Learning Recursive Program Schemes

Bachelor Seminar
Cognitive Systems Group
University of Bamberg

Thorsten Spieker Christophe Quignon

April 12, 2010

1 Introduction

In inductive programming a system tries to find a program to solve a problem using inductive reasoning. Current approaches and systems that learn programs given a set of inputs and outputs try to find a program just on the base of these specifications.[HKS09] They work independantly from one program to the next without using already learned programs as knowledge for future programs. Having such background knowledge to use in the inductive process could help an inductive programming system by both improving performance in terms of speed and in terms of quality of found algorithms. The problem lies in representing such background knowledge to make use of it. Just saving already learned programs will not bring the improvement we hope to get. One approach to this is a partially ordered program scheme repository which consists of already learned programs and more generic program schemes that are computed by taking two programs or program schemes and computing the anti-instance of the two. This will result in a partially ordered repository from the most generic program scheme to specific programs that result from instantiating program schemes.

An inductive programming system could then try to find similarities between a given problem specification and the problem specifications of already learned programs and use more generic program schemes than the learned program corresponding to the most similar problem specification to find a new instantiation of program schemes that solve the new problem. Our method to find program schemes from two specific programs uses tree representations of the programs in a form where tree nodes are functions and the children of the nodes are the functions arguments. Leafs of the tree will be variables, also representing constant values. The benefit of using this representation is that we can use a tree matching algorithm to find similarities between the trees and what operations transform one tree into the other. With this information we find the program scheme using transformation operations to create a generic tree and a set of mapping instructions. To create the two programs instantiation of the functions and constants und the program scheme is used. Our implementation is described in further detail in section 3.

2 Background

A program is defined as a term, in particular a set of equality tests on functions, variables and constants. A program scheme is a generic program with generic function and variable placeholders that have to be instantiated with concrete functions and variables or constants to obtain a working program. A program scheme that generalizes over two programs can be obtained by calculating the anti-instance of the two programs since both are represented as terms.[SSW00] The anti-instance will represent the program scheme that can be instantiated to obtain both programs.

An anti-instance is defined as the anti-unification of two terms.[Rey70] There are two main methods to obtain an anti-instance. One obtains only a first order anti-instance which does not generalize over functions and one obtains a second order anti-instance which does generalize over functions but includes some problems. First order anti-unifications allow only the substitution of variables in terms by other variables or a function (Figure 1). It then obtains the maximally specific generalization of two programs by finding substitutions that make both programs equal. When calculating the set of possible anti-instances by using substitution this leads to a finite set because there is only a finite number of different substitutions that can be generated. This is an important property of the first order anti-unification algorithm. However there is an important drawback. Since only variables are being substituted the first-order anti-unification cannot find the similarity of two terms that use different functions on the same argument: The anti-unification of $G(a)$ and $F(a)$ yields a single variable x as their anti-instance. Therefore this method is not sufficient for finding similarities and computing the program scheme of two recursive programs.[Sin00] See the two examples of first order anti-unification in figure 2 and 3.

Second order anti-unification also allows the substitution of a function by another function. This improves the detection of similarities by many positive aspects, like argument permutation, generalizing over functions of different arity et cetera, but also has one important drawback. The computation of all possible anti-instances now results in an infinite set of anti-instances. This comes from the possibility to endlessly insert and remove arguments from functions. To still use benefits of second order anti-unification it is important to restrict the operations allowed to get from the anti-instance to the input terms to the following operations: substitution of variables, substitution of function names and deletion of function arguments. What is not allowed is the insertion of arguments.[Wag02] With those restrictions the second order anti-unification is a much better fit for finding a proper program scheme of two programs as it finds more similarities and still results in a finite set of possible anti-intances.

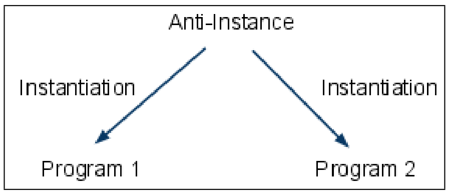

Figure 1: Anti instance

After introducing two different methods for computing anti-instances of two terms the question remains what properties such anti-instance must hold to qualify as a program scheme. An obvious property is correctness which is supplied by the algorithms themselves. Next to correctness the anti-instance should be of the kind, that the two input terms (programs) can be obtained with the minimal amount of instantiations from the anti-instance. This ensures that there are no unnecessary variables or functions in the anti-instance which are not needed to obtain the input terms. This means the anti-instance needs to be the maximally specific generalization of the two terms. This is given by the first order algorithm but remains a problem with the second order algorithm. Given a tree representation of the programs a tree matching algorithm computes how similar two programs are.

