SYSTEMATIC SYNTHESIS OF λ-TERMS

PIETER KOOPMAN AND RINUS PLASMEIJER

Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands
e-mail address: pieter,rinus@cs.ru.nl

Abstract. In this paper we show how to generate terms in the λ-calculus that match a given number of function argument result pairs. It appears that the number of λ-terms is too large to find terms reasonably fast based on the grammar of λ-calculus alone. By adding knowledge such as the desired number of arguments it is possible to synthesize λ-terms effectively for some interesting examples. This yields surprising terms that are unlikely to be found by a human.

An interesting subproblem is the determination of suitability of candidate terms based on equivalence of terms. We used an approximation of equivalence by a finite number of reduction steps. This implies that the test for equivalence can also yield the value undefined. Fortunately the test system used is able to handle undefined test results.

For Henk Barendregt on his sixtieth birthday

1. Introduction

In computer science one often looks for reducts of λ-expressions (the λ-expression is seen as a functional program representing the desired value), or general properties of λ-calculus (like the famous Church-Rosser property). The construction of λ-terms possessing some desirable property is commonly done manually. In this paper we describe a technique to synthesize such λ-terms automatically. Typical examples are: find a term Y such that $\forall f.\, Y\, f = f\,(Y\, f)$, or find a term s such that $\forall n \geq 0.\, s\ n = \sum_{i=0}^{n} i$. This technique can be used to find rather complicated terms, or terms that are not very intuitive. Although the (mathematical) examples shown in this paper are merely a proof of concept. There exists also serious applications of this kind of synthesis techniques, like the generalization of behavior in adaptive systems. In this paper we concentrate on untyped λ-terms as described by Barendregt in [1] extended with numbers and some operations on numbers. See [3] for an introduction to typed λ-calculus.

The approach to generate λ-terms obeying some property is inspired by our previous work [9]. In that paper we described how one can synthesize functions matching a property such as $f\ 4 = 5 \wedge f\ 5 = 8$. In general there are umpteen functions matching a given set of input-output pairs. In program synthesis it is our goal to find small functions that

2000 ACM Subject Classification: D.1.2, I.2.2.

Key words and phrases: Program Synthesis, Automatic Programming, λ-Terms, programming by example.

generalize the given behavior. We prefer a nonrecursive function over a recursive function, and a recursive function is highly preferred over a sequence of conditionals that exactly coincides with the given argument result pairs (like $f\ x =$ if $(x == 4)\ 5$ (if $(x == 5)\ 8\ 0)$). The approach in [9] is to define a data type that represents the grammar of the candidate functions, for our example primitive recursive functions of type `Int→Int` will do. Using the generic generation of instances of this type, abstract syntax trees of candidate functions are generated [8]. Such a syntax tree is turned into the equivalent function. The suitability of these functions is determined by the automatic test system G∀st [7]. The test system is used to find functions f matching the desired property, by stating that such a function does not exists, e.g. $\forall f.\neg(f\ 4 = 5 \wedge f\ 5 = 8)$. The counterexamples found by G∀st are exactly the functions wanted. For $f\ 4 = 5 \wedge f\ 5 = 8$ the test system finds the function `f x = if (x≤1) 1 (f(x–2)+f(x–1))`, the well known Fibonacci function.

In this paper we do not define a specific grammar for candidate functions, since we want to find ordinary λ-terms matching the given property. For the synthesis of λ-terms we start with the same approach. First we define a data type representing λ-terms and synthesize instances of this type from small to large as candidates. Then we check with the test system if these instances of the data type represent a function (λ-term) obeying the given constraints. However, there are some significant differences compared to the generation of the functions mentioned above. These difference corresponds to problems that needs to be tackled in order to make the systematic synthesis of λ-terms work. First, the termination of computations needed to determine the suitability of candidate λ-terms is an issue. In the generation of ordinary functions, we constructed the functions such that termination of reduction is guaranteed. In our new approach it is not possible to guarantee termination of reduction without too serious restrictions on the terms considered. Second, in order to obtain interesting λ-terms (corresponding to recursive functions) it is essential to have higher order functions. The use of higher order and potentially nonterminating expressions makes the equivalence of λ-terms an issue. Theoretically it is known that the equivalence of λ-terms is in general undecidable. Third, the λ-terms corresponding to recursive functions like the Fibonacci function mentioned above are relative large as well. This is caused by the fine grain computations in λ-calculus. Experience shows that the number of candidate terms to be considered becomes impractically large. Hence we need some guidance in the generation of candidate terms. A fourth difference with previous work is that we will use also logical properties like $\forall x,\ y\,.\,k\ x\ y = x$ instead of only input-output pairs like $k\ 1\ 2 = 1$.

