Synthesis of Functions Using Generic Programming

Pieter Koopman and Rinus Plasmeijer

Nijmegen Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands
{pieter,rinus}@cs.ru.nl

Abstract. This paper describes a very flexible way to synthesize functions matching a given predicate. This can be used to find general recursive functions or λ-terms obeying an input–output behavior specified by a number of examples. Generating complex algorithms from just a small number of simple input-output pairs is the goal of inductive programming. This paper illustrates that our approach works well in some challenging examples.

1 Introduction

Inductive programming aims to synthesize functions or programs from a small number of input-output pairs. In general there will by many functions that have the desired behavior. From this family of solutions we are interested in the smallest or simplest solution. In some situations there are (often well-know) algorithms to construct such functions, for instance for fitting a linear function through a set of points in the $\mathbb{R}^2$. In general it is very hard to construct functions for arbitrary data types in this way. Instead of constructing a function that has the desired behavior we use a generate-and-test based approach. Our system generates a sequence of more and more complex candidate functions, the system verifies if these candidates have the desired behavior and yields the first candidate that passes this test.

Since there are enormous many candidate functions one has to guide this search process in one way or another to synthesize the desired function in reasonable time. In this paper we show how we can control the synthesis of candidate functions effectively by defining a tailor made data type for the grammar of the candidate functions. The instances of these data types represent the candidate functions, in fact the generated instance of the data type are the abstract syntax trees of the corresponding functions. In contrast with real functions, these syntax trees can be easily inspected and manipulated.

To reduce the manual effort in defining algorithms to generate candidate candidate functions we introduce a generic algorithm that enumerates the instances of any (recursive) data type from small to large. We show how we can use this to generate tailor-made candidate functions with very little effort. Usually we only have to specify the constants to be used explicitly, everything else is done by the generic algorithm and the type definitions.

U. Schmid, E. Kitzelmann, and R. Plasmeijer (Eds.): AAIP 2009, LNCS 5812, pp. 25–49, 2010.

It appears that the generic algorithm for generating instances of a data type that is used to generate test suites in the model-based test tool G∀st is very effective to synthesize candidate functions in inductive programming. In order to verify if a synthesized abstract syntax tree represents the correct function, the system needs to be able to execute it as a function. This is done by a user defined function that transforms the abstract syntax tree to the corresponding function.

Using a test system to generate candidate functions and check their suitability has additional advantages. Instead of specifying just input output pairs for the functions one can specify an arbitrary predicate in first order logic.

For a new application domain the user has to define the grammar of candidate functions as a data type and how instances of this data type are transformed to functions. Next the user specifies a predicate about the specific function wanted. The system synthesizes the instances and tests the candidates until one (or more) functions with the desired behavior are found.

This paper first gives an explanation of the generate and test approach to synthesize functions in section 2. In section 3 we explain how the candidate functions can be synthesized using generic programming. Section 4 shows how the desired functions can be selected from the candidates with the model-based test tool G∀st.Then we show some nontrivial examples of our approach. Keppler's third law relating the distance of planets and their period is rediscovered empirically in section 5. Next we show how one can synthesize primitive recursive function in section 6. Section 7 shows how to synthesize complex λ-terms. Finally we discuss related work, section 8, and conclude in section 9.

2 The Generate and Test Approach to Synthesis Functions

It is a challenging idea to create a computer system that is able to produce the function we have in mind based on just a few examples from input and output. On one hand it is obvious that such a system cannot exist for arbitrary functions, e.g. we cannot expect a function that solves the halting problem based on some examples of terminating and nonterminating functions. On the other hand there are couple of examples in the literature (e.g. [8,3,6,11,14]) that show that these kind of systems can be constructed and that these systems are capable to find solutions in a number of situations.

In this paper we are looking for a system that synthesizes a function based on some partial specification, usually a small number of typical input–output pairs. Since the specification given by these input–output pairs is partial (usually the given inputs are only a small fraction of the domain of the function) there are generally many functions that match this specification. A trivial one is the function that maps only the given inputs to the associated outputs. Such a function is not what we are looking for. Apart from mapping the given inputs to outputs we have the following constraints:

1. A function with a small body is considered to be better than a function with a big body. We will use the size of the abstract syntax tree as a measure for the size of the function body.
2. As a consequence of the previous point we generally prefer a nonrecursive definition over a recursive one and single recursion over double recursion.
3. The function should contain at most a few special case for specific arguments. A recursive function needs of course some stopping criterion, but we do not want many special cases. If there are special cases the should preferably handle the non-recursive alternatives of a recursive data type (e.g. empty list or empty tree), or common stop criteria for other functions (e.g. the numbers 0 or 1).

These additional properties are not added to the specification. In order to specify that one function is smaller or simpler than another solution we need both functions and compare them. Instead we will use a predicate to capture only the constraints like input–output pairs. The numbered constraints will be met by the kind and order in which candidate functions are generated. When we do not generate functions with excessive pattern matching on input arguments, such a function will never be found. By generating candidate functions from small to large the first function matching the constraints will be the smallest function we are looking for.

An example illustrates the preference of functions. Suppose we are looking for a function f that has the following behavior: $f(0) = 0$, $f(2) = 4$ and $f(3) = 9$. Some functions displaying the required input–output behavior are:

```
f1 0 = 0
f1 2 = 4
f1 3 = 9
f1 x = x

f2 x = x*x

f3 x = x^2

f4 x = g x
where
    g 0 = 0
    g y = x + g (y-1)
```

From these functions `f1` is clearly undesirable, it contains too much specific patterns for the given input–output pairs. The functions `f2` and `f3` are equally good, they are small and meet the desired input–output relations with a general pattern. Function `f4` implements the multiplication by repeated addition. Since its definition is larger than the definition of `f2` and `f3` we do not prefer the solution `f4`.

2.1 Partial Specification of the Functions

We prefer a richer specification language than just input–output pairs. We also want to be able to express properties like $f(1) \geq 0$, or even $\forall\, x\,.\, f(x) \geq (0)$ and $\forall\, x\,.\, f(x) \geq (0) \Rightarrow f(x+1) > f(x)$. To be able to specify this kind of properties

we use first order logic as specification language for the functions we are looking for rather than only input–output pairs.

The predicate corresponding to the input–output pairs $f(0) = 0$, $f(2) = 4$ and $f(3) = 9$ becomes $p_1(f) = f(0) = 0 \wedge f(2) = 4 \wedge f(3) = 9$. In our implementation this is modeled by a Boolean–valued function in the functional programming language Clean [13].

```
p1 :: (Int→Int) → Bool
p1 f = f 0 = 0 && f 2 = 4 && f 3 = 9
```

Our test system G∀st provides a full range of logical operators. Using some of these logical operators the predicate $f(2) = 4 \wedge \forall\, x\,.\, f(x) \geq (0) \Rightarrow f(x{+}1) > f(x)$ can be written as

```
p2 :: (Int→Int) → Property
p2 f = f 2 = 4 ∧ ForAll λx . f x ≥ 0 ⟹ f (x+1) > f x
```

Each property that uses logical operators from G∀st yields `Property` instead of `Bool`. This is necessary in the implementation, but the user can consider this type as an equivalent for Booleans.

Since these predicates in general do not pinpoint the desired functions completely, the predicates are partial specifications.