We are using an algorithm for ordered trees to find the edit distance of the two tree representations of our programs. The algorithm also returns the operations needed to transform one tree into the other. Since the algorithm finds the minimal amount of operations we can use these operations to find the smallest anti-instance by directly computing it from the trees and the operations instead of computing a set of all possible anti-instances and then selecting the smallest like the second order algorithm would do. The algorithm that computes this smallest anti-instance is described in section 3.2. Note that the algorithm only handles ordered trees which means it does not allow argument permutation within functions which would only be possible with a tree matching algorithm that compares unordered trees. However we did not find a proper algorithm for unordered trees that serves our needs and therefore had to find a work-around to make argument permutation possible with neither an insert operation from the anti-instance to the specific programs nor a tree matching algorithm for unordered trees. The tree matching algorithm and the work-around are further presented in section 3.1.

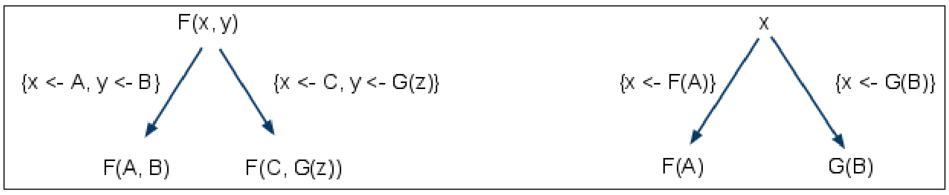

Figure 2: First order anti unification

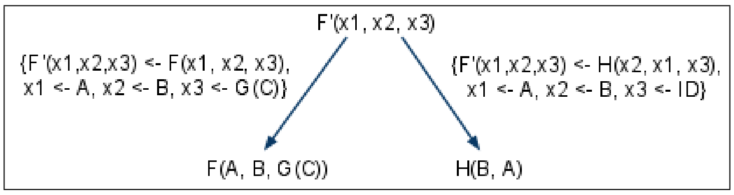
F'(x1, x2, x3)
{F'(x1,x2,x3) <- F(x1, x2, x3), x1 <- A, x2 <- B, x3 <- G(C)}
{F'(x1,x2,x3) <- H(x2, x1, x3), x1 <- A, x2 <- B, x3 <- ID}
F(A, B, G(C))
H(B, A)

Figure 3: Second order anti unification

3 Implementation

Our implementation is written in the functional programming language Haskell. The two programs from whom the anti-instance shall be generated need to be written to our input file in a tree representation where a node is represented by a string and its children follow this string where each child is encapsulated in brackets.[TlZJS92] Figure 4 is an example how to convert a functions into their tree representations. At first the function name is pulled from infix to prefix and both, the whole term as well as both function arguments, are encapsulated in brackets. This is repeated until we reach the tree representation.

Function: $add(x, y)$ = $x + y$
Tree:
$(= (add(x, y))\ (x + y))$
$(= (add(x)(y))\ (x + y))$
$(= (add(x)(y))\ (+(x)(y)))$
Function:
$sumList\ x = if\ (x == [])\ then\ 0\ else\ (head\ x)\ +(sumList(tail\ (x)))$
Tree:
$(sumList$
$(= (sumList([]))(0))$
$(= (sumList(x))\ (+(head(x))\ (sumList(tail(x))))))$

Figure 4: Example of function splitting

Afterwards, the program reads the input file and processes the trees. It runs an external program that calculates the distance between the two trees and the transform operations needed to transform one tree into the other. The number of operations is called the distance. This is described in the section 3.1. The program then passes the operations to the anti-instance part. This is described in section 3.2. Afterwards the new program scheme and the mappings to generate the input programs from the program scheme by instantiation are written to an output file.

3.1 Tree Matching

Within our program we use an internal representation of the input programs. The data structure is an abstract syntax tree with a few modifications. To differentiate between commutative functions (like addition or multiplication) and non-commutative functions (like subtraction and division) we introduce an attribute for commutative functions in our data structure. Using this attribute our algorithm is able to find function similarities between two programs that result from argument permutation without destroying the semantics of a function and their program. One example is $div(x, y)$ and $sub(y, x)$. While both functions seem to be similar by permuting the arguments, it would destroy the semantics of substraction and division which are both non-commutative functions. With our internal representation it is possible to find argument permutation similarities with the described restriction. However, since we are working with a string representation of the trees as an input to our program that is needed for the tree matching algorithm, described in the previous section, we cannot predict if a function is commutative or not when reading the input file. Therefore we do not allow argument permutation at all in our current implementation.

