Chapter 3

Systematic Synthesis of Functions

Pieter Koopman[1], Rinus Plasmeijer[1]

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 the abstract syntax tree of the candidate functions is specified at a high level of abstraction. We use a recursive data type to represent the syntax tree of the candidate functions. The test system G∀ST is used for the systematic synthesis of candidate functions and the selection of functions matching the given condition. The representation of candidate functions as data structures gives us full control over them and the transformation of the syntax tree to the actual function is straight forward. Instances of the syntax tree are generated by a generic algorithm that can be tailored easily to specific needs. This yields a very flexible system to synthesize clear (recursive) function definitions efficiently.

3.1 INTRODUCTION

At TFP'05 Susumu Katayama [7] 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 above recursive solution first. Katayama's work is one of the latest steps in a long research effort to synthesize pure functional programs from examples. Some key steps are Summers 1977 [16], Banerjee 1987 [1], and Cypher 1993 [4].

Programming by example is not only a curious toy, but it is used within areas like adaptive user interfaces [4, 5] and branches of AI-like rule learning for

[1]Nijmegen Institute for Computer and Information Science, Radboud University Nijmegen, The Netherlands; E-mail: pieter@cs.ru.nl, rinus@cs.ru.nl

planning [8, 18]. Using proof checkers one has sometimes to invent a function matching given conditions. Jeuring et al [6] use our approach as an inspiration to generate generic functions that perform a task defined by some typical examples. The goal of all these programming by example systems is not to replace programming in general, but to have a tool that within some limited area is able to synthesize a function that is a generalization of the input-output behavior specified by a (small) number of input-output pairs. In this paper we extend this to a small number of expressions containing an application of the function. Since we have the synthesized candidate function available as a data structure representing its abstract syntax tree, it is easy to do symbolic manipulations of the synthesized candidate functions (like selecting or eliminating functions of some kind, or determination of the derivative of the candidate function).

There are various approaches in the generation of functions matching the given input result pairs. The research based on *computation traces* [16, 4] orders the examples in a lattice and synthesizes the desired function by folding the steps in these computation traces. The 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 [15, 17] maintains a set of "promising functions". 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 [7] generates candidate functions 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 predefined functions from the component library provide also the necessary recursion patterns. A dynamic type system is used to generate λ-expressions with the desired type. A detailed comparison between his work and this paper is given in Section 3.7.

In this paper we show how the exhaustive generation of candidate functions can be improved. Instead of λ-expressions, we generate functions that can be directly recursive. 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 a data structure in a functional programming language. The test system G$\forall$ST [9] is used to generate instances of this data type in a systematic way [10] and to judge the suitability of the generated candidates.

Existing test systems like QUICKCHECK [2] and G$\forall$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 the pseudo random generator. In QUICKCHECK the function $\sigma \to$ `int` has to be provided by the user

as an instance of the class `coarbitrary`, in G∀ST it was[2] 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. By its nature the system will never generate a decent (recursive) algorithm. Due to these limitations this generic generation is not suitable for the problem treated in this paper and it has been removed from G∀ST.

In this paper we show how the generation and print problems are solved by defining the grammar as a 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. In [11] we have used basically the same technique, but in a less sophisticated way, to test properties over higher order functions. There was the goal to test a (universally quantified) property of a higher order function by generating a huge number of functions as test arguments. Here we are interested in the synthesized function obeying the restrictions given by some applications of the function. This can be viewed as the mirror image of testing universal quantified properties: there to goal is to find arguments that falsify the property.

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 [14], and repopularized by Danvy [3]. Defunctionalisation is a program transformation to turn higher-order programs into first-order ones. It has been proven to be meaning preserving. The basic idea is to replace every lambda abstraction with a data constructor that will carry the environment needed, and replace every application of the higher-order function with an apply function that will interpret the data structure. Here we will generate a list of instances of a recursive type representing the grammar of the candidate functions. This implies that each and every function is generated as a data structure. Whenever desired this data structure is transformed to the function it represents by the corresponding instance of the class `apply`.