2.2 Automatic Test Systems

Automatic test systems like G∀st [10] and QuickCheck [5] are designed to handle these kind of predicates. The test system is designed to falsify a property by finding a counterexample. A typical example of such a property for the functions `abs` that computes the absolute value of an integer is $\forall\, i \in \mathrm{Int}\,.\,\mathrm{abs}(i) \geq 0$. Expressed as a Boolean function that can be handled by G∀st this is:

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

This property can be tested automatically by executing

```
Start = test pAbs
```

To test this property the system executes the following subtasks.

Test suite generation. The test suite is the collection of values that will be used in the test. For our test tool the test suite is a, potentially infinite, list of values.

In this example the function `test` detects that the property `pAbs` ranges over integer values. A test suite for the type integer and other predefined types is provided by G∀st.

If we want to deviate in a specific test from the predefined test suite we can use the operator `For`. The property `pAbs` can be tested for integers between -100 and 100 by executing

```
Start = test (pAbs For [−100..100])
```

Test execution. Since the property is generated and the test suite is given as a list of values, test execution is basically just a map of the property over the test suite.

Generating a verdict. The test system generates a verdict by inspecting the first N (by default 1000) Boolean values in the list generated by test execution. Basically the property passes the test is all Booleans have the value `True` and fails otherwise.

In reality the verdict is a little more detailed. Possible verdicts are:

Proof. The property holds for all elements of the test suite. Such a proof by exhaustive testing is only possible when the size of the test suite is smaller than the maximum number of tests to be done.

Executing the test `Start = test (pAbs For [−100..100])` yields Proof since there are only 201 test cases and they all succeed.

Pass. If all tests done are successful, but there are more values in the test suite than the maximum number of tests to be done, the test result is Pass.

Counterexample. If one of the test results is `False` the property obviously does not hold. The test system G∀st does not only yield the test result Counterexample, but also prints the test value that causes this counter example. Moreover, it is possible to indicate that one wants at most M, by default 1, counterexamples in the first N test cases.

Executing the test `Start = test pAbs` yields the counterexample -2147483648, which is the minimum integer number of the 32-bit integers. This counterexample is found almost immediately since integers that are known to be often good test values (like 0, 1, ,-1, maxint, and minint) are placed near the head of the test suite for integers.

Testing of properties is not restricted to properties with a single universal quantified variable, or predefined data types. Suppose we have a function `rev :: [x] → [x]` that reverses lists. A desirable property is given by the list law $\forall\, xs,\ ys\,.\, rev\ (xs{+}{+}\, ys) = rev\ ys +\!\!+ rev\ xs$. This law [4] can be directly used to formulate a property to be tested by G∀st. We only have to add a data type to be used in the test and make sure to use a defined instance for the equality.

```
pRev :: [Color] [Color] → Bool
pRev xs ys = rev (xs++ys) == rev ys ++ rev xs

:: Color = Red | Yellow | Blue
```

In the next section we show generic, also called polytypic, programming [2] removes the burden to define these things from the user of the test system. The test system has generic definitions for operations like generation of test suites, equality of elements and showing the elements. The desired operations can be derived by the compiler from the generic definitions. The user just has to write

```
derive ggen    Color   // generic generation of the list of all Colors
derive gEq     Color   // generic equality for Color
derive genShow Color   // generic show (transformation to strings) for Color
```

Now the property can be tested by executing `Start = test pRev`. For a correct function `rev` the test result will be Pass. For a correct function there will be no counterexamples, but the generated test suite is an infinite list of lists of colors. For an incorrect implementation a typical test result is: `Counterexample 1 found after 1 tests: [Yellow] [Blue]`. The generic algorithm generates test

values from small to large. This implies that the counterexamples found are the smallest counterexample that exists. Having small counterexamples is beneficial since this makes it usually much easier to find the bug in the system under test (SUT, here the function `rev`).

Although the goal of these test systems is just opposite to inductive programming, we can reuse the automatic test machinery in inductive programming.

2.3 Selecting Functions with an Automatic Test System

Above we developed a property for G∀st that captures the desired input–output behavior. The goal of inductive programming is to find a function that satisfies this predicate, and hence posses the desired input–output relation. A test system is designed to find counterexamples, which is just the opposite of finding evidence that such a function exists. Instead of change the test system to an inductive programming system we will change the properties. Constructing a new system requires additional work to change the system. Moreover, we need to maintain two systems.

By a small change of the properties we obtain exactly the desired effect. Instead of specifying what properties the desired function has, we specify that all functions does not poses the desired properties. Counterexamples found by the test system are exactly the functions we are looking for.

For example we replace property `p1` from section 2.1 by

```
p1' :: (Int→Int) → Bool
p1' f = ¬(f 0 = 0 && f 2 = 4 && f 3 = 9)
```

Using De Morgan's law this can also be written as:

```
p1'' :: (Int→Int) → Bool
p1'' f = f 0 ≠ 0 || f 2 ≠ 4 || f 3 ≠ 9
```

The generation of candidate functions is now the only missing part. This is handled in the next section.

3 Generic Synthesis of Functions

The crux of the synthesis of functions using generic programming is the systematic generation of candidate functions. In order to limit the search space we will use a data type that corresponds directly to the grammar of the candidate functions.

As an example we start with arithmetic expression with a single variable. The syntax is:

$$
\begin{aligned}
Expr &= IConst \mid Var \mid BinOp\ Expr \\
BinOp\ e &= e\ +\ e \mid e\ -\ e \mid e\ \times\ e \mid e\ \hat{}\ PConst \\
Var &= X \\
IConst &= 1..5 \\
PConst &= 2..4
\end{aligned}
$$

We have used a higher order grammar rule for *BinOp* in order to reuse it later with a different argument.

Each grammar rule is directly mapped to an algebraic data type. In order to reduce the number of constructors needed we will use the data type OR to indicate a binary choice.

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

The data types corresponding to the grammar rules above are:

```
:: Expr    = Expr (OR (OR Var IConst) (BinOp Expr))
:: BinOp e = OpPlus e e | OpMinus e e | OpTimes e e | OpPower e PConst
:: Var     = X
:: IConst  = IConst Int
:: PConst  = PConst Int
```

For the constants IConst and PConst we have added a constructor to make it a data type on its own instead of using the type synonym `:: IConst := Int`. These separate data types appear to be convenient in the generation of instances.

Using these data types the expression $(X + 1)\hat{}2$ is represented by a data type of the form `OpPower (OpPlus (L X) (R (IConst 1))) (R (PConst 2))` of type `BinOp (OR Var IConst`.

The next step is generating instances of these data types that are going to be used as candidate function bodies. Rather than defining this for each and every data type over and over again we are going to define one generic algorithm that is able to enumerate the instances of any data type.

3.1 Generic Programming

The basic idea of generic programming is very simple. It is based on a uniform representation of arbitrary, user defined, data types. The language compiler can transform instances of an arbitrary data type to this uniform representation and from this representation back to the original data type. If we need a class of similar function we define the function on the generic representation instead of on all types individually. Famous examples are operations like equality and pretty printing etcetera. Generic programming is however by no means limited to these simple examples.

Generic Representation of Values. The uniform representation of data types is constructed with ordinary algebraic data types. These data type are used to construct binary trees representing the usual constructors. The basic types to construct these binary trees are:

```
:: UNIT          = UNIT                 // leaf
:: EITHER a b = LEFT a | RIGHT b        // choice
:: PAIR    a b = PAIR a b               // grouping
```

The type UNIT represents the leaves of the binary tree. The type EITHER is used to indicate a choice. Using these choices the representation indicates what

constructors is actually used. This is very similar to the type OR introduced above[1]. The type PAIR is used to glue things together, typically arguments to constructors.

