Applying Anti-Unification Strategies to Matching and Generalization of Recursive Functions

-

Investigating a Second-order approach for Learning from Examples

Masterarbeit

im Studiengang Angewandte Informatik
der Fakultät Wirtschaftsinformatik
und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

Verfasser: Sebastian Boosz

Gutachterin: Prof. Dr. Ute Schmid

Acknowledgement

I would like to thank Prof. Dr. Ute Schmid for suggesting the interesting and exciting topic. Her enthusiasm about the progress and achievements has been inspiring and motivating. I thank her for the ideas and help she provided during our meetings.

I would also like to thank Dr. Temur Kutsia and Alexander Baumgartner of the Research Institute for Symbolic Computation belonging to the Johannes Kepler University Linz. They created the anti-unification library which has been essential for the thesis. They also helped me to get started with the usage of their library by constructing and describing two insightful examples.

Abstract

Programmers seldom invent completely new programs, instead they tend to adapt existing programs by analogical reasoning. After a suitable source program has been found, it is a feasible strategy to picture the execution behavior of that program for a certain input. The programmer then tries to envision the execution behavior of the desired program for the very same input. Comparing, she can draw conclusions about what both programs have in common and where the differences are located. Adapting the existing program by applying those differences appropriately, yields the desired program.

In this thesis a programming by analogy approach is investigated for recursive functions. Based on concrete unfoldings of source and target functions, higher-order anti-unification is used to find a generalization, revealing differences between source and target. A set of heuristics was developed which is used to match and apply those differences to the source function in order to transform it into the corresponding target function. The success of the heuristics is evaluated and possible means of improvement are suggested.

Keywords: *recursion, recursive functions, higher-order anti-unification, generalization, matching, programming by analogy, learning from examples, heuristics*

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

Douglas Hofstadter [Hof79]

Contents

List of Figures

List of Tables

1 Introduction

Programming by analogy is a promising approach to ease the life of programmers and to allow for more convenient program reuse techniques, as programmers seldom invent completely new programs. Instead they use analogical transfer to transfer programs they have already written into new programs. While programming by analogy approaches are far from market maturity, they have a lot of potential, especially since cognitive sciences agree that analogy and analogical reasoning play seminal role in the development of our abilities and minds.

Learning programs according to analogical principles yields another benefit. If a program has been learned from examples, the result may be more comprehensible for human users as it was created according to the way of thinking we might utilize ourselves.

This thesis introduces our programming by analogy approach for recursive functions, which we chose due to their importance for functional programming and their fascinating underlying idea. Such function definitions are concise and intuitive, but nevertheless expressive. They are ideal candidates for learning from examples, as so called unfoldings, which act as exmaples, can be derived directly from their definitions. These unfoldings act as examples and we aim to learn the corresponding recursive function definition by adaption of another recursive function, that is already known.

As a prerequisite for the automatic transformation of a base function into a target function, the commons structures and differences between base and target have to be identified. For this we use a technique called anti-unification.

The thesis has the following structure: Chapter 2 introduces the foundations the thesis is based on, i.e. recursive functions, analogy and programming by analogy as well as anti-unification.

In chapter 3 we present our approach in detail. We give a top-down overview of the whole procedure, then describe our representation of recursive functions and which functions we investigated. We explain the heuristics and mechanics that we have developed. Finally, we give two detailed examples.

Chapter 4 is an evaluation our approach. We do an assessment of the achieved success. We mention typical problems and deficits. Additionally, we provide some ideas which could help to further improve our approach.

Concluding, chapter 5 summarizes what has been done in the thesis and gives an outlook on possible further developments.

2
Foundations

In this chapter we introduce the foundations on which our approach is constructed upon. First the concepts of *recursion* and *recursive functions* are covered. Then we will provide information about analogies and mention some existing approaches for *Programming by Analogy*. Eventually we will address the topics *unification* and *anti-unification*, which is the enabling technique for our approach.

2.1. Recursion and Recursive Functions

According to Hofstadter [Hof79, Chapter 5] recursion is a very general concept. It can be applied to all phenomena which include nesting, e.g. "stories inside stories, movies inside movies, paintings inside paintings, Russian dolls inside Russian dolls."

In daily life recursion occurs when we "postpone completing a task in favor of a simpler task, often of the same type". Sometimes, for completing a bigger task, it might also be necessary to fulfill a set of smaller tasks first, which only solve a part of the problem. For example, if we want to clean the house, we have to clean the kitchen (which is a part of the house) and in order to clean the kitchen, we have to clean the kitchen sink (which is a part of the kitchen).

For formalizing recursion, we have to utilize *recursive definitions*, which denote concepts that are actually defined in terms of themselves. However, to avoid paradoxes and

infinite loops, a sound recursive definition, must not define the solution of a problem exactly in terms of itself, but in a simpler version of itself. [Hof79].

Summation, for example, is the addition of a sequence of numbers. In mathematics it is written as $\sum_{i=1}^{n} i$, which means "add all numbers from one to n". While there exists a formula[1] which computes the result in one step, we can also apply a simple recursive definition for that task:

Definition 2.1

$$sum(n) = \begin{cases} 0 & \text{for } n = 0 \\ n + sum(n-1) & \text{else} \end{cases}$$

The solution of $sum(n)$ is $n + sum(n-1)$, in the next recursive step $sum(n-1)$ is $(n-1) + sum(n-2)$. As can be seen the parameter n will always become smaller and smaller, until the so called *base case* is reached: When n is equal to zero, $sum(0) = 0$. One can say that we have split the big problem of computing $sum(n)$ into n small problems. One might refer to that proceeding as *divide and conquer* approach [OHP07].

For illustrating how a recursive function is evaluated, we consider the sum function for $n = 4$:

```
sum(4) = 4 + sum(3)
       = 4 + 3 + sum(2)
       = 4 + 3 + 2 + sum(1)
       = 4 + 3 + 2 + 1 + sum(0)
       = 4 + 3 + 2 + 1 + 0
       = 4 + 3 + 2 + 1
       = 4 + 3 + 3
       = 4 + 6
       = 10
```

Repeated calls to a recursive function like in the example ($sum(4)$, $sum(3)$, $sum(2)$, ...) are also called *incarnations* of the function. In the example the chain of incarnations *terminates*, as the base case $n = 0$ is reached.

[1] $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Termination is an important property of recursive functions. If a recursive function does not terminate, it can be considered as *undefined* and therefore, not computable. In case of a non-terminating recursive function in computer programs the computer will just keep on working until it is running out of memory or the user interrupts processing.

While recursive definitions are not prevalent in imperative programming, which operates mainly with *for-* and *while-loops* to express reiterated computations, recursive definitions play a seminal role in functional programming languages like *HASKELL*, *ML* and *F#*. They allow for concise and elegant function definitions, which are often very close to the underlying mathematical theory. [Pep02]

Considering the importance of recursion for functional programming, it is not surprising that recursion and recursive functions have been thoroughly researched. Different types of recursion are distinguished. In the following section we introduce those types.

2.1.1. Types of Recursion

There are four different types of recursion possible in a single function definition. However, there is an additional type which occurs when recursive functions may call other recursive functions. The type of a recursion is determined by the pattern of how recursive calls are used in function definitions. The types are [Pep02]:

Tail Recursion This is the most restricted type of recursion. For every branch of the function there must not be more than one recursive call, and if there is a recursive call in a branch, it must be the outermost operation. The recursive variant of the *modulo* function is a typical example for tail recursion. It is defined as:
Definition 2.2

$$modulo(n, m) = \begin{cases} n & \text{for } n < m \\ modulo(n - m, m) & \text{else} \end{cases}$$

As can be seen there are two branches. Provided, n is less than m no more recursive calls have to be done. The solution is n. For the $n \geq m$ branch, the result is a recursive call to the modulo function itself, where the new n is reduced by m and the value for m remains the same. We can see why this function is

a valid definition. For each recursive call n gets smaller and smaller, while m always remains the same. Eventually, n will be less than m.

Linear Recursion Functions which are linear recursive, are often used as the typical examples of recursion. Recall the definition of the sum function (def 2.1). One of the constraints of tail recursion still holds: For each branch there may be at most one recursive call. However, the recursive call does not need to be the outermost operation. For the sum function, the branch containing the recursive call is: $n + sum(n-1)$. For each recursive call, the whole term increases in size. This is shown by the exemplary evaluation of an instance of the sum function on page 4.

Tree Recursion For this type of recursion, the limit of a maximum amount of one recursive call per branch is lifted. This results in a tree-like growth of the whole term. A typical representative is the *Fibonacci* function (*fib*):
Definition 2.3

$$fib(n) = \begin{cases} 0 & \text{for } n = 0 \\ 1 & \text{for } n = 1 \\ fib(n-1) + fib(n-2) & \text{else} \end{cases}$$

The Fibonacci function defines a sequence of numbers where each number in the sequence is the addition of its two predecessors. While this behavior can be efficiently implemented in an iterative way or by introducing a counter variable, the recursive function is quite inefficient and has high demand for processing power and memory.

The function has two base cases. For input 0 the result is 0. For input 1 the result 1. For all inputs greater than 1 the result is the addition of two recursive calls: $fib(n-1) + fib(n-2)$. We can see why there have to be two base cases. $fib(2)$ entails two recursive calls: $fib(0)$ and $fib(1)$. Both of these calls must result in a single natural number. Otherwise there would be recursive calls with negative numbers as input.

Nested Recursion Nested recursion allows for having recursive calls as parameters in recursive calls. The most popular nested recursive function probably is the

Ackermann function. Its result grows extremely fast with bigger inputs. It is defined as:

Definition 2.4

$$ack(m,n) = \begin{cases} n+1 & \text{for } m = 0 \\ ack(m-1,1) & \text{for } m > 0 \wedge n = 0 \\ ack(m-1, ack(m, n-1)) & \text{for } m > 0 \wedge n \neq 0 \end{cases}$$

While some nested recursive functions can also be expressed by other recursion types, the *Ackermann* function is an example for a computable function, which cannot be realized by another recursion type. It disproved the belief according to which each nested recursion could be simplified to other recursion types.[2]

When considering more than one recursive function at a time, *Mutual Recursion* is possible. It happens, when a recursive function f calls another recursive function g, which calls f once again. An example function is a realization of functions *odd* and *even*:

Definition 2.5

$$even(n) = \begin{cases} true & \text{for } n = 0 \\ \textbf{odd}(n-1) & \text{else} \end{cases}$$

$$odd(n) = \begin{cases} false & \text{for } n = 0 \\ \textbf{even}(n-1) & \text{else} \end{cases}$$

In theory there can be any number of recursive functions involved.

It is noticeable that recursion types do not depend on the number of parameters of recursive functions at all, but only on the pattern of usage of recursive functions.

This section introduced the concepts of recursion and recursive functions. As our method - working on recursive functions - can be classified as a *Programming by Analogy* approach, this topic will be addressed in the next section.

[2] `http://mathworld.wolfram.com/AckermannFunction.html`, last access Sunday 22nd March, 2015

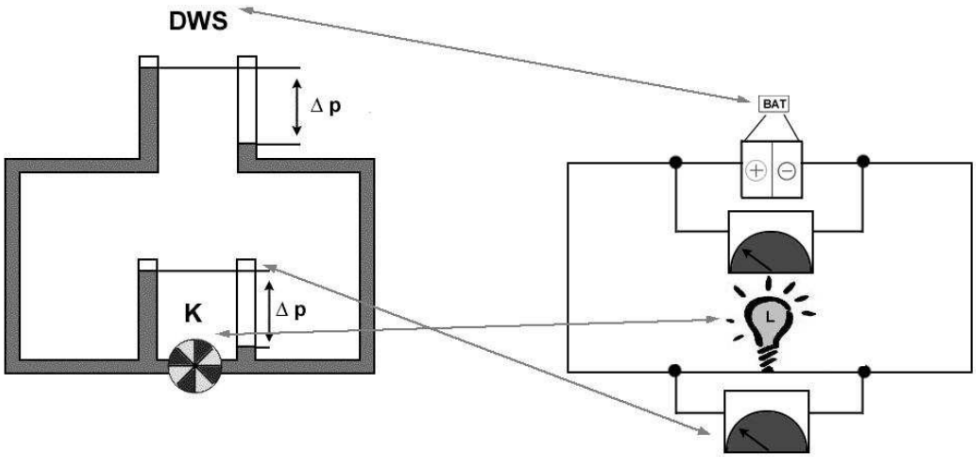


Figure 2.1.: Mapping between two domains: water flow and electricity [WKS08]

2.2. Analogy

According to Gentner in the *MIT encyclopedia of the cognitive sciences* [WK01] analogy shows in two manners:

1. the similarity in which the same relations hold between different domains or systems,
2. inference that if two things agree in certain respects they probably agree in others.

Those two meanings are interconnected. Analogy is thought to be an essential mechanism in cognitive sciences. Analogy must be taken into account when talking about *learning* and transfer across domains. Successful analogical reasoning can result in new *mental models* which - in turn - can lead to the understanding of new domains: Gentner has shown that people may use the concepts of water flow to reason about electricity [Gen83]. Wiese [WKS08] investigated whether the whole structure of a suitable base domain is transferred to a target domain or rather only those sub-structures required to solve an actual problem. She also referred to water flow and electricity. Figure 2.1 illustrates a possible mapping between the two domains: The waterwheel in the water flow acts a resistor, just like the light bulb in the electrical circuit. The decrease in water height corresponds to the voltage drop in the circuit.

Furthermore, Gentner has shown that analogy is a key element in *Problem Solving* and pivotal to creativity. It also occurs in the interaction between people as part of communication. In his book about general reasoning, Polya [Pol14] emphasizes the importance of analogy as well:

> *Analogy pervades all our thinking, our everyday speech and our trivial conclusions as well as artistic ways of expression and the highest scientific achievements. Analogy is used on very different levels. People often use vague, ambiguous, incomplete, or incompletely clarified analogies, but analogy may reach the level of mathematical precision. All sorts of analogy may play a role in the discovery of the solution and so we should not neglect any sort.*

The central topic in analogy research is on the *mapping* process. It is mapping, which allows us to understand one situation in terms of another [WK01]. However, mapping is only a part of the whole story. Therefore, the next section will briefly introduce some analogy theory.

2.2.1. Analogical Theory

When talking about analogy and how analogy works, the term must first be distinguished from other forms of inference of information. As prerequisite for analogy, Gentner makes the following assumptions:

1. Domains or situations which act as sources and targets of analogy, are considered to be object systems, which consist of:
   - *objects*, which can be real-world entities, parts of those entities or a collection of instances of that entity.
   - *object-attributes*, which capture properties of objects and always refer to the object.
   - *relations*, those are relations between objects. They express relationships between objects.

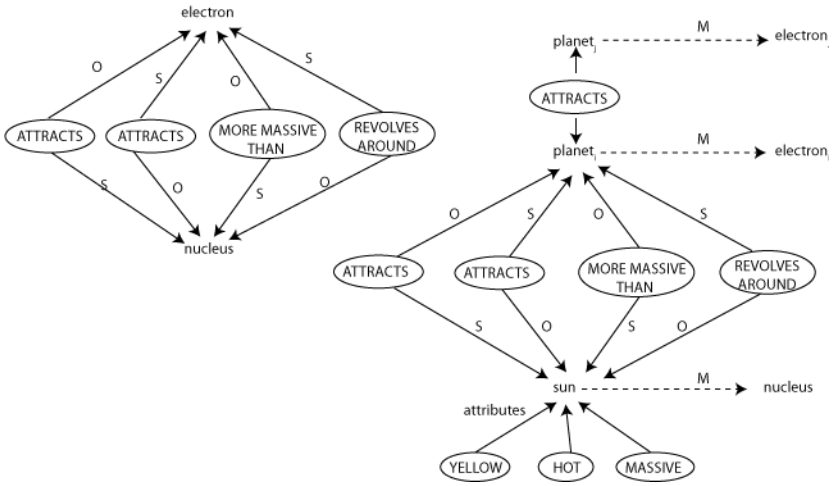


Figure 2.2.: The Rutherford Analogy depicted in [Gen83][3]

2. Knowledge within a domain is captured by a propositional network of nodes and predicates. Predicates are object-attributes and relations. They express propositions about concepts. Figure 2.2 depicts such a network which is known as the Rutherford analogy. It shows that an atom has a similar structure compared to a solar system.

3. Object-attributes as well as relations are are realized as predicates (labeled ellipses). Attributes are unary predicates. They express that the sun is yellow, hot and massive. Relations are binary or have an even higher cardinality. In the figure there are only binary relations. Each relation is connected to two objects, one acts as a subject (s) and the other as object (o), e.g. the sun is more massive than a planet.

   There is another type of relation. There can be relations, which do not just express the relationships between objects (*first-order*) but also between relations (*second- and higher-order*).

4. The representation of knowledge is supposed to be similar to the natural way people express knowledge, e.g. binary relations should not be reduced to attributes. It might be logically feasible, but neither intuitive nor natural.

[3]Alternative visualization acquired from: `http://www.sussex.ac.uk/Users/christ/crs/gc/rutherford-analogy-gentner-schematic.png`, last access Sunday 22nd March, 2015

Based on these foundations, different domains can be compared each other. However, only a certain way of comparison can be called analogy. When mapping from one domain into another, we can differentiate between the number of object attributes that are mapped and the number of relations that are mapped.

If many object attributes as well as many relations can be transferred from the source into the target domain, Gentner refers to ***literal similarity***. Considering water flow and electric circuits (fig. 2.1) it is obvious that not many attributes can be mapped. A water reservoir and a battery do not share many inherent properties, however they both serve as subject for powering water wheels or light bulbs.

The opposite of literal similarity is called ***anomaly***. The condition for an anomaly is that only a few attributes as well as a few relations can be mapped. It occurs when we try to map from and to domains, which have nothing in common.