Problem one and two are covered by using normal order (left most) single step reduction in the comparison of equations. If equivalence is not found in a given number of steps, the equivalence is decided to be *undefined*. The test system G∀st is well equipped to handle these undefined test results. The third problem is handled by a simple yet effective and flexible generation algorithm for candidate terms. Although the data type used allows more terms, the synthesization algorithm used generates only instances of a grammar that allows a smaller number of terms. Since we still handle terms prescribing potentially infinite computations, transforming the terms to ordinary expressions and evaluating these is unsafe. For the reduction algorithm however, it is crucial that all terms used are described by a small and simple data type.

In the remainder of this paper we first give a quick overview of testing logical properties using the test tool G∀st. In section 3 we introduce the data structure that will be used to represent λ-terms. The equivalence of λ-terms is treated in section 4. Next we use the generic algorithm from G∀st to generate λ-terms in section 5 and look at the effectiveness of

this approach with some examples. In section 6 we introduce more effective algorithms to generate λ-terms matching a given condition. Finally, there is a conclusion and a discussion of related work in section 7.

2. Testing logical properties with the test tool G∀st

The test system G∀st checks properties in first order logic by evaluating the property for a (large) number of arguments. G∀st is implemented as a library for the functional programming language Clean [10]. This library provides operators corresponding to the logical operators from logic as well as a class of functions called `test` to check properties.

The test system treats every ordinary function argument as a universally quantified variable if the function is used as a logicalproperty. Consider the function `pAbs n = abs n≥0`, where `abs` is a function from the standard library that computes the absolute value of integer arguments. This function is interpreted by G∀st as the property $\forall n \in \mathsf{Int}\,.\,\mathsf{pAbs}\ n$. By unfolding the function definition this is $\forall n \in \mathsf{Int}\,.\,\mathsf{abs}\ n \geq 0$. This property can be tested by executing `Start = test pAbs`. By the type of the function `pAbs` to be tested `Int → Bool`, the test system determines that the property should be evaluated for a large number of integers. These test values are synthesized systematically (from small to large for recursive types) by the generic function `ggen`. If G∀st finds a counterexample within the first `maxTest` tests, the test result is `CE`. Apart from this test result, the test system gives also information about the counterexample found (like the number of tests done and the values used to find this counterexample). `maxTest` is the default number of tests, by default 1000. It is easy to change this number in general or in a specific test. Otherwise, the result is `OK` if G∀st detects that the number of test values is less than `maxTest` and the property holds for all these test values. If no counterexample is found in the first `maxTest` tests and there are more than `maxTest` values in the list generated by `ggen`, the test result is `Pass`. The test result `Pass` is weaker then `OK`: doing additional test might show a counter example if the result is `Pass`, while `OK` indicates proof by exhaustive testing.

In the example `pAbs n = abs n≥0` the test system finds a counterexample corresponding to the minimum integer value in the domain for `test pAbs`. The instance of the generic test suite generator, `ggen`, for integers generates test values like `0`, `1`, `–1`, maxint, and minint that are known to cause often issues somewhere in the beginning of each test suite for integers.

With the operator ∃ we can test existentially qualified expressions like $\exists x.f\ x = x$. The operator ∃ takes a function as argument. The type of the argument determines the type of test arguments generated by `ggen` in G∀st. In the example `pFix` above, we used a nameless function (λ-expression) as argument for the operator ∃. Compared with the ordinary logical notation, we have to write only an additional λ between ∃ and the variable. Of course one can use any function as argument for the operator ∃. Clean's type inference system detects in this example that `x` must be an integer. Hence, G∀st will generate integer values.

The test system G∀st is used to handle undefined values. For any function `f :: Int → Int` we can test if the function has a fixed point ($\exists x.f\ x = x$) by defining the property:

```
pFix :: (Int → Int) → Property
pFix f = ∃ λx.f x==x    // the type of x is determined by the context, here Int
```

We can test if the function `g x = x+1` has a fixed point by executing `Start = test (pFix g)`. The test system generates a fixed number of integer values (by default 500) and checks if one of these values makes `g x==x` true. If such a value does not occur the test system can neither decide that the property holds, nor that there is a counterexample. The test system

uses the value `Undef` to indicate that a positive test result has not been encountered within the tests to be done, but such a value might exist. Hence, the possible test results are:

```
:: Result = Pass | OK | CE | Undef
```

The difference between this test system and a model checker is that the test system evaluates properties using the ordinary evaluation mechanism. A model checker uses an abstraction of the system (the model) as basis for reasoning rather than actual code. A model checker uses also abstract evaluating steps to check the validity of the model (e.g. $\forall x \,.\, \neg \neg\ x = x$). This implies that a model checker is able to prove properties that can only be tested partially by a test system. Advantages of a test system are that no separate model is needed and that the actual code is used rather than a model of this code.

