Systematic Synthesis of Functions

Pieter Koopman and Rinus Plasmeijer

Nijmegen Institute for Computer and Information Science, The Netherlands
{pieter,rinus}@cs.ru.nl

Abstract

In this paper we introduce a new technique to synthesize functions matching a given set of input output pairs. Using techniques similar to defunctionalisation one can specify the abstract syntax tree of the candidate functions at a high level of abstraction. We use the test system G∀ST for the systematic synthesis of candidate functions and the selection of functions matching the given condition. The generation of candidate functions is controlled by the types representing them. Instances are generated by a generic algorithm that can be tailored to specific needs. This yields a very flexible system to synthesize clear primitive recursive function definitions efficiently.

1 INTRODUCTION

At TFP'05 Susumu Katayama [5] presented an intriguing system that was able to synthesize a general function that fits a number of argument result pairs. For instance, if we state `f 2 = 2`, `f 4 = 24`, and `f 6 = 720`. we expect a factorial function like `f x =` **if** `(x≤0) 1 (x*f (x-1))`. There are of course thousands of functions that match the given condition, but by generating candidate functions in an appropriate order and form, the system should find the shown recursive solution first. Katayama's work is a step in a long research effort to synthesize pure functional programs from examples. Some key steps are Summers 1977 [14], Banerjee 1987 [1], and Cypher 1993 [2]. Programming by example is not only a curious toy, but it is used within some restricted areas like adaptive user interfaces [2, 10] and branches of AI like rule learning for planning [8, 15]. Using proof checkers one has sometimes to invent a function matching given conditions. We will investigate of our techniques can be used there.

There are various approaches in the generation of functions matching the given input result pairs. The research based on *computations traces* [14, 2] orders the examples in a lattice and synthesizes the desired function by folding the steps in this computational trace. Main problem with this approach is the construction of the lattice from individual examples. It is far from easy to generate a usable lattice for the example above. The *genetic programming* approach [13, 16] maintains a set of "promising Programs". By heuristic exchange and variation of subexpressions one tries to synthesize a matching function. The main topics in this approach are sensible variations of the candidate functions and the determination of their fitness. A third approach uses the *exhaustive enumeration* of candidate functions. The challenge here is to generate a candidate function matching the given examples in reasonable time. Katayama [5] generates expressions as λ-expressions of

the desired type. Apart from the usual abstraction, application and variables, his anonymous λ-expressions contain a small number of predefined functions. These functions from the component library provide also the needed recursion patterns. A dynamic type systems generates λ-expressions with the desired type.

In this paper we show how the exhaustive generation of candidate functions can be improved. Instead of λ-expressions, we generate functions that are direct primitive recursive, or not recursive at all. These functions are not composed of λ-expressions, but their structure is determined by the type of their abstract syntax trees. This syntax tree is represented as data structure in a functional programming language. The test system G∀ST [6] is used to generate instances of this data type in a systematic way [7]. The main differences with Katayama's approach are:

- The type system selects statically the grammar used to generate values of the desired function result, instead of dynamic system that controls the generation of λ-expressions.
- The direct definition of primitive recursive functions, instead of searching λ-expressions containing an instance of a paramorphism as recursion builder. By including a clause for a recursion builder, like `fold`, in the grammar, the corresponding recursion pattern will be generated.
- We use data types to control the generation of candidate functions. Using these types, the system becomes more open and much easier to adept to special wishes. The generation of instances of these types is done by our general generic algorithm [7] instead of an ad-hoc algorithm.
- A general test system is used to generate candidate functions, and to select and print matching functions.

Existing test systems like QUICKCHECK [3] and G∀ST have limited capabilities for the generation of functions. The generation of a function of type $\sigma \to \tau$ is done in two steps. First the argument of type σ is transformed to an integer. In the second step this integer is used to select an element of type τ. Either a value is selected from a list of values, or the integer is used as seed for a the pseudo random generater. In QUICKCHECK the function $\sigma \to$ `int` has to be provided by the user as instance of the class `coarbitrary`, in G∀ST it was derived automatically by the generic generation algorithm. A multi-argument function of type $\sigma \to \tau \to \upsilon$ is transformed to a function $\tau \to \upsilon$ by providing a pseudo randomly generated element of type σ. In this way all information of all arguments is encoded in a single integer. This approach is not powerful enough for more complex functions, and has as drawback that it is impossible to print these functions in a descent way. By its nature the system will never generate a descent (recursive) algorithm. Due to these limitations this generic generation is is not suitable for the problem treated in this paper and it is removed from G∀ST.

In this paper we show how we the generation and print problems are solved by defining a grammar as data type and a simple translation from instances of this data type to the corresponding functions. For the generation of instances of the data type the existing generic capabilities of our test system G∀ST are used.

It turns out that a similar representation of functions by data types is used at different places in the literature. The technique is called *defunctionalisation*, and the function transforming the data type is usually called *apply*. This technique was introduced by Reynolds [12], and repopularized by Danvy [4]. Defunctionalisation is usually applied as a program transformation technique to replace higher order functions by an instance of a data structure. Here we will generate a list of instances of a recursive type representing the grammar of the candidate functions.