The key contributions of this paper are the use of data structures to guide the synthesis of candidate functions in a flexible and controlled way, and the use of a general test system to find the functions matching the given input-output pairs. The use of a test system seems rather surprising since it is geared towards finding counterexamples and here we need functions that match the given predicate. The key step is a rephrasing of the property: if we state that such a function does not exist, the counterexamples produced by the test system are exactly the functions we are looking for.

[2]For technical reasons the mapping of values to integers had to be integrated in the generic generation algorithm. Since this slows down the generic generation algorithm, the increased memory consumption and the limited use of functions generated in this way, this feature has been removed from G∀ST.

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. The ways to control the synthesis of candidate functions are discussed in Section 3.3. In Section 3.4 we illustrate how this approach can handle multi-argument functions. The generation of functions handling other types, like lists, is covered in Section 3.5. Section 3.6 illustrates that this approach enables more powerful properties than just matching input-output pairs. Section 3.7 provides a comparison of our work with Katayama's system. Finally we draw some conclusions.

3.2 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 3.2.1 we review the basic operations of the automatic test system G∀ST. In 3.2.2 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.

3.2.1 Basic verification by automatic testing

First we explain the basic architecture of the logical part of 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, a given fixed number N (say 1000) of tests are done. There are three possible test results. The result *Proof* indicates that the test succeeded for all values in the test suite. This can only be achieved for a 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 | Undefined
```

The function `testAll` implements the testing of universally 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 [10]. G∀ST also reports any counterexample found (if any), handles properties over multiple variables, and has a complete set of logical operators.

A similar test function exists for existentially 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 none of the first N values makes the predicate true, the result is *Undefined*. The result *Undefined* means that within the given bounds G∀ST was neither able to find a counterexample, nor a value that makes this predicate hold. Hence, its value is undefined.

A typical example is the rule that the absolute value of any number is greater than 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 additional arguments to `testAll`. The test suite is obtained as instance of the generic class `ggen` [10]. G∀ST almost immediately finds the counterexample -2147483648, which is the minimal integer that can be represented in 32 bit numbers. This value is one of the common border values that are in the front of any test sequence of integers, other border values are `-1`, `0`, `1` and `maxint`.

3.2.2 The function selection problem as a predicate

In this section we will show how G∀ST can be used to synthesize candidate functions and to select 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 straightforward 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 an 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 it reports successes in an existentially quantified predicate is not very difficult, but undesirable from a software engineering point of view.

We search for a function by stating that a function matching the given examples does not exist $\neg\exists f \,.\, P(f)$ or more conveniently 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 4 == 24 && f 6 == 720)
```

where $\sim$ is the negation operator. Any counterexample found by G∀ST is a function that matches the given input-output pairs. As outlined in the introduction, functional test systems like QUICKCHECK and G∀ST are not very good in generating functions and printing them. Instead of `prop0` we use a property over the data type `Fun`. The type `Fun` represents the grammar of candidate functions, see 3.2.3. The function `apply`, see 3.2.5, turns an instance of this data type in the actual function.

```
prop1 :: Fun → Bool
prop1 d = ∼(f 2 == 2 && f 4 == 24 && f 6 == 720) 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 function we are looking for, the well-known factorial function. More examples will be given below. In the next subsection we treat the structure of the type `Fun` and the synthesis of instances.

3.2.3 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 can become nonterminating as well. Termination can either be guaranteed by an upper limit on the number of recursive calls (if the candidate function does not terminate in N calls, it is rejected), or by only generating functions that are terminating by construction.

We will construct only terminating (primitive recursive) functions. For the integer domain, these functions either do not recurse, 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. An expression is either a variable, an integer constant or a binary operator applied to two 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 expression in an else-part is either a variable, a constant or a binary operator over a variable, a constant, or a recursive function application:

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

Note that the grammar rule for *BinOp* is parameterized by the arguments for the binary operators. This is convenient since we can now use this rule for *Expr* as for *Expr2*. This reuse of data types carries over directly to the implementation. Although the principle of parameterizing grammar rules is not completely standard, it is known as two level grammar, or Van Wijngaarden grammar, and is at least as old as the Algol 68 report.

This grammar is directly mapped to a data type in CLEAN [13]. We use the type `OR` to mimic the choice operator, |, used in the grammar.

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

The composition of types allows us to use a choice between types. This saves us from 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 abstract 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 similarly 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.

3.2.4 Generating candidate functions

The generic algorithm `ggen` [10] 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 set 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 commonly 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]
```

None 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. Using a maximum that is too small prevent the desired function from being found. Katayama uses only $f(x-1)$ in his recursion pattern.