In addition to the basic types it appears to be convenient to have some additional types carrying information about objects and constructors. We only introduce the type CONS indicating explicitly that there is a constructor at this spot in the generic representation. In the Clean version of generic programming this constructor is able to provide information about the actual constructor (like name, arity, type it belong to etc.).

```
:: CONS a = CONS a
```

The generic representation of the type Color introduced in section 2.2 is

```
:: Color_g = EITHER (CONS UNIT) (EITHER (CONS UNIT) (CONS UNIT))
```

The generic representation of the constructors Red, Yellow, and Blue from this type become:

```
Red_g    = LEFT (CONS UNIT)
Yellow_g = RIGHT (LEFT (CONS UNIT))
Blue_g   = RIGHT (RIGHT (CONS UNIT))
```

If a constructor has an argument, this argument replaces the place holder UNIT in the generic representation. For example the generic representation of the type IConst is:

```
IConst_g = CONS Int
```

As an example of of grouping things together by PAIR we give the generic representation of x1. This value represents the expression $x + 1$ as an algebraic data type of type BinOp (OR Var IConst) (a binary operator expression over variables or constants).

```
x1 :: BinOp (OR Var IConst)
x1 = OpPlus (L X) (R (IConst 1))
```

The generic representation of this expression is:

```
LEFT (LEFT ((CONS (PAIR (LEFT (CONS (CONS UNIT)) (RIGHT (CONS (CONS 1))))))))
```

Here the PAIR glues both arguments of OpPlus together. This generic form of $x+1$ is huge and quite incomprehensible. Fortunately, those generic representations of expressions are usually generated. The transformation between the generic representation and the usual representation of data types can always be handled automatically by the compiler.

Generic functions. The power of generic programming is that an operation can be applied to an arbitrary data type by defining it only for the basic generic types (UINT, EITHER, PAIR, and CONS). Since the transformation of the data type to

[1] In fact there is no need to introduce the type OR, we can use EITHER equally well. We have introduced OR only to prevent confusion between the ordinary domain of data types and the domain of generic representation of these data types.

its generic representation is done by the language implementation all we have to do is to provide an instance for the four generic data types.

The classical example of generic programming is equality. First we define the general generic function, similar to a class definition.

```
generic gEq a :: a a → Bool
```

Next we define instances for the generic types and the basic types used in our program.

The instant for UNIT is very simple, there is only one constructor in the data type (line 1 in the code block below). Without looking at the argument supplied we known that the elements must be equal.

The type EITHER indicates a choice. The given elements can only be equal if the make the same choice between LEFT and RIGHT. If they make the same choice we have to compare the arguments (line 2 and 3). In contrast with a class the comparison of the arguments of LEFT and RIGHT is not done by an overloaded recursive call of geq. In the generic programming variant implemented in Clean the functions to compare the type arguments of EITHER are supplied as additional arguments by the generic system. In the code below we call these functions fl and fr.

Since the type PAIR has also two type arguments the generic instance of geq for PAIR has also two additional functions as arguments. The type PAIR has only one alternative (line 5). Hence we can immediately start with comparing the arguments of the constructor PAIR using the given functions.

For the single argument type CONS we have only one additional function. Since there is again only one constructor in the type, the only task we have is to compare the function arguments using the given function f (line 6).

In this example the only basic type needed is Int. Integers are compared using the ordinary equality on integers (line 7).

All code for defined instances of geq together is:

```
geq{|UNIT|}                _                _                = True                                  1
geq{|EITHER|} fl fr (LEFT x)      (LEFT y)      = fl x y                                 2
geq{|EITHER|} fl fr (RIGHT x)     (RIGHT y)     = fr x y                                 3
geq{|EITHER|} fl fr _             _             = False                                  4
geq{|PAIR|}   fx fy (PAIR x1 y1) (PAIR x2 y2) = fx x1 x2 && fy y1 y2                   5
geq{|CONS|}   f     (CONS x)      (CONS y)      = f x y                                  6
geq{|Int|}          x             y             = x == y                                 7
```

For any other data type we can define an instance like the instances shown above. The power of generic programming however is that we can *derive* these instances.

```
derive geq OR, BinOp, Var, IConst, PConst
```

Now you can use the operation geq for all types mentioned. The Clean system implements those operations by transforming the instances of the type to their generic representations and comparing those representations using the definitions given above.

Using those definitions we can compare the values IConst 5 and IConst 7 by executing

```
Start = geq{|*|} (IConst 5) (IConst 7)
```

As we might expect and hope the result is `False`.

One might wonder what the result of comparing `IConst 3` and `PConst 3` world be. These values have the same generic representation, but a different type. If one tries to compare them in an expression like `geq{|*|} (IConst 3) (PConst 3)`, this fails rather than yielding `True`. This expression contains a type error since the generic function definition `generic gEq a :: a a → Bool` requires that both arguments have the same type.

It is important to realize that generic programming is by no means limited to the simple classical examples like equality and pretty printing. In this paper we will use it to generate the instances of data types.

3.2 Generic Generation of Instances of a Data type

Now we consider the task of generating a list of values for all types. We approach this task by generic programming. Our algorithm will generate the generic instances of those values and the Clean system will convert those generic values to ordinary values whenever desired.

What we need is a generic function that yields a list of all values of a given type. This is:

```
generic gengen a :: [a]
```

Again we define the instances for the generic types and derive the instances of other types whenever possible.

The instance for UNIT is again very simple (line 1 in the numbered code block below). There is only one value of this type: the constructor UNIT. So, the list of values contains only this constructor.

For the type CONS we only have to apply the constructor CONS to all possible arguments (line 2). The list `i` of possible arguments is supplied by the generic system, just as the functions to compare arguments in `geq` above.

For the type PAIR we have to combine the elements from the given lists is all possible ways. We use the library function `diag2` to ensure that the elements are mixed in a 'fair' way. This prevents that we take the first element of one of the lists and pair it with all elements of the second list before we consider the second argument of the first list. To illustrate this mixing of list elements with some ordinary types we consider the unbounded lists of integers `[0..]` and the list of characters `['a'..]`. The expression `diag2 [0..] ['a'..]` yields

```
[(0,'a'),(1,'a'),(0,'b'),(2,'a'),(1,'b'),(0,'c'),(3,'a'),(2,'b'),(1,'c'),(0,'d'),..
```

An ordinary combination of list elements with `[(i,c) \\ i←[0..], c←['a'..]]` yields

```
[(0,'a'),(0,'b'),(0,'c'),(0,'d'),(0,'e'),(0,'f'),(0,'g'),(0,'h'),(0,'i'),(0,'j'),..
```

Here only the integer `0` is used.

For the choice between elements from two lists in the type EITHER we apply the combinators LEFT and RIGHT to the elements in the given lists (line 4). The function `merge` merges the resulting list by taking repeatedly one element from the first list and one element from the second list.