In the next section we will show how such a function is found by our test system. First we will limit our system to functions of type `Int → Int`. We illustrate the power of our approach with a number of examples. In section 4 we illustrate how this approach can handle multi argument functions. The generation of functions with handling other types, like lists, is covered in section 5. The next section illustrates that this approach enables more powerful properties than just matching input output pairs. Finally we draw some conclusions.

2 FUNCTION GENERATION

In this section we will show how functions of type `Int → Int` can be generated using a grammar. The grammar specifies the syntax tree of the candidate functions. Our test system uses the type to generate candidate functions. The restriction to functions of type `Int → Int` in this section is by no means a conceptual restriction of the described approach. We use it here just to keep the explanations simple, a similar approach can be used for any type.

In section 2.1 we review the basic operations of the automatic test system G∀ST. In 2.2 we state the function synthesis problem as a test problem. The rest of this section covers the generation and manipulation of the data structures used to represent the syntax tree of the candidate functions synthesized.

2.1 Basic verification by automatic testing

First we explain the basic architecture of the logical part our test system G∀ST. The logical expression $\forall t : T.P(t)$ is tested by evaluating $P(t)$ for a large number of values of type T. In G∀ST the predicate P is represented by a function of type `T → Bool`. The potentially infinite list of all possible values of type `T` is used as test suite. In order to obtain a test result in finite time at most N, some given fixed number (say 1000), tests are done. There are three possible test results. *Proof* indicates that the test succeeded for all values in the test suite. This can only be achieved for type with less than N values. The result *Pass* indicates that no counterexamples are found in the first N tests. The result *Fail* indicates that a counterexample was found during the first N tests.

The result of testing in G∀ST will be represented by the data type `Verdict`:

```
:: Verdict = Proof | Pass | Fail
```

The function `testAll` implements the testing of universal quantified predicates:

```
testAll :: Int (t→Bool) [t] → Verdict
testAll n p []   = Proof
testAll 0 p list = Pass
testAll n p [x:r]
    | p x       = testAll (n-1) p r
    | otherwise = Fail
```

The list of values of type `T` is the test suite. It can be specified manually, but is usually derived fully automatically from the type T by the generic algorithm described in [7]. The actual implementation of G∀ST also reports the counterexample found and handles properties over multiple variables and a complete set of logical operators.

A similar test function exists for existential quantified logical expression of the form $\exists t : T \,.\, P(t)$. The test system returns *Proof* if a test value is found that makes $P(t)$ true. The result is *Fail* if none of the values of type T makes the predicates true. If non of the first N values makes the predicate true, the result is *Undef*.

A typical example is the rule that the absolute value of any number is greater or equal to zero, $\forall i \,.\, \mathsf{abs}(i) \geq 0$. In G∀ST we have to choose a type for the number in order to allow the system to generate an appropriate test suite. Using integers as test suite this property reads:

```
propAbs :: Int → Bool
propAbs i = abs i ≥ 0
```

This property can be tested by executing the start rule `Start = test propAbs`. The function `test` provides the number of tests and the test suite as addinitonal arguments to `testAll`. The test suite is obtained as instance of the generic class `ggen` [7]. G∀ST almost immediately finds the counterexample -2147483648, which is the minimal integer that can be represented in 32 bit numbers.

2.2 The function selection problem as a predicate

In this section we show how G∀ST can be used to synthesize candidate functions and select to functions obeying the desired properties. It is not difficult to state a property about functions that expresses that it should obey the given input output pairs. For our running example, `f 2 = 2`, `f 4 = 24` and `f 6 = 720`, we state $P(f) = f(2) = 4 \wedge f(4) = 24 \wedge f(6) = 720$. Using a straight forward approach, the property to test becomes $\exists f \,.\, P(f)$. Test systems like QUICKCHECK and G∀ST are geared towards finding counterexamples. This implies that testing yields just *Proof* if such a f is found, and yields *Undefined* if such a function is not found in the given number of tests. Here we want a function that makes the predicate true. Changing the test system such that is reports successes in an existentially quantified predicate is not very difficult, but undesirable for a software engineering point of view.