When only a few object attributes can mapped to a target domain, but a lot of relations can be mapped, two cases must be distinguished. If both domains did not have a lot of object attributes in the first place, it is called ***abstraction***. An example for that would be "*The atom is a central force system*". Finally, an ***analogy*** occurs when at least one of the domains does actually have attributed objects, but mapping mainly relies on relations: "*The atom is like our solar system*". An important aspect in Gentner's Theoretical Framework for Analogy [Gen83] is the so called *systematicity principle*. It states that mappings of large and interconnected structures of relations are to preferred. The same amount of individual relations which are separately mapped, have less chance of successful analogical transfer.

Although mapping is a central element and the focus of research, the whole process of inferring knowledge from a base or source domain to a target domain involves more than mere mapping. Mapping is just a sub-process. As our approach will involve other components of the analogical process as well, we briefly introduce those subprocesses as well[WK01]:

Representation This mechanism describes the possibility to alter one of the domains or both to allow for a better matching and mapping. However, in our opinion it can also be seen as enabling mechanism for analogy. After all domains must be represented in suitable mental models which can actually be compared.

Retrieval In a realistic scenario, when people think about a target domain, the do not immediately know which domain could act as a promising source domain. Retrieval is the process of searching for promising source domains.

Evaluation When the mapping from source to target domain has been done, the result is evaluated. It is checked for overall soundness and actual usefulness.

While we described analogy as a general concept above, we will now address *Programming by Analogy*.

2.2.2. Programming by Analogy

Considering the typical tasks a programmer has to fulfill, it immediately comes into one's mind, that analogy probably plays a large role. In his approach on Programming by Analogy, Dershowitz [Der86] states:

> [...] *a small fraction of a programmer's time is typically devoted to the creation of original programs ex nihilo. Rather, most of his effort is normally directed at debugging incorrect programs, adapting known techniques* [...] *and abstracting ideas of general applicability into "subroutines".*

While we do not want to introduce the approach of Dershowitz in detail, some the ideas presented are valuable for our approach. Dershowitz demands clearly *structured program specifications*, which include input and output behavior, as well as *assertions*, which are invariants, i.e. properties that always have to hold.

Together with other specialized *statements* types, this allows for the mentioned debugging of programs. When the programmer has implemented a program which fulfills its specification, the created program can be **modified** to implement other program specifications.

The key idea is depicted in figure 2.3. The boxes with blue background denote elements which are already present, i.e. there is a program specification p_1 and an implementation fulfilling the implementation. The task is to create a new program fulfilling a certain specification. By analogical reasoning a mapping from p_1 to the new program specification is done. This mapping provides insights about used operators or variables,

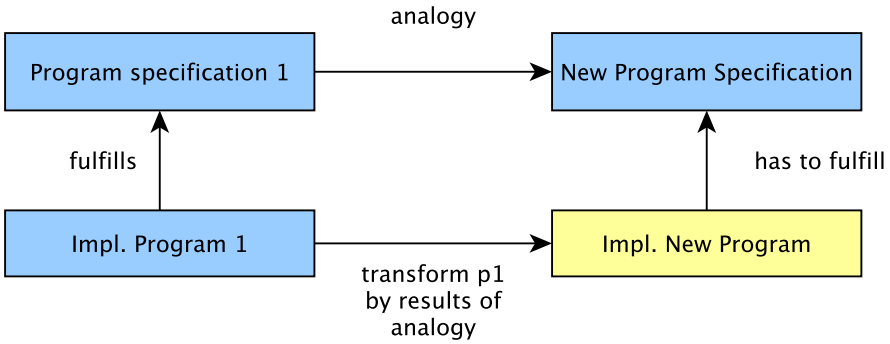


Figure 2.3.: An existing implementation of a specified program can be used to find a program which fulfills a new specification

e.g. we know that operator x in the program specification p_1 can be mapped to operator y in the new specification. In a last step the existing implementation is altered by applying those analogical coherences.

Dershowitz describes other processes as well. **Abstraction** can be done when there are at least two complete programs (complete meaning specification and corresponding implementation). Abstraction results in an abstract program, which, captures the strategy or the essence of the strategy which the programs use. In his paper, Dershowitz conducts abstraction on programs which both use the idea binary search. Of course, the programs abstracted have to follow a common strategy in the first place to achieve feasible results.

Finally, Dershowitz introduces **instantiation**. Given an abstracted program and a program specification, the abstract program can be instantiated with the specific operators and terms in the specification to derive an instance of the abstract program, which fulfills the specification.

More recently, Repenning and Perrone [RP01] suggested an approach for enriching *Programming by Examples* techniques with analogies. It is called *Programming by Analogous Examples*. Programming by Examples is a paradigm which allows end-users of applications to program by executing a task manually. The actions of the user's are recorded and turned into a program. This allows for more automation in the users'

workflows. The recording of macros or the Flash Fill[4] feature in Microsoft Excel are examples of Programming by Examples.

While Programming by Examples is useful, it lacks general applicability, as hardly any program reuse is possible, unless the user is able to modify the underlying code. When repeating tasks slightly differ from each other or are corresponding to different domains, the user must record actions manually again. The authors try to address this deficit with analogical reasoning. They provide an example which is realized with *AgentSheets*[5], which is a tool for creating visual games and simulations.

In the example there have been rail tracks and trains which moved on those tracks. AgentSheets utilizes a syntactic rule base to determine the next state of the simulation. Users must explicitly show situations and their solution to define behavior. In the example, the behavior of trains on tracks had already been implemented. They wanted to add cars and streets. Instead of defining all situations for cars and streets again, they added a possibility to model analogy explicitly, to express that cars move on streets just like trains move on tracks. Their approach allows users to reuse programs which were created by Programming by Example techniques by providing a way of defining analogies between two domains explicitly.

Abstraction in Proportional Analogies

The approach, which is quite close to ours, relies on abstraction. As Weller and Schmid in [WS06] have shown, abstraction allows for methods for solving *proportional analogies*. Those are well-known task of the form $A : B :: C : D$, literally "A is to B like C is to D". A, B and C are given and D is unknown.

Those tasks often emerge in the string domain, *abc : abd :: ijk : ?* is an instance of those problems. For application in the string domain there exists the *Copycat* system, which is described in [HM+94]. It's aim is to solve proportional analogies in a psychologically plausible way with statistical methods. Therefore, it is non-deterministic. The authors of [WS06], however, argue that statistical processes cannot explain human analogy solving.

[4] `http://research.microsoft.com/en-us/um/people/sumitg/flashfill.html`, last access Sunday 22nd March, 2015

[5] `http://www.agentsheets.com`, last access Sunday 22nd March, 2015

They propose the usage of *anti-unification* which is introduced in detail in the next section. Simply put, anti-unification takes two terms or structures and generates a common, generalized structure. The common substructure, which occurs in both terms is preserved. Differences are abstracted and replaced by variables. The result also encompasses the substitutions for the variables introduced, i.e. the two original terms can be restored. This allows for a matching process which does not only map objects to each other, but which can also consider the common structure of the terms.

The idea presented for solving proportional analogies is to anti-unify A and C.[6] As a result we obtain the common structure of A and C, which is a term that captures differences between the two terms in variables. As a byproduct there is also τ_1, which holds the information, which variables in the generalized term have to substituted by which subterms to obtain the original term A. There is τ_2 as well which captures the information about variable replacements to restore term C. τ_1 can be inversed to obtain τ_1^{-i}. The resulting substitutions are no longer mapped from variable to term, instead they tell us which concrete terms are substituted by variables.

To solve the proportional analogy τ_1^{-i} (a mapping from terms to variables) is applied to B. The result is the common structure between B and D. Applying the substitutions of τ_2 (a mapping from variables to terms) to that common structure results in D.

Schmid et al. also proposed the usage of anti-unification for other types of analogies [SBW03]:

Predictive Analogies Predictive analogies which try to draw conclusions about a second domain based on knowledge about the source domain. The Rutherford analogy depicted in fig 2.2 is an example of a predictive analogy.

Analogical Problem Solving This approach is usually occurs within a certain domain. The idea is to transfer a solution from a known problem to a structurally similar problem. As Anderson and Thompson have described [AT89], analogies are indeed important for programming. They agree with Dershowitz, who said that programmers seldom have to invent a completely new program. Instead they think about programs they have already written and try to use that knowledge.

[6] This is only an informal description to illustrate the idea. *A, B, C* and *D* are actually supposed to be regular tree grammars.

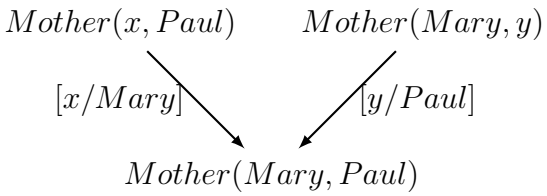


Figure 2.4.: Simple unification of a predicate expression

Our approach can be classified as analogical problem solving. It will be described in detail in chapter 3. The approach heavily relies on abstraction by anti-unification as well. Because of this, we will cover the topic of anti-unification first.

2.3. Unification and Anti-Unification

In this section we will cover the topics of unification and anti-unification. We will also introduce a library for second-order anti-unification, which has been developed at the Johannes Kepler University Linz.

2.3.1. Unification

For understanding the concept of anti-unification, it makes sense to attend unification first. It is the dual operation to anti-unification, well-researched and constitutes the basis for *resolution*, the main technique for the evaluation of formulas in the logic programming language Prolog.

Unification is an operation, which takes two or more terms and computes a new, single term from them [Hog84, chapter 1]. The computed term must be obtainable by renaming variables or by replacing variables with other terms. Figure 2.4 shows an exemplary unification: The terms unified are $Mother(x, Paul)$ and $Mother(Mary, y)$. In this example $Mother$ is a binary predicate. $Mary$ and $Paul$ are constants, x and y are variables. The unification result $Mother(Mary, Paul)$ is obtained, by replacing x by $Mary$ and y with $Paul$. It is important to note that the the whole substitution, i.e. $[x := Mary, y := Paul]$, is applied to both terms. However, in our case, where terms do not share any common variable, this is not immediately visible. Another important aspect is that this is a merely syntactically process. Although the term

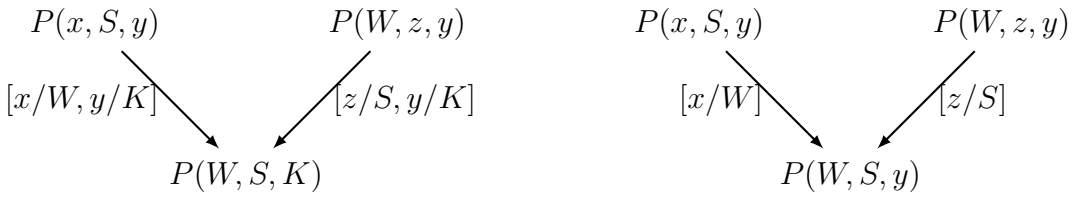


Figure 2.5.: A valid unifier (left) and the most general unifier (right)

$Mother(Mary, Paul)$ probably shall express that Mary is the mother of Paul, we do not know that.

Formally, the unification of a set of terms is done by applying the a *unifier* called Θ to each term. A unifier is a set of substitutions of the form $\Theta = \{v_1 := t_1, \ldots, v_n := t_n\}$, where $v_i := t_i$ means "substitute v_i by t_i". The unifier must constituted in such a way that for each application of the unifier to any expression E_i the same term results. $E\Theta$ denotes the application of the substitutions in Θ to E. Θ is a unifier for a set of expressions, if $E_1\Theta = \cdots = E_n\Theta$.

An important concept is the *most general unifier* called *mgu*. It denotes the minimal set of substitutions necessary to unify the terms in question. For each unification problem there is one most general unifier, if the terms involved can be unified.[7] Figure 2.5 illustrates the concept. The unifier $\Theta_v = \{x := W, y := K, z := S\}$ is valid, applied to both terms the result will be $P(W, S, K)$. However the substitutions are more restrictive than necessary: The most general unifier $\Theta_{mgu} = \{x := W, z := S\}$ suffices to unify both expressions, resulting in $P(W, S, y)$. Formally, a unifier Θ is the most general unifier, if for all other unifiers Θ_i there exists an extra substitution σ, such that $\Theta_i = \Theta\sigma$.

Summarizing, one can say that unification is a process which takes terms and transforms them into one single term by specializing the original terms, e.g. by constraining variables to certain constant values. A lot of research on unification has been conducted. There are approaches to combine unification with *equational theories* [BS01] and methods for *higher-order* unification [SG89].[8] In this thesis, however, unification is

[7] In case of first-order unification and not considering the names of introduced variables, as the name of a new variable does not make a difference semantically.

[8] The advanced topics on anti-unification theory given in sections 2.3.2 and 2.3.2, may allow getting a grasp on the corresponding approaches for unification.

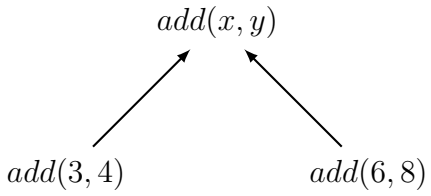


Figure 2.6.: Simple anti-unification of a binary function term

mainly important for its role as dual concept to anti-unification, which will be presented in the following subsection.

2.3.2. Anti-Unification

In contrary to unification, the main idea of anti-unification is to *generalize* terms, i.e. compute a single, common term from two or more terms. Subterms, which are the same for all the terms, are preserved. Subterms which differ are generalized, e.g. by introducing a new variable for those places. The idea was first introduced by Plotkin in 1970 [Plo70], however in the same year Reynolds published a related work [Rey70].

Figure 2.6 illustrates a so called *syntactical* or *first-order* anti-unification. The two terms t_1 and t_2 which are anti-unified are $add(3, 4)$ and $add(6, 8)$[9]. *add* is a function symbol and the numbers are constants. Anti-unification has to find a generalized term, which allows us to reconstruct both original terms by instantiating the variables introduced. A possible solution is term $add(x, y)$, where x and y are variable symbols. By computing the generalized term, two substitutions have been generated implicitly: $\sigma_1 = \{x := 3, y := 4\}$ and $\sigma_2 = \{x := 6, y := 8\}$, which - when applied to the generalized term - will restore the original terms. In his bachelor thesis, Weller [Wel05] presents an algorithm for syntactical anti-unification based Reynolds:

```
function au(x,y)
  if  x = y
    x
  else if  x = f(x_1, ..., x_n)
    and  y = f(y_1, ..., y_n)
```

[9]For examples of anti-unification we use *prefix* notation, as the terms we anti-unified in our approach also had to be in prefix notation.

$$f(au(x_1, y_1, \ldots, au(x_n, y_n)))$$

else

ϕ

ϕ is a fresh variable introduced. When the ϕ is introduced a new rule is added to each σ_i:

$$\sigma_1 = \sigma_1 \cup \{\phi := x\}$$

$$\sigma_2 = \sigma_2 \cup \{\phi := y\}$$

The algorithm immediately unveils a big shortcoming. Considering the anti-unification of $g(a, b, c)$ and $f(a, b, c)$, the result will be a single variable x. That means, that there the algorithm does not recognize any of the common structure in the terms, although the functions arguments are the same for both terms. However, it is also evident, that anti-unification of two terms will always produce a result. In case terms do not share a common structure or algorithmic restrictions prevent a more precise result, each terms can be restored from a single variable.

In spite of its inherent deficits, first-order anti-unification is successfully utilized in some domains. Bulychev and Minea investigated a system for detecting duplicate code in Java and Python programs [BM09]. As duplicate code unnecessarily bloats up programs and increases the probability of occurring bugs, it's a reasonable approach.

Just like for unification, where the most general unifier usually is of most interest, for anti-unification we are interested in the *least general generalizations* (*lgg*) [Plo71]. Since anti-unification is an operation to identify common structures, the fewer subterms are abstracted by variables, the more meaningful the result is. If the generalized term is the *lgg*, we know that as much information as possible has been preserved. There has been research on more advanced anti-unification techniques. In the following, we introduce two important concepts.

E-Generalization

A successful approach to increase the power of anti-unification is the *Anti–Unification Modulo Equational Theory* (also called *E-Generalization*), developed by Heinz in 1995. Burghardt and Heinz released an implementation of the algorithm in the following year [BH14]. The idea of *E-Generalization* is to overcome the mere syntactic perspective

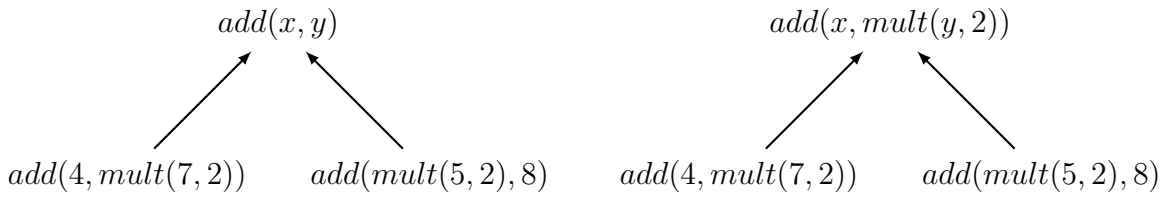


Figure 2.7.: Syntactic anti-unification (left) and one possible solution for E-Generalization (right)

when dealing with anti-unification, by allowing equality transformations on the term. Which transformation conserve equality must be constituted by a theory of equations. Figure 2.7 show an example. Our equational theory consists of two equations:

$$add(n, m) = add(m, n),$$

$$mult(n, m) = mult(m, n)$$

Given that *add* and *mult* denote addition and multiplication, the equations define commutativity for those two operations. On the left side the figure illustrates syntactic unification. On the top level for both terms there is the *add* function, i.e. the generalized term will also use that function. Now the parameters of the functions are anti-unified pairwise. For the pair 4 and $mult(5, 2)$ a new variable has to be introduced. Analogously for the pair $mult(7, 2)$ and 8 no common structure can be identified. Therefore, the resulting generalized term does not hold a lot of information. It is $add(x, y)$, with:

$$\sigma_1 = \{x := 4, y := mult(7, 2)\}$$

$$\sigma_2 = \{x := mult(5, 2), y := 8\}$$

On the right side of the figure there is one of the possible results of *E-Generalization*. Unlike the syntactic anti-unification, E-Generalization does not operate directly on terms. Based on the terms in question and the equational theory, *regular tree grammars* are created, which are anti-unified afterwards. In [Bur05], Burghardt further elaborated the idea by using standard grammar algorithms and confirmed the feasibility of the approach by providing concrete applications. One of the was the generation of candidate lemmas in inductive proofs.