3.2.5 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`, and the type of value `v` to be generated[3].

```
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
```

[3] In Haskell one would have to write
`class apply d e v where apply :: d → e → v` instead of this shorthand notation.

```
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 into an environment containing 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))
```

Note that the transformation of the syntax tree into the corresponding function is done only once for all recursive applications of the function (the generated function `f` is passed in the environment of the else-part). This more sophisticated implementation results in a faster execution than repeated interpretation of the data structure for recursive calls (by passing `rf` to the recursive calls).

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` and 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
```

3.2.6 Pretty printing generated functions

If we would derive showing of candidate functions in the generic way, we would obtain the following representation for the factorial function from Section 3.2.2.

```
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 generic instances of the print

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

TABLE 3.1. Input-output pairs and the synthesized functions.

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.

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 parentheses around compound expressions are needed. The third argument is the object to be printed. The last argument, `rest`, is a continuation. This continuation is the list of strings representing 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 X            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]]]
```

3.2.7 Examples

In order to demonstrate the power of our approach we list some examples in table 3.1. The first column of the table contains the input-output pairs the function has to match. The next columns contain the first matching function found, the number of tests and the time needed (in seconds) to generate this function. We used a 1 GHz AMD PC running Windows XP and the latest versions of CLEAN and G∀ST.

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 func-

tions per second. Private communication with Katayama indicates that our implementation is more than one order of magnitude faster then Katayama's. 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[4] $f\,0 = 1, f\,1 = 1, f\,2 = 2, f\,3 = 6, f\,4 = 24$ is

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

Using the paramorphism [12] `nat_para`, twice, as recursion pattern. The first occurrence of `nat_para` handles the recursion in the factorial function. The second instance of `nat_para` implements multiplication by repeated addition.

```
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 the fourth example in the table above, indicates that our system generates functions that are better readable. In addition our approach synthesizes a matching function faster, and the generated function is more efficient. Moreover, Katayama's system needs more input-output pairs to generate the desired factorial function. Katayama's system can be improved by adding primitive functions, like addition and multiplication, to the library.

3.3 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 = 9`. The three different ways to control the synthesis of functions are:

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 non-recursive, 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 prove 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.

[4] 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.

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

TABLE 3.2. Generating 10 matching functions in different ways.

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.

These techniques are demonstrated by synthesis of functions matching `f 1 = 3`, `f 2 = 6`, and `f 3 = 9`. G∀ST searches for non recursive solutions by testing:

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

G∀ST quickly finds functions like $fx = 0+((x+x)+x)$, $fx = (x-x)+((x+x)+x)$, and $fx = x+(x+x)$. In the property `pFun` we replace the type `Expr` of `pExpr` by `Fun`. When G∀ST tests this property, it will also generate recursive candidate functions. In table 3.2 we see that it takes longer to generate 10 matching functions for `pFun` than for `pExpr`. Since the synthesized recursive functions do not match the given condition, `pFun` has a lower success rate.

In order to get rid of redundant expressions like `x-x` or `x+0` in the generated functions, we filter them with the predicate `fit`:

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

The predicate `fit` is implemented as a class. The instance for binary operators is given as an example. A subtraction is fit if the arguments are unequal and each of the arguments is fit. An addition is fit if both arguments are unequal to the constant zero, checked by `is0`, and fit. A multiplication is fit if both arguments are unequal to 0 and 1 and fit.

```
class fit a :: a → Bool

instance fit (BinOp x) | gEq{|*|} x & isConst, fit x
where fit (OpMinus x y) = x =!= y && ~(is0 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
```

Defining a type without these redundant expressions is somewhat tricky, but doable. The key-step is to define operators as a separate type with different type-arguments as left and right arguments:

```
:: Sub x y = Sub x y
:: Es = Es (OR (Sub Var NConst) (Sub IConst Var))
```

In table 3.2 we see that it takes considerably more time to generate 10 matching functions if we filter redundant expressions. This is not surprising since also the rejected candidates are generated and all candidates are tested. Using more sophisticated types is more efficient since no fitness tests and generation of redundant expressions occurs, but requires more programmer insight.