We search for a function by stating that a function matching the given examples does not exists $\neg\exists f \,.\, P(f)$ or more convenient for testing $\forall f \,.\, \neg P(f)$. Counterexamples found by G∀ST are exactly the desired functions. Now these functions are counterexamples and will be shown by the test system. We state in G∀ST:

```
prop0 :: (Int→Int) → Bool
prop0 f = ∼ (f 2 == 2 && f 6 == 720 && f 4 = 24)
```

Where $\sim$ is the negation operator. Any counterexample found by G$\forall$ST is a function that matches the given input output pairs. As outlined in the introduction, functional test systems like QUICKCHECK and G$\forall$ST are not very good in generating functions and printing them. Instead of `prop0` we will use a property over the data type `Fun`. The type `Fun` represents the grammar of candidate functions, see 2.3. The function `apply`, see 2.5 turns an instance of this data type in the actual function.

```
prop1 :: Fun → Bool
prop1 d = ∼(f 2 == 2 && f 6 == 720 && f 4 == 24) where f = apply d
```

This predicate can be tested by executing a program with `Start = test prop1` as starting point. Our system yields the following result:

```
Counterexample 1 found after 30808 tests: f x = if (x≤0) 1 (x*f (x-1))
Execution: 1.02  Garbage collection: 0.15  Total: 1.17
```

This counterexample is exactly the general primitive recursive we are looking for, the well-known factorial function. More examples will be given below.

2.3 A grammar for candidate functions

In the generation of candidate functions we have to be very careful to generate only terminating functions. If one of the generated functions happens to be nonterminating for one of the examples, testing becomes nonterminating as well. This termination can either be guaranteed by carrying the number of recursive call done around in the function and put an upper limit on the number of recursive calls, or by only generating functions that are terminating by construction.

We will construct only terminating (primitive recursive) functions. For the integer domain, these functions are either not recursive, or use as stop criterion a conditional of the form `x≤c`, where `x` is the function argument and `c` is some small integer constant. The then-part is an expression containing no recursive calls. The else-part contains only recursive calls of the form `f (x-d)`, where `d` is a small positive number. Since we want to generate only primitive recursive functions, recursive calls are not nested.

The body of a function is either a non-recursive expression, or a recursive expression of the described form. Expressions are either a variable, an integer constant or a binary operator applied to expressions. This is captured by the following grammar.

$$
\begin{aligned}
\mathit{Fun} &= \mathbf{f}\,\mathbf{x} = (\mathit{Expr} \mid \mathit{RFun}) \\
\mathit{RFUN} &= \mathbf{if}\ (\mathbf{x} - \mathit{IConst})\ \mathit{Expr}\ \mathit{Expr2} \\
\mathit{IConst} &= \mathsf{positive\_integer} \\
\mathit{Expr} &= \mathsf{Variable} \mid \mathsf{integer} \mid \mathit{BinOp}\ \mathit{Expr} \\
\mathit{BinOp}\ e &= e + e \mid e - e \mid e * e
\end{aligned}
$$

The expressions in an else-part are either a variable, a constant, or a binary operator over a variable, a constant of a recursive function application:

$$\mathit{Expr2} \quad = \quad \mathsf{Variable} \mid \mathsf{integer} \mid \mathit{BinOp}\ (\mathsf{Variable} \mid \mathsf{integer} \mid \mathbf{f}(\mathbf{x} - \mathsf{integer}))$$

This grammar is directly mapped to a data type in CLEAN [11]. We use the type `OR` to mimic the choice operator, |, used in the grammar.

```
:: OR s t = L s | R t
```

This composition of types allow us to use a choice between types. This saves us the burden of defining a tailor made type for each choice.

In the definition of the data types representing the grammar we represent only the variable parts of the grammar. Literal parts of the grammar (like **f x** =) are omitted (as in any syntax tree). Constructors like `IConst` are introduced in order to make the associated integer a separate type, this is necessary in order to generate values of this type in a different way than standard integers.

Constructs that behave similar are placed in the same type (like `BinOp`). A separate type is used for recursive parts in the grammar, parts that are used at several places, or for clarity.

```
:: IConst  = IConst Int
:: BinOp x = OpPlus x x | OpMinus x x | OpTimes x x
:: Var     = X
:: Expr    = Expr (OR (OR Var IConst) (BinOp Expr))
:: FunAp   = FunAp Int
:: TermVal = TermVal Int
:: RFun    = RFun TermVal Expr
                  (OR (OR Var IConst) (BinOp (OR (OR Var IConst) FunAp)))
:: Fun     = Fun (OR Expr RFun)
```

These data types are used to represent recursive functions as illustrated above. The design of these types controls the shape of the candidate functions. It is very easy to add additional operators like division or power.

2.4 Generating candidate functions

The generic algorithm `ggen` used by G∀ST generates a list of all instances of a (recursive) type from small to large. The only thing to be done is to order CLEAN to derive the generic generation for these types.

derive `ggen OR, BinOp, Var, Expr, RFun, Fun`

For the constants we do not use the ordinary generation of integers. A much smaller sets of values is used to speed up the synthesis of matching candidates functions. After studying many examples of recursive functions in text books and libraries the values 0..2 appear to be used as termination value. The occurring recursive calls for integer functions are usually of the form $f(x-1)$ or $f(x-2)$. The occurring integer constants are in the range 0..5. These values are used in the following tailor

defined instances of the corresponding types in CLEAN. The variables `n` and `r` can be used to make a pseudo random change in the order of the values. This is not needed nor wanted here.

```
ggen {|TermVal|} n r = map TermVal [0..2]
ggen {|FunAp|}   n r = map FunAp   [1..2]
ggen {|IConst|}  n r = map IConst  [0..5]
```

Neither of these upper limits is critical. Making the maximum `IConst` 50 (or even unbounded) instead of 5 slows the discovery of most functions down by a factor of 2. Using 3 as maximum, instead of 5, usually gives a speedup of a factor of 2.