As can be seen in the figure the E-Generalization is able to detect more of the commonalities of the terms. One of the solutions is $add(x, mult(y, 2))$, with:

$$\sigma_1 = \{x := 4, y := 7\}$$

$$\sigma_2 = \{x := 8, y := 5\}$$

As can be seen, enriching anti-unification with the ability to transform terms according to an adequate equational theory can improve the resulting generalizations. For syntactic anti-unification the information that both terms compute a multiplication of a number by two, has been lost.

As has been stated, there are multiple possible solutions, as the result is not a term, but also a regular tree grammar, i.e. all possible solutions can be generated by that grammar. Of course, the notion of least general generalizations is still applicable and desired.

Higher-Order Anti-Unification

While E-Generalization is a useful approach, especially when doing the anti-unification in a domain, where a meaningful equational theory can be established, it cannot completely overcome the flaws of a first-order approach.

In his PhD thesis [Has95, chapter 5] Hasker developed an approach for doing second-order anti-unification with combinator terms. There have been different approaches to higher-order anti-unification. Based on Hasker's findings, Wagner [Wag02] developed a computable, restricted higher-order anti-unification algorithm and used it for analogical programming in context of the IPAL project. Krumnack et al. [KSGK07] developed a restricted higher-order anti-unification algorithm for making analogies using the *Heuristic-Driven Theory Projection* framework. Baumgartner and Kutsia [BK14b] developed a second-order anti-unification technique. They also created an implementation of their algorithm. It is their implementation, that we use for our approach. For that reason, we will stick to their notation when presenting the general idea of higher-order anti-unification.

The second-order anti-unification allows for more specific generalizations, when two terms actually do have similar structures, but differ in their head symbol or the similarities occur in different contexts.

Considering the terms $add(4, 3)$ and $mult(4, 3)$, one can see that they have the same parameters, however one of them is an instance of the *add* function, the other an instance

of the *mult* function. In first-order anti-unification the only possible generalization would be a single variable x with

$$\sigma_1 = \{x := add(4,3)\}$$

$$\sigma_2 = \{x := mult(4,3)\}.$$

In the higher-order case, however, a lot more information can be preserved. It allows the introduction of higher-order variables, which represent contexts. The higher-order solution is the term $X(4,3)$, with X being a higher-order variable and

$$\sigma_1 = \{X := add(\circ)\}$$

$$\sigma_2 = \{X := mult(\circ)\}.$$

The $\circ$ symbol is called *hole*. X is also called a context variable or just context. Contexts are terms which have one $\circ$. They can be applied to other arguments. When applied to an argument, the hole in the context is replaced by the argument, i.e. $X(4,3)\sigma_2 = mult(\circ)(4,3)$ is evaluated to be $mult(4,3)$.

The terms $add(1, mult(2,3))$ and $mult(2,3)$ share the $mult(2,3)$ term, but at different depths. Again, first-order anti-unification cannot find a good solution and goes with x as result. Second-order anti-unification will find $X(mult(2,3))$ with

$$\sigma_1 = \{X := add(1,\circ)\}$$

$$\sigma_2 = \{X := \circ\}.$$

Applying the σ_i to restore terms, for σ_1 we have

$$X(mult(2,3))\sigma_1 = add(1,\circ)(mult(2,3) = add(1, mult(2,3))$$

and for σ_2:

$$X(mult(2,3))\sigma_2 = \circ(mult(2,3)) = mult(2,3).$$

Higher-order anti-unification also allows for swapping of arguments.[10] The hole operator can also be used for swapping arguments. For $t_1 = add(1,3)$ and $t_2 = mult(3,1)$, the generalization will be $X(1)$ with

$$\sigma_1 = \{X := add(\circ,3)\}$$

[10] Hasker uses so called projections π_1 and π_2 [Has95] which allow for reordering of function arguments. A projection π_i is just a function which chooses the i^{th} argument in a list of arguments.

$$\sigma_2 = \{X := mult(3, \circ)\}.$$

There is also the case that both terms have the same head symbol, e.g. $t_1 = add(1, 3)$ and $t_2 = add(3, 1)$. Those are anti-unified to $add(x, 1, y)$, where x and y are first-order variables. Here, they are also called hedge variables. The corresponding substitutions are:

$$\sigma_1 = \{x := \epsilon, y := 3\}$$

$$\sigma_2 = \{x := 3, y := \epsilon\}.$$

The ϵ denotes a removal or deletion of an argument. Naturally higher-order anti-unification is capable of simply substituting a subterm as well: $t_1 = add(1, 3), t2 = add(1, add(2, 2))$ with generalization $add(1, x)$ and

$$\sigma_1 = \{x := 3\}$$

$$\sigma_2 = \{x := add(1, 1)\}.$$

Summarizing, one can say that higher-order anti-unification:

- allows for deletion, insertion and replacing of terms,
- generalizes terms even which have a different head symbol and
- can leverage common structures, even if those are not on the same depth.

The examples we gave represent second-order anti-unification. The context variables are functions, which can be applied to a context and transform it into a term. In theory there is no limit for higher-order anti-unification. Third-order variables could be functions, which create functions which create terms. The next section will be dedicated to the particularities of an implementation of second-order anti-unification.

2.3.3. An Anti-Unification Library

As mentioned, the anti-unification introduced above is called *Unranked Second-Order Anti-Unification*. It is part of a suite of anti-unification algorithms which have been implemented in the Java programming language.[11] The algorithms are specialized

[11] The library is available online: `http://www.risc.jku.at/projects/stout/`, last access Sunday 22nd March, 2015. The different techniques can also be tried out online as well.

on different use cases, e.g. there is a first-order algorithm as well as a second-order algorithm for anti-unifying lambda terms. All have been described in [BK14a].

The unranked second-order approach [BK14b] was made for anti-unifying two so called *hedges*. A hedge is an arbitrary number of terms of the form $(t_1, \ldots, t_n)$, where terms can be function expressions, hedge (first-order) variables or context (second-order) variables. The reason the approach is called unranked is that there is no fixed limit for terms in a hedge and that function symbols occurring in terms also do not have a fixed arity. Although we want to anti-unify terms, instead of hedges, we can still use the approach. In our case the two input hedges will always contain only one term (*singleton* hedges), which represent a function expression.

The authors introduce the notion of *rigid* generalizations, which are subject to some constraints. Those constraints ensure that the computed generalization will be a least general generalization. The constrains are:

- σ_1 and σ_2 must only contain singleton contexts, i.e. context variables may only have one hole operator.
- In the generalization there may neither be two hedge variables next to each other, nor *vertical* chains of variable, vertical meaning nested, e.g. an anti-instance $X(Y)$, where X and Y are context variables, is forbidden.
- A context variable must not contain a hedge variable as first or last argument, i.e. $X(x, \ldots)$ and $X(\ldots, x)$ are forbidden.

The Java implementation of the algorithm is convenient to use. It also supports some properties of E-Generalization. Function symbols can be declared to be commutative and/or associative. However, we did no use those options. It is a nice feature, that the inputs can be parsed from strings. The algorithm builds its own internal representation of the underlying hedge.

An example shows what the output of the algorithm looks like. The results consists of three artifacts. The input terms (hedges) are

$$t_1 = if(eq(n, 0), 0, add(n, if(eq(subtract(n, 1), 0), 0)))$$

$$t_2 = if(eq(m, 0), 1, mult(m, if(eq(subtract(m, 1), 0), 1)))$$

The first step in the algorithm is the computation of an admissible alignment, which matches elements of the first term to elements of the second. It has the following form:

```
if<1, 1> eq<1.1, 1.1> 0<1.1.2, 1.1.2> if<1.3.2, 1.3.2>
eq<1.3.2.1, 1.3.2.1> subtract<1.3.2.1.1, 1.3.2.1.1>...
```

Each element in the alignment denotes an actual element present in both terms (e.g. if). In angle brackets the position of that subterm is given for both input terms, as the position will not always the same for both terms. The second if has the position 1.3.2, i.e. it is the second argument in the third argument of the whole term.

The algorithm also computes the generalized term, which is also called generalization or anti-instance:

```
(if(eq(#14, 0), #16, $4(if(eq(subtract(#14, 1), 0), #16))))
```

The introduced variables are referred to by a number: first-order (hedge) variables are denoted by the # symbol. Second-order (context) variables are denoted by $. From the generalized term, the original terms can be reconstructed, as the algorithm also computes the so called *sigma store*, which has the following syntax:

```
#4: () =^= (); $4: add(n, @) =^= mult(m, @);
#14: n =^= m;  $14: @ =^= @;
#16: 0 =^= 1;  $16: @ =^= @
```

In the sigma store, for a given variable name (e.g. 4), substitutions of both types of variables (hedge and context) are given, although most of the time one of them does not occur in the generalized term. On the left side of `=^=` are the substitutions for restoring the first input term, on the right side are the substitutions for restoring the second input term. `()` is the deletion of an element (ϵ), `@` is the hole operator ($\circ$).

This concludes the chapter about the foundations and related topics of our approach. We introduced recursive functions, programming by analogy and second-order anti-unification. Those will be the basics for the next chapter, where we introduce our approach.

3 Approach

This chapter presents our approach for utilizing anti-unification and a matching algorithm to learn recursive schemes from examples.

First an overview about the general workflow for our approach is presented. Then we describe, how we can represent recursive functions in our approach and provide details about the recursive functions we investigated. After that, we will give details about the anti-unification and the further processing by heuristics. The procedure will be illustrated by two insightful examples.

3.1. General Procedure

Underlying Idea

Our work orientates itself on parts of the work of Schmid et al. [SMW98], which we will introduce briefly. The work is about a programming by analogy approach which covers all aspects of analogical reasoning: retrieval, mapping, adaption and generalization. We omit the retrieval aspect and use generalizations only while learning function definitions, rather than storing them in a permanent memory.

Figure 3.1 depicts the architecture of that programming by analogy approach. The programs which are learned, are so called *recursive program schemes* (RPS) which describe the structure of a recursive program and - ideally - are also executable. Those program

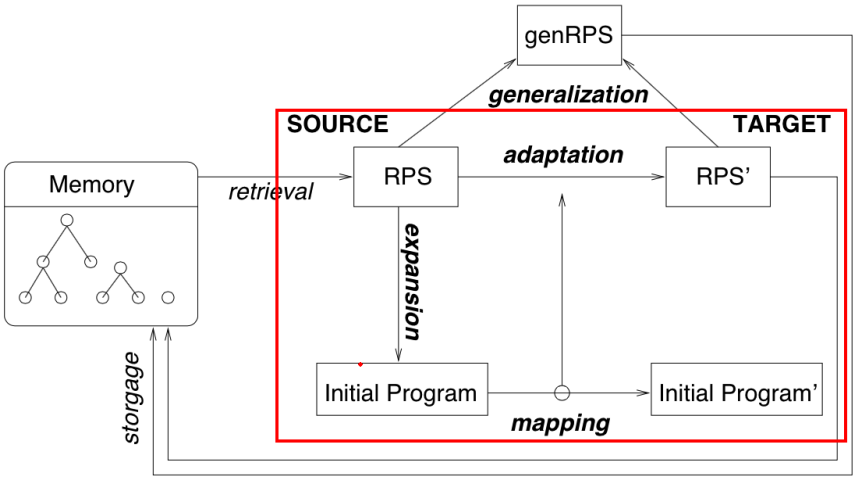


Figure 3.1.: Programming by analogy architecture by Schmid et al. in [SMW98]

schemes can be unfolded/expanded. The expansion can be seen as a handsimulation. An example given in the text is:

```
if eq0(x) then 0                              [else]
if eq0(pred(x)) then x                        [else]
if eq0(pred(pred(x))) then
   plus(x, pred(x))                           [else]
if eq0 (pred(pred(pred(x)))) then
   plus(x, plus(pred(x), pred(pred(x)))).
```

This is an initial program for the sum function up to $x = 3$. An unfolded recursive program scheme is called *initial program*. The idea is that for the function to be learned another initial program ($InitialProgram'$ in the figure) is provided. A suitable source RPS is then unfolded and analogical matching is done between two initial programs. The findings of the mapping process are then used to adapt the source RPS to form the target RPS. It is similar to the program modification scheme by Dershowitz, which we described in 2.2.2. However, here there are no program specifications and implementations, but rather recursive schemes and initial programs. The red box in the figure depicts the scope of our approach. We do not utilize the concept of a memory for generalized recursive program schemes.

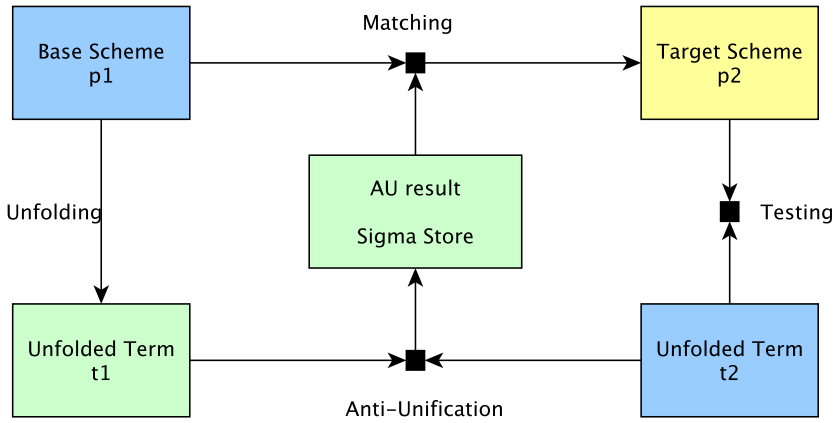


Figure 3.2.: General workflow for anti-unification and Matching of recursive functions

Our Approach

Our approach also works on recursive functions, however our representation of unfolded terms is different as can be seen in section 3.4.1. We also choose to use a different naming scheme: we call the base recursive function definition p_1 and the target function scheme p_2. They are the counterpart to recursive program schemes. Instead of initial programs we use unfolded recursion terms called t_1 and t_2. In this section only a coarse summary of the steps is given. All details will be presented in the following sections.

Figure 3.2 depicts the workflow for our approach. Light blue elements (the base/source scheme p_1 and the unfolded term t_2 of the target function) represent the components which are considered to be given initially. Green elements(the unfolded term t_1 and the anti-unification result) are immediate results of processing and constitute the foundation for the following steps. t_1 is obtained by unfolding p_1, and the anti-unification result is computed harnessing the anti-unification library as described in 2.3.3. Adapting p_1 by matching the results of the anti-unification to generate p_2 is the actual aim of processing.

After unfolding, t_1 and t_2 are anti-unified. The important results of the anti-unification are the generalized term t_g and the sigma store σ. As described, the generalized term captures all differences between base and target term in labeled variables, while maintaining those parts which are common in both terms. The sigma store allows us to reconstruct the base term t_1 as well as the target term t_2. For each labeled variable in

the generalized term the store σ holds two expressions. The first expression represents the actual content of t_1 in the position of the variable. The second expression is the actual content of t_2. If all variables in the generalized term are substituted by their corresponding parts in the sigma store, the original terms can be restored.

As the resulting sigma store is created by anti-unification of unfolded terms rather than recursive schemes, we have to find a way to use the information in the sigma store σ to adapt the source function p_1. We create a *reversed sigma store* σ_r. The actual sigma store computed by the anti-unification contains substitutions of the form

$$var : repl_1 \texttt{ =\^{}= } repl_2,$$

where $repl_1$ is the substitution for variable var in the first term, and $repl_2$ is the substitution for var in the second term. In the reversed sigma store we abstract from hedge and context variables. Instead we form substitutions of the form $repl_1 := repl_2$.

As the subterms on the left side of those substitutions origin from the unfolded base term t_1, usually some of those subterms are also part of the recursive base scheme p_1. Heuristics are utilized to match substitutions in σ_r to elements in p_1. The aim is to modify p_1 by the findings of the anti-unification of t_1 and t_2 to obtain the recursive target scheme p_2.

As an indicator of whether or not this transformation yields the correct result, we test the resulting recursive scheme p_2 by instantiating occurring variables[1] with the same parameters, which were used for the unfolding of the term. The instantiated scheme as well as the unfolded term are evaluated. If the outcome for both terms is the same there is typically a high probability that the recursive scheme p_2 is correct. The result of the evaluation of the unfolded term will always be correct.

As this is an automated approach, we had to develop a suitable way to represent functions and terms, which is the topic of the following section.

[1] The variables mentioned are not the first- or second-order variables of the anti-instace, but rather the regular variables in recursive schemes. For example, the n in the $sum(n)$ definition

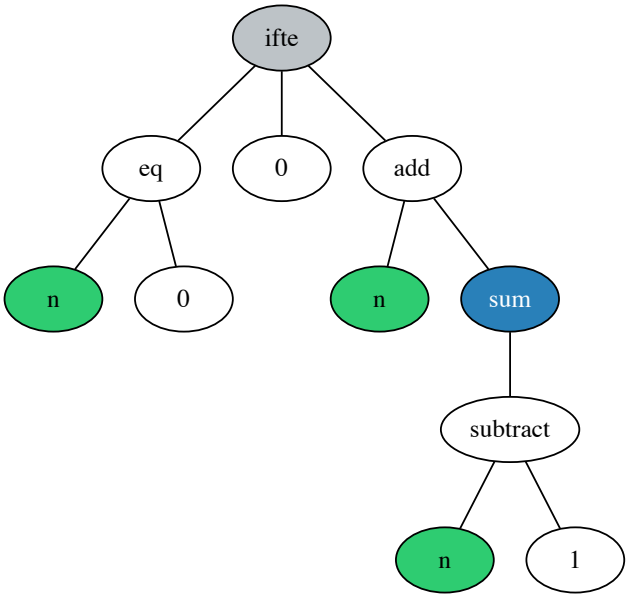


Figure 3.3.: The sum function in our tree representation

3.2. Function Representation

We generalize terms, which represent recursive functions. Therefore, functions must be represented in such a way, that they can be processed by the anti-unification library introduced in 2.3.3. It must also be flexible enough, so that parts of the function can be easily exchanged. The anti-unification library has some restrictions: it cannot process the usual operator symbols, like $+$, $-$, ... For creating the alignment which is the foundation for the anti-unification, terms must well formed and the scope of functions must be given explicitly by parentheses. While the library has not been explicitly created for algebraic terms, we can still use it in our approach.