3.4 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, integer 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 generated.

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

In the next section we will show functions having a list and an integer as example.

3.5 SYNTHESIS OF FUNCTIONS OVER OTHER DATA TYPES

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

given example	generated function	tests	time
g [1,2,3] = [1,2,3]	g y = y	1	0.01
g [1,2,3] = [1,4,9]	g y = map f y where f x = x*x	34	0.05
g [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

TABLE 3.3. Input-output pairs of type `[Int]→[Int]` and synthesized functions.

`apply` and `genShow`. We derive the generation of all types introduced in this section.

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 a slightly more advanced example we show how function over lists of integers, that is, of 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)
```

Although this are very restricted functions and not all that interesting, it shows how data types generating function can be reused. Some examples of its use are listed in Table 3.3.

In exactly the same way we can synthesize functions destructing recursive data types like lists and trees. As an example we let G∀ST synthesize product functions over a list of integers with the property:

```
pProduct :: ListFun → Property
pProduct d = fit d ⟹ ~(f [1,2,3] == 6 && f [] == 1 && f [5] == 5)
where f = apply d
```

Note that by changing `f [] == 1` to `f [] == 0` we will obtain the sum rather than the product.

The key to success is of course an appropriate type for `ListFun` and the associated instances of `apply` and `genShow`. Direct recursive functions can be synthesized by:

```
:: LFUN = LFUN IConst LEx              // expressions for nil and cons
:: LEx  = LEx (OR (OR Var IConst) (OR Rec (BinOp LEx)))  // note the recursion
:: Rec  = Rec                          // recursive call

:: Env = Env ([Int]→Int) Int [Int]  // environment: function, head, and tail

instance apply LEx Env Int where apply (LEx lex) = apply lex
instance apply Var Env Int where apply X = λ(Env f x l) → x
instance apply Rec Env Int where apply Rec = λ(Env f x l) → f l
```

```
instance apply LFUN [Int] Int
where apply (LFUN nil cons) = f
      where f []    = apply nil 0
            f [x:l] = apply cons (Env f x l)
```

One often prefers functions over lists with an accumulator in order to reduce the stack space needed by the synthesized function. This requires just another data type for functions:

```
:: AFun = AFun IConst AEx    // initial accumulator and body for recursion
:: AEx  = AEx (OR (OR Var IConst) (OR A (BinOp AEx))) // note the recursion
:: A    = A                  // accumulator

:: AEnv = AEnv Int Int       // environment: accumulator and head

instance apply AEx AEnv Int where apply (AEx ex) = apply ex
instance apply A   AEnv Int where apply A = λ(AEnv a x) → a
instance apply Var AEnv Int where apply X = λ(AEnv a x) → x
instance apply AFun [Int] Int
where apply (AFun c ex) = f (apply c 0)
      where f a [] = a
            f a [x:l] = f (apply ex (AEnv a x)) l
```

By choosing `(OR AFun LFUN)` for `ListFun` in the property above, G∀ST synthesizes both kinds of functions is one test. The first three matching functions are:

