward, J. Swan, *Template method hyper-heuristics*, in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation, GECCO Comp '14* (ACM, New York, 2014), pp. 1437–1438

38. T. Yu, Structure abstraction and genetic programming, in *Proceedings of the Congress on Evolutionary Computation*, vol. 1, ed. by P.J. Angeline, Z. Michalewicz, et al. (IEEE Press, Washington, 1999), pp. 652–659
39. T. Yu, A higher-order function approach to evolve recursive programs, in *Genetic Programming Theory and Practice III, volume 9 of Genetic Programming, chapter 7*, ed. by T. Yu, R.L. Riolo, B. Worzel, et al. (Springer, Ann Arbor, 2005), pp. 93–108
40. T. Yu, C. Clack, Recursion, lambda abstractions and genetic programming, in *Genetic Programming 1998*, ed. by J.R. Koza, W. Banzhaf (Wisconsin, USA, 1998), pp. 422–431. Morgan Kaufmann

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.