2.5 Transforming data structures into functions

Until now we generate the syntax trees representing candidate functions, but for the determination of the fitness of a candidate function we need the function corresponding to this syntax tree. The class `apply` will be used to transform a syntax tree into the corresponding actual function. Although `apply` can also be defined in a generic way, we prefer an ordinary class here. The generic definition is not shorter, and the ordinary class is more efficient. The class `apply` contains only the function `apply`. The class is parameterized by the data type `d` to be transformed, the environment `e` used to determine its value, and the type of value `v` to be generated.

```
class apply d e v :: d → e → v
```

We will use two different environments. The first type of environment contains only the integer used as function argument. The second type of environment is a tuple containing the recursive function and the function argument.

The interesting cases using the environment are:

```
instance apply Var Int     Int where apply x = λi.i
instance apply Var (x,Int) Int where apply x = λ(_,i).i
instance apply FunAp (Int→Int,Int) Int
where apply (FunAp d) = λ(f,i).f (i-d)
```

In the definition of a recursive function, `RFun`, an environment containing the integer argument is transformed in an environment containg the recursive function and the argument. The recursion is constructed by the cycle in the definition of `f`.

```
instance apply RFun Int Int
where apply rf=:(RFun (TermVal c) then else) = f
      where f i = if (i≤c) (apply then i) (apply else (f,i))
```

The definition of the `apply` for expressions of type `Expr` is somewhat smart. Expressions do not contain calls of the recursive function. Hence it is superfluous to pass it to all nodes of the syntax tree.

```
instance apply Expr Int Int where apply (Expr f) = apply f
instance apply Expr (x,Int) Int where apply (Expr f) = λ(_,i).apply f i
```

The instance of apply for binary operators takes care of the computations. The instance of `apply` for `BinOp x` requires that there is an instance of `apply` for `x` this environment `e` and result of type `v`. Moreover it is required that the operators `+`, `-`, and `*` are defined for type `v`.

```
instance apply (BinOp x) e v | apply x e v & +, -, * v
where apply (OpPlus  x y) = λe.apply x e + apply y e
      apply (OpMinus x y) = λe.apply x e - apply y e
      apply (OpTimes x y) = λe.apply x e * apply y e
```

The other instances of apply just pass the environment to their children, e.g:

```
instance apply (OR x y) b c | apply x b c & apply y b c
where apply (L x) = apply x
      apply (R y) = apply y
```

2.6 Pretty printing generated functions

When we derive showing of candidate functions in the generic way, we would obtain the following representation for the factorial function from section 2.2.

```
Fun (R (RFun (TermVal 0) (Expr (L (R (IConst 1))))
                         (R (OpTimes (L (L X)) (R (FunAp 1))))))
```

Although this data structure represents exactly the recursive factorial function listed above, it is harder to read. Instead of deriving the generic print routines for the data types representing the grammar, we use tailor made definitions in order to obtain nicely printed functions instead of the data structures representing them. See section 2.2.

The generic function `GenShow` yields a list of strings to be printed. It has a separator `sep` as argument that is used between constructors. The second argument, `p`, is a boolean indicating whether parenthesis around compound expressions are needed. The third argument is the object to be printed. The last argument, `rest`, is a continuation. This list of strings represents the rest of the result of `genShow`.

The dull code below just takes care of the pretty printing of candidate functions. It just adds the constant parts of the grammar not represented in the syntax tree and removes some constructors. We list some typical examples.

```
genShow{|OR|} f g sep p (L x)        rest = f sep p x rest
genShow{|OR|} f g sep p (R y)        rest = g sep p y rest
genShow{|IConst|} sep p (IConst c)   rest = [toString c:rest]
genShow{|Var|}    sep p v            rest = ["x":rest]
genShow{|Expr|}   sep p (Expr e)     rest = genShow{|*|} sep p e rest
genShow{|RFun|}   sep p (RFun c t e) rest
 = ["if (x≤":genShow{|*|} sep False c
    [") ":genShow{|*|} sep True t [" ": genShow{|*|} sep True e rest]]]
genShow{|BinOp|} f sep p (OpPlus x y) rest
 = [if p "(" "": f sep True x ["+": f sep True y [if p ")" "":rest]]]
```

2.7 Examples

In order to demonstrate the power of our approach we list some examples. The first column of the table contains the input/output pairs the function has to obey. The next columns contain the first matching function found, the number of test and the time needed (in seconds) to generate generate this function. We used a 1 GHz AMD PC running Windows XP and the latest versions of CLEAN and G∀ST.

given examples	generated function	tests	time
f 1 = 1	f x = 1	1	0.01
f 1 = 1, f 2 = 4	f x = x*x	69	0.02
f 1 = 1, f 2 = 5	f x = if (x≤1) 1 5	160	0.02
f 2 = 2, f 6 = 720, f 4 = 24	f x = if (x≤0) 1 (x*f (x-1))	30808	1.17
f 4 = 5, f 5 = 8	f x = if (x≤1) 1 (f (x-2)+f (x-1))	2791	0.16
f (-2) = 2, f 5 = 5, f (-4) = 4	f x = if (x≤0) (0-x) x	678	0.05

These examples show that a small number of examples are sufficient to generate many well-known functions. From top to bottom these functions are known as: the constant one, square, a simple choice, factorial, fibonacci, and absolute value.