```
Counterexample 1 found after 8 tests:
f []    = 1
f [x:l] = (f l)*x
Counterexample 2 found after 677 tests:
f []    = 1
f [x:l] = x*(f l)
Counterexample 3 found after 1039 tests:
f l = g 1 l
where g a []    = a
      g a [x:l] = g (a*x) l
```

3.6 OTHER PROPERTIES

Having the candidate function available as a 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. In this section we show some other kinds of properties that can be stated about the desired functions. One possibility is to use the fully fledged test system to specify for instance properties containing additional for-all operators. Another possibility is to use the availability of the functions as data structures for symbolic manipulation.

Using the capabilities of the test systems it is for instance possible to search for nonrecursive functions that obey the rule $f0 = 0$ and $\forall x.2f(x) = f(2x)$. This

can directly be stated in 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 include recursive functions in the search space, we have to take care that the integers tried as arguments 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 infeasible 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 trees of the candidate functions are available, it is easy to manipulate the candidate functions. As an 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 X = toExpr (IConst 1)
instance ddx IConst where ddx c = 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. Having the candidate function available as a data structure also enables symbolic manipulations like computing the derivative.

3.7 RELATED WORK

This paper presents an application of the concept of systematic generation of functions to the area of programming by example. The basic idea to synthesize functions via the synthesis of a data structure representing their syntax tree is presented in [11]. There the functions generated are used for the automatic testing of higher order functions. The generated functions serve only as test arguments for the properties to be tested. We were interested in the property over the higher order function rather than the set of functions generated as test suite. The main quest was there to find errors in a library of continuation based parser combinators.

Here we are interested in the generated properties themselves, since the goal is to find a general function matching the given input-output pairs. The techniques of synthesizing functions is improved by the introduction of the type `OR`. This type allows us to model the choice of elements of two existing data types. The advantage is that data types, and hence the components of functions modeled by them, can be reused. Furthermore, this paper systematically shows what has to be done if the system generates undesirable (for instance non terminating) functions. The options are: 1) improve the data type such that the dangerous functions cannot be represented, 2) replace the generic generation of instances of this type by a tailor made generation such that only the desired instances are generated (this is only attractive for non recursive definitions), or 3) define a predicate over the data type that rejects unwanted candidates before they are tried.

In the area of programming by example through systematic synthesis of candidate functions and selecting a match candidate the most related work is [7] of Katayama. The main differences between our work and Katayama's approach are:

Type correctness of generated candidates The type system selects statically the grammar used to generate values of the desired function result, instead of a dynamic system that controls the generation of λ-expressions.

Recursion in the synthesized functions Our system is able to synthesize definitions of (primitive) recursive functions directly, instead of searching λ-expressions containing an instance of a paramorphism as recursion builder. Although our examples are primitive recursive functions, this is not an inherent limitation of the approach. By including a clause for a recursion builder, like `fold` for lists, in the grammar, the corresponding recursion pattern can be generated. In Katayama's system any recursion pattern wanted should be supplied as a (higher order) function in the library.

Control of the synthesis In Katayama's system the candidate functions are synthesized from ordinary λ-expressions and a library of functions. The recursion pattern has to be supplied in this library, since the generation of λ-expressions is not capable to generate recursion (for instance by a Y-combinator). By default Katayama's library of primitive functions provides two paramorphisms: one recursion pattern for integers and one for lists (similar to a fold function). This is sufficient since his system only handles recursive functions over integers and lists. Katayama's system generates type correct expressions in a breadth first way. The exact algorithm used is not revealed.

We use data types to control the generation of candidate functions. Using these types, the system becomes more open and much easier to adapt to special wishes. The generation of instances of these types is done by our general generic algorithm [10] instead of an ad-hoc algorithm. Using the techniques discussed in this paper the synthesis can be fine tuned if necessary. Due to the tailor made data types the functions synthesized are not restricted to integers and lists nor to specific recursion patterns. The price we have to pay for this flexibility is that we have to define new a data type and associated instance of `apply` for each new recursion pattern and data type. We have shown that it is possible to reuse (parts of) existing solutions.