After reading the input file and parsing the string into our data structure both trees are passed to the external tree matching algorithm.[1] The algorithm computes the edit distance of two trees and returns the distance plus the operations to obtain the distance. The edit distance is defined as the minimal number of operations to transform one tree into the other.[TlZJS92] Figure 5 shows the three operations, their description and representation. The positions are in post traversal order.

- $RENAME$ p1 n1 p2 n2
  - renames a node n1 at position p1 in the first tree to node n2 at pos p2 in the second tree
  - Example: (2 fib 2 $fact$)
- $INSERT$ _ _ p2 n2
  - inserts a node with label name at position pos in the second tree
  - Example: (_ _ 12 y)
- $DELETE$ p1 n1 _ _
  - deletes a node with label name at position pos in the first tree
  - Example: (6 fib _ _)

Figure 5: Operations used by the algorithm to transform the trees

3.1.1 Example output

The example output is given in figure 6. The first two lines print the string representation of the tree as described in the introduction of section 3. The third line prints the edit distance, the smallest number of operations needed to transform one tree into the other. From the 9th line until the end all the operations are listed which correspond to the smallest distance. The first operation is a renaming operation but since both names are equal, this operation is not counted towards the edit distance. Only real renaming operations as the second operation are counted. Operation number 9 deletes the node 9 with label $reverse$ in the first tree. Operation

[1]Generously provided by Prof. Dennis Shasha from New York University

number 12 inserts a node with label $sumList$ at position 11 in the second tree. Note that the first number in each operation is not the number of the operation but the position of the node that the operation is performed on.

```
Tree1:
(reverse
(= (reverse([]))(0))
(= (reverse(x))(++ (reverse(tail(x)))(head(x)))))
Tree2:
(sumList
(= (sumList([]))(0))
(= (sumList(x))(+(head(x))(sumList(tail(x))))))
Distance:   8
( 1 []   1 [] )
( 2 reverse   2 sumList )
( 3 0   3 0 )
( 4 =   4 = )
( 5 x   5 x )
( 6 reverse   6 sumList )
( 7 x   7 x )
( 8 tail   8 head )
( 9 reverse )
( 10 x   9 x )
( 11 head  10 tail )
( 11 sumList )
( 12 ++  12 + )
( 13 =  13 = )
( 14 reverse  14 sumList )
```

Figure 6: Example output for tree matching of $sumList$ and $reverse$

This example is not the smallest distance of the two programs if argument permutation is enabled. If the permutation option is enabled our implementation will compute the set of all possible permutations of both trees and send each pair into the tree matching algorithm to find the smallest distance. Note that a smaller distance is a smaller number of operations to transform a tree. This means that a commutative function with two arguments will result in a distance that is smaller by at least 2 (2 renaming operations for the arguments) but possibly a lot more considering that an argument could be a whole subtree. The minimal distance therefore is found by finding the lowest distance among all combinations of permutations of the two trees and enables us to find the most specific generalization of both programs given the restrictions described in section 2. In the above example you can see the recursive call of the respective function in the last operation ($+$ and $++$ respectively). The call is the first argument in the $reverse$ program and the second in the $sumList$ program. If we would use the knowledge that the $+$ operation is commutative we could swap the arguments in the second program and observe a smaller distance from the tree matcher. Instead of the $delete$ operation (number 9) and the add operation (number 12) we would have only one renaming operation. Also the renaming operations of tail to head and vice versa (number 8 and 11) would not be necessary. After we obtained the smallest distance (either with or without the permutation of arguments) the results are then read including the edit operations and then passed to the anti-instance part of our implementation.

3.2 Anti-Instance

The Anti-Instance Algorithm itself need as input 2 programs, represented by their abstract syntax tree ”AST” and a list of ”commands” which describe the editing distance between the two trees. This is exactly what the TreeMatching program delivers. Both trees AST_a and AST_b get their abstract syntax tree form internally by simple parsing. Without any additional information about the semantics behind the program, we assume that their arguments are not commutative.