Depending on the amount of memory (32 – 64 M) and the details of the generated functions, our implementation generates 10 to 25 thousand candidate functions per second. Private communication with Katayama indicates that our implementation is more than one order of magnitude faster. When lists are excluded from his implementation it needs 25 seconds on Katayama's faster (3 GHz Pentium 4) machine for the factorial function. His solution for[1] $f\,0 = 1, f\,1 = 1, f\,2 = 2, f\,3 = 6, f\,4 = 24$ is

```
λa.nat_para a (λb.inc b) (λb c d.c (nat_para b d (λe f.c f))) zero
```

Using the paramorphism [9] `nat_para`, twice, as recursion pattern.

```
nat_para :: Int a (Int a → a) → a
nat_para 0 x f = x
nat_para i x f = f (i-1) (nat_para (i-1) x f)
```

Comparison with our running example, repeated as example 4 in the table above, indicates that our system generates functions that are better readable for most people, that our approach synthesizes a matching function faster, and the generated function is more efficient.

3 CONTROLLING THE CANDIDATE FUNCTIONS

The generation of candidate functions can be controlled in three ways. In this section we will discuss these ways, and show their effect by searching for functions matching `f 1 = 3`, `f 2 = 6`, and `f 3 = 6`. The three different ways to control the synthesis of functions are:

[1] Katayama's system needs more input output pairs to find the factorial functions. With the pairs used as running example his system finds another function. This is just an effect of the order of generation of candidate functions.

Designing types By far the most important way to control the synthesis of candidate functions is the design of the data types used to represent the candidate functions. Only candidate functions that can be represented can be generated and will be considered.

In this paper we used this to guarantee that candidate functions are either nonrecursive, i.e. the function body is an arithmetic expression, or the candidate function is primitive recursive containing an appropriate stop condition.

Generating instances of types The test system G∀ST generates instances of these types in its struggle to approve or falsify the statement that there is no function obeying the given input output pairs. One of the advantages of G∀ST is that the generation of instances for types can be done by the generic algorithm `ggen`. The instance of `ggen` for a specific type just yields the list of candidate values. This implies that one can decide to specify a list of values by hand instead of deriving them by the generic algorithm.

We used this in the generation of constants. Although there is no conceptual limitation to leaves of the syntax trees to be generated, it is convenient to use it only there. One can use a general type for constants and easily control the actual constants used. It is possible to use this also for types with arguments, but that brings the burden of controlling the order of generating instance back to the user.

Selection of generated instances Finaly, G∀ST has the possibility to apply a predicate to candidate functions, or actually their syntax tree, before they are used. If the predicate does not hold, the test value is not used. In fact it is not even counted as a test.

This is often used for partial functions. A typical example is the square root function that is only defined for nonnegative numbers. For these numbers we can state that the square of the square root of any nonnegative rational number should be equal to that number: $\forall r\,.\,r \geq 0 \rightarrow sqrt(r)^2 = r$. This can be expressed directly in G∀ST as:

```
pSqrt :: Real → Property
pSqrt r = r ≥ 0.0 ⟹ (sqrt r)^2.0 == r
```

Using this mechanism we can eliminate undesirable candidate functions from the tests, and hence from the synthesis of matching functions.

In order to demonstrate the effects of these techniques we introduce four variants of the property searching for functions matching the input output pairs `f 1 = 3`, `f 2 = 6`, and `f 3 = 6`. Some functions matching these pairs are `f x = 3*x` and `f x = x+x+x`. The first ten functions synthesized are listed in table 1. The time needed and the number of candidates tried and rejected are given in table 2.