In the recursion pattern we use here as an example for functions of type `Int→Int` we use various values as stop condition and step size in the recursive call. In Katayama's systems all of the desired combinations should be stated as separate recursion patterns, or the needed constants should be included as functions in the library. A consequence of that last action would be that these constants would be used in each and every position where the type fits. As shown above, this can be controlled very easily and accurately in our approach.

Tool support Our general test system is used to generate candidate functions, and to select and print matching functions. No changes of the tool are required whatsoever. Katayama uses a tailor-made tool.

The advantage of Katayama's system is that it is in principle able to generate any function over integers and lists. The system should be extended in order to handle other other types, like trees. The advantage of our approach is that it works for any type and any kind of function wanted. Since it synthesizes only instances of the defined abstract syntax trees, it is usually faster. Moreover, it tends to require less input-output pairs to find nice (recursive) functions. The price to be paid is that we have to define new data types and associated instances of `apply` for new kinds of functions.

Jeuring et al [6] generate generic functions that performs a task defined by some typical examples. They use Djinn to generate arms of a generic function for instances of the user-specified generic signature of the desired function on the type indices of generic functions. This is partly based on their misconception that our approach is not suited to generate higher order functions[5]. The selection of candidate functions is very similar to our approach, although they use QUICKCHECK rather than G∀ST. In [11] we show how higher order functions can be generated using this approach. It is interesting future work to find out of the generation of generic functions can be done based on our technique.

3.8 CONCLUSION

In this paper we have shown how functions matching given input-result pairs can be synthesized in a clear and flexible way. By defining an appropriate type for

[5] Jeuring at al state in their introduction: "..the approach of Koopman and Plasmeijer [15] does not seem to be able to generate higher-order functions.."

syntax trees as data structure, the user can control the structure of the synthesized functions. We have shown non-recursive functions as well as various recursion patterns. If other kind of functions are wanted (other data types and or recursion patterns) we just have to define a data type representing their syntax tree, derive their instantiation, and add an instance of `apply` that turns the syntax tree into the corresponding function.

Generating the instances of the data types representing the syntax tree 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` instead of `0` and `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 most functions a single page of additional CLEAN code is sufficient. This approach is more transparent, flexible and efficient than existing systems like [7]. Although the described system works excellently for many examples, synthesizing functions involving very large expressions or very large constants will take a very long time. This is due to the size of the search space and the systematic search.

Acknowledgement

The authors thank the anonymous referees for their contributions to improve this paper.

REFERENCES

[1] Debasish Banerjee. A methodology for synthesis of recursive functional programs. *ACM Transactions on Programming Languages and Systems*, 9(3):441–462, 1987.

[2] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 268–279. ACM Press, 2000.

[3] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *ACM SIGPLAN conference on Principles and Practice of Declarative Programming (PPDP)*, pages 162–174, 2001.

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

[5] Henry Lieberman (editor). *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[6] Johan Jeuring, Alexey Rodriguez, and Gideon Smeding. Generating generic functions. In *WGP '06: Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 23–32, New York, NY, USA, 2006. ACM Press.

[7] Susumu Katayama. Systematic search for lambda expressions. In *Proceedings Sixth Symposium on Trends in Functional Programming (TFP2005)*, pages 195–205, 2005.

[8] E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs. In *AISC 2002 and Calculemus 2002*, volume 2385 of *LNCS*, pages 26–37. Springer, 2002.

[9] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In Ricardo Peña and Thomas Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 84–100. Springer, 2003.

[10] Pieter Koopman and Rinus Plasmeijer. Generic Generation of Elements of Types. In *Proceedings Sixth Symposium on Trends in Functional Programming (TFP2005)*, Tallin, Estonia, Sep 23-24 2005.

[11] Pieter W. M. Koopman and Rinus Plasmeijer. Automatic testing of higher order functions. In *Fourth Asian Symposium on Programming Languages and Systems (APLAS)*, pages 148–164, 2006.

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

[13] Rinus Plasmeijer and Marko van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.ru.nl/~clean/.

[14] John C. Reynolds. Defuncitional 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).

[15] Ute Schmid and Jens Waltermann. Automatic synthesis of XSL-transformations from example documents. In M.H. Hamza, editor, *Artificial Intelligence and Applications Proceedings (AIA 2004)*, pages 252–257. Acta Press, 2004.

[16] Philip Summers. A methadology for LISP program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175, 1977.

[17] Malcolm Wallace and Colin Runciman. Recursion, lambda abstractions and genetic programming. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 422–431, 1998.

[18] Fritz Wysotzki and Ute Schmid. Synthesis of recursive programs from finite examples by detection of macro-functions. *Forschungsberichte des Fachbereichs Informatik der TU Berlin Nr. 2001-2*, (2).