Internally the recursive functions we investigate are realized as tree structures. Figure 3.3 shows an example of such a recursive tree for the recursive *summation* function. The actual anti-unification process works on textual representation of those trees. We create string representations by traversing the tree in *prefix* order. For example, the sum function given in the figure has the textual representation:

$$if(eq(n,0),0,add(n,sum(subtract(n,1))))$$

In the following we will briefly introduce all relevant building blocks for representing terms:

Constant A constant is the simplest term. We restricted the investigated functions to functions on natural numbers and lists of natural numbers. Lists, however, may also contain other lists. As the anti-unification cannot deal with a standard like $[1,2,3]$ for lists and we did not want to introduce an operator without fixed arity for representing lists, lists are transformed into *constructive definitions* for the textual representation. For example the textual representation of the list $[1,2,3]$ is $cons(1,cons(2,cons(3,empty)))$, where *empty* denotes the empty list.

Variable In our case, *variables* in recursive terms are only first-order, i.e. variables can hold constants (see previous description). They are denoted by a variable name. In our approach we do not assign variable names manually, they are assigned by the system when instantiating functions. Usually the variable names are n, m, k or l. Actually any name is possible. However, they are written in lowercase. A variable taken by itself is a valid term.

Operator Operators take an arbitrary number of inputs. However in the recursive functions we investigated, only three arities occur:
constant operators($empty()$), unary ($isempty(l)$) and binary ($add(x,2)$) operations. A operator expression with valid terms as operands is a valid term. We differentiate *comparative* operator, which produce truth values and regular operators, which produce constants.

Table 3.1 shows all operators we defined[2]. Their in- and output types are given as well as the actual operation and the "short name" which we utilize in our program. The operands are internally managed as a list. Hence, we allow to access operands by their index: if we consider the expression $x = add(3,4)$, the first operand 3 is given by $x[0]$ and the second by $x[1]$. That is why we define the semantics of operators by lambda expressions, which have one argument at most. The single argument of the lambda expression is the list of operands.

If-Then-Else As was stated in section 2.1, recursion works by reducing a computational problem every step. However, this process must not go on infinitely. Instead, we

[2] $\mathbb{X}$ must be a natural number or a list. The $++$ operator is the conc

Operation	Identifier	Semantics	Input	Output
Regular Operators				
Addition	*add*	$\lambda x.x[0] + x[1]$	$\mathbb{N} \times \mathbb{N}$	$\mathbb{N}$
Subtraction	*subtract*	$\lambda x.x[0] - x[1]$	$\mathbb{N} \times \mathbb{N}$	$\mathbb{N}$
Multiplication	*mult*	$\lambda x.x[0] * x[1]$	$\mathbb{N} \times \mathbb{N}$	$\mathbb{N}$
Empty	*empty*	$\lambda.[\,]$	-	$\mathbb{L}$
Head	*car*	$\lambda x.x[0][0]$	$\mathbb{L}$	$\mathbb{X}$
Tail	*cdr*	$\lambda x.x[0][1:n]$	$\mathbb{L}$	$\mathbb{L}$
Construct	*cons*	$\lambda x.x[0]{+}{+}x[1]$	$\mathbb{X} \times \mathbb{L}$	$\mathbb{L}$
Comparative Operators				
Equality Test	*eq*	$\lambda x.x[0] = x[1]$	$\mathbb{N} \times \mathbb{N}$	$\mathbb{B}$
Less Than	*less*	$\lambda x.x[0] < x[1]$	$\mathbb{N} \times \mathbb{N}$	$\mathbb{B}$
Greater Than	*greater*	$\lambda x.x[0] > x[1]$	$\mathbb{N} \times \mathbb{N}$	$\mathbb{B}$
Check For Empty List	*isempty*	$\lambda x.x[0] = [\,]$	$\mathbb{L}$	$\mathbb{B}$

Table 3.1.: Defined operators used in recursive functions

have to be able to check if execution has reached the base case. We realize this by allowing *if-then-else* constructs. We implemented the McCarthy Conditional [McC60] for modeling decisions: Our if-then-else has an *if-part*, a *then-part* and an optional *else-part*. The if-part must have a comparative operator as head element. When evaluated it produces a truth value. If the resulting value is *true*, the *then* part is evaluated and the *else* discarded. In case it is false, the *else* part is further processed instead.

An if-then-else statement is a valid term, if the if-part is a comparative operator, the then part is a valid term and if present, the else-part is a valid term.

Recursive Call A recursive calls are quite similar to operators. Internally, they also utilize a list to represent their parameters. Recursive calls, however, can be unfolded, which will be further explained in section 3.4.1. A recursive function is a valid function, if its name is defined and its parameters are valid terms.

The building blocks we introduced here, suffice to represent the recursive functions we want to investigate. In the next section we will give a brief of overview of those recursive functions.

Function	Recursion Type	Input	Output
$sum(n)$	*linear*	$\mathbb{N}$	$\mathbb{N}$
$nsum(n)$	*linear*	$\mathbb{N}$	$\mathbb{N}$
$fac(n)$	*linear*	$\mathbb{N}$	$\mathbb{N}$
$fib(n)$	*tree*	$\mathbb{N}$	$\mathbb{N}$
$countdown(n)$	*linear*	$\mathbb{N}$	$\mathbb{L}$
$last(l)$	*tail*	$\mathbb{L}$	$\mathbb{X}$
$unpack(l)$	*linear*	$\mathbb{L}$	$\mathbb{L}$
$addMult(n, m)$	*linear*	$\mathbb{N} \times \mathbb{N}$	$\mathbb{N}$
$power(n, m)$	*linear*	$\mathbb{N} \times \mathbb{N}$	$\mathbb{N}$
$square(n)$	*linear*	$\mathbb{N}(\times\mathbb{N})$	$\mathbb{N}$
$binom(n, m)$	*tree*	$\mathbb{N} \times \mathbb{N}$	$\mathbb{N}$
$modulo(n, m)$	*tail*	$\mathbb{N} \times \mathbb{N}$	$\mathbb{N}$
$append(l, k)$	*linear*	$\mathbb{L} \times \mathbb{L}$	$\mathbb{L}$
$enum(n, m)$	*linear*	$\mathbb{N} \times \mathbb{N}$	$\mathbb{L}$
$iterate(n, m)$	*linear*	$\mathbb{X} \times \mathbb{N}$	$\mathbb{L}$
$member(n, l)$	*tail*	$\mathbb{N} \times \mathbb{L}$	$\mathbb{B}$

Table 3.2.: Investigated functions

3.3. Investigated Functions

For our investigation we chose 16 recursive functions, which we implemented according to the building blocks presented in the previous chapter. We implemented the functions so that the recursive calls, always occur in the *else* part of if-then-else statements. This is essential for unfolding the terms. All functions are shown in table 3.2. For each function its name, the recursion type and input as well as output types are given. All functions are also presented in much greater detail in appendix part A.

Among the functions are 11 *linear recursive*, 2 *tree recursive* and 3 *tail recursive* functions (cf. chapter 2.1). We omit nested recursion and mutual recursion for their innate increased complexity. It is reasonable to avoid that complexity for a first investigation of the feasibility of our approach.

Seven functions are unary working on only one variable, nine are binary functions. Nine functions produce natural numbers as output, for five the output type is list. The *last*

function[3] can produce a list or a natural number depending on the input. The *member* function is another special case. It decides whether a certain number is member of a given list and is supposed to return a truth value. We did not want to introduce boolean values as an additional output type. Therefore, the member function returns *0* (in case the element is not part of the list) or *1* (in case it is).

In the next section we will describe the steps, which are executed during the anti-unification and matching workflow.

3.4. Processing Steps

Our approach tries to learn the correct recursive definition of a target function p_2. This process described here, is always applied on a pair of functions (e.g. on sum and faculty function). The recursive function definition of the base function p_1 is known as well as an unfolding of the target function t_2.

3.4.1. Unfolding and Anti-unification

The first step is to unfold the recursive definition of the source function. The unfolding is done for a certain input. In our current implementation we do not determine the unfolding depth algorithmically. Instead, function objects are instantiated with certain values as parameters. When a function is expanded, syntactic unfolding is done until the base case is reached. That means the recursive call is replaced by the original recursive definition and all occurrences of a variable in the recursive definition are replaced by the corresponding term in the recursive call, e.g. for the first expansion of the sum function, all occurrences of the variable n are replaced by $subtract(n, 1)$ as the recursive call has been $sum(subtract(n, 1))$.

As an example we present an interim result of the unfolding of $sum(2)$[4]:

```
if (eq(n, 0),
    0,
    add(n, if (eq(subtract(n, 1), 0),
```

[3] $last(l)$ returns the last element in list l.

[4] The recursive definition tree for *sum* was given on page 30

```
            0,
            add(subtract(n, 1), if (eq(subtract(subtract(n, 1), 1), 0),
->              0,
                add(subtract(subtract(n, 1), 1),
                sum(subtract(subtract(subtract(n, 1), 1), 1))))))))
```

As can be seen the term is actually two big: for $n = 2$ the base case is reached on the line marked with `->`. Also we do not want to include recursive calls in our unfolded terms. As we have already checked that the base case has been reached at the current step, the excess *else* part can be removed. This results in the following unfolding:

```
if (eq(n, 0),
    0,
    add(n, if (eq(subtract(n, 1), 0),
        0,
        add(subtract(n, 1), if (eq(subtract(subtract(n, 1), 1), 0),
            0)))))
```

Figure 3.4 on page 36 depicts the unfolding for the *sum* function with $n = 2$. One can see that our unfolded terms differ from the initial programs used in [SMW98].[5] The initial program provides the solutions for all inputs up to x. Instead, the unfolded terms give the solution for exactly one input in a verbose manner. After the unfolding step there are two unfolded terms: t_2, the unfolding of the target term and t_1, the unfolding of the base term.

The next step is the anti-unification of t_1 and t_2. The results are the generalized term t_g and the sigma store σ, which is a combined representation of σ_1 and σ_2, i.e. the substitutions necessary to obtain the original terms t_1 and t_2 from the generalized term t_g.

As the output substitutions of the anti-unification library are just strings, we transform them into the terms according to our function representation. For example, a 0 as a substitution is just a string literal 0. After parsing the same string is handled as constant object with value 0. As we described in 3.1 we create the reversed sigma store σ_r. Here is an example:

$\sigma =$ `#4: 1 =^= 0; $8: mult(@) =^= add(@)`

[5] An example for an initial program has been given on page 27.

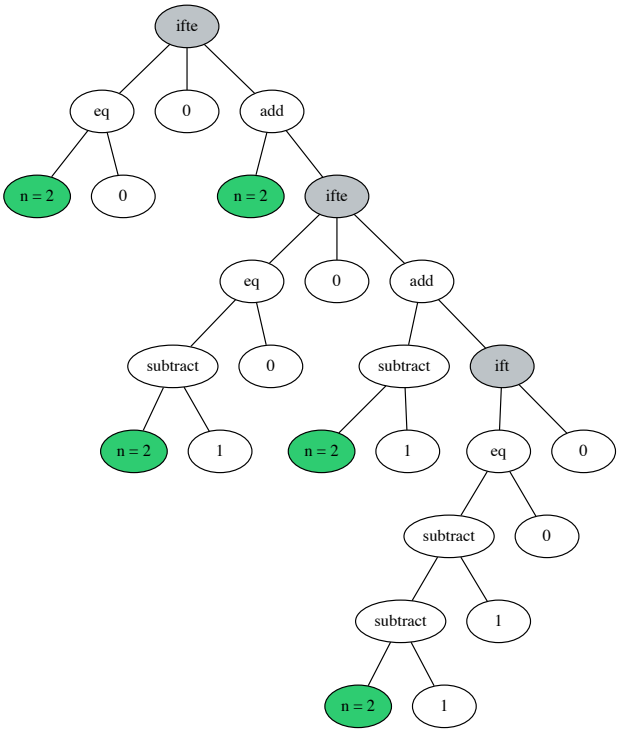


Figure 3.4.: The unfolded sum function for $n = 2$

$$\sigma_1 = \{\#4 := 1, \$8 := mult(\circ)\}$$

$$\sigma_2 = \{\#4 := 0, \$8. = add(\circ)\}$$

σ_1 and σ_2 can be obtained by the raw output of the sigma store σ. The reversed sigma abstracts from the anti-unification variables and maps substitutions from t_1 to t_2 directly:

$$\sigma_r = \{1 := 0, mult(\circ) := add(\circ)\}$$

In the reversed sigma store there are no more anti-unification variables, i.e. no hedge or context variables, however the hole operation remains for substitutions originating from context variables. In the following, when talking about variables, we refer to the variables in the recursive functions. The reversed sigma store σ_r is the basis for the following matching.

3.4.2. Matching

Subsequent to the anti-unification, the matching process is a deterministic sequence of steps to find the underlying recursive definition p_2 of the corresponding unfolding of the target term. We did not include any *try-and-error* or *bruteforce* mechanics. We call the processing steps *rules*. Each rule will be presented with its name and a description of its effects.

As a basis for finding p_2 we copy the recursive definition of the base function p_1, i.e. starting out $p_1 = p_2$. However we rename the recursive call (in case of linear and tail recursion), the recursive calls (for tree recursion) respectively, in p_2 to *rec*, to express that we actually do not know the target recursive function. In the following we will describe the major steps in the matching procedure. Before starting to adapt parts of p_2, we first investigate the rules in the sigma store to find inconsistent substitutions and possible variable replacements.

Inconsistent Substitution

In our function representation, terms are trees with exactly one root node. There are additional restrictions as well. For example, a if-then-else node must have 2 or 3 child nodes to allow for *if*, *then* and potentially *else* subterms.

The left- and right-hand sides of substitutions in σ_r also have to have a single root node. We have found that this constraint is often violated for if-then-else substitutions.

Rule 3.1 *Conditional Completion*
All functions we investigated share the property of having a if-then-else as topmost element. Therefore, for all function definitions a if-then-else is the root node. When anti-unifying such terms, the initial if will always be part of the generalized term t_g, as there is no difference between the terms at that position. This can lead to problems:

$t_1 = if(eq(n, 0), 0, \ldots$

$t_2 = if(less(m, l), 1, \ldots$

As $eq(n, 0), 0$ and $less(m, l), 1$ do not share any commonalities, the expressions are abstracted by a variable in the generalized term t_g, which begins with $if(\#4, \dots)$. When creating the reversed sigma store for that substitution we get: $eq(n, 0), 0 := less(m, l), 1$. The expressions on both sides are no terms by our definition as a single root element is missing. When this happens, we do the following adaption: if the first element on both sides is a *comparative operator* and there are two head elements in each expression, we transform both sides to be *if-then* terms, i.e. $if(eq(n, 0), 0) := if(less(m, l), 1)$.

If-then terms obtained by that rule can still be applied to if-then-else terms, the *else* part remains unchanged. In other cases, when at least one of the two expressions of a substitution is no valid term, we just remove that substitution from σ_r for now.

Variable Replacements

Variable replacements are supposed to offer a solution for the problem which emerges from the fact, that t_1 and t_2 do not share variable names. Instead, t_1 might use variables n and m, while t_2's variables are called l and k. Variable substitution rules try to find a reasonable way to match t_1's variables to their correct counterparts in t_2.

Rule 3.2 *Plain Variable Substitution*
If there is a substitution of the form $x := y$ in the reversed substitutions σ_r and x, y are variables, replace x by y in p_2 and in all other occurrences of variable x in σ_r. This rule must not be applied, if there is another substitution of the form $x := z$, i.e. in case not all occurrences of x are replaced by the single variable y.

Rule 3.3 *Contextual Variable Substitution*
This rule builds a *key-value* data structure, called *dict* here, where keys are variable names and the values are sets, i.e. when adding the same element twice, the set still contains that element only once. The program iterates over all substitutions in the reversed sigma store σ_r.

For each substitution, every variable name appearing for the first time on the left side of a substitution is added as key to *dict* and the corresponding

value is initialized as empty set. From the substitution's right side, all occurring variable names are extracted as well. For each variable on the left side, each variable name on the right side is added to the corresponding set.

The result is a key-value mapping, where keys are the variables appearing on the left side of substitutions in σ_r and the values are sets containing all variables co-occurring on the right sides of the substitutions.

At the end, if there is a key k in *dict*, for which the corresponding value set only contains a single variable v, replace k by v in p_2 and in all other occurrences of variable k in σ_r.

After this preparatory step of matching and renaming variables in unambiguous cases, we check for possibilities to apply substitutions in σ_r to p_2.

Finding Applicable Substitutions

The first step for finding matching candidates is to iterate over all substitutions in σ_r again. For each substitution a list of candidates is initialized as an empty list. Then the left hand side ls of each substitution is compared to each element of the recursive definition p_2.[6]

Rule 3.4 *Candidate Matching*
For each substitution in σ_r and for each element e_i (i.e. for each subterm) of p_2: if the left side of the substitution ls and the current e_i are of the same type, e.g. variable and variable, check whether they match in values:

- For constants their values must match.
- Variables have to share the same variable name.
- Operators have to be the same operation.
- For if-then-else objects type equality is sufficient.

If the conditions are met, the current e_i is added to the list of matching candidates for that substitution.

[6] This can be seen as a tree traversal in prefix order again.