The instance of the generic generation of lists of values for integers is defined such that it yields the list `[0,1,-1,2,-2,3,-3,..` (line 9).

```
gengen{|UNIT|}          = [ UNIT ]                                          1
gengen{|CONS|}    l     = map CONS l                                        2
gengen{|PAIR|}    l m   = [ PAIR a b \\ (a,b) ← diag2 l m ]                  3
gengen{|EITHER|}  l g   = merge (map LEFT l) (map RIGHT m)                  4
where                                                                       5
    merge []     m      = m                                                 6
    merge l      []     = l                                                 7
    merge [a:r]  m      = [a:merge m r]                                     8
gengen{|Int|}           = [0:[j\\i←[1..],j←[i,-i]]]                         9
```

After these preparations we can derive the generation of our data types by

```
derive gengen OR, BinOp, Var
```

From the syntax in section 3.2 we see that the values for IConst vary from 1 to 5 and the values for PConst range from 2 to 4. This implies that they cannot be derived. By deriving those values all integers would occur. Instead of deriving we use tailor made definitions for these types.

```
gengen{|IConst|} = map IConst [1..5]
gengen{|PConst|} = map PConst [2..4]
```

Using this we can generate a list of expressions of type BinOp (OR Var IConst) just by writing

```
l :: [BinOp (OR Var IConst)]
l = gengen{|*|}
```

The first 10 expressions generated are:

```
[OpPlus (L X) (L X)                     // x+x
,OpTimes (L X) (L X)                    // x×x
,OpMinus (L X) (L X)                    // x-x
,OpPower (L X) (PConst 2)               // x^2
,OpPlus (R (IConst 1)) (L X)            // 1+x
,OpTimes (R (IConst 1)) (L X)           // 1×x
,OpMinus (R (IConst 1)) (L X)           // 1-x
,OpPower (R (IConst 1)) (PConst 2)      // 1^2
,OpPlus (L X) (R (IConst 1))            // x+1
,OpTimes (L X) (R (IConst 1))           // x×1
]
```

The mechanism to produce instances of data types introduced here appears to be very general. If we want an abstract syntax tree for an other grammar, we just define a new data type that mimics this syntax. For the generation we derive whatever possible and use a tailor made definition for the other types. The pattern seen here appears to be common, everything except the constants represented by basic types can be derived. The required manual definitions are very simple.

For instance if we want recursive expression given by the syntax

$$Expr = Var \mid IConst \mid BinOp\ Expr$$

we define the recursive data type Expr.

```
:: Expr = Expr (OR (OR Var IConst) (BinOp Expr))
```

After deriving gengen for Expr we can generate those expressions. The first 10 expressions generated are:

```
[Expr (L (L X))                                                    // x
,Expr (R (OpPlus (Expr (L (L X))) (Expr (L (L X)))))               // x+x
,Expr (L (R (IConst 1)))                                           // 1
,Expr (R (OpTimes (Expr (L (L X))) (Expr (L (L X)))))              // x×x
,Expr (L (R (IConst 2)))                                           // 2
,Expr (R (OpMinus (Expr (L (L X))) (Expr (L (L X)))))              // x-x
,Expr (L (R (IConst 3)))                                           // 3
,Expr (R (OpPower (Expr (L (L X))) (PConst 2)))                    // x^2
,Expr (L (R (IConst 4)))                                           // 4
,Expr (R (OpPlus (Expr (R (OpPlus (Expr (L (L X)))
                                  (Expr (L (L X))))))              // (x+x)+x
                 (Expr (L (L X)))))]
```

The actual generic generation algorithm `ggen` used by G∀st uses a pseudo random choice between the list with LEFT elements and RIGHT elements instead of a strict interleaving. As a result the order of elements in the resulting lists has a slight pseudo random perturbation compared with the algorithm presented here. Testers are found of such randomness. Here it does not harm us, but neither is a big advantage. In the rest of the paper we will use the generic function `ggen` from G∀st instead of the somewhat simpler version `gengen` introduced here.

3.3 Transforming Syntax Trees to Functions

Now we are able to generate abstract syntax trees of candidate functions in a convenient and high level way. Just by changing the algebraic data types representing the syntax trees, we can change the candidate functions considered.

However, in order to evaluate a predicate over a candidate function we do need the function instead of its abstract syntax tree. In order to construct these functions we define the class `apply`. The functions in this class produce a value `v` given a data type instance `d` and an environment `e`. As usual in interpreters and semantically descriptions this environment is used to store bindings of variables to values.

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

The first instance is for the type OR. The type restriction `apply x b c & apply y b c` says the we need to be able to apply the types `x` and `y` for the given binding `b` and value `v`[2]. All this function `apply` does is removing the constructor LEFT or RIGHT and apply the appropriate function `apply` to the argument of the constructor.

```
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
```

Slightly more interesting is the instance of `apply` for binary operations, `BinOp x`. The definition just transforms the arguments of the operator to a value of type `v` by `apply x b v` or `apply PConst b v` and applies the indicated operator to the result. The class restriction just requires that all the operations are available.

[2] In Haskell one write such a type restriction as

```
instance (apply x b c, apply y b c) => apply (OR x y) b c
  where ..
```

```
instance apply (BinOp x) b v | apply x b v & +, −, ∗, ^ v & apply PConst b v
where
    apply (OpPlus  x y) = λb.apply x b + apply y b
    apply (OpMinus x y) = λb.apply x b − apply y b
    apply (OpTimes x y) = λb.apply x b ∗ apply y b
    apply (OpPower x p) = λb.apply x b ^ apply p b
```

Our very simple expressions of type Expr from section 3.2 just have one variable X. The environment needed to evaluate these expressions can be accordingly simple, we just have to store the value of this single variable. If we assume that this variable is of type Int we have:

```
instance apply IConst b   Int where apply (IConst i) = λb.i
instance apply PConst b   Int where apply (PConst i) = λb.i
instance apply Var    Int Int where apply X          = λi.i
instance apply Expr   Int Int where apply (Expr f)   = apply f
```

For the constants IConst and PConst we just ignore the binding environment b and yield the stored value. For a variable, Var, we produce the value stored in the environment. For an expression, Expr, we just remove the combinator and continue recursively.

Here it pays off to use the type OR instead separate constructors for all alternatives. If we had used separate constructors for the alternatives we would need one alternative of apply for each constructor. In our current approach the instance of apply handles all choices in the syntax.

After all these preparations we can reformulate our predicate and start the test system. The difference between this version of the predicate, p1e, and the predicate p1‘ from section 3.2 is that p1e ranges over Expr while p1‘ ranges over functions of type Int→Int. The test system generates instance e of type Expr effectively by the given instance of ggen. The generated abstract syntax tree e is transformed to the desired function f by the appropriate instance of apply. In this example we execute at most 1000 test and stop after finding 10 counterexamples, hence we use testnm instead of test since test will produce only one counterexample.

```
p1e :: Expr → Bool
p1e e = ¬(f 0 = 0 && f 2 = 4 && f 3 = 9) where f = apply e

Start = testnm 1000 10 p1e
```

The result produced by G∀st in 0.4 seconds is:

```
Counterexample 1 found after 16 tests: (x∗x)
Counterexample 2 found after 22 tests: ((1∗x)∗x)
Counterexample 3 found after 38 tests: (x^2)
Counterexample 4 found after 358 tests: ((x∗1)^2)
Counterexample 5 found after 381 tests: ((x^2)+(1−1))
Counterexample 6 found after 453 tests: ((x+(x∗x))−x)
Counterexample 7 found after 491 tests: ((x∗x)−(x−x))
Counterexample 8 found after 582 tests: (((x^2)+x)−x)
Counterexample 9 found after 713 tests: ((1+(x∗x))−1)
Counterexample 10 found after 762 tests: (1∗(x∗x))
```

These counterexamples of the predicate p1 are all functions matching the input–output patterns $f\ 0 = 0$, $f\ 2 = 4$, and $f\ 3 = 9$.

Obviously we used a tailor made instance of the generic show function rather than deriving an instance. Our instance removes all unnecessary constructors and

prints the binary operations as infix operators. The pretty printer here gives only the body of the function found. The first solution found should be understood as `f x = x*x`.