3.2.1 Definitions

As the type definition given in figure 7 shows, an AST consists of an identifier in string form, a boolean, indicating if the function is permutativ or not and a list of children, which are also ASTs.

$$\boxed{\begin{array}{l} AST : \{\mathbb{S}, \mathbb{B}, (AST)^*\} \\ COMM : ((\mathbb{N}, \mathbb{S}), (\mathbb{N}, \mathbb{S})) \\ MAP : ((\mathbb{N}, AST_a), (\mathbb{N}, AST_b)) \end{array}}$$

Figure 7: Type definitions. $\mathbb{S}$ represent strings, $\mathbb{B}$ booleans and $\mathbb{N}$ integers

The different parts within this types can be accessed as stated in figure 8. With the index a (b) one can get the first (second) position of a tuple. The pos function returns the integer value from there and the val function the second position of one tuple.

$$\boxed{\begin{array}{l} c \in COMM \\ c \leftarrow ((n1, s1), (n2, s2)) \\ \\ c_a = s1; c_b = s2; \\ pos(c_s) = n1; \\ val(c_s) = \mathbb{S} \leftarrow AST_s \end{array}}$$

Figure 8: Subtype accessor definitions

The external information about the shortes editing path (figure 6) is parsed into a list of tuple of tuples, called ”COMM”. One $COMM$ of this list represents at first position the element of AST_a and the second element of AST_b. Both elemts are represented by their postorder position in their AST and the string representing their name. If the TreeMatcher does suggest not a mapping, but a insertion or deletion by giving just one tuple, the missing one is instantiated with an empty position and the string ”ID”, meaning that the function may stay the same without this subAST.

In the stated algorithm the positions can be reached by the index of their AST eg. $COMM_a$. The numeric value of the elemt can be obtained by the function $pos(COMM_i)$ and the function name by $val(COMM_i)$. Beneath the AST which is the Anti-Instance of both given AST_s, an list of mappings MAP is returned. One element of MAP has an index S_i and a list of instances represented by that index. All instances may be instanciated while generalizing the Anti-Instance. Figure 9 and 10 show examples of the datatypes. The algorithm itself is given in figure 11.

There are two critical points in this algorithm:

- The definition of ”new” element
- The check, if one element is also mapped

As one may point out, the ”new” element is an error in type theory, for it is an AST and not a tuple of strings. This exactly is the critical point, the need to find a representation of the mapping in the returned AST. The ”new” element can, as stated her, be achieved by simple mapping of one atomar element to another atomar element. This reduces the effort to a bare minimum, but does not account the fact, that this atomar element may represent a complex function, with underlying structure and constraints of commutation. This can lead lead to errors when multiple insertions occur. To merge these multiple insertions to one insertion of a complex AST will solve the problem. The checking of already mapped functions is critical if the mappings shall be minimal and especially without circular dependencies, which may fault the reinstanciation of the Anti-Instance. We check and reduce our mapping list against double occurrences.

(11 head 10 tail) $\rightarrow COMM$((11, ”head”), (10, ”tail”))
(11 sumList) $\rightarrow COMM$((0, ”ID”), (11, ”sumList”))

Figure 9: Example for $COMM$

```
AS1{"sumList", 1, (
      AST {"=", 0, (
           AST {"sumList", 0, (
                AST {"[]"})},
           AST{ "0"})},
      AST {"=", 0, (
           AST {"sumList", 0, (
                AST { "x"})},
           AST {"+", 1, (
                AST {"head", 0, (
                     AST {"x"})},
                AST {"sumList", 0, (
                     AST {"tail", 0, (
                          AST { "x"})})})})})})}
```

Figure 10: The AST to the "sumList" program, that adds all values of a list

```
$MAP \leftarrow []$
for all $c \in COMM$ do
  $new \leftarrow (val(C_a), val(C_b))$
  $dest \leftarrow pos(C_a)$
  $a_{dest} \leftarrow new$
  $MAP \leftarrow MAP + new$
end for
$i \leftarrow 0$
for all $m \in MAP$ do
  for all $p \in AST_a$ do
    if $m \subseteq p$ then
      $p \leftarrow S_i$
    end if
    $m \leftarrow (S_i, p)$
  end for
  $i \leftarrow i + 1$
end for
return $(AST_a, MAP)$
```