3. A data type to represent λ-terms

The first step is to construct a data type to represent λ-terms. Apart from variables, abstraction, and application we introduce numerical constants and constructors `Plus` and `If` for a primitive addition and conditional in the terms treated. We use the functional programming language Clean for the algorithms in this paper.

```
:: Expr = Var V | Abs V Expr | Ap Expr Expr | Const C | Plus | If   // λ-expression
:: V = V Int                                                       // variable
:: C = C Int                                                       // constant
```

The additional types `V` and `C` are superfluous for the syntax trees describing λ-terms, but are convenient to control the generation of variables and constants.

Although that it is known that the numerical constants and the constants `Plus` and `If` are theoretical superfluous [1], it is convenient to introduce them. The use of these constants makes computations much more efficient. Moreover, this representation is much more compact than the representation of constants by Church numbers.

By using tailor made instances of the generic `show` function, instances of these types can be printed as usual in λ-calculus. The term `Abs (V 1) (Ap (Var (V 0)) (Ap (Var (V 1)) (Var (V 1))))` will be printed as `λb.a (b b)`. This is more compact and better readable.

4. Equivalence of λ-terms and reduction

A key step in the search for λ-terms is determination of the equivalence of terms. Looking for some term I such that $I\,x = x$ the system needs to be able to determine that $(\lambda a.a)\,x$ and x are equivalent. If we write $N = M$ we mean equality modulo reduction: $N =_\beta M$. The terms M and N are β-convertible if they are equal, if one reduces to the other (i.e. $M \rightarrow_\beta N$ or $N \rightarrow_\beta M$), or there is a common reduct L of M and N (i.e. $\exists L \in \Lambda . M \rightarrow_\beta L \wedge N \rightarrow_\beta L$). In general checking whether $N =_\beta M$ is undecidable [1].

The undecidability of convertibility does not imply that it is impossible to look for equivalent terms. It just says that there are terms where the convertibility is unknown. For many terms we can determine whether they are convertible by reducing them a finite number of steps. We will use the normal order (left most) reduction strategy for these reductions since it is known to find a normal form if it exists [1]. If we find a common reduct (modulo α-conversion) within this finite number of reduction steps the terms are clearly convertible. If we obtain unequal normal forms the terms are obviously not convertible. If one of the terms shows a cyclic reduction (like $w\ w$ where $w = \lambda x.x\ x$ that has the property that

$w\ w \rightarrow_\beta w\ w$) and the other is not a redex the terms are also unconvertible. In all other situation the convertibility is considered to be undefined.

To reduce the space consumption in this paper we will only list the rules for the traditional λ-calculus here and ignore the constants. Adding constants is straightforward: two constants are convertible if they are syntactically equal, otherwise they are unconvertible. A constant is unconvertible to any other term in normal form. The convertibility of a constant to a redex is undefined. The system will do a finite number of reduction steps to determine if it is possible to determine convertibility. We use the constructor `OK` from the type `Result` to represent convertibility and `CE` to indicate inconvertibility.

A single reduction step on an expression is done by the function `hnf1 :: Expr → (Bool, Expr)`. The Boolean in the resulting tuple indicates whether a reduction step is done.

```
hnf1 :: Expr → (Bool, Expr)
hnf1 (Ap (Abs v e) a) = (True, sub e v a)
hnf1 (Ap n a)
    # (r, m) = hnf1 n      // the symbol # introduces a let definition in Clean
    = (r, Ap m a)
hnf1 e = (False, e)        // reduction to weak head normal form: no reduction under an abstraction.
```

The notion of substitution, $e[v := a]$, in λ-calculus is implemented as `sub e v a`. The function `sub :: Expr V Expr → Expr` replaces each free occurrence of the second argument in the first argument by the third argument.

```
sub m=:(Var v) x n = if (v==x) n m
sub m=:(Abs y e) x n
    | x==y
        = m                          // stop substitution, every occurrence of x in m bound by this λ x.
    | isMember y (freeVars n)    // x ≠ y: makes this λ y an undesirable binding in m?
        = Abs v (sub (sub e y (Var v)) x n)      // yes, rename y to v in m before actual substitution
        = Abs y (sub e x n)                      // no, continue with substitution in e
where v = newVar startVal [x:freeVars m++freeVars n]   // a fresh variable
sub (Ap f a) x n = Ap (sub f x n) (sub a x n)
sub m x n = m
```

The function `freeVars` yields a list containing the free variables in the given expression. The expression `newVar n l` yields the first variable starting at `V n` that does not occur in the list of variables `l`. This is used to prevent undesirable binding of variables in examples such as $\lambda a.a\ b[b := a\ c]$. By the renaming of variables this is transformed to $\lambda d.d\ b[b := a\ c]$.