4 Selecting Candidate Solutions

Looking at the solutions found at the end of the previous section we notice that a desired solution is found quickly and is the first solution found. However, many of the other solutions have a rather undesirable form. For instance, for a human it is obvious that the second solution `f x = (1*x)*x` represents semantically exactly the same solution as the first one `f x = x*x`.

There are at least three ways to avoid those kind of undesirable solutions.

1. One can design a better syntax that excludes those undesirable solutions. It is obvious how this should be done. Since the undesirable candidates cannot be represented in the new data types, they will never be considered. The advantage is that we obtain a complex syntax, and hence data type, for simple expressions. We will not elaborate on this since it is obvious how it should be done and we prefer a simple syntax (and hence data types).
2. We can adapt the generation of instances such that the undesirable candidate functions are never considered. This is an unattractive solution since we now have to define a generation algorithm manually instead of reusing the generic algorithm.

   In section 7 however we will provide an elegant solution that combines a simple data structure, generic generation and tailor made instances.
3. Finally we can at runtime exclude undesirable candidate solutions. This is possible since the candidate function is available as an abstract syntax tree. We can easily write a predicate `fit` that inspects the syntax tree and yields a Boolean indicating if this candidate function should be used. We will illustrate this solution here.

We will illustrate the selection of candidate functions here. First we define a class `fit` that determines if the candidate is healthy.

```
class fit a :: a → Bool
```

The maximum penalty for making the predicate not advanced enough is that a candidate function is considered that actually has not the desired form. The instances of this class presented below are pretty straightforward, of course we can make these predicates as cleaver as desired.

For the choice type `Or` the instance of `fit` determines what alternative we have at hand and applies the appropriate version of `fit` recursively.

The real work happens in the instance of `fit` for binary operations. For $x - y$ we require that $x \neq y$, $y \neq 0$, x is fit, y is fit, and that x and y are not both constants. For the other alternatives we impose similar constraints.

An expression of the form `Expr e` is fit if `e` is fit.

```
instance fit (OR s t) | fit s & fit t
where
    fit (L x) = fit x
    fit (R y) = fit y

instance fit (BinOp x) | gEq{|*|} x & isConst, fit x
where
  fit (OpMinus x y) = x =!= y   && ¬(is0 y)  && fit x && fit y && ¬(isAny x && isAny y)
  fit (OpPlus x y)  = ¬(is0 x)  && ¬(is0 y)  && fit x && fit y && ¬(isAny x && isAny y)
  fit (OpTimes x y) = ¬(is01 x) && ¬(is01 y) && fit x && fit y && ¬(isAny x && isAny y)
  fit (OpPower x (PConst p)) = p>1 && fit x

instance fit Expr where fit (Expr e) = fit e
```

In the code above we used the following predicates to decide if some data type represents the constant 0 (`is0`), the constant 0 or 1 (`is01`), or any constant (`isAny`).

```
is0 x   = isConst (λi.i==0) x
is01 x  = isConst (λi.i==0 || i==1) x
isAny x = isConst (λi.True) x
```

These predicates are built on top of the class `isConst` defined as:

```
class isConst a :: (Int→Bool) a → Bool

instance isConst (OR s t) | isConst s & isConst t
where
    isConst p (L s) = isConst p s
    isConst p (R t) = isConst p t
instance isConst IConst where isConst p (IConst i) = p i
instance isConst Expr   where isConst p (Expr e)   = isConst p e
instance isConst a      where isConst p a          = False
```

Using this predicate we can update our predicate to find functions matching $f(0) = 0$, $f(2) = 4$ and $f(3) = 9$ to:

```
p2 :: Expr → Property
p2 d = fit d ⟹ ¬(f 0 == 0 && f 2 == 4 && f 3 == 9) where f = apply d
```

In order to test the first 1000 candidates, the test system rejects 738 candidates that are not `fit` and not counted as a test. Within one second the test system produces the following result:

```
Counterexample 1 found after 13 tests: (x*x)
Counterexample 2 found after 24 tests: (x^2)
Counterexample 3 found after 253 tests: ((x+(x*x))-x)
Counterexample 4 found after 332 tests: (((x^2)+x)-x)
Counterexample 5 found after 419 tests: ((1+(x*x))-1)
Counterexample 6 found after 654 tests: (((x^2)+(x+x))-(x+x))
```

This shows that a number of undesirable results are removed. One might argue that the solutions from 3 up to 6 are all undesirable. They can be excluded by improving the predicate `fit` as well.

This concludes the generation of candidates for this simple example. Perhaps the reader wonders that we needed quite a heavy equipment to find rather simple functions. There are two answers to this concern. First and foremost, the test tool G∀st including the generic generation of instances of data types is existing technology. It is treated here to make this paper self-contained, but only the application as inductive programming tool is new. Second, the approach introduced here can also be applied to many other and more complicated problem areas. We show a couple of those applications in the next sections.

5 Keppler's Third Law

Kepler (1571-1630) studied the motion of planets of the Sun. He is famous for formulating 3 laws about the motion of planets. His third law was formulated more than ten years after the first two laws. This third law quantitatively relates orbital period and distance of the planet to the Sun. Apparently it was hard for him to find this law. This may be partially caused by the kind of equipment and data available in those days.

In order to test the power of our approach we try to rediscover Kepler's third law from data about the planets found on Wikipedia [1]. The basis of our data is table 1 containing the diameter, mass, orbital radius, and orbital period of the planets of our Sun. These parameters are given in astronomical units (AU), which means that these parameters are relative to the parameter for the Earth.

Table 1. Parameters of the planets

Name	Equatorial diameter (AU)	Mass (AU)	Orbital radius (AU)	Orbital Period (years)
Mercury	0.382	0.06	0.39	0.24
Venus	0.949	0.82	0.72	0.62
Earth	1.00	1.00	1.00	1.00
Mars	0.532	0.11	1.52	1.88
Jupiter	11.209	317.8	5.20	11.86
Saturn	9.449	95.2	9.54	29.46
Uranus	4.007	14.6	19.22	84.01
Neptune	3.883	17.2	30.06	164.8

In our synthesis this table is represented as a list of 5-tuples. Each tuple represents one line in the table.

```
// [(name, diameter, mass, orbital radius, and orbital period)]
planetTable :: [(String,Real,Real,Real,Real)]
planetTable
  = [("Mercury" ,0.382  ,0.06  ,0.39  ,0.24)
    ,("Venus"   ,0.949  ,0.82  ,0.72  ,0.62)
    ,("Earth"   ,1.00   ,1.00  ,1.00  ,1.00)
    ,("Mars"    ,0.532  ,0.11  ,1.52  ,1.88)
    ,("Jupiter" ,11.209 ,317.8 ,5.20  ,11.86)
    ,("Saturn"  ,9.449  ,95.2  ,9.54  ,29.46)
    ,("Uranus"  ,4.007  ,14.6  ,19.22 ,84.01)
    ,("Neptune" ,3.883  ,17.2  ,30.06 ,164.8)
    ]
```

For Keppler's third law we are looking for a function giving the period as function of the mass and the distance, that is a function of type $f\,(mass,\ distance) = period$.