Figure 11: The core algorithm

4 Results

To present that our algorithm works we have compiled a list of comparison programs and computed their program schemes. Note that some of the comparisons are similar to already computed generalizations in other works.[Sin00] [SSW00] [Wag02]

4.1 List of programs

We have picked four recursive programs to perform our comparisons with which have a number of properties that made the computation of program schemes both fast and simple yet meaningful. All programs where hand-written from memory without searching for a proper implementation. First we picked a program that computes the n-th fibonacci number. This recursive program has two synchronous recursive calls and therefore shows what happens to the second call when comparing it to a recursive program with only a single recursive call. It also contains two base cases instead of one which is more common in recursive programs. Second we picked a program that computes the factorial of a number n. This program has the mentioned single recursive call and both programs work on natural numbers. Third we picked two programs that work on lists instead of natural numbers which are very similar to each other. One program computes the sum of a list of natural numbers and the other reverses a list with elements of some arbitrary type. Both programs have only a single recursive call on the tail of the list and perform some operation on the head and the result from the recursive call.

1. $Fibonacci$:
   $(fib$
   $\quad(= (fib(0))(1))$
   $\quad(= (fib(1))(1))$
   $\quad(= (fib(x))(*(fib(-(y)(1)))(fib(-(z)(2))))))$

2. $Factorial$:
   $(fact$
   $\quad(= (fact(0))(1))$
   $\quad(= (fact(a))(*(b)(fact(-(c)(1))))))$

3. $SumList$:
   $(sumList$
   $\quad(= (sumList([]))(0))$
   $\quad(= (sumList(x))(+(head(x))(sumList(tail(x))))))$

4. $Reverse$:
   $(reverse$
   $\quad(= (reverse([]))(0))$
   $\quad(= (reverse(x))(++(reverse(tail(x)))(head(x)))))$

4.2 Example Anti-Instances

4.2.1 (1) $Fibonacci$ and (2) $Factorial$:

The program scheme resulting from the $fibonacci$ and the $factorial$ program contains both recursive calls of the $fibonacci$ program and shows that one of them is deleted when instantiated with the identity function to obtain the $factorial$ program. The same happens to one of the base cases of the $fibonacci$ program.

Function:
$(s1$
  $(= (s1(0))(1))$
  $(s6(s7(s8))(s8))$
  $(= (s1(s12))(*(s1(-(s16)(1)))(s7(s19(s20)(s21))))))$

Mappings:
$s1 \rightarrow (fib; fact), s6 \rightarrow (=; ID), s7 \rightarrow (fib; ID),$
$s8 \rightarrow (1; ID), s12 \rightarrow (x; a), s16 \rightarrow (y; c),$
$s19 \rightarrow (-; ID), s20 \rightarrow (z; ID), s21 \rightarrow (2; ID)$

4.2.2 (1) $Fibonacci$ and (3) $SumList$:

The program scheme resulting from the $fibonacci$ program and the sumList program shows a generalization over functions where one is a recursive call and the other a simple head of a list. It also shows that the algorithm does not care about types at all.

Function:
$(s1$
  $(= (s1(s4))(s5))$
  $(s6(s7(s8))(s8))$
  $(= (s1(x))(s13(s14(s15(s16)(s8)))(s1(s19(s20)(s21))))))$

Mappings:
$s1 \rightarrow (fib; sumList), s13 \rightarrow (*; +), s14 \rightarrow (fib; ID),$
$s15 \rightarrow (-; head), s16 \rightarrow (y; x), s19 \rightarrow (-; tail),$
$s20 \rightarrow (z; x), s21 \rightarrow (2; ID), s4 \rightarrow (0; []),$
$s5 \rightarrow (1; 0), s6 \rightarrow (=; ID), s7 \rightarrow (fib; ID),$
$s8 \rightarrow (1; ID)$

4.2.3 (4) $Reverse$ and (2) $Factorial$:

The program scheme resulting from the $reverse$ program and the $factorial$ program is another example over type generalization.