Rule 3.5 *Checking Structural Equality*
The previous rule created a (potentially empty) list of candidate matches for each substitution. Those candidates are then inspected more thoroughly. While the finding of a candidate does not involve recursive checks, investigating structural equality is a recursive process. It takes the subterms of candidates into account, therefore, it has no effect on constant and variable matches, as those are always leaves in the tree terms.

For operators the corresponding operands are checked. The substitution's ls has to be the same operator type, as that is ensured by rule 3.4. However, operands have to be match as well. For example, if the substitution's ls is $add(1, \circ)$ and the candidate subterm was $add(n, 2)$, the 1 in the precondition of the substitution does not conform to the n in the candidate term.

Operators are structurally equal, if for each operand op_i of the actual term and of the corresponding operand in the substitution sub_i one of the following holds:

- The substitution's ls operator only has one operand, which is the hole $\circ$.
- op_i is arbitrary and sub_i is the hole $\circ$, or
- $op_i == sub_i$.

For if-then-else constructs as substitution precondition the individual parts are verified: *if*, *then* and (if present) *else*. sub_i denote those parts in the substitution and $ifte_i$ denotes the parts of the if-then-else in the candidate term. For structural equality one of the following conditions has to hold for each part i:

- $ifte_i$ is arbitrary and sub_i is the hole $\circ$.
- $ifte_i == sub_i$, or
- $ifte_i$ is structurally equal to sub_i.
- $ifte_i$ is the *else* part, and for sub_i no *else* exists.

If the check for structural equality fails for a candidate term, it is removed from the candidate list.

After execution of the two previous rules, we obtain a list of tuples $\langle sub, candidates \rangle$, the first element is the substitution coming from σ_r, the second element is the potentially empty list of candidate elements from p_2 where the corresponding substitution could be applied. For cases where there are multiple possible matching candidates, we strive to create a better basis of decision-making. That is the reason for observing the context of matching candidates:

Rule 3.6 *Context Extraction*
We define the context of a subterm as the sequence of *parent* nodes or parent elements in the term. Recalling that our terms are just trees, obtaining the context is simple. If we have

```
if (eq(n, 0), 0, add(n, sum(subtract(n, 1))))
```

as p_2, a substitution $subtract(\circ) := add(\circ)$ in σ_r, and the candidate list only contains the $subtract(n, 1)$ subterm, the context contains: recursive call $sum \rightarrow add$ operator $\rightarrow$ *else* part of if-then-else.

A substitution of the form $0 := 1$ will have two candidates, as there are two constants 0 in p_2. The first occurrence has the context: *eq* operator $\rightarrow$ *if* part. The second 0 has the context: *then* part.

We represent the context of a candidate as list of parent elements and store the context as part of each candidate in each substitution.

At that point, we have obtained a list of tuples of the form: $\langle sub, [candidate_i, context_i] \rangle$, where *sub* is a substitution originating from σ_r and the second element is a list of candidates $candidate_i$, which denote subterms of the recursive definition p_2 and corresponding context $context_i$, which contains information about the position and role of $candidate_i$ in p_2. The context allows us to classify candidate substitutions. For example, we can tell whether a candidate constant is the base case value (*then* part) or is the value a parameter adopt to obtain the base case (*if* part).

Before the actual matching begins, in some cases we rearrange the order of operator substitutions in σ_r. It turned out, that it can happen that multiple substitutions can be applied on the same operator in p_2. It's obvious that in such cases a sensible decision has to be made. If there is a substitution $add(\circ) := mult(\circ)$ and another substitution $add(1, \circ) := mult(2, \circ)$, and both are applicable on the same term in p_2, only one can

actually be applied: As the operator symbol inevitably changes to *mult* after the first substitution, the remaining substitution can no longer be applied.

Normally, we just apply substitutions in the order they are provided by anti-unification. The reason is that, the anti-unification works from the left side to the right side in the unfolded terms t_1 and t_2. Typically the leftmost subterms are also most similar to the underlying recursive definitions. On the left side there has been no or at most one expansion of recursive calls. In contrast, the rightmost subterm has been obtained by the maximum amount of recursion expansions.

Nevertheless, it has shown that for our investigated functions it is reasonable to watch out for a *balance* of lengths of the precondition and the actual replacement of a substitution.

> **Rule 3.7** *Balance of Operator Substitutions*
> If there are two substitutions sub_i and another substitution sub_j, whose preconditions (left sides) are both operators with the same operator name, we potentially change the order of these substitutions in the reversed sigma store σ_r. We call the operators op_i and op_j and their corresponding replacements $repl_i$ and $repl_j$.
>
> We consider the differences in the absolute lenghts of the substituted and replacement operator expressions. Only the string length of the whole operator expression is considered. For example, $add(\circ, n)$ has nine characters, $mult(\circ, n)$ has ten characters. We denote the absolute length of an operator as $|op|$.
>
> Whenever two substitutions from σ_r are operator substitutions that can be applied to the same subterm in p_2, we compute the differences in lengths between the left and the right side of that substitutions, i.e. $diff_i = |repl_i| - |op_i|$ and $diff_j = |repl_j| - |op_j|$.
>
> If $diff_i$ is less than $diff_j$, substitution corresponding to i is getting a higher priority in σ_r, i.e. it is moved upwards in the list of all substitutions and is thus applied first.

After the rearranging has been executed, we try to apply the substitutions in the reversed sigma store in order of their occurrence. Applying a substitution is really

Candidate	Substitution	Resulting Operator
$add(x, y)$	$add(\circ) := mult(\circ)$	$mult(x, y)$
$add(x, y)$	$add(x, \circ) := mult(z, \circ)$	$mult(z, y)$
$add(x, y)$	$add(x, \circ) := mult(\circ, z)$	$mult(y, z)$
$add(y, x)$	$add(\circ, x) := mult(\circ, z)$	$mult(y, z)$
$add(y, x)$	$add(\circ, x) := mult(z, \circ)$	$mult(z, y)$

Table 3.3.: Patterns that can occur in substitutions originating from higher-order variables

just the replacing of a candidate subterm in p_2 by the corresponding right side of the substitution. If a substitution can no longer be applied, because the subterm in question has already been changed by earlier substitutions, it is ignored.

Substitutions which are based on higher-order (context) variables, often occur when an operator is replaced by another operator, while at least one of the operands remains the same. Table 3.3 shows the results for possible combinations of original term and substitutions. In the given patterns, y is always a common operand.

It can happen that for a substitution multiple replacements candidates exist, i.e. the list of corresponding candidates has more than one entry. When such a case occurs, we have to decide which replacement candidate is the most promising and is actually substituted. For doing that, we developed a simple *scoring* mechanism, that can be extended in the future for more elaborated decision-making. The candidate with the highest score is used for the actual substitution. Currently we only use a small set of rules.

Rule 3.8 *Candidate Scoring*
The scoring is done for each candidate of a substitution. The initial score is always 0. The score value can be increased or decreased depending on types of candidate terms and substitution, context and the absolute position of a candidate term in p_r.

Investigating our functions, it often showed that substitutions of constants, especially those involving number 0 typically have multiple candidates. For constants we have established the following scoring:

- if a constant occurs in the scope of a comparative operator, it is less likely, that it has to be replaced and the score is reduced by 1. For example, many recursive functions involve a base case test of the form $eq(n, 0)$ so the 0 must not be replaced.
- in contrast, if a constant is in the context of a *then* part of an if-then-else statement we increase the score by 1. This seems reasonable, as the *then* part represents the base case value, which often differs, e.g. for *sum* and *faculty* functions.

It can happen that there are two or more candidates for substituting an operator term. This happens mostly for non-comparative operators. **Comparative** operators actually only occur in *if* condition statements and are thus unique in most terms. The same **regular** operator, however, can definitely occur twice. For example in the recursive definition of the *negative sum* function: $if(eq(n, 0), 0, subtract(nsum(subtract(n, 1)), n))$

There are two *subtract* operators, one of them is in the recursive call $nsum$. In such a case we prefer to substitute the outer operation, which is not in the scope of the recursive call:

- If a substitution candidate is an operator in the scope of a recursive call, decrease its score by 1.

After the application of all admissible substitutions to the recursive definition p_2, we take a closer look at the recursive call(s) in p_r. As our unfolded terms t_1 and t_2 do not contain recursive calls, inconsistencies can emerge from the lack of information. There are typical cases which can be handled.

Rule 3.9 *Inconsistencies in Recursive Calls*

- When learning functions which only use one parameter from functions which have two parameters, often, the recursive call in the learned, unary function p_2 still has two parameters:

  $$if(eq(m, 0), 1, mult(m, rec(subtract(m, 1), m)))$$

The function above represents the *faculty* function. However the second m in the recursive call $rec(subtract(m, 1), m)$ does not make sense. We handle this case in the following way: if p_2 only uses one variable name (m) and a recursive call has two parameters, remove the parameter which is just the plain variable (i.e. remove m and not $subtract(m, 1)$. A variable that remains unchanged inside the recursive call cannot contribute to reaching the base case eventually.

- Another problem is that p_2 might be a recursive function which uses two parameters n and m in the whole term, but the corresponding recursive call is of the form $rec(subtract(n, 1), n)$, e.g.

  $$if(eq(m, 0), 0, add(n, muladd(n, subtract(n, 1))))$$

  In such a case we check the comparative operator in p_2, which is $eq(m, 0)$, and extract the variables occurring. If variable m is involved in a comparative operator, it has to be the variable that gets changed in the recursive call. So we replace the n in $subtract(n, 1)$ in p_2 by m.

After handling the recursive calls in p_2 we can test the resulting recursive definition.

3.4.3. Testing

As we have stated earlier, the unfolding of a recursive definition is done for certain parameters, e.g. $sum(2)$. The resulting term can be evaluated. The tree in figure 3.4 on page 36, for example, would be evaluated to 3.

When we have learned the *sum* function from another function we obtain a recursive definition p_2, which is executable as well. We capitalize on this. As variable names in the learned function correspond to the ones, during the unfolding, we can instantiate the variables with the values during unfolding, i.e. we instantiate the variable in the learned *sum* function, say n, with the constant 2, as well. The resulting instantiated recursion can be computed.

We then compare the result of evaluating t_2 to the result of the evaluation of the instantiated p_2. If they are equal, we consider p_2 to be the correct recursive definition of the learned function. If they are not equal or p_2 cannot be executed, e.g., because the

learned function is not a valid term, the recursive definition will always be considered to be wrong.

To avoid false positives among the working functions, we conduct additional tests for some functions, e.g. on *member* whose possible output is limited to 0 or 1. Additional tests can be defined for each function. Such a test must provide input values and the expected outcome. In the following section we present two detailed case studies for our anti-unification and matching technique.

3.5. In-depth Examples

This section provides two cases of trying to learn a recursive function from another recursive definition. We will provide all information created during anti-unification, talk about heuristics, that are used in the individual cases and present the resulting recursive definition. The presentation will be close to the actual processing in our program.

3.5.1. Sum and Faculty

We try to learn the *faculty* function with *sum* as base function. As parameters we choose $sum(2)$ and $fac(2)$.

Starting out we have:

```
p_1 = if (eq(n, 0), 0, add(n, sum(subtract(n, 1))))
t_2 =
if (eq(m, 0),
    1,
    mult(m, if (eq(subtract(m, 1), 0),
        1,
        mult(subtract(m, 1),
        if (eq(subtract(subtract(m, 1), 1), 0), 1)))))
```

Unfolding the sum function yields:

```
t_1 =
if (eq(n, 0),
    0,
    add(n, if (eq(subtract(n, 1), 0),
        0,
        add(subtract(n, 1),
        if (eq(subtract(subtract(n, 1), 1), 0), 1)))))
```

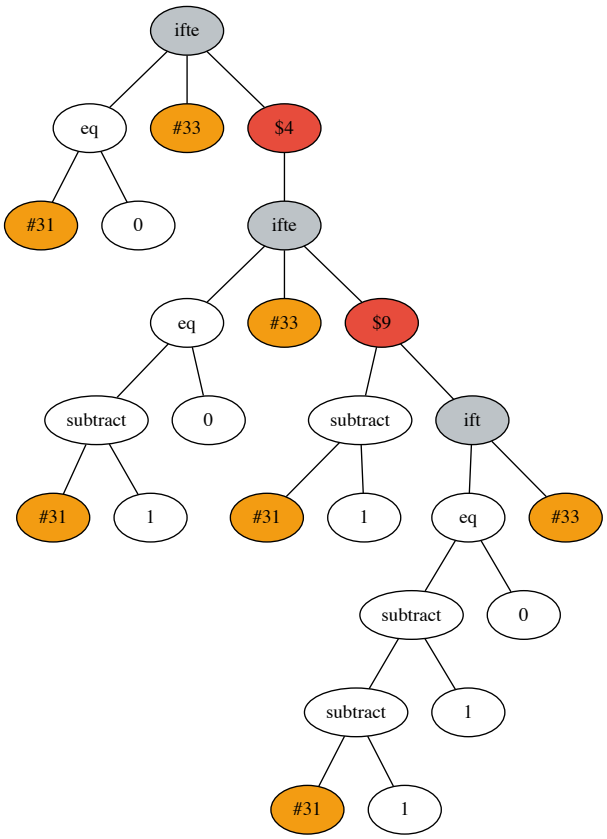


Figure 3.5.: The generalized term t_g, obtained from the anti-unification of t_1 and t_2

Anti-unification can now be done for t_1 and t_2. Figure 3.5 on the next page shows the generalized term t_g in the tree representation. Orange nodes, whose labels begin with # are first-order variables. Red nodes, with labels beginning with $ are second-order. Before matching we initialize p_2 as copy of p_1, and rename the recursive call to *rec*.

The sigma store σ contains the following substitutions:

```
#4: () =^= (); $4: add(n, @) =^= mult(m, @);
#9: () =^= (); $9: add(@) =^= mult(@);
#31: n =^= m; $31: @ =^= @;
#33: 0 =^= 1; $33: @ =^= @;
```

From the sigma store we create the reversed sigma store σ_r:

$$\sigma_r = \{add(n, \circ) := mult(m, \circ), add(\circ) := mult(\circ), n := m, 0 := 1\}$$

The *Plain Variable Substitution* causes the replacement of n by m, because there is the substitution $n := m$.

$$p_2 = if(eq(m, 0), 0, add(m, rec(subtract(m, 1))))$$

After variable replacement the remaining substitutions are:

$$\sigma_r = \{add(m, \circ) := mult(m, \circ), add(\circ) := mult(\circ), 0 := 1\}$$

A valid substitution candidate for both, $add(m, \circ)$ and $add(\circ)$ is the operator

$$add(m, rec(subtract(m, 1)))$$

in p_2. For the $0 := 1$ substitution, there are two candidates: The 0 in $eq(m, 0)$ and the 0, which is the *then* part of the if-then-else. Now the application of substitutions begins. The first substitution in σ_r is

$$add(m, \circ) := mult(m, \circ)$$

There is only one candidate, so the substitution is immediately executed. The hole operator is replaced by the recursive call $rec(subtract(m, 1))$. The resulting recursive definition is:

$$p_2 = if(eq(m, 0), 0, mult(m, rec(subtract(m, 1))))$$

Substitution $add(\circ) := mult(\circ)$ can no longer be applied, as the candidate term now is a *mult* operator. For the substitution of 0 by 1 there are two candidates. That is why we have to look at the scores of the candidates:

$$score(0, eq(m, 0), context_1) = -1$$

$$score(0, 0, context_2) = 1$$

The score for replacing the 0 in the *then* part is higher, i.e. that is where the 1 is inserted:

$$p_2 = if(eq(m, 0), 1, mult(m, rec(subtract(m, 1))))$$

There are no inconsistencies in the recursive call, so we can test t_2 and the instantiated p_2 with $m = 2$.

$$eval(t_2) = 2$$

$$eval(p_2) = 2$$

Both results are equal, i.e. we consider the learned recursive definition to be correct.

3.5.2. Sum and Last

We try to learn the *last* function from the *sum* function. *Last* takes a list of values as input and returns the last element in the list. As all other elements are discarded, *last* is tail recursive. As parameters we choose $sum(2)$ and $last([1, 2])$.

Starting out we have:

```
p_1 = if (eq(n, 0), 0, add(n, sum(subtract(n, 1))))
t_2 =
if (isempty(cdr(m)),
    car(m),
    if (isempty(cdr(cdr(m))),
        car(cdr(m))))
```

Unfolding the sum function yields:

```
t_1 =
if (eq(n, 0),
    0,
    add(n, if (eq(subtract(n, 1), 0),
        0,
        add(subtract(n, 1),
        if (eq(subtract(subtract(n, 1), 1), 0), 1)))))
```

The next step is the anti-unification can of t_1 and t_2. Before matching we initialize p_2 again as copy of p_1, and rename the recursive call to *rec*. The sigma store contains the following substitutions:

```
#2: () =^= (); $2: add(n, @) =^= @;
#3: (eq(subtract(n, 1), 0), 0, add(subtract(n, 1),
      if(eq(subtract(subtract(n, 1), 1), 0), 0)))
          =^=
      (isempty(cdr(cdr(m))), car(cdr(m)));
      $3: @ =^= @;
#4: (eq(n, 0), 0) =^= (isempty(cdr(m)), car(m)); $4: @ =^= @
```

Altering the sigma store we obtain the reversed sigma store σ_r. For #3 as well as for #4 the *Conditional Completion* rule comes into effect. The expressions in both substitutions are not valid, but they all start with a comparative operator. So they are transformed into *if-then* terms.

$$\sigma_r = \{add(n, \circ) := \circ,$$

$$if(eq(subtract(n, 1), 0), \cdots := if(isempty(cdr(cdr(m))), car(cdr(m))),$$

$$if(eq(n, 0), 0) := if(isempty(cdr(m)), car(m))$$

The substitution given by hedge variable #3 is so big, that it can never be matched to a part of p_2, so we will ignore it from now on. The *Contextual Variable Substitution* causes the replacement of n by m, as all substitutions with variable n on their left side, use variable m on their right side. For p_2 that means:

$$p_2 = if(eq(m, 0), 0, add(m, rec(subtract(m, 1))))$$

The remaining substitutions are:

$$\sigma_r = \{add(m, \circ) := \circ, if(eq(m, 0), 0) := if(isempty(cdr(m)), car(m))$$

As for each substitution there is only one candidate, we will omit the finding of candidates here and instead skip to the application of substitutions directly. The first substitution $add(n, \circ) := \circ$ is interesting, as it causes the deletion of $add(n,$. As a consequence the function becomes tail recursive. The updated p_2 is:

$$p_2 = if(eq(m, 0), 0, rec(subtract(m, 1)))$$

The last applicable substitution is $if(eq(m, 0), 0) := if(isempty(cdr(m)), car(m))$. We apply to the if-then-else statement, resulting in

$$p_2 = if(isempty(cdr(m)), car(m), rec(subtract(m, 1)))$$

There are no inconsistencies in the recursive call, so we try to test t_2 and the instantiated p_2 with $m = 2$.

$$eval(t_2) = 2$$

$$eval(p_2) = undefined$$

Unfortunately p_2 cannot be evaluated as the operation $subtract(m, 1)$ in the recursive call is not defined on lists. However the inferred p_2 is identical to the real recursive definition of last apart from the recursive call. The correct recursive definition is:

$$last(m) = if(isempty(cdr(m)), car(m), last(cdr(m))).$$

The output of our program for all pairs of investigated functions is provided in .html files, which we described in appendix part C. These files also provide mention the underlying function parameters. In the next chapter we evaluate our approach for generalization and matching.