The complete function `hnf1` also contains alternatives for the constants `Plus` and `If`. When both arguments of the `Plus` are constants the expression is replaced by a new constant, otherwise `hnf1` tries to evaluate arguments of the addition. For the conditional expression numbers are interpreted as Booleans. Positive numbers are interpreted as the Boolean value True, all other values as False. If the subterm `c` in a term `Ap (Ap (AP If c) t) e` is not a constant, the functions `hnf1` tries to reduce it.

As a first step to determine convertibility we define α-equality. Two expressions are α-equal (result `OK`) if they can be made identical by α-conversion of variables introduced by abstractions within the expressions. If the expressions are not α-equal and are not in normal form the result is undefined (`Undef`). Otherwise the expressions are clearly not α-equal, and the result is `CE`.

```
alphaEQ :: Expr Expr → Result
alphaEQ (Var x)    (Var y)   = if (x==y) OK CE
```

```
alphaEQ (Var x)    (Abs v e) = CE
alphaEQ (Abs v e)  (Var y)   = CE
alphaEQ (Abs x e1) (Abs y e2)
    | x == y
        = alphaEQ e1 e2
    | not (isMember x (freeVars e2))
        = alphaEQ e1 (sub e2 y (Var x))                  // α-conversion: e2[y:=x]
    | not (isMember y (freeVars e1))
        = alphaEQ (sub e1 x (Var y)) e2                  // α-conversion: e1[x:=y]
        = alphaEQ (sub e1 x (Var v)) (sub e2 y (Var v)) // α-conversion: e1[x:=v] and e2[y:=v]
where v = newVar startVal (freeVars e1++freeVars e2)     // a fresh variable
alphaEQ e1=:(Ap f x) e2=:(Ap g y)
 = case alphaEQ f g of
    OK = case alphaEQ x y of
            CE = if (isRedex e1 || isRedex e2) Undef CE
            r  = r
    CE = if (isRedex e1 || isRedex e2) Undef CE
    Undef = Undef
alphaEQ e1 e2 = Undef
```

The equivalence (convertabity) of expressions is determined by the infix operator $\equiv$.

```
(≡) infix 4 :: Expr Expr → Result
(≡) x y = redEQ maxReductions [x] [y]
```

The constant `maxReductions` determines the maximum number of reductions done on the expressions. This is a trade-off between speed and the ability to determine the convertibility of expressions. In the tests reported in this paper the value 500 was used to full satisfaction. If we use examples where more reductions are needed to determine equality, the constant `maxReductions` should be increased. For most examples a value of 50 is more than enough. The real work to determine convertibility is done by `redEQ`.

The function `redEQ` gets the number of reduction steps to be done and two stacks of expressions as arguments. If the number of steps to be done is zero, the result of determining equality is `Undef` unless one of the terms shows cyclic reduction and the other is in head normal form (in that case the result is `CE`). If the number of steps to be done is not zero, the function `redEQ` determines α-equality of the top of one stack (the most recent expression) and one of the elements in the other stack is unequal to `Undef`. Any results unequal to `Undef` determines the result of `redEQ`. If all comparisons for α-equality yield `Undef`, we try to reduce the most recent expressions one single step. If such a reduction is possible for at least one of the terms, the function `redEQ` continues recursively with the new expressions. Otherwise we decompose an application or abstraction if it occurs in both expressions to be compared, and continue with the fragments of the expressions to be compared. In all other situations the given expressions are unequal under reduction: the result is `CE`.

```
redEQ :: Int [Expr] [Expr] → Result
redEQ n lx=:[x:xs] ly=:[y:ys]
    | n==0
        | (isMem x xs == OK && ¬(isRedex y)) || (isMem y ys == OK && ¬(isRedex x))
            = CE
            = Undef
    # rx = isMem x ly                                // n ≠ 0; check if x =α y or one of its ancestors
    | rx ≠ Undef
```

```
        = rx
    ♯ ry = isMem y lx                          // check if y =α x or one of its ancestors
    | ry ≠ Undef
        = ry
    ♯ (bx,x1) = hnf1 x                         // single step reduction of x
      (by,y1) = hnf1 y                         // single step reduction of y
    | bx && by = redEQ (n–1) [x1:lx] [y1:ly]   // x and y are reduced
    | bx = redEQ (n–1) [x1:lx] ly              // only x could be reduced
    | by = redEQ (n–1) lx [y1:ly]              // only y could be reduced
redEQ n lx=:[Ap f a:xs] ly=:[Ap g b:ys]        // x and y are applications in hnf
    | redEQ (n/2) [f] [g] == OK                // compare left arguments of applications
        | isMem a lx == OK || isMem b ly == OK// on a cycle?
            = Undef
            = redEQ (n/2) [a] [b]              // compare right arguments of applications
redEQ n lx=:[Abs a c:xs] ly=:[Abs b d:ys]      // x and y are abstractions
    | isMem c xs == OK || isMem d ys == OK     // on a cycle?
        = Undef
        | a==b
            = redEQ (n–1) [c] [d]              // compare bodies of the abstractions
        ♯ v = newVar startVal (freeVars c ++ freeVars d)
        = redEQ (n–1) [sub c a (Var v)] [sub d b (Var v)]
redEQ n lx ly
    = CE
```