The first version of this property looks only for nonrecursive functions. This is achieved by using syntax trees of type `Expr`, rather than `Fun`.

```
pExpr :: Expr → Bool
pExpr d = ∼(f 1 == 3 && f 2 == 6 && f 3 == 9) where f = apply d
```

TABLE 1. First 10 functions synthesized for variants of properties requiring the input output pairs `f 1` = 3, `f 2` = 6, and `f 3` = 6.

pExpr: synthesize expressions	pFun: synthesize functions
f x = 0+((x+x)+x)	f x = (x+(x+x))+0
f x = (x-x)+((x+x)+x)	f x = x+(x+x)
f x = x+(x+x)	f x = 0+((0+(x+x))+x)
f x = ((x+((x+0)+x))+x)-x	f x = 3*x
f x = x+((x+x)+0)	f x = (x+x)+x
f x = (x+x)+x	f x = x+((0+(x+x))+0)
f x = ((x+((x+0)+x))+x)-(0+x))	f x = (0+(x+x))+x
f x = ((x+((x+0)+x))+0)-0	f x = ((x+x)+x)+0
f x = (x+x)+(0+x)	f x = x*3
f x = (0*x)+((x+x)+x)	f x = (x-x)+((0+(x+x))+x)
pFit: use only fit functions	**pExpr2: new type for expressions**
f x = x+(x+x)	f x = 3*x
f x = 3*x	f x = x+(2*x)
f x = (x+x)+x	f x = 3*(1+(x-1))
f x = x*3	f x = 3*((x+1)-1)
f x = ((x+x)+(x+x))+(0-x)	f x = 3*(2+(x-2))
f x = x-(0-(x+x))	f x = 3+(3*(x-1))
f x = (x+x)+((x+x)-x)	f x = 3*(3+(x-3))
f x = if (x≤1) 3 (f (x-1)+f (x-2))	f x = 3*(4+(x-4))
f x = (x+x)-(x-(x+x))	f x = 3*(5+(x-5))
f x = if (x≤0) x (f (x-1)+3)	f x = x*((x+3)-x)

G∀ST quickly synthesizes functions matching these conditions, but many generated functions contain redundant subexpressions of forms like `0+x` or `(x-x)+y` where we would prefer the expressions `x` and `y` respectively.

By changing the data type of syntax trees to `Fun`, G∀ST also synthesizes primitive recursive functions. From the tables we see that more candidates are needed in order to find ten matching functions. Non of the first ten matching functions synthesized is recursive. This illustrates the first possibility to control the candidate functions mentioned above: using different data types, the system will generate different candidate functions. Since we use different data types the order of candidate functions synthesized is somewhat different than for the previous case. This explains why the functions synthesized are not exactly equal.

```
pFun :: Fun → Bool
pFun d = ∼(f 1 == 3 && f 2 == 6 && f 3 == 9) where f = apply d
```

The third version of the property used also generates candidates from the type `Fun`. However, only those instances from that type that obey the predicate `fit` are used in the tests. Generated instances of `Fun` that do not obey the predicate `fit` are

TABLE 2. Execution time, number of tests and rejections needed to synthesize the results from table 1.

property	execution time (S)	number of tests	candidates rejected
pExpr	0.02	180	0
pFun	0.03	429	0
pFit	0.21	1525	2860
pExpr2	0.12	2126	0

rejected instead of tested.

```
pFit :: Fun → Property
pFit d = fit d ⟹ ∼(f 1 == 3 && f 2 == 6 && f 3 == 9) where f = apply d
```

The predicate `fit` is implemented as a class. The default instance yields true for all arguments. A typical instance of this class for the type of binary operators is given. A subtraction is fit is the arguments are not equal and each of the arguments is fit. An addition is fit if both arguments are not equal to the constant zero, checked by `is0`, and fit.

```
class fit a :: a → Bool

instance fit (BinOp x) | gEq{|*|} x & isConst, fit x
where fit (OpMinus x y) = x =!= y && fit x && fit y
      fit (OpPlus x y)  = ∼(is0 x) && ∼(is0 y) && fit x && fit y
      fit (OpTimes x y) = ∼(is01 x) && ∼(is01 y) && fit x && fit y

instance fit a where fit a = True // default instance
```

From the tables we see that this does indeed remove many undesirable function definitions. Removing additional undesirable expressions is achieved by improving the associated instances of `fit`. As a result of removing undesirable candidates, there are now two recursive definitions found matching the examples. The first one is similar to the Fibonacci function and might be somewhat surprising. The second one implements multiplication by repeated addition.

Instead of filtering undesirable function candidates, one can also prevent their synthesis. Doing this in the instance of `ggen` is quite tricky since expressions are recursive. A better way is to use a new data type `E` for expressions.

```
pExpr2 :: E → Bool
pExpr d = ∼(f 1 == 3 && f 2 == 6 && f 3 == 9) where f = apply d
```

In the data type `E` we prevent undesirable expressions of the form `x-0`, `x*1`, `0+x`, `1+1`, and `x-x` by removing them from the data types.

```
:: Mul x y = Mul x y
:: Add x y = Add x y
:: Sub x y = Sub x y
```

```
:: E1 = E1 (OR (Mul MConst Var) (Mul (OR MConst Var) E23))
:: E2 = E2 (OR (Add Var NConst) (Add (OR NConst Var) E13))
:: E3 = E3 (OR (OR (Sub Var NConst) (Sub IConst Var))
           (OR (Sub (OR NConst Var) E23) (Sub E23 (OR NConst Var))))
:: E12 :== OR E1 E2
:: E23 :== OR E2 E3
:: E13 :== OR E1 E3
:: E   :== OR (OR Var IConst) (OR E1 (OR E2 E3))
```

This techniques works, but makes the type definitions used larger. Ensuring that all wanted instances are included in the type and nothing more is at least as tricky as stating a predicate `fit` that removes all candidates that might be considered undesirable.