4 Evaluation

For evaluating our approach, we use the following setup: there are 16 recursive functions, which have been introduced in 3.3. In appendix chapter A starting on page 66 all recursive functions are presented with corresponding formal definition and their tree representation. For evaluation we apply our process of anti-unification and matching to every pair of recursive functions, i.e. from each function we try to learn the recursive definitions of all other functions. Concerning function parameters, we mostly chose parameters in such a way that the base case was reached on the expansion of the third recursive call.

We imposed some restrictions concerning function pairs, that currently cannot be handled. Learning a recursive definition from its own definition has been excluded (e.g. learn *sum* from *sum*). It is obvious that this is a trivial task in theory, as the generalization of two identical term is identical to both inputs. Therefore, we won't obtain anti-unification variables or substitutions. So the base function remains unaltered and is equal to the target function immediately. On the technical side, the version of the second-order anti-unification library which we use, does not output anything for identical terms, which is problematic for our processing.

In addition we exclude those function pairs from evaluation, for which the base function operates on one recursion parameter, but the target function needs two (fac and *power*). The other direction (from two parameters to one) is allowed, e.g. *enum* and *sum*. This is reasonable, as it is a lot harder to insert a parameter than to delete one, which can be done by simple renaming of variables.

	sum	nsum	fac	fib	countd	last	unpack	muladd	power	square	binom	modulo	append	enum	iterate	member
sum		✓	✓		✓	✶	✶									
nsum	✓		✓		✓	✶	✶									
fac	✓	✓			✓	✶	✶									
fib																
countd	✓	✓	✓			✶										
last																
unpack	✶	✶	✶													
muladd	✓	✓	✓		✓	✶	✶		✓	✓		✶	✶	✓	✓	
power	✓	✓	✓		✓	✶	✶	✓		✓		✶	✶	✓	✓	
square	✓	✓	✓		✓	✶	✶	✓	✓			✶	✶	✓	✓	
binom																
modulo						✶										
append	✶	✶	✶					✶	✶	✶		✶				
enum	✓	✓	✓		✓	✶		✓	✓	✓		✶			✓	
iterate	✓	✓	✓		✓	✶		✓	✓	✓		✶		✓		
member																

Excluded

Table 4.1.: All investigated functions with result of pairwise anti-unification and matching

Following the same reasoning and considering recursion types we impose another restriction. fib and *binom* are tree recursive functions. Learning a tree recursive function with the help of a linear or tail recursive function is impossible, as the unfolded terms do not contain information about recursive calls. In the following section we present the results of the evaluation.

4.1. Quantitative Results

The plain results of pairwise anti-unification and matching are given in table 4.1. The table has to be interpreted as follows: every table cell is given by a row and a column. The labels of rows and columns correspond to the recursive functions we investigated. The row denotes the base function and the column denotes the target function, i.e.

Function Pairs		256	
Exclusions			
From 1 to 2 parameters		63	
Base and target identic		16	
Learning tree recursions		23	
Remaining pairs		154	100%
Results			
Working	✓	52	33,8%
Working except for rec. call	✶	34	22,1%
Not working	□	68	44,2%

Table 4.2.: Quantitative figures on the performance of our approach

"use ⟨ row ⟩ function p_1 to learn ⟨ column ⟩ function p_2". For example, cell (*sum*, fac) gives the result of trying to learn the recursive definition of $faculty$ by using the recursive definition of *sum* and an unfolding of $faculty$.

The restrictions described above are illustrated in the table. The excluded area in the upper right corner contains function pairs for which the base function has one parameter and the target function has two. The light grey cells are function pairs, whose base and target are the same function. Cells in dark grey occur in the column for fib and *binom*. They denote that the base function is tail or linear recursive and the target is tree recursive.

The remaining cells are considered to be actually *manageable*. The symbols used have the following meaning:

- ✓ The anti-unification and matching for the corresponding function pair has led to the correct recursive definition p_2.
- ✶ The resulting recursive definition is correct except for parameters in the recursive call. This case occurred for sum and last, which was presented in 3.5.2.
- □ Cases, for which the learned recursive function was outright wrong, have empty cells.

Table 4.2 shows the aggregated results. Overall, there are 256 possible pairs of functions. Subtracting the excluded pairs, 154 manageable pairs remain. Of those, 52

pairs (33,8%) produce the correct recursive definition, 34 resulting definitions (22,1%) are correct except for the recursive call and 68 pairs (44,2%) produce no reasonable output. The relatively high amount of results, whose definitions are correct except for their recursive call, had to be expected, as the unfoldings do not explicitly provide information about recursive calls.

The overall success rate is not groundbreaking. However, for 55,9 % of the cases the result is reasonable (i.e. at least correct or correct save for recursive calls), which is a satisfying value. Aside from that we are sure, that our approach has potential for great improvements in performance. We give ideas about possible improvements, which we deem to be promising, in 4.3.

It's standing out that the tree recursive functions fib and *binom* as well as the *member* function are the least successful. Neither can they be learned from other functions, nor do they serve as working base functions.

The best performing base functions are $muladd, power$ and $square$. From 13 possible target functions, they enable the correct adaption of 8 functions and the partially complete adaption of 4 other functions. The most successful target functions are $sum, nsum$ and fac which are learned correctly from 8 of 15 possible functions and are learned partially correct in two further cases.

4.2. Findings

In this section we give ideas about properties and specialties of some functions. The aim is to understand why the adaption of recursive functions works in some cases and fails in others.

One finding is that pairs of *isomorphic* functions are ideal for inferring correct results. For defining isomorphy we orientate ourselves on graph isomorphy as described in [SS13, chapter 7]. Our tree representation for functions can be seen as a directed graph. Isomorphy answers the question, whether or not two graphs have the same basic structure: two graphs g_1 and g_2 are isomorphic, when a mapping from every node of g_1 to a node in g_2 exists. Such a mapping must ensure that the graph nodes have the same relations (i.e. graph edges) as in the original g_1. Nodes can be renamed in the process.

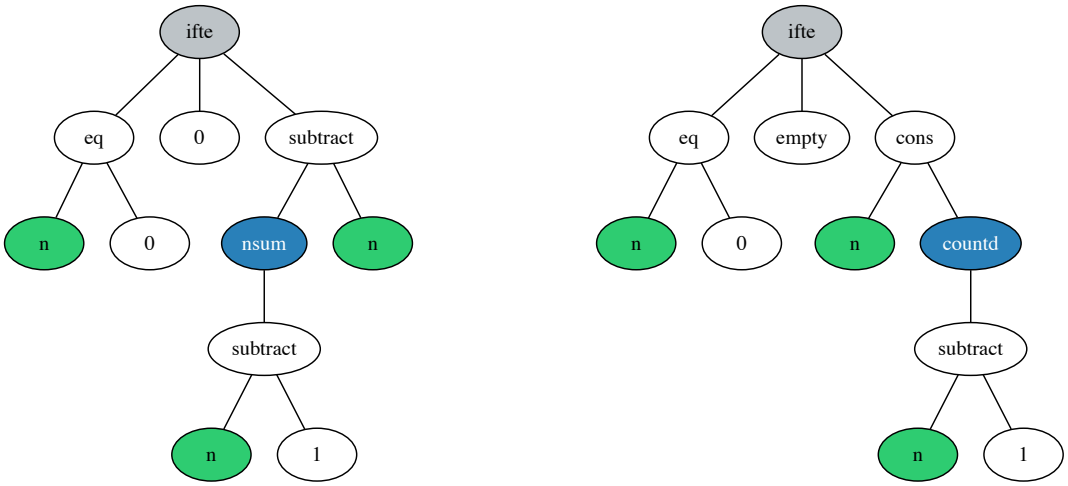


Figure 4.1.: *Negative sum* and *countdown* are isomorphic functions

The concept does not fit completely to our scenario, as our trees do not just have edges but there is an ordering on the edges. i.e. $subtract(2, 8)$ is not the same as $subtract(8, 2)$. Nevertheless, we can observe great performance on isomorphic functions. Figure 4.1 shows the tree representations of *nsum* and *countd*. Even without renaming nodes, they are very similar. We have identified two isomorphic groups among our sample of recursive functions. In each group every function can be learned from any other function of that group. The groups are:

1. *sum*, *nsum*, fac and *countd*
2. *muladd*, *power*, *square*, *enum* and *iterate*

The two groups are also similar among themselves. All pairs of functions, for which the anti-unification and matching process outright works, are part of those two groups. All of them are linear recursive.

One inherent problem of our current approach is unveiled when inspecting the *last* function. As a target function many other functions can successfully learn the definition save for the recursive call. With *last* as base function, however, there is no other target function which achieves that. The reason is that we did not implement a rule to actually insert subterms *ex nihilo*. All we can do is to replace a subterm by a bigger subterm

and thus do an insertion implicitly. This deficit is obvious for *last* and *sum*, *sum* being the target function. At some point during the matching we have:

$$p_2 = if(eq(n, 0), 0, rec(cdr(n))),$$

$$\sigma_r = \{\circ := add(n, \circ)\}$$

That means, there actually is a substitution, which would make p_2 linear recursive as it should be. However, we did not specify how candidate terms for single hole operators $\circ$ can be found. So the saving grace substitution is not applied and the matching ends, resulting in a wrong definition. For *last* as base function, substitutions of the kind $\circ := x$ appear for all linear recursive target functions. Therefore, this is definitely a deficit. Also, such real insertions do not only occur for second-order variables, but also for first-order variables ($\epsilon := x$).

Another problem occurs when learning or learning from functions with two base cases (functions which use nested *if-then-else* constructs). *member* is such a function: It has two possible return values, 0 and 1. If its input list is empty, the result is 0, if the element given as parameter is the first element in the input list, the result will be 1. When learning *member* from a function, which only has one base case, e.g. from *sum*, the following substitution emerges: $\epsilon := isempty(cdr(l))$. That is the same problem described above. In the other direction, i.e. learning *sum* from *member*, we would need to delete the excess *if* statement in the *member* definition.

However, the corresponding sigma store does not contain a substitution, which would allow us to do so. In the unfolding of *sum* there are 4 *if-then-else* statements, the unfolding of *member* has six. The only substitution created, which encompasses the deletion of an *if-then-else* is:

$$if(isempty(cdr(cdr(m))), \circ, if(eq(n, car(cdr(cdr(m)))), 1)) := \circ$$

As one can see this substitution cannot be applied to the recursive definition of member, i.e. the information about deletion of *if-then-else* statements is not usable.

Summarizing, the two main problems, which have to be tackled for improving the performance of our approach, are insertions without preconditions and blown up substitution terms which cannot be matched on a subterm of the processed recursive function.

In the following section we want to provide some ideas for improving our approach.

4.3. Potential Improvements

Use Semantic Knowledge

An imaginable improvement could consist in the modeling of more semantic knowledge. Currently, only the semantics of operators and their arities are defined. However, knowledge about the relations of operations to each other could be modeled as well. The potential benefit can be investigated based on the *last* function. As we have stated, learning *last* by other functions (e.g. fac) yields

$$if(isempty(cdr(n)), car(n), rec(subtract(n, 1)))$$

as p_2. The correct recursive call would be $rec(cdr(n))$. The *subtract* and *cdr* operations are similar in their role. Both can be seen as a *reduction* operator. By the context in which variable n occurs (*cdr* in *if*, *car* in *then*) we can deduce that n has to be instantiated with a list. However, if n hast to be a list, the $subtract(n, 1)$ in the recursive call cannot be correct, as subtraction on lists is not defined. Assumed there was a knowledge base for relations of operations, we could determine that $subtract(n, 1)$ has the role of a reduction and hence look up what operations represent reductions on lists, resulting in *cdr*. As operator arities are known, transforming $subtract(n, 1)$ into $cdr(n)$ is simple. This behavior could be implemented as additional rule which is executed after rule 3.9: *Inconsistencies in Recursive Calls*.

As a consequence, many functions which are learned correct except for their recursive call could be fixed, if we could reason about the roles of operators in terms based on extra semantic knowledge.

Use Information Given in the Generalization

Our approach has a more fundamental possibility to improve. Now, the anti-unification is actually only used to obtain the sigma store σ with its substitutions. So our approach actually represents the following scenario: a person is given a complete recursive function definition and a list of substitutions of the form $x := y$ and has the task to apply the substitutions on the list to transform the recursive function into another function. So, our approach solely concentrates on differences between two unfolded terms. Thereby we neglect that anti-unification also preserves common structures between the

unfoldings by finding the so called anti-instance or generalized term t_g. It provides additional information. Admittedly, the substitutions in σ also contain knowledge about the common structure implicitly as the substitutions are created around those common parts.

As can be seen in figure 3.5 on page 47 these generalized terms are actually regular terms supporting two additional term types: first- and second-order variables. The sigma store contains these variables as well, i.e. we know which substitution originates from which anti-unification variable. This information could be used to get more information about the context of substitutions. We present an example for $last(n = [1, 2])$ and $sum(m = 2)$, sum being the target function. We argued that it is hard to insert terms from substitutions like $\circ := x$ as we do not have information about the position the new term should be at.

However, the generalized term can be used to obtain information about the context of the substitution. The following is the generalized term:

```
(if(#4, $2(if(#3))))
```

The sigma store contains:[1]

```
#2: () =^= (); $2: @ =^= add(m, @);
#4: (isempty(cdr(n)), car(n)) =^= (eq(m, 0), 0); $4: @ =^= @
```

The reversed sigma store is:

$$\sigma_r = \{\circ := add(m, \circ), if(isempty(cdr(n)), car(n)) := if(eq(m, 0), 0)\}$$

and

$$p_2 = if(isempty(cdr(m)), car(m), rec(cdr(m))).$$

The idea is to use the generalized term to obtain information about the correct position of the $\circ := add(m, \circ)$ substitution. We can see that the anti-unification variable corresponding to that substitution, is `$2`. To get correct results we have to instantiate all other first- and second-order variables (`#4` and `#3`) in their original form, i.e. before potentially performing adaptions to the rules. This results in:

```
(if(isempty(cdr(n)), car(n)), $2(if(...))))
```

[1] #3 is omitted for more clarity.

Just like for candidate terms, we can apply rule 3.6 to investigate the context of $2. As a result we obtain the information that $2 is the head element of the *else* part of an *if-then-else* statement.

That information enables us to enter exactly that context in p_2 and add the subterm which is there as a substitution candidate. In the example the subterm described by the context is $rec(cdr(m))$. When applying the substitution $\circ := add(m, \circ)$, we actually insert a subterm:

$$p_2$$
$$= if(isempty(cdr(m)), car(m), add(m, \circ)rec(cdr(m)))$$
$$= if(isempty(cdr(m)), car(m), add(m, rec(cdr(m))))$$

As can be seen p_2 is now a linear recursive function. The remaining substitution can be applied the same way as before, resulting in

$$p_2 = if(eq(m, 0), 0, add(m, rec(cdr(m))))$$

Combining this result with the semantic knowledge we described in the first part of this section, we could even construct the correct recursive call. Using the generalized term to get hints on the context a substitution might even render *Candidate Matching* (rule 3.4) obsolete.

Summarizing, we propose two extensions to our approach which complement each other: the addition of more semantic knowledge to be able to reason about the role operators have in terms and the utilization of the generalized term to infer the correct candidate position for "real" insertion substitutions.

5 Conclusion

In this thesis we have developed an automated programming by analogy approach for learning recursive definitions from examples. We covered the basics of recursion, analogy and anti-unification. For the purpose of our approach we invented a way of representing recursive function as tree structures in Python. Those representations are suitable for generalization with second-order anti-unification, which is the enabling technique for this work. For matching and adapting base functions, we have created heuristics and rules. We employed a stock of 16 recursive functions to evaluate our approach.

The evaluation has shown that combining the idea of programming by analogy with anti-unification techniques does not solve all problems instantly or trivially. However, we could point out that in our approach anti-unification is an excellent method for identifying common structures and differences between recursive functions and their unfoldings. The information about differences and matching parts in structures is essential for mapping in analogical reasoning. Therefore, one of the important conclusions is that the potential use of anti-unification greatly depends on the suitable representation of terms and structures.

Personally, we think that a anti-unification is a powerful approach, not just for learning recursive functions. Instead, we believe that it can be applied to a wide variety of problems. A prerequisite - which perhaps is also the biggest problem - is the appropriate modeling of domains. In our opinion, another interesting finding is that that our results confirm the properties Gentner demanded for analogical reasoning. When we interpret

functions working on natural numbers and those working on lists as different domains, we observe that common structure is much more critical to the successful transfer than the usage of common operators or operators of the same arity.