Clearly we need slightly different expressions as above. Here we have two real numbers as argument instead of one integer. Moreover, there might be other operators involved. Apart from the expressions $x_1 - x_2$, $x_1 + x_2$, $x_1 \times x - s$ and $x\hat{}p$ (power) considered above, we include sin x, cos x and $\sqrt{x}$. This are the usual operations found in any handbook of physics. The corresponding data types are:

```
:: Op x
   = OpPlus x x | OpMinus x x | OpTimes x x | OpPower x PConst
   | OpDivide x x | OpRoot x PConst | Sin x | Cos x
:: Var = Var Int
:: Expr = Expr (OR (OR Var RConst) (Op Expr))

:: RConst = RConst Real
:: PConst = PConst Real
```

Exactly as above we derive generation of instances of these types for all complicated types. The only interesting cases are the generation of variables and constants. In this situation we know that the arity (number of arguments) of the desired function is two. So, only the variables `Var 0` and `Var 1` make sense.

```
ggen{|Var|} n r = map Var [0..arity-1]
arity = 2
ggen{|PConst|} n r = map PConst [2.0..3.0]
ggen{|RConst|} n r = map RConst [1.0, pi, pi4]
```

The environment should here not consider a single integer value as above, but a real value for each argument. This is represented by a list of reals. The only slightly interesting instance is the one that looks up a variable in the environment.

```
instance apply Var [Real] Real where apply (Var i) = λe.e!!i
```

All other instances of `apply` are exactly similar to the once shown above. The only difference is that the resulting value is of type `Real`. Whenever necessary the environment should be given the type `[Real]`.

After these preparations we can immediately state the property for functions implementing Keppler's third law: The function k_3 should be *fit* and $k_3\,(m,\, r) \approx\ p$.

```
pKepler :: Expr → Property
pKepler k3 = fit k3 ⟹ ¬((λ(name,d,m,r,p). apply k3 [m,r] ≈ p) For planetTable)
```

In order to compensate for small errors in real calculations and finite precision the the numbers in the planet table we use $\approx$ instead of $=$. The operator $\approx$ considers two numbers equal if their relative difference is less than some δ, e.g. 1%.

```
(≈) infix 4 :: Real Real → Bool
(≈) x y = x=y || (abs (x-y)/(abs x+abs y)) ≤ delta
```

Within 0.5 second this system generates the first version of Keppler's third law. The first 5 functions generated are:

```
Counterexample 1 found after 4838 tests: k3 x0 x1 = (x1^(1/2))^3
Counterexample 2 found after 6121 tests: k3 x0 x1 = (x1^(1/2))*x1
Counterexample 3 found after 12286 tests: k3 x0 x1 = (x1^3)^(1/2)
Counterexample 4 found after 54331 tests: k3 x0 x1 = (x1*x1)/(x1^(1/2))
Counterexample 5 found after 80598 tests: k3 x0 x1 = (x1^2)/(x1^(1/2))
```

Note that the mass, `x0`, of the planet does not occur in the body of these functions. Apparently it plays no rôle in the law. If we had known this before, we could have searched for a function obeying the predicate $k_3\,(r) \approx p$. We pretended

that we had just like Keppler no idea of the relation to be found. The generated functions are all equivalent to the official versions of Kepplers third law:

$$K_3 : \ p = \sqrt{r}^3$$

Our system found this law within one second. Even if we include the time to construct our function synthesis system this is much faster than the ten years Keppler needed. This is of course by no means a fair comparision. For instance we can lookup the data of planets simply at Wikipedia and have quite powerful computers available. However, this example shows that our approach is capable to solve nontrivial problems.

Until know we have shown that it is possible to generate functions with a body that is some expression over containing variables, and predefined operators. In the next sections we try to find recursive functions and λ–expressions obeying some predicate.

6 Synthesizing Primitive Recursive Functions

The principle introduced above can also be applied to recursive functions. However, the presence of functions imposes one additional concern. Suppose we synthesize a nonterminating function and start evaluating the predicate. This will start a nonterminating computation. We can look for three kind of solutions:

1. At first glance extending the predicate `fit` to allow only terminating functions looks tempting. Unfortunately termination of computations is an undecidable problem. Of course we make a safe approximation and allow only functions that are known to terminate. In case of doubt, the function is considered to be not `fit`.
2. A better solution is to synthesize only functions that are known to terminate always. We will explore this approach in this section for primitive recursive functions.
3. Another approach is to reduce functions only a finite amount of reduction steps, say 1000 step. If the predicate is not reduced to `True` in these steps we reject this candidate function. Of course this includes the risk to reject matching functions, but we avoid nontermination. This approach is explored in the next section.

In order to generate primitive recursive functions that are guaranteed to terminate we extend the syntax for expression from section 3.2 with the following syntax for recursive functions.

$$\begin{aligned}
Fun &= \mathbf{f}(\mathbf{x}) = \{\ Expr \mid RFun\ \} \\
RFun &= \mathbf{if}\ (\mathbf{x} \leq\ Termval\ )\ \mathbf{then}\ Expr\ \mathbf{else}\ PRex \\
PRex &= FunAp \mid Var \mid IConst \mid BinOp\ PRex \\
FunAp &= \mathbf{f}\,(\,\mathbf{x} -\ FConst\,) \\
TermVal &= 0\,..\,2 \\
FConst &= 1\,..\,2
\end{aligned}$$

Again we map this directly to a data type that includes all variable parts of the functions. There is no reason to store constant parts in the abstract syntax tree. For this reason FunAp only contains the constant subtracted from the argument x in the recursive call.

```
:: PRex    = PRex (OR (OR FunAp Var) (OR IConst (BinOp PRex)))
:: FunAp   = FunAp Int
:: TermVal = TermVal Int
:: RFun    = RFun TermVal Expr PRex
:: Fun     = Fun (OR Expr RFun)
```

Like usual we derive the generation of everything in the abstract syntax trees but the constants. The generation of constants has the familiar pattern:

```
gengen{|TermVal|} = map TermVal [0..2]
gengen{|FunAp|}   = map FunAp   [1..2]
```

In order to evaluate the application of such syntax trees we use an environment that contains the recursive function as well as the value of the current argument: `(Int→Int,Int)`.
In the instance for variable we select the appropriate field from this environment:

```
instance apply Var (x,Int) Int where apply X = λ(_,i).i
```

The interesting instances of apply are the recursive function call FunAp, and the initial function definition by RFun. In the instance for RFun we transform the abstract syntax tree to a function of type Int→Int and put it in the initial environment together with the current argument.

```
instance apply FunAp (Int→Int,Int) Int where apply (FunAp d) = λ(f,i).f (i-d)
instance apply RFun Int Int
where apply rf=:(RFun (TermVal c) t e) = f
      where f i = if (i≤c) (apply t i) (apply e (f,i))
```

We can synthesize functions for the input–output patterns from above: $f(0) = 0$, $f(2) = 4$ and $f(3) = 9$ by:

```
p3 :: Fun → Property
p3 d = fit d ⟹ ¬(f 0 = 0 && f 2 = 4 && f 3 = 9) where f = apply d

Start = testnm 1000 5 p1
```

Note that compared to p2 only the type used in the predicate is changed from Expr to Fun, this is all we need to do to change the search space. The results are:

```
Counterexample 1 found after 25 tests: f x = x*x
Counterexample 2 found after 57 tests: f x = x^2
Counterexample 3 found after 6241 tests: f x = ((x+x)-x)^2
Counterexample 4 found after 7500 tests: f x = ((x*x)+x)-x
Counterexample 5 found after 8336 tests: f x = if (x≤1) (x+x) (f (x-2)+(f (x-1)+x))
```

As expected the first results are identical to synthesizing using the type Expr since this is the first option in the type Fun. The fifth function synthesized however is a recursive function.