The function `isMem` looks for a result unequal to `Undef` in the list of results. If such a value exists, the result of the application is the value of the first list element unequal to `Undef`, otherwise the result is `Undef`.

```
isMem x [] = Undef
isMem x [a:r]
    = case alphaEQ x a of
        Undef = isMem x r
        result = result
```

This is sufficient to compare λ-terms. In a number of examples the result of comparison might be undefined, but for each property the test system will generate a lot of test arguments. Usually some of these arguments will show whether the property holds or not. Some improvements of the algorithm to compare expressions are possible. For instance, expressions with a different type, such as `λa.a` and `0`, will always be unequal.

5. Generic generation of λ-terms

In order to find λ-terms matching some property, the test system needs to generate candidate expressions. Since G∀st contains a generic algorithm to generate the members of a type, we can completely derive the generation of candidates. In order to limit the search space (and hence speed up the finding of matching λ-terms) we limit the number of variables to 3, and use only the often occurring constants −1, 1 and 2:

```
ggen {|V|}  n r = map V [0..2]
ggen {|C|}  n r = map C [−1,1,2]
```

The generation of expressions is done by the generic algorithm. We derive an instance of this algorithm for this type by:

```
derive ggen Expr
```

If we would have used the generic algorithm to derive the generation of values for the types `V` and `C`, values like `V minint` and `C maxint` would have been generated. We considered this undesirable in this situation and hence we used a tailor made instance of `ggen` for these types. This is all we need to get started.

5.1. **Some examples.** Let's start with a very simple example: a λ-term i with the property $\forall\, x.i\, x =_\beta x$. We state that such a term does not exist: $\forall\, i \in \Lambda\,.\neg\, \forall x \in \Lambda\,(i\, x =_\beta x)$. The test system will try to find counterexamples of this properties. The counterexample found are exactly the λ-terms obeying $i\, x =_\beta x$. The property expressed in G∀st reads.

```
pI :: Expr → Property
pI i = ¬(∀ λx.Ap i x ≡ x)
```

G∀st uses the generic algorithm to generate candidate expressions for i and d. The first ten identity functions found by testing this property are: `λa.a`, `λb.b`, `λc.c`, `(λa.a) (λa.a)`, `λa.((λb.a) Plus)`, `λb.((λa.a) b)`, `λb.((λa.b) Plus)`, `λa.((λb.a) (λa.a))`, and `λa. ((λc.a) Plus)`. For these ten matching λ-terms the system had to generate only 464 candidate expressions. Note that we use here a property with a universal quantifier rather than some input-output pairs (like `i 1 ≡ 1`, `i (λa.a) ≡ λa.a` and `i ((λa.a a) (λa.a a)) ≡ (λa.a a) (λa.a a)`). After the preparations described above, G∀st is very well capable of testing this kind of properties. In our opinion the property shown above is clearer and more elegant than the explicit input-output pairs of the function `i`. Of course it is still possible to search for functions using input-output pairs.

In the same spirit we can look for terms representing the K-combinator by:

```
pK :: Expr → Property
pK k = ¬(∀ λx y.Ap (Ap k x) y ≡ x)
```

As expected the system produces terms like `λa.λb.a` and `λb.λa.b` within the same number of candidates. The system finds also some less obvious terms like `λa.λb.(λa.a) a` and `If −1`.