In chapter 4 we also made suggestions for improving the performance of our approach. So far we exclusively focused on the sigma store for adapting functions. Leveraging the information given in the generalized terms, seems to be particularly promising to us. In perspective it could also be possible to automate the unfolding of base functions to a suitable depth. Another worthwhile question to answer is whether or not appropriate base functions can be retrieved reliably by the system.

We hope that this thesis and especially our implementation of the approach in the Python programming language can serve as a framework or starting point for future work. We tried to design our implementation flexible to allow for future changes and additions. More information on our program is given in appendix part B. The context mechanism and scoring techniques can be adapted easily. We also tried to make the output of the results of our program more comprehensible and meaningful by creating styled HTML files.

By all means, programming by analogy is a deeply interesting topic, not least because of its closeness to the processes taking place in our own minds.

Bibliography

[AT89] John R Anderson and Ross Thompson. Use of analogy in a production system architecture. *Similarity and analogical reasoning*, pages 267–297, 1989.

[BH14] Jochen Burghardt and Birgit Heinz. Implementing anti-unification modulo equational theory. *arXiv preprint arXiv:1404.0953*, 2014.

[BK14a] Alexander Baumgartner and Temur Kutsia. A library of anti-unification algorithms. In *Logics in Artificial Intelligence*, pages 543–557. Springer, 2014.

[BK14b] Alexander Baumgartner and Temur Kutsia. Unranked second-order anti-unification. In *Logic, Language, Information, and Computation*, pages 66–80. Springer, 2014.

[BM09] Peter Bulychev and Marius Minea. An evaluation of duplicate code detection using anti-unification. In *Proc. 3rd International Workshop on Software Clones*. Citeseer, 2009.

[BS01] Franz Baader and Wayne Snyder. Unification theory. *Handbook of automated reasoning*, 1:445–532, 2001.

[Bur05] Jochen Burghardt. E-generalization using grammars. *Artificial intelligence*, 165(1):1–35, 2005.

[Der86] Nachum Dershowitz. Programming by analogy. *Machine Learning: An artificial intelligence approach*, 2:395–423, 1986.

[Gen83] Dedre Gentner. Structure-mapping: A theoretical framework for analogy*. *Cognitive science*, 7(2):155–170, 1983.

[Has95] Robert W Hasker. *The replay of program derivations*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[HM+94] Douglas R Hofstadter, Melanie Mitchell, et al. The copycat project: A model of mental fluidity and analogy-making. *Advances in connectionist and neural computation theory*, 2(31-112):29–30, 1994.

[Hof79] Douglas R. Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books, Inc., New York, NY, USA, 1979.

[Hog84] Christopher John Hogger. *Introduction to logic programming*. Academic Press Professional, Inc., 1984.

[KSGK07] Ulf Krumnack, Angela Schwering, Helmar Gust, and Kai-Uwe Kühnberger. Restricted higher-order anti-unification for analogy making. In *AI 2007: Advances in Artificial Intelligence*, pages 273–282. Springer, 2007.

[Lut13] Mark Lutz. *Learning Python*. O'Reilly Media, Inc., 2013.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[OHP07] John O'Donnell, Cordelia Hall, and Rex Page. *Discrete mathematics using a computer*. Springer Science & Business Media, 2007.

[Pep02] Peter Pepper. *Funktionale Programmierung: In Opal, Ml, Haskell Und Gofer*. Springer, 2002.

[Plo70] Gordon D Plotkin. A note on inductive generalization. *Machine intelligence*, 5(1):153–163, 1970.

[Plo71] Gordon D Plotkin. A further note on inductive generalization. *Machine intelligence*, 6(101-124), 1971.

[Pol14] George Polya. *How to solve it: A new aspect of mathematical method*. Princeton university press, 2014.

[Rey70] John C Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine intelligence*, 5(1):135–151, 1970.

[RP01] Alexander Repenning and C Perrone. Programming by analogous examples. *Your Wish is My Command: Programming by Example. San Francisco, CA: Morgan Kaufmann Publishers*, pages 351–369, 2001.

[SBW03] Ute Schmid, Jochen Burghardt, and Ulrich Wagner. Analogy needs abstraction. *Verfügbar unter http://www. vorlesungen. uos. de/informatik/fp02/iccm03. pdf*, 2003.

[SG89] Wayne Snyder and Jean Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(1):101–140, 1989.

[SMW98] Ute Schmid, René Mercy, and Fritz Wysotzki. Programming by analogy: Retrieval, mapping, adaptation and generalization of recursive program schemes. In *Proc. of the Annual Meeting of the GI Machine Learning Group, FGML*, volume 98, pages 140–147, 1998.

[SS13] Gunther Schmidt and Thomas Ströhlein. *Relationen und Graphen*. Springer-Verlag, 2013.

[Wag02] Ulrich Wagner. Combinatorically restricted higher order anti-unification. an application to programming by analogy. *Unpublished master's thesis, Dept. of Electrical Engineering and Computer Science, TU Berlin, Germany.(http://user. cs. tu-berlin. de/˜ xlat/)*, 116, 2002.

[Wel05] Stephan Weller. Solving proportional analogies by application of anti-unification modulo equational theory. *Available o n http://www-lehre. inf. uos. de/ stweller/ba*, 2005.

[WK01] Robert Andrew Wilson and Frank C Keil. *The MIT encyclopedia of the cognitive sciences*. MIT press, 2001.

[WKS08] Eva Wiese, Uwe Konerding, and Ute Schmid. Mapping and inference in analogical problem solving–as much as needed or as much as possible. In *Proceedings of the 30th Annual Conference of the Cognitive Science Sociecty*, pages 927–932, 2008.

[WS06] Stephan Weller and Ute Schmid. Analogy by abstraction. In *Proceedings of the 7th International Conference on Cognitive Modeling*, pages 334–339, 2006.

A

Investigated Functions

On the following pages all recursive functions which have been investigated in this thesis are given in-depth. We have investigated unary and binary functions.

For each recursive functions the following information is given:

- function name
- information on what the function computes
- formal definition
- programmatic definition (representation which is used in our program)
- visual graph representation of the function

A.1. Unary Functions

Sum

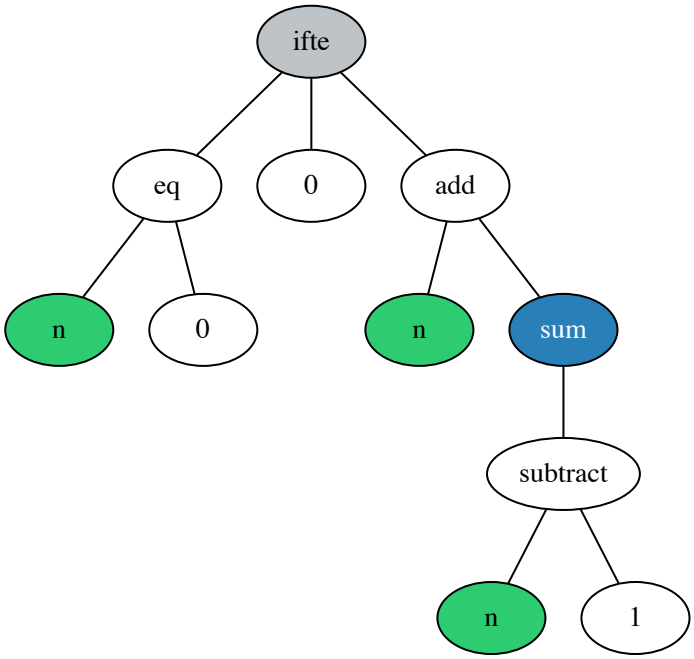
Computes the summation from one to n.

Formal definition:

$$sum(n) = \begin{cases} 0 & \text{for } n = 0 \\ n + sum(n-1) & \text{else} \end{cases}$$

Programmatic definition:

```
if (eq(n, 0),
    0,
    add(n, sum(subtract(n, 1))))
```



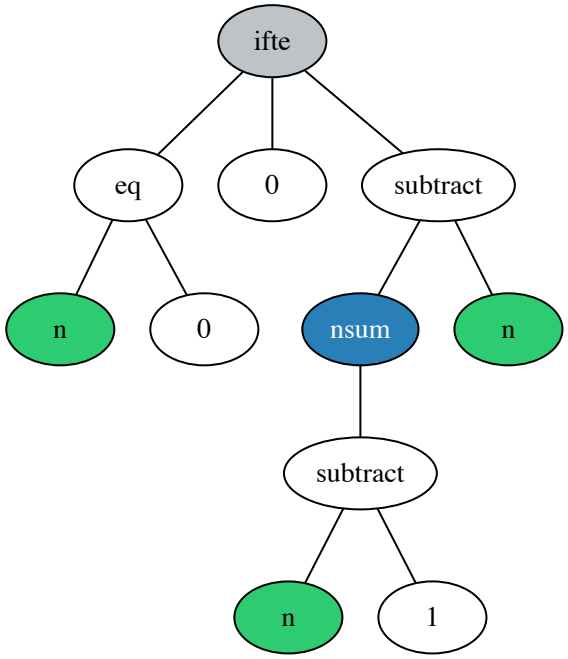
Negative Sum

Computes the negative summation function from one to n.
Formal definition:

$$nsum(n) = \begin{cases} 0 & \text{for } n = 0 \\ nsum(n-1) - n & \text{else} \end{cases}$$

Programmatic definition:

```
if (eq(n, 0),
    0,
    subtract(nsum(subtract(n, 1)), n))
```



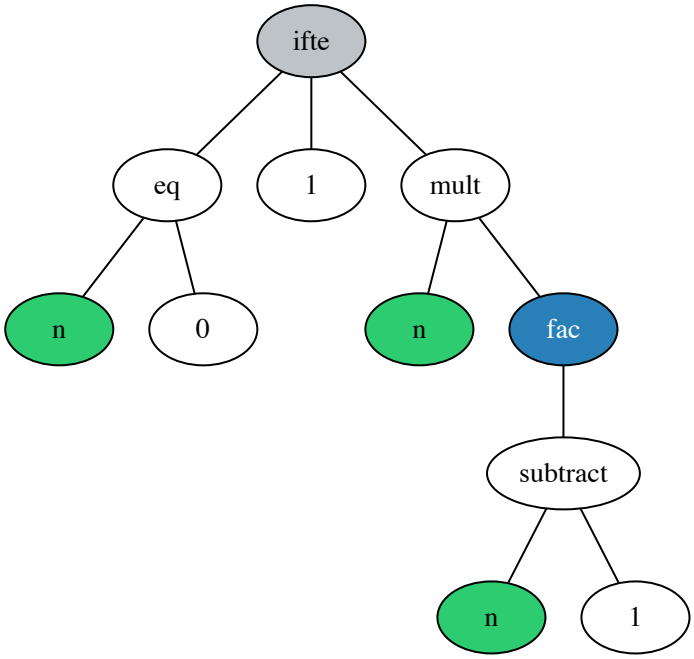
Faculty

Computes the faculty of n, i.e. $n!$.
Formal definition:

$$fac(n) = \begin{cases} 1 & \text{for } n = 0 \\ n * fac(n-1) & \text{else} \end{cases}$$

Programmatic definition:

```
if (eq(n, 0),
    1,
    mult(n, fac(subtract(n, 1))))
```



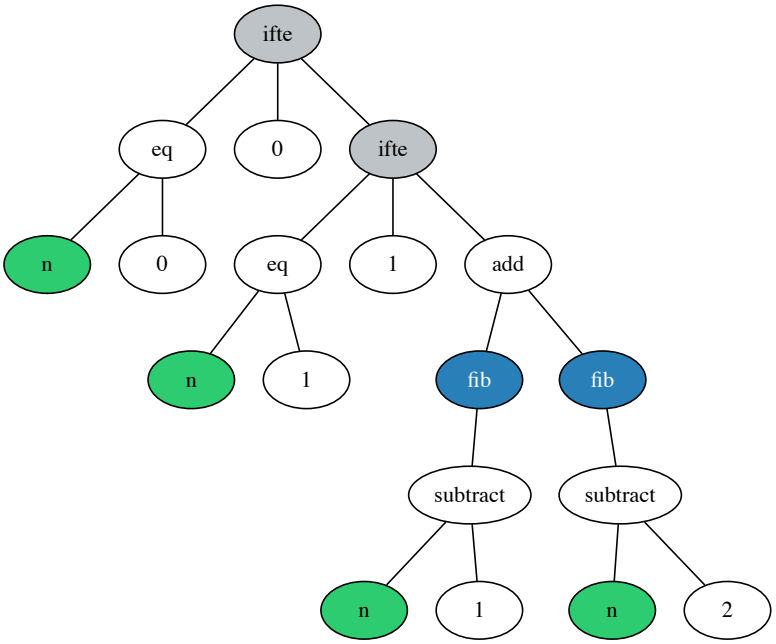
Fibonacci

Computes the n-th number in the Fibonacci sequence.
Formal definition:

$$fib(n) = \begin{cases} 0 & \text{for } n = 0 \\ 1 & \text{for } n = 1 \\ fib(n-1) + fib(n-2) & \text{else} \end{cases}$$

Programmatic definition:

```
if (eq(n, 0),
    0,
    if (eq(n, 1),
      1,
      add(fib(subtract(n, 1)), fib(subtract(n, 2)))))
```



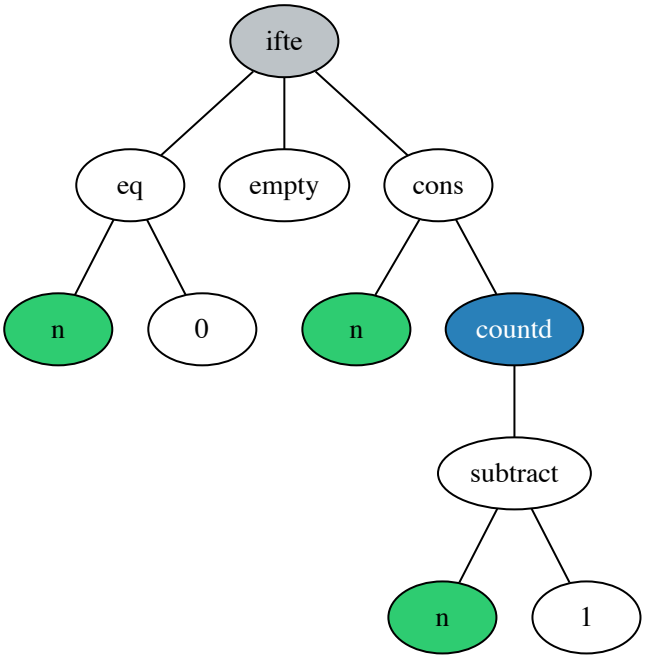
Countdown

Results in a list with numbers from n to one.
Formal definition:

$$countd(n) = \begin{cases} [\,] & \text{for } n = 0 \\ n{+}{+}countd(n-1) & \text{else} \end{cases}$$

Programmatic representation:

```
if (eq(n, 0),
    empty,
    cons(n, countd(subtract(n, 1))))
```



Last

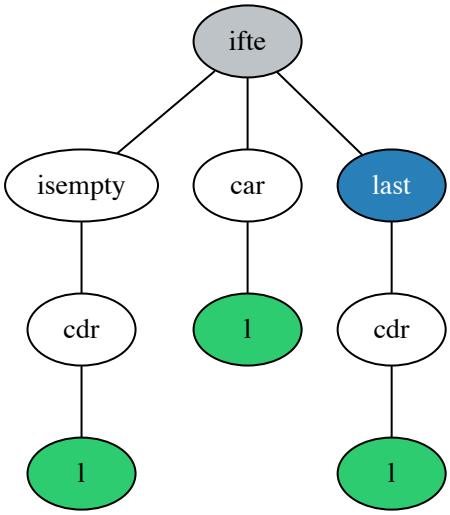
Returns the last element in list l.

Formal definition:

$$last(l) = \begin{cases} car(l) & \text{for } empty(cdr(l)) \\ last(cdr(l)) & \text{else} \end{cases}$$

Programmatic representation:

```
if (isempty(cdr(n)),
    car(n),
    last(cdr(n)))
```



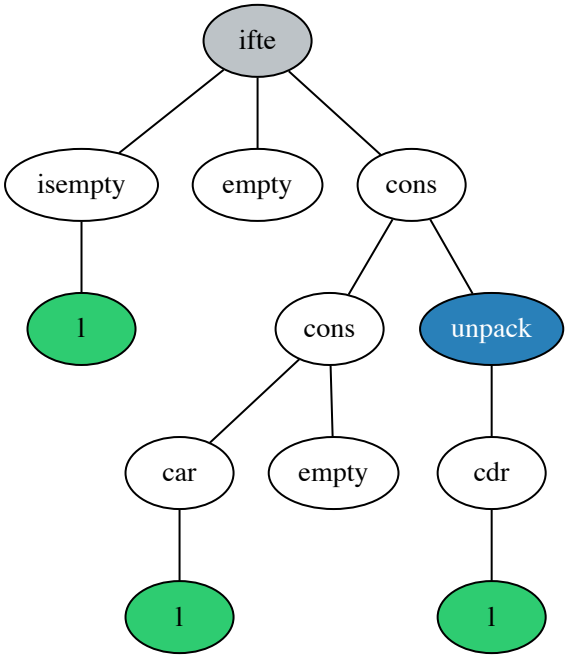
Unpack

Creates a new list, where each element from l has been put into an individual list. Formal definition:

$$unpack(l) = \begin{cases} [\,] & \text{for } empty(l) \\ [car(l)] \mathbin{++} unpack(cdr(l)) & \text{else} \end{cases}$$

Programmatic representation:

```
if (isempty(n),
    empty,
    cons(cons(car(n), empty), unpack(cdr(n))))
```



A.2. Binary Functions

Multiplication by Addition

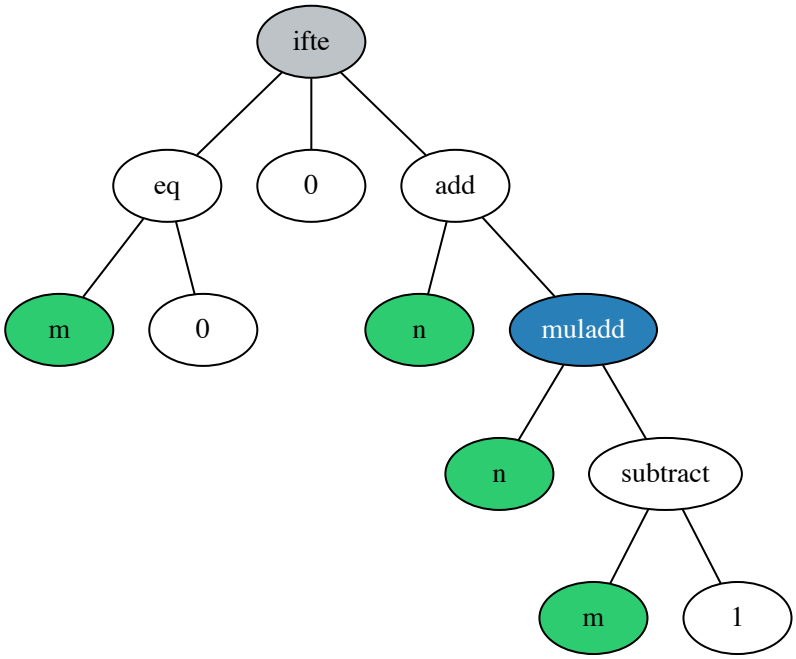
Computes the multiplication of n and m by adding n m times.