4 GENERATION OF MULTI ARGUMENT FUNCTIONS

All generated functions above are of type `Int→Int`. This was chosen deliberately to keep things as simple as possible, but it is not an inherent limitation of the approach. To demonstrate this we show how to handle functions with `Arity`, e.g. 2, integers arguments. The type for variables is changed such that it represents a numbered argument.

```
:: VarN = VarN Int
```

The environment in `apply` will now contain a list of values.

instance `apply VarN [Int]  Int` **where** `apply (VarN n) = λ  l .l !! n`

The instance of `ggen` takes care that only valid argument numbers are generted.

```
ggen {|VarN|} n r = map VarN [0..Arity-1]
```

5 SYNTHESIS OF FUNCTIONS OVER OTHER DATA TYPES

The manipulation of other types than integers can be handle by defining a suitable abstract syntax tree for these functions, and the associated instances of `ggen`, `apply` and `genShow`.

The synthesis of functions of type `Real→Real` with the same structure as the functions of type `Int→Int` used above is very simple, we only have to supply suitable instances of `apply`. As slightly more advanced example we show how function over list of integers, type `[Int]→[Int]`, can be handled that are either the identity function, or the map of a function of type `Int→Int` over the argument list.

```
:: LFun = ID | MAP Fun

instance apply LFun [Int] [Int]
where apply ID = λl.l
      apply (MAP f) = map (apply f)
```

derive `ggen LFun`

Some examples of its use are:

given example	generated function	tests	time
f [1,2,3] = [1,2,3]	g y = y	1	0.01
f [1,2,3] = [1,4,9]	g y = map f y where f x = x*x	34	0.05
f [1,2,5] = [1,2,120]	g y = map f y where f x = if (x≤1) x (f (x-1)*x)	67573	3.89

6 OTHER PROPERTIES

Having the candidate function available as real function enables us to write also other conditions, like `twice f 1 == 4` or `f 1 ≠ 5`.

However, there is no reason to stick to these simple predicates on the synthesized candidate functions. It is possible to search for nonrecursive functions that obey the rule $f = 0$ and $\forall x \,.\, 2f(x) = f(2x)$. This can directly be stated is G∀ST as:

```
pfExpr :: Expr → Property
pfExpr d = fit d ⟹ ∼(f 0 == 0 ∧ ForAll (λx.2*f x == f (2*x)))
where f = apply d
```

Note that we limit the search to fit candidates. The system promptly synthesizes functions like `f x = 0`, `f x = x`, `f x = x+((x+x)+x)`, `f x = 0-(x+x)`.

If we also includes recursive functions in the search space, we have to take care that the integers tried as argument by G∀ST are not too large. Computing the result of synthesized primitive recursive functions, like factorial and Fibonacci, for a typical test value like `maxint` uses undesirable amounts of time and space. The numbers used in the tests can be limited by computing them modulo some reasonable upper bound, like 15, or by stating a range of values directly. A typical example is:

```
pfFun :: Fun → Property
pfFun d = fit d ⟹ ∼(f 1 == 1 ∧ ((λx.(f x)/x == f (x-1)) For [1..10]))
where f = apply d
```

The factorial function `f x = if (x≤0) 1 (f (x-1)*x)` is synthesized quickly.

Since the syntax tree of the candidate functions is available, it is easy to manipulate the candidate functions. As example we show how we can obtain the derivative of functions of type `Real→Real` and how it is used in properties. The derivative $\frac{d}{dx}$ of expressions is computed by the class `ddx`. The rules are taken directly from high school mathematics:

```
class ddx t :: t → Expr

instance ddx Var where ddx v = toExpr (IConst 1)
instance ddx IConst where ddx v = toExpr (IConst 0)
instance ddx (BinOp t) | ddx t & toExpr t
where ddx (OpPlus  s t) = toExpr (OpPlus (ddx s) (ddx t))
```

```
ddx (OpMinus s t) = toExpr (OpMinus (ddx s) (ddx t))
ddx (OpTimes s t)
 = toExpr (OpPlus (toExpr (OpTimes (ddx s) (toExpr t)))
                  (toExpr (OpTimes (toExpr s) (ddx t))))
```

This can be used in properties over a function `f` and its derivative `f'`. For example $f(0) = 1$ and $\forall x . f'(x) = 2x$. In G∀ST this is:

```
pddx :: Expr → Property
pddx d = ~(f 0.0 == 1.0 ∧ ForAll (λx. f' x == 2.0*x))
where f  = apply d; f' = apply (ddx d)
```

After 145 test cases G∀ST synthesizes the first matching function: `f x = (x*x)+1`.