For functions that only have to work on arguments of a specific type, e.g. numeric constants of the form `Const (C i)`, the ∀ operator will generate undesired arguments if the type of arguments is `Expr`. It is not relevant to know what a plus operator does on free variables or arguments like `λa.a`, hence we should exclude them from the property and the tests. This problem can easily be tackled by using a quantification over type `Int` and the needed type conversion in the property.

```
pPlus :: Expr → Property
pPlus p = ¬(∀ λa b.Ap (Ap p (Const (C a))) (Const (C b)) ≡ Const (C (a+b)))
```

This will produce correct λ-terms for `p` like `Plus`, `(λa.a) Plus`, `λa.Plus a`, `λa.Plus b`, `(λa.Plus) a`, and `(λa.Plus) (−1 −1)`. If we do not want to use all integers in the property or have only specific input-output combination available, the property will not contain a ∀-operator. We use the given input-output pairs in the property. For example:

```
pF1 :: Expr → Result
pF1 p = ¬(f 1 1 1 &&& f 2 2 4 &&& f 3 3 6)
where f a b c = Ap (Ap p (Const (C a))) (Const (C b)) ≡ Const (C c)
```

The operator `&&&` is the logical and for values of type `Result`. Matching λ-terms found are `λa.λb.(Plus b) b`, `λa.λb.(Plus a) b`, `λa.λb.(Plus b) a`, and `λa.λb.(Plus ((Plus b) b)) 0`.

Although these examples work fine, they show also that expressions are generated that are usually considered undesirable like `λa.Plus b` (where `b` is a free variable) and `(λa.Plus) (−1 −1)` (with the constant `−1` at a function position). In these examples these terms are only curious, but they do occupy space in the search space and hence time during the search for the desired λ-terms. If we search λ-terms implementing the Y-combinators by testing:

```
pY :: Expr → Property
pY y = ¬(∀ λf.Ap y f ≡ Ap f (Ap y f))
```

no success is found in the first 1,000,000 tests. The search space is simply too large to find a suitable term in a reasonable time.

6. Smarter generation of λ-terms

There are umpteen way to reduce the search space. An unattractive alternative is to reject candidates that represent wrongly typed λ-terms in a property, as these terms will still be generated and hence consume resources. It will be better to prevent the generation of terms that are clearly unsuitable. Take care not to eliminate the wanted terms. In this section we describe an approach to generate better candidate terms.

With a few simple restrictions we can generate much better candidate λ-terms. First, we will always generate a number of abstractions that corresponds to the number of arguments needed by the function at hand. Second, there is no need to generate open λ-terms. In the generation we keep track of the bound variables and only generate them at applied occurrences. In principle that can be further improved by keeping track of the type of these variables, as done by Katayama [6]. Third, we will generate the right number of arguments for constants like `Plus` and `If`. Forth, it is useless to generate numerical constants as first argument of an `Ap`. Fifth, if the right constants are generated then there is no need to generate complex subexpressions containing only constants.

For `pY` defined above we clearly need a higher order function. Hence we add the introduction of new abstractions in the generated expressions. The function `ho` generates higher order candidate functions. The first argument of `ho` is the number of arguments needed, the second argument a list of the bound variables, and the last argument the name (number) of the next argument.

```
ho :: Int [V] Int → [Expr]
ho 0 vs x = r
where
    r = l |. [Const (C i) \\ i←[−1,1]]
    l = [Var v                  \\ v ← vs]
        |. [Ap e1 e2            \\ (e1,e2) ← diag2 l r]
        |. [Abs (V x) e         \\e ← ho 0 [V x:vs] (x+1)]
        |. [Ap (Ap Plus e1) e2  \\ (e1,e2) ← diag2 l r]
ho i vs x = [Abs (V x) e \\ e ← ho (i−1) [V x:vs] (x+1)]
```

As a grammar this is:

$$
\begin{aligned}
ho\ n\ m &= \lambda v_n \ldots \lambda v_1 \,.\, r \\
\text{where } r &= l \mid -1 \mid 1 \\
l &= v_m \mid \ldots \mid v_1 \mid l\ r \mid \lambda v_{n+1} \,.\, ho\, 0\ (m+1) \mid (Plus\ l)\ r
\end{aligned}
$$

The infix operator |. merges two lists into a single list by taking elements from the argument lists in turn.

```
(|.) infixl 4 :: [x] [x] → [x]
(|.) [a:x] y  = [a: y |. x]      // note the swap of argument
(|.) []    y  = y
```

Now the first 1,000,000 candidates generated contain 7 Y-combinators. Some examples are λa.(λb.b b) (λb.a (b b)), λa.(λb.a (b b)) (λb.a (b b)), and λa.((λb.b) (λb.b b)) (λb.a (b b)).