Function:
$(s1$
$(= (s1(s4))(s5))$
$(= (s1(x))(s9(s10(s11(x)))(s13(x)))))$

Mappings:
$s1 \to (reverse; fact), s4 \to ([]; 0), s5 \to (0; 1),$
$s9 \to (++; *), s10 \to (reverse; ID), s11 \to (tail; ID),$
$s13 \to (head; -)$

4.2.4 (4) $Reverse$ and (3) $SumList$ (without permutation):

The program scheme resulting from the $reverse$ program and the $sumList$ program shows the differences of enabling argument permutation and disabling it. When disabling it most of the functions have to be generalized…

Function:
$(s1$
$(= (s1([]))(0))$
$(= (s1(x))(s9(s10(s11(x)))(s13(x)))))$

Mappings:
$s1 \to (reverse; sumList), s9 \to (++; +), s10 \to (reverse; ID),$
$s11 \to (tail; head), s13 \to (head; tail)$

4.2.5 (4) $Reverse$ and (3) $SumList$ (with permutation):

…while some can be kept (namely $head$ and $tail$) when enabling argument permutation.

Function:
$(s1$
$(= (s1([]))(0))$
$(= (s1(x))(s9(s1(tail(x)))(head(x)))))$

Mappings:
$s1 \to (reverse; sumList), s9 \to (++; +)$

These results show that our algorithm works with programs of any type, any number of recursive calls as recursive calls are just handled like any other non-recursive function, any number of base cases and therefore equations in the program. It is also impossible to find a program scheme that obtains the input programs with a lesser number of instantiations, therefore the program schemes are the most specific generalizations given the restriction on argument permutation and the prohibition of insertions while keeping the deletion operations using the identity function instantiation to delete a variable or function.

5 Future Work

The goal of this seminar was to find an algorithm that computes a program scheme as a generalization over two precursive programs. We have found a way to make use of second order anti-unification and its benefits plus implemented a prototype that uses an external program to compute the edit distance. We have shown that the algorithm computes correct anti-instances and therefore program schemes that are also minimal given the number of instantiations to retrieve the input programs as a minimality property. We have also shown that the algorithm the algorithm is type free and generalizes both over multiple recursive calls and multiple base cases or equations in a program.

We did not integrate a tree matching algorithm in our own implementation nor define an input representation that makes use of the argument permutation option of our algorithm.

There are a lot of possibilities to extend this work in the future. First of all, to make use of our permutation possibility if one would either define an input format that represent commutative functions or make the algorithm work with a background knowledge repository of commutative functions. However the implementation of this would have been beyond the limits of the seminar so it stays as future work while the functionality of permutation has been implemented. Also it would be a great speed improvement to port or interface the tree matching algorithm into Haskell to completely integrate it in our program.

Further possibilities also include the integration of this method in an inductive programming learner. This however can only be the case if a proper repository representation has been introduced and implemented including save and lookup methods to work with the repository. Another problem is that there is no direct benefit to have all learned programs and their program schemes in a repository as an inductive programming learner only works on problem specifications. This means that such a background repository would need to store the problem specifications to its corresponding learned programs through which the learner could find similarities to the current problem and then try out several program schemes based on the similarity of the problem specification. However similarity of problem specifications is another big task to find and implement. Given such a problem similarity it would be a pretty simple task to find a program scheme from the repository that could help in finding a program for a given new problem. One would only need to compute the similarities to all already learned problem specifications and then retrieve all program schemes that generalize over the most similar problem. Now an algorithm can try to instantiate the program schemes ordered from most specific to most general to solve the given problem. If a new program is found the repository needs to be updated with new program schemes given the new program. Over time a program learner would become faster and produce more efficient programs.

References

[HKS09] Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. A unifying framework for analysis and evaluation of inductive programming systems. In *Proceedings of the Second Conference on Artificial General Intelligence (Second Conference on Artificial General Intelligence (AGI-09)*. Goertzel, Ben and Hitzler, Pascal and Hutter, Marcus, 2009.

[Rey70] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence*, volume 5, pages 135–152. Meltzer, Bernard and Michie, Donald, 1970.

[Sin00] Uwe Sinha. Gedächtnisorganisation und abruf von rekursiven programmschemata beim analogen programmieren durch typindizierte anti-unifikation. Diplomarbeit, 2000. FB Informatik, TU Berlin.

[SSW00] Ute Schmid, Uwe Sinha, and Fritz Wysotzki. Generalizing recursive program schemes with anti-unification (abstract), 2000.

[TlZJS92] Jason Tsong-li, Wang Kaizhong Zhang, Karpjoo Jeong, and Dennis Shasha. A system for approximate tree matching. *IEEE Transactions on Knowledge and Data Engineering*, 6:559–571, 1992.

[Wag02] Ulrich Wagner. Combinatorically restricted higher order anti-unification. an application to programming by analogy. Diplomarbeit, 2002. FB Informatik, TU Berlin.