In exactly the same way we can synthesize familiar primitive recursive functions from a few input–output pairs. For instance:

```
p4 :: Fun → Property
p4 d = fit d ⟹ ¬(f 1 = 1 && f 4 = 24) where f = apply d
```

yields the factorial function.

```
Counterexample 1 found after 2785 tests: f x = if (x≤0) 1 (x*f (x−1))
```

and

```
p5 :: Fun → Property
p5 d = fit d ⟹ ¬(f 5 = 8 && f 7 = 21) where f = apply d
```

yields the famous Fibonacci function

```
Counterexample 1 found after 1167 tests: f x = if (x≤1) 1 (f (x−2)+f (x−1))
```

The Ackermann function cannot be synthesized in this way since it is not primitive recursive.

This approach is not restricted to function having a single integer variable as argument. Above we have shown how multiple arguments can be handled. In exactly the same way as functions over integers we have also synthesized recursive functions over lists and other recursive data types.

7 Synthesizing Lambda Expressions

In this section we show how λ-expressions with specific properties can be synthesized. This imposes two problems. First it is not possible to use a data type that represents only terminating functions. If we need to generate λ-expressions, the abstract syntax tree to be used reflects the structure of those λ-expressions. The second problem is a consequence of using those λ-expressions: it is hard to determine an interesting class of λ-expressions that is known to terminate.

We solve these problems one by one. First we define a data type to represent λ-expressions. Apart from the well-known variables (`Var V`), abstractions (`Abs V LExpr`), and applications (`Ap LExpr LExpr`), we have integer constants (`Const C`), and binary operator constants (`OpConst String`). These binary operations are integer manipulations like `"+"` and `"-"`.

```
:: LExpr = Var V | Abs V LExpr | Ap LExpr LExpr | Const C | OpConst String
:: V = V Int     // variable
:: C = C Int     // constant
```

We do not include a build-in conditional for our λ-expressions. The Booleans and the conditional are represented by the expressions $\lambda\, v_0 \,.\, \lambda\, v_1 \,.\, v_0$ for True, $\lambda\, v_0 \,.\, \lambda\, v_1 \,.\, v_1$ for False, and the identity function $\lambda\, v \,.\, v$ for if. Represented as syntax trees this is:

```
TRUE  = Abs v0 (Abs v1 (Var v0))
FALSE = Abs v0 (Abs v1 (Var v1))
If    = Abs v0 (Var v0)
```

It is completely standard to write a reducer for λ-expressions of the form `LExpr`. We omit the details here for brevity and assume that we have a reducer to head normal form according to the lazy (left-most, outer-most) strategy.

```
hnf :: LExpr → LExpr
```

The only thing special about this reduce is that it does at most N reduction steps in order to ensure termination. In our test we used 1000 as upper limit for the number of reduction steps.

As a first approach we generate instance of LExpr in the now familiar way: we manually generate some appropriate constants and variables and derive generation for the other types. We reformulate property p1 from above to find λ-expression matching $f(0) = 0$, $f(2) = 4$ and $f(3) = 9$ as:

```
pL1 :: LExpr → Bool
pL1 f = ¬(p 0 0 && p 2 4 && p 3 9)
where p x y = hnf (Ap f (Const (C x))) == (Const (C y))
```

Unfortunately our synthesis technique does not find an answer in reasonable time. Our approach fails since the search space is too large. Most of the generated expressions are ill-formed, like $\lambda a\,.\,b$, $\lambda a\,.\,+$, and so on.

There are several solutions for this problem. For instance we can define the generation of instances of LExpr manually as we did in [12]. This works well, but this approach is not very elegant. Another solution is to keep track of the type of the generated expressions during generating and make sure to yield only well-typed expressions. Katayama [8] uses this approach quite successfully. We find it less appealing since it further complicates the generation algorithm.

In this paper we propose a new method to control the generation of λ-expressions: we introduce an additional data type that corresponds to a high level language that describes the functions we want to consider. We synthesize instances of these high level data type in the usual way; define the instances of constants manually and derive the generation of the rest. Next we convert the instances of these high level syntax trees to λ-expressions. For this purpose we can introduce the class conv.

```
class conv a :: a → LExpr
```

However, it is more interresting and convenient to define a generic conversion.

```
generic gconv a :: a → LExpr
```

The instances for EITHER, PAIR and CONS do nothing else than applying the given conversion function to the arguments.

```
gconv{|EITHER|} gf gg (LEFT x)    = gf x
gconv{|EITHER|} gf gg (RIGHT y)   = gg y
gconv{|PAIR|}   gf gg (PAIR x y) = Ap (gf x) (gg y)
gconv{|CONS|}   g      (CONS x)   = g x
```

For all leaves of the tree we have to think what should be done. For this reason we do not provide an instance of UNIT.

The conversion of high-level functions to λ-expressions can be found in any textbook on semantics or implementation of functional languages.

We use a data type very similar to the one for primitive recursive functions shown in the previous section.

```
:: Op     = Op String
:: Oper x = Oper Op x x
:: X      = X
:: RecAp  = RecAp Int
:: Ex     = Ex (OR (OR X C) (Oper Ex))
```

```
:: Pr     = Pr (OR (OR X C) (OR (Oper Pr) RecAp))
:: RFun   = RFun C Ex Pr
:: Fun    = Fun Ex
```

As usual we define the generation of constants manually and derive generation for all other data types.

```
gengen{|Op|}    = map Op ["+","*"]
gengen{|RecAp|} = map RecAp [1..2]
derive gengen Oper, X, Ex, Pr, RFun, Fun
```

Similar to the previous section we can define an instance of `fit` for those types. This contains no surprises at all.

The conversion of these types to the corresponding types for λ-expressions is very simple for most types. Some typical examples are:

```
gconv{|C|} c = Const c
gconv{|V|} v = Var v
```

The conversion of a given body to a function is rather simple. We only have to add an abstraction to the converted body.

```
gconv{|Fun|} (Fun b) = Abs v0 (gconv{|*|} b)
```

Only the conversion of recursive functions deserves some attention. First we need to decide how we handle the recursion. Usually this is done by a Y-combinator defined as $\mathsf{Y}\ f = f\ (\mathsf{Y}\ f)$. Here we unfold the Y-combinator at conversion time. This implies that every recursive call gets its own function as λ-term as its first argument. By convention the function argument is represented by v_0 and the recursive function by v_1. This implies that the recursive call $f\,(x - c)$ is represented by the term $v_1\ v_1\ (-\ v_0\ c)$. In terms of our data types this is:

```
gconv{|RecAp|} (RecAp c)
 = Ap (Ap (Var v1) (Var v1)) (Ap (Ap (OpConst "-") (Var v0)) (Const (C c)))
```

A recursive function definition rearranged the argument and the function such that it can be recursively applied. That is the function is represented by the λ-expression $(\lambda v_4\,.\,\lambda v_3\,.\,v_4\ v_4\ v_3)\ f$ where f is the λ-expression corresponding to the primitive recursive function. This function f as a λ-expression gets itself and the argument `x` as arguments $(\lambda v_1\,.\,\lambda v_0\,...)$. In a conditional expression (see `If` defined as the identity function above) it checks wether `x` is less or equal to the given constant: $\leq\ v_0\ c$. Depending on this condition it either executes the converted then-part `t`, or else-part `e`.

```
gconv{|RFun|} (RFun c t e)
 = Ap (Abs v4 (Abs v3 (Ap (Ap (Var v4) (Var v4)) (Var v3)))) f
where f = Abs v1 (Abs v0 (Ap (Ap (Ap If (Ap (Ap (OpConst "≤") (Var v0)) (Const c)))
                                        (conv t)) (conv e)))
```