Most interesting functions are recursive. However, this does not imply that we need to generate higher order λ-terms. It is sufficient to generate terms containing an application of a predefined Y-combinator. Moreover, for recursive functions that yield a nonrecursive type, like Int, it is essential to contain a stop condition. That is, after the Y-combinator there should be a conditional (an If) before the recursive occurrence of the recursive function. This is exactly what the generator of λ-terms fun does.

```
fun :: Int → [Expr]
fun n = [abs 1 n e \\ e ← r []] |. [Ap exprY (abs 0 (n+1) e) \\ e ← rFun]
where
    vars = [Var (V v) \\ v ← [1..n]]        // V 0 is the recursive fun if it exists
    r c  = [Const (C i) \\ i ← [−1,1,−2]] |. e c
    e c  = vars |. [Ap (Ap Plus e1) e2 \\ (e1,e2) ← diag2 (e c) (r c)] |. c
    rFun = [Ap (Ap (Ap If c) t) e
            \\ (c,t,e) ← diag3 simple (e (rApp n)) ([Const (C i) \\ i ← [0,1]] |. vars)]
    rApp 0 = [Var (V 0)]
    rApp n = [Ap f a\\(f,a)← diag2 (rApp (n−1)) simple]
    simple = vars |. [Ap (Ap Plus v) c\\(v,c)←diag2 vars [Const (C i) \\ i ← [−1,−2]]]
    abs n 0 e = e
    abs n m e = Abs (V n) (abs (n+1) (m−1) e)
```

Using this generation function we will look for a term that implements multiplication by repeated addition. Since we want to prevent (very) large values as arguments for this multiplication function (it is $O(n)$), we select some test values manually rather than using a quantification over all integers.

```
pTimes p = ¬(f 0 3 &&& f 2 4 &&& f 7 5 &&& f 3 0)
where f a b = Ap (Ap p (Const (C a))) (Const (C b)) ≡ Const (C (a*b))
```

We look for 2-argument terms generated by the function fun by testing pTimes **For** (fun 2). The system produces multiplication functions for non-negative numbers of the form:

```
Y (λa.λb.λc.((If c) ((Plus ((a ((Plus c) −1)) b)) b)) 0),
Y (λa.λb.λc.((If c) ((Plus ((a ((Plus c) −1)) b)) b)) c)
```

and terms that a human is more likely to write

```
Y (λa.λb.λc.((If c) ((Plus ((a b) ((Plus c) −1))) b)) 0),
Y ((λa.λb.λc.((If b) ((Plus ((a ((Plus b) −1)) c)) c)) 0)
```

In these expressions we use Y as abbreviation of the term (λa.(λb.a (b b)) (λb.a (b b))). The first two terms are somewhat peculiar due to the swap of arguments. As a direct recursive function the second term is:

```
f b c = if (c>0) (b+f (c−1) b) c
```

Although this function looks extraordinary through the swap of arguments, it computes the desired product of non-negative arguments.

Terms for $s\ n = \sum_{i=0}^{n} i$ are found by testing `pSum` **For** `(fun 1)` with

```
pSum p = ¬(f 3 &&& f 5) where f a = Ap p (Const (C a)) ≡ Const (C (sum [1..a]))
```

The first term found is `Y (λa.λb.((If b) ((Plus (a ((Plus b) −1))) b)) 0)`.

In the same way we can look for λ-terms matching $f\ 4 = 5 \wedge f\ 5 = 8$ from the introductions by looking for counterexamples for:

```
pFib p = ¬(f 4 5 &&& f 5 8) where f a b  = Ap p (Const (C a)) ≡ Const (C b)
```

The first solutions found by testing `pFib` **For** `(fun 1)` are not the Fibonacci function found in our earlier work, but nonrecursive terms such as

```
λb.(Plus ((Plus ((Plus ((Plus ((Plus b) −1)) −1)) −1)) b))
         ((Plus ((Plus ((Plus ((Plus b) −1)) −1)) −1)) −1)
```

and some single-recursive term like

```
Y (λa.λb.((If ((Plus b) −2)) ((Plus (a ((Plus b) −1))) ((Plus ((Plus b) −1))−1))) b)
```

Counterexample 13 found after 1583 test is the first (double-recursive) Fibonacci function:

```
Y (λa.λb.((If ((Plus b) −1)) ((Plus (a ((Plus b) −2))) (a ((Plus b)−1)))) 1)
```

By adding $f\ 6 = 13$ to the patterns to be matched, this is the first term found.