These examples show that it pays to use a general test system for the synthesis of functions. The matching of given pairs nicely integrates with the general logical expressions. Have the candidate functions available as data structure also enable symbolic manipulations like computing the derivative.

7 CONCLUSION

In this paper we show how functions matching given input-result pairs can be synthesized. In fact, we generate functions that are either not recursive, or primitive recursive and are guaranteed to terminate by their structure.

The syntax tree of the candidate functions is determined by a data type. Generating the instances of these data types and selecting the correct corresponding functions can be done very well with our general test system G∀ST. There are three ways in which the synthesis of functions is controlled. The first and most important control mechanism is the type of the syntax tree representing the functions. The second control mechanism is the generation of instances of these types. It is very convenient to derive the generation of instances from the generic algorithm of G∀ST, but that is not required. Any list of values can be used. We use this in the generation of constants: the type is very general, but the used instances of `ggen` generate only a small list of desired values. The third and final way to control which functions are used in the test is by using a predicate in the property. In this paper we used the predicate `fit` to eliminate candidates representing undesirable subexpressions (like `x-x`, and `0+x`). By defining more sophisticated types, the other ways to control the synthesis become superfluous. The user decides what is most convenient and effective.

The test system does most of the work and provides an excellent platform. For the functions of type `Int→Int` only one page of additional CLEAN code is needed. This approach is more transparent, flexible and efficient than existing systems like [5]. Just like [5] the system will not synthesize functions that does not fit in the syntax tree, like `f list = [sum list]` for the very simple functions of type `[Int]→[Int]` used in this paper. This can be fixed by extending the grammar of candidate functions. Although the described system works excellent for many examples, synthesizing functions involving very large expressions or very large

constants will take very much time, e.g. hours or days. This is due to the size of the search space and the systematic search. Our system does not use any information available in the predicate during synthesizing candidate functions.

REFERENCES

[1] Debasish Banerjee, *A methodology for systhesis of recursive functional programs*. ACM transactions on programming languages and Systems, 9(3) 441–462, 1987.

[2] Allen Cypher (editor) *Watch What I Do: Programming by Demonstration* MIT Press, 1993.

[3] K. Claessen, J. Hughes. *QuickCheck: A lightweight Tool for Random Testing of Haskell Programs.* ICFP, ACM, pp 268–279, 2000.

[4] Olivier Danvy and Lasse R. Nielsen. *Defunctionalization at Work.* PPDP '01 Proceedings, 2001, pp 162–174.

[5] Susumu Katayama: *Systematic search for lambda expressions* Proceedings 6th Symposium on Trends in Functional Programming TFP 2005, pp. 195–205, www.cs.ioc.ee/tfp-icfp-gpce05/tfp-proc and privite communication.

[6] Pieter Koopman, Artem Alimarine, Jan Tretmans and Rinus Plasmeijer: *Gast: Generic Automated Software Testing*, Peña: IFL'02, LNCS 2670, pp 84–100, 2002.

[7] Pieter Koopman, Rinus Plasmeijer: *Generic Generation of the Elements of Types*, Proceedings 6th Symposium on Trends in Functional Programming TFP 2005, pp. 167–179, www.cs.ioc.ee/tfp-icfp-gpce05/tfp-proc.

[8] E. Kitzelmann, U. Schmid, M. Mühlpfordt, and F. Wysotzki *Inductive Synthesis of Functional Programs* In J. Calmet et al (Eds.), AISC 2002 and Calculemus 2002, LNCS 2385, 26–37, 2002.

[9] Lambert Meertens. *Paramorphisms*, Formal Aspects of Computing, 4, pp 413–424, 1992.

[10] Henry Lieberman (editor) *Your Wish is My Command: Programming by Example* The Morgan Kaufmann, ISBN 1-55860-688-2, 2001

[11] Rinus Plasmeijer and Marko van Eekelen: *Concurrent Clean Language Report (version 2.1.1)*, 2005. www.cs.ru.nl/∼clean.

[12] John C. Reynolds. *Denitional interpreters for higher-order programming languages.* Higher-Order and Symbolic Computation, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[13] Ute Schmid, Jens Waltermann. *Automatic Synthesis of XSL-Transformations from Example Documents*. In: M.H. Hamza: Artificial Intelligence and Applications Proceedings (AIA 2004), pp 252–257, Acta Press, 2004.

[14] Philip Summers *A methadology for LISP program construction from examples* JACM 24(1) 161–175, 1977.

[15] Fritz Wysotzki, Ute Schmid *Synthesis of Recursive Programs from Finite Examples By Detection of Macro-Functions* Forschungsberichte des Fachbereichs Informatik der TU Berlin Nr. 2001-2, ISSN 1436-9915, 2001. http://ki.cs.tu-berlin.de/pubs.html

[16] T. Yu and C. Clack, *Recursion, Lambda Abstractions and Genetic Programming*, Genetic Programming 1998: Proceedings of the Third Annual Conference, Pages 422–431, Morgan Kaufmann, 1998