The conversion of `Ex`, `Pr`, and `OR` can be derived.

```
derive gconv Ex, Pr, OR
```

After all these preparations it is easy to generate high quality λ-expressions. We simply convert the fit instance of functions `Fun` and recursive functions `RFun`.

```
ggen{|LExpr|} n r = map gconv{|*|} (filter fit (es n r))

es :: [OR Fun RFun]
es = gengen{|*|}
```

With these generator for instance of LExpr the test system finds solutions for predicate pL1 quickly:

```
Counterexample 1 found after 4 tests: (λa.(* a) a)
Counterexample 2 found after 748 tests: (λa.(+ ((* −1) a)) ((* a) ((+ a) 1)))
Counterexample 3 found after 863 tests: (λa.(+ ((* a) −1)) ((* a) ((+ a) 1)))
Counterexample 4 found after 1294 tests: (λa.(+ ((+ 1) ((* a) a))) −1)
Counterexample 5 found after 1484 tests: (λe.e e)(λb.λa.(((λa.a)((≤ a) −1)) 1)((* a)a))
```

In a similar way we can state some input–output pairs of the Fibonacci function:

```
pL2 :: LExpr → Bool
pL2 f = ¬(p 3 3 && p 4 5 && p 5 8 && p 7 21)
where
    p x y = hnf (Ap f (Const (C x))) == (Const (C y))
```

The first Fibonacci function in λ-calculus found is:

```
(λe.e e) (λb.λa.(((λa.a) ((≤ a) 1)) 1) ((+ ((b b) ((− a) 2))) ((b b) ((− a) 1))))
```

This λ-term corresponds exactly to the most common double recursive definition.

Of course it is also possible to select the desired (primitive recursive) functions in a way similar to the previous section and transform the matching functions to λ-terms. We prefer the route outline here. The synthesized λ expressions are really used to determine if they obey the given predicate. This gives much more confidence that they really are the expressions we are looking for. In the alternative approach mistakes in compiling high-level functions to λ-terms will pass unnoticed.

8 Related and Future Work

Many attempts have been described to construct inductive programming systems. The synthesize candidates and test approach here is just one of the possibilities. See for instance [14] for an overview.

Closely related to our work is the approach of Katayama [8]. He generates λ-expressions using type information and a set of user-defined functions in the functional programming language Haskell. Recursion for a data type that is used as argument of the generated functions has to be defined as one of the primitives in Haskell. The actual generation of λ-terms is a black box. Since our approach is based on a general test system supporting first–order logic, our system is able to handle a wider range of predicates. We can control the generation of candidate functions very easily by changing the appropriate data types. this make our approach more flexible. In [9] Katayama proposes to use a test based approach to determine the equivalence of functions as alternative to our function fit. It is very easy to add this to our system, but unnecessary. The given definitions of fit removes equivalent candidate functions effectively. Since fit only has to look at the current candidate it is more efficient if the system has to generate large number of candidates. The amount of work needed to compare a new candidates with the candidates seen before will increase if the number of candidates seen increases. An even better approach is to use a more sophisticated grammar and associated data-types that exclude many of the redundant function candidates. Using such a grammar we can for instance ensure

that constants for operations like addition and multiplication only occur in one of the branches, and try to avoid subexpressions that contain only arithmetic operations applied to constants.

Our approach works for any kind of functions. The user has to supply only a data type representing the abstract syntax trees, a function apply that assigns a meaning to those syntax trees, and the generation of the trees. We have show that generic programming can really help to reduce the amount of functions that has to be defined manually.

In the future we want to develop a generic version of apply. This function has now a lot of dull instances that should be derived from a generic definition.

The current examples do not need much constants. In many kind of functions there are a lot of constants involved. Determining these constants by a generate and test approach will not be very effective. We want to investigate if it is possible to determine the shape of the functions by the techniques outlined in this paper and select the appropriate constants by a conventional technique like hill climbing.

The real challenge is of course to generate more complex functions using inductive programming. Despite all our efforts the search space for complicated functions still grows rapidly. Hence it will take much time to find such a function by a generate and test approach. There are two directions of optimizations possible. First we can generate more appropriate candidate expressions by adding knowledge to our system. If we somehow know what suitable building blocks of good candidates are, we add these primitives as additional items to our data types representing the candidate functions. Second we can try to split the problem area in smaller pieces and find solutions to these pieces separately, see [7]. In a next phase try to find solutions for the full problem by combining these partial solutions.

9 Conclusions

In this paper we have shown a very general and flexible approach to do inductive programming by a generate and test approach. The user defines the syntax of the functions to be generated by a set of algebraic data types. Using generic programming support the user defines the semantic of these syntax trees in function apply. The generation of instances of the algebraic data type representing the syntax of the candidate functions is done by a generic algorithm. Only the generation of constants deserves manual definitions. Using the model-based test tool G∀st one can specify high level predicates about determining the functions wanted.

In this paper we have shown that this system works for nonrecursive function, primitive recursive functions and λ-expressions. We are convinced that there are many more application areas.

Acknowledgement

We thank the anonymous reviewers for their useful feedback.

References

1. http://en.wikipedia.org/wiki/planet
2. Alimarine, A., Smetsers, S.: Efficient generic functional programming. Technical report niii-r0425, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands (2004)
3. Banerjee, D.: A methodology for synthesis of recursive functional programs. ACM Transactions on Programming Languages and Systems (TOPLAS) 9(3), 441–462 (1987)
4. Bird, R.: Introduction to functional programming using Haskell, 2nd edn. Prentice Hall, Englewood Cliffs (1998)
5. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming, Montreal, Canada, pp. 268–279. ACM Press, New York (2000)
6. Cypher, A.: Watch what I do: programming by demonstration. MIT Press, Cambridge (1993)
7. Henderson, R.: Incremental learning in inductive programming. In: Schmid, U., Kitzelmann, E., Plasmeijer, R. (eds.) AAIP 2009. LNCS, vol. 5812, pp. 74–92. Springer, Heidelberg (2010)
8. Katayama, S.: Systematic search for lambda expressions. In: Proceedings of the 6th Symposium on Trends in Functional Programming (TFP 2005), pp. 195–205 (2005)
9. Katayama, S.: Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In: Ho, T.-B., Zhou, Z.-H. (eds.) PRICAI 2008. LNCS (LNAI), vol. 5351, pp. 199–210. Springer, Heidelberg (2008)
10. Koopman, P., Plasmeijer, R.: Fully automatic testing with functions as specifications. In: Horváth, Z. (ed.) CEFP 2005. LNCS, vol. 4164, pp. 35–61. Springer, Heidelberg (2006)
11. Koopman, P., Plasmeijer, R.: Systematic synthesis of functions. In: Nilsson, H. (ed.) Selected Papers of the 7th Symposium on Trends in Functional Programming, TFP 2006, Nottingham, UK, April 19-21, pp. 68–83 (2006), Intellect Books, ISBN 978-1-84150-188-8
12. Koopman, P., Plasmeijer, R.: Systematic synthesis of λ-terms. In: Barendsen, E., Capretta, V., Geuvers, H., Niqui, M. (eds.) Reflections on Type Theory, λ-Calculus, and the Mind - Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday, December 17, pp. 211–222 (2007) ISBN 978-90-9022446-6
13. Plasmeijer, R., van Eekelen, M.: Concurrent Clean language report (version 2.0) (December 2001), http://www.cs.ru.nl/~clean/
14. Schmid, U. (ed.): Inductive Synthesis of Functional Programs. LNCS (LNAI), vol. 2654. Springer, Heidelberg (2003)