The speed of generating and testing candidate functions depends strongly on the condition that has to be evaluated and the size of the expression. On a rather slow (1GHz) Windows XP laptop we measured a speed of 500 to 200,000 candidate terms per second.

7. Discussion and related work

In this paper we demonstrate that it is possible to find λ-terms matching some condition by systematic synthesis of candidate expressions. Since we want to be able to find terms like the Y-combinator, restricting ourselves to terminating expressions is no option. This implies that testing the suitability of a candidate expression is rather delicate. The equivalence of terms is known to be undecidable. In this paper we used a simple approximation: a finite (and rather small) normal order reduction steps are done on the terms to be compared. If the reduction sequences contains elements that are α-equal the terms are obviously α-equivalent. If the terms are unequal normal forms the terms are non-equivalent. Otherwise the equivalence has the value *undefined*.

It appears that the number of λ-terms is too large to find most interesting terms by brute force search in reasonable time. We have introduced two rather simple but effective generators for expressions. The first one generates higher order terms like the famous Y-combinator. The second one generates (recursive) functions like multiplication by repeated addition and the Fibonacci function. By using type information it is possible to generate candidate functions even more effectively. Katayama [6] uses this in his generation of functions matching examples. He generates only first order terms, all other things (like recursion) have to be defined as a recursion pattern in a library of primitive functions. Broda [5] and Wang [11] discuss algorithms to generate λ-terms randomly. Broda uses a grammar to specify the type of the terms, somewhat similar to our generation functions. Henk Barendregt has touched the generation of λ-terms via enumeration in [1, 2, 4].

By adding constructs like the Y-combinator and multiplication to the terms, the generated terms become more powerful. Hence complex functions will be found quicker.

Without the struggle for nontermination, it is more elegant to introduce a type class `eval` to evaluate instances of various grammars represented as type [9]. The grammar of the

candidate terms can then elegantly and effectively be determined by more specific types. The types control the generation of candidate functions at a higher level of abstraction than the generation functions used in section 6. When intermediate terms in the reduction need to be compared, as needed to compare λ-terms for equivalence, this is not possible.

We find λ-terms generalizing the behavior of the given input-output pairs or properties. Both the obvious functions and more surprising λ-terms are synthesized. If the goal is to find only primitive recursive functions the direct approach in [9] is more effective. This paper shows that it is possible to find the primitive recursive functions as well as other λ-termslike the Y-combinator. The advantage of the approach introduced in this paper is that it is able to synthesize λ-terms for general properties without the need to define a very precise grammar for the candidate functions. Some guidance is needed to find larger terms, but the generators might produce totally wrong candidates (like ill-typed terms or terms with nonterminating reduction sequences) without causing any trouble.

Acknowledgement

The authors wish to thank the anonymous referees, the editors and Peter Achten for their suggestions to improve this paper.

References

[1] H. Barendregt. *The Lambda Calculus, its Syntax and Semantics. Revised edition*, volume 103 of *Studies in Logic*. North-Holland, 1984.
[2] H. Barendregt. Enumerators of lambda terms are reducing. *J. Funct. Program.*, 2(2):233–236, 1992.
[3] H. Barendregt. Lambda calculi with types. In M. Abramsky, Gabbay, editor, *Handbook of Logic in Computer Science*, volume 2. Oxford university press, 1992.
[4] H. Barendregt. Enumerators of lambda terms are reducing constructively. *Ann. Pure Appl. Logic*, 73(1):3–9, 1995.
[5] S. Broda and L. Damas. Generating normal inhabitants of types with a common structure. Technical Report DCC-2001-1, DCC-FC & LIACC, Universidade do Porto.
[6] S. Katayama. Systematic search for lambda expressions. In *Proceedings Sixth Symposium on Trends in Functional Programming (TFP2005)*, pages 195–205, 2005.
[7] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In R. Peña and T. 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.
[8] P. Koopman and R. Plasmeijer. Generic Generation of Elements of Types. In *Proceedings Sixth Symposium on Trends in Functional Programming (TFP2005)*, Tallin, Estonia, Sep 23-24 2005.
[9] P. Koopman and R. Plasmeijer. Systematic Synthesis of Functions. In H. Nilsson, editor, *Proceedings Seventh Symposium on Trends in Functional Programming, TFP 2006*, pages 68–83, Nottingham, UK, The University of Nottingham, April 19-21 2006.
[10] R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.ru.nl/∼clean/.
[11] J. Wang. Generating random lambda calculus terms. Master's thesis, Boston University, 2005.