Formal definition:

$$muladd(n, m) = \begin{cases} 0 & \text{for } m = 0 \\ n + muladd(n, m - 1) & \text{else} \end{cases}$$

Programmatic definition:

```
if (eq(m, 0),
    0,
    add(n, muladd(n,subtract(m, 1))))
```



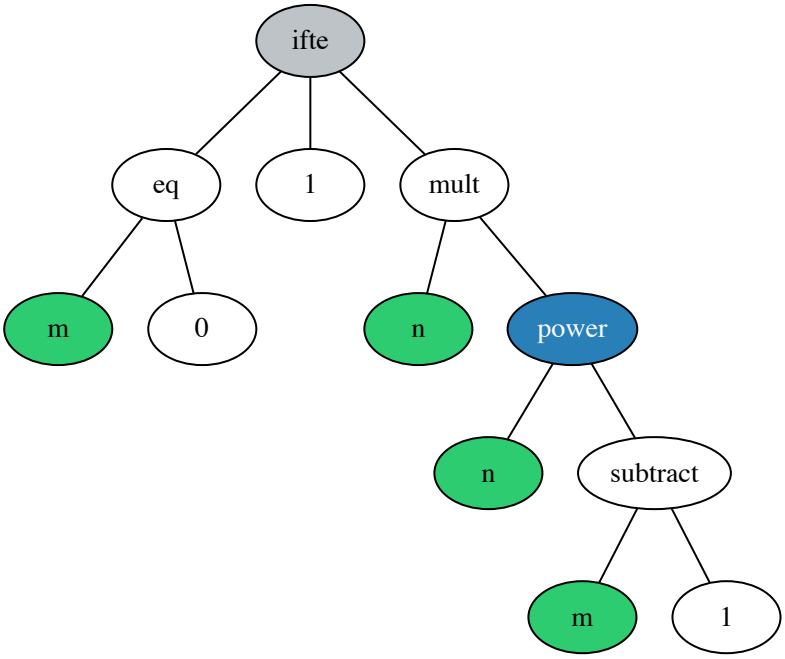
Power

Computes the power of n^m by multiplying n, m times.
Formal definition:

$$power(n, m) = \begin{cases} 1 & \text{for } m = 0 \\ n * power(n, m - 1) & \text{else} \end{cases}$$

Programmatic definition:

```
if (eq(m, 0),
    1,
    mult(n, power(n,subtract(m, 1))))
```



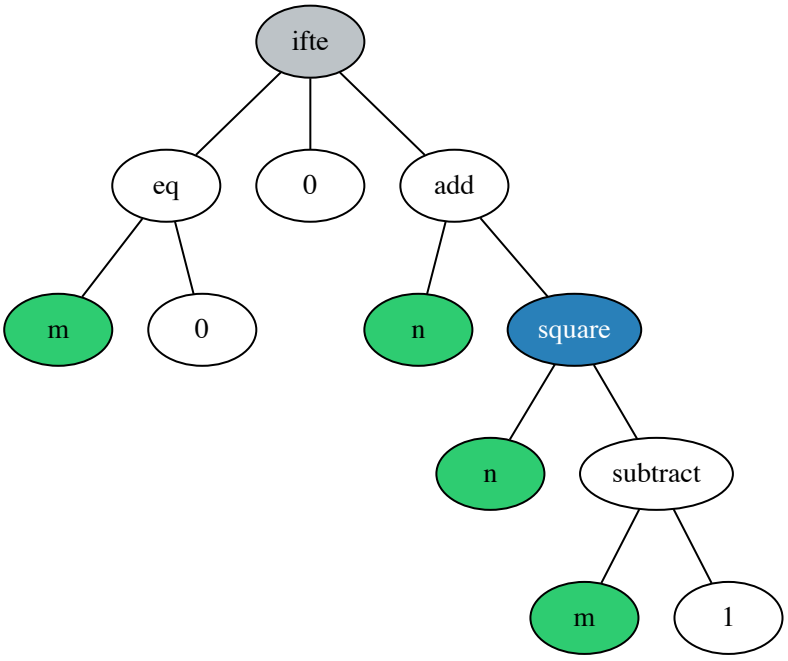
Square

Computes n^2 by adding n, n times. The square function is called with only one parameter (e.g. *square(2)* but uses two internally.
Formal definition:

$$square(n, m) = \begin{cases} 0 & \text{for } m = 0 \\ n + square(n, m - 1) & \text{else} \end{cases}$$

Programmatic representation:

```
if (eq(m, 0),
    0,
    add(n, square(n,subtract(m, 1))))
```



Binomial

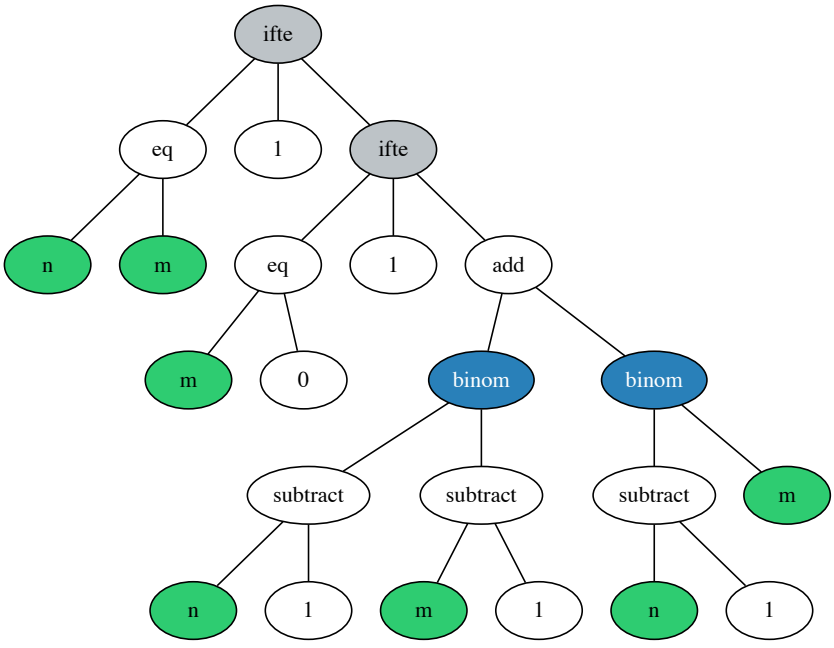
Computes the binomial of n choose m: $\binom{n}{m}$. The first parameter for the binomial function must be greater or equal.

Formal definition:

$$binom(n, m) = \begin{cases} 1 & \text{for } m = 0 \text{ or } n = m \\ binom(n-1, m-1) + binom(n-1, m) & \text{else} \end{cases}$$

Programmatic representation:

```
if (eq(n, m),
    1,
    if (eq(n, 0),
        1,
        add(binom(subtract(n, 1), subtract(m, 1)),
            binom(subtract(n, 1), m))))
```



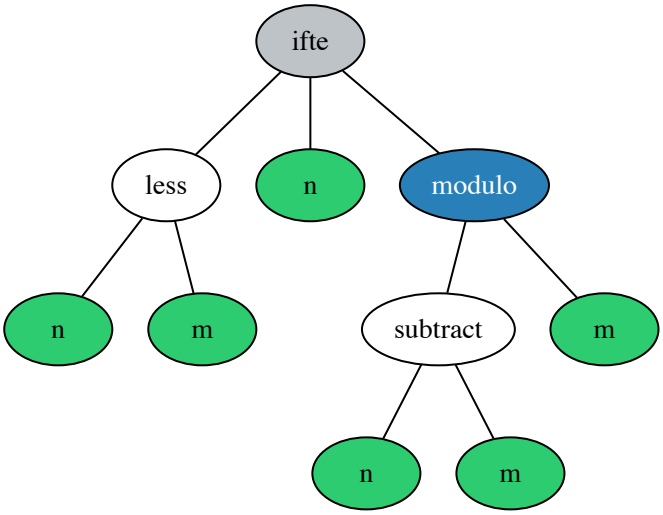
Modulo

Computes the residual of the integer division n/m.
Formal definition:

$$modulo(n,m) = \begin{cases} n & \text{for } n < m \\ modulo(n-m,m) & \text{else} \end{cases}$$

Programmatic representation:

```
if (less(n, m),
    n,
    modulo(subtract(n, m),m))
```



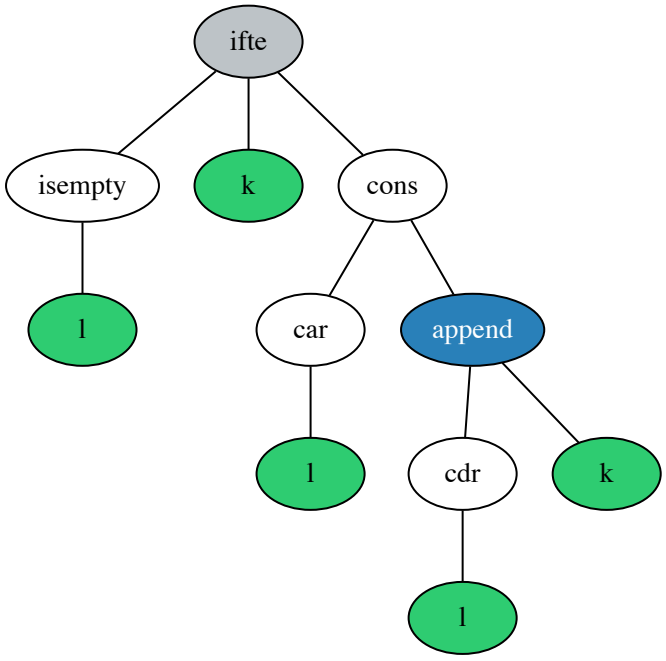
Append

Computes a new list, which contains all elements from l in their original order, followed by all elements from k in their original order.
Formal definition:

$$append(l,k) = \begin{cases} k & \text{for } empty(l) \\ car(l) \mathbin{++} append(cdr(l),k) & \text{else} \end{cases}$$

Programmatic representation:

```
if (isempty(l),
    k,
    cons(car(l), append(cdr(l),k)))
```



Enumerate

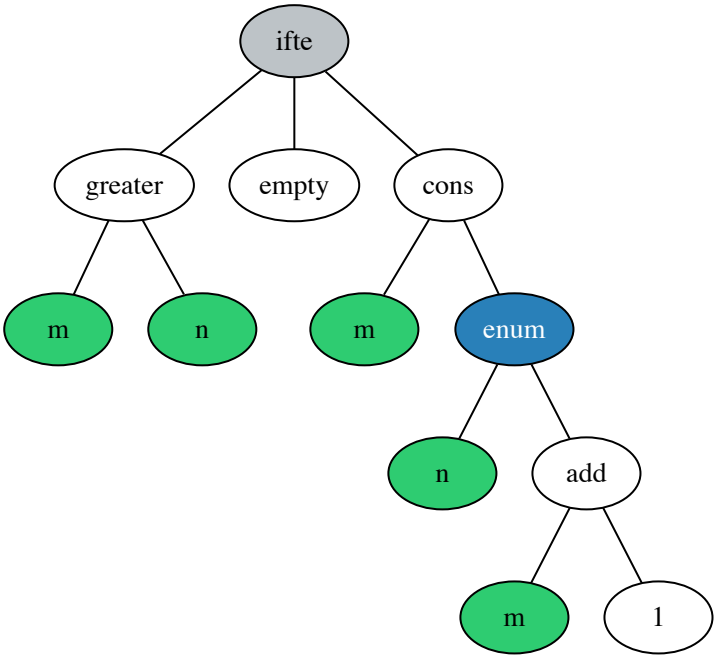
Creates a list containing the numbers from m to n.

Formal definition:

$$enum(n, m) = \begin{cases} [\,] & \text{for } m > n \\ m \mathbin{+\!\!+} enum(n, m+1) & \text{else} \end{cases}$$

Programmatic representation:

```
if (greater(m, n),
    empty,
    cons(m, enum(n, add(m, 1))))
```



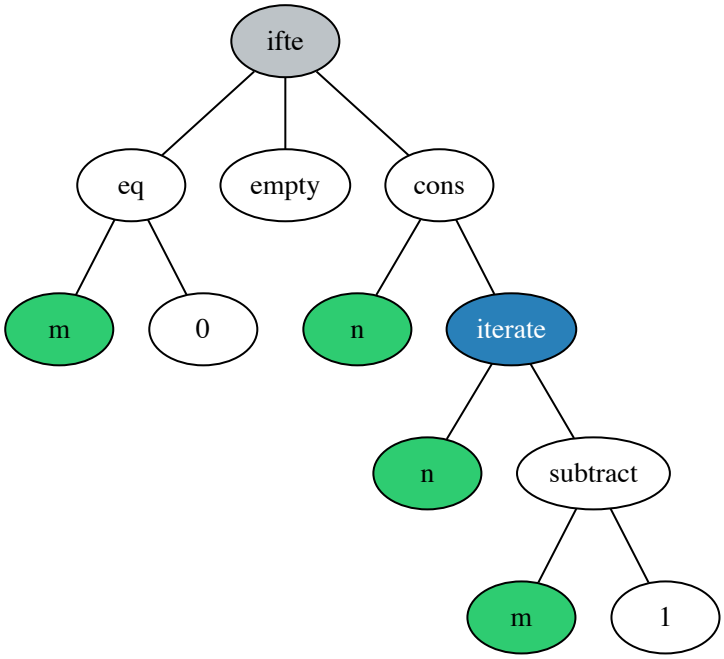
Iterate

Creates a list containing element n, m times.
Formal definition:

$$iterate(n, m) = \begin{cases} [\,] & \text{for } m = 0 \\ n{+}{+}iterate(n, m-1) & \text{else} \end{cases}$$

Programmatic representation:

```
if (eq(m, 0),
    empty(),
    cons(n, iterate(n,subtract(m, 1))))
```



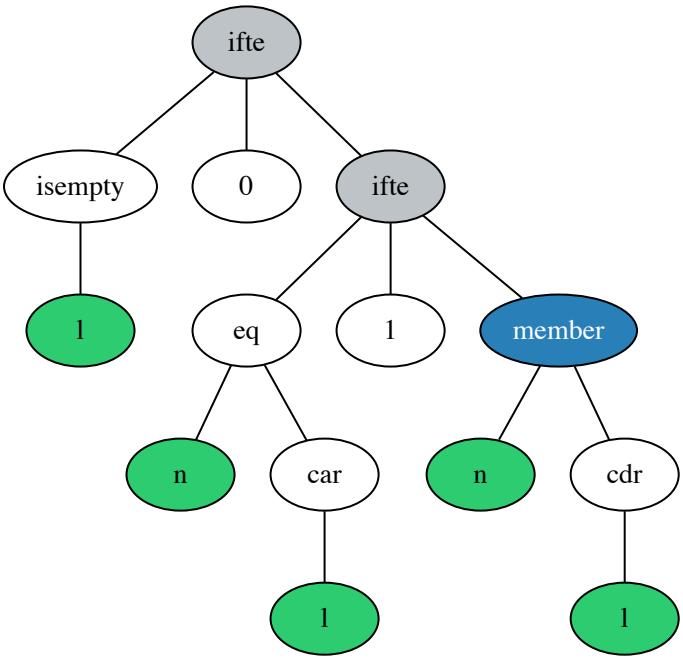
Member

Decides whether number n is an element of list l. Formal definition:

$$member(n,l) = \begin{cases} 0 & \text{for } empty(l) \\ 1 & \text{for } n = car(l) \\ member(n, cdr(l)) & \text{else} \end{cases}$$

Programmatic representation:

```
if (isempty(l),
    0,
    if (eq(n, car(l)),
        1,
         member(n,cdr(l))))
```



B

Program Design

The main program we developed for investigating the possibilities of matching and learning recursive functions by analogy, is implemented in the Python programming language[1] (version 3.x). We chose Python as it is a quite popular language according to the TIOBE Programming Community Index[2], has a readable syntax, offers good community support and well-defined libraries, supports a mixture of object-oriented and procedural programming and - most important in the given use case - supports lambda constructs. [Lut13]

The program also uses a very small part which was implemented in Java for performing the Anti-Unification. The source files for this program are also present on the CD enclosed. In our main program it creates a *subprocess* which is just offering another interface for the Unranked Second-Order Anti-Unification[3] which has been shown in chapter 2.3.3. For the purpose of convenience, we created an executable Java Jar file which handles the whole Anti-Unification and is present inside the root of the project directory. This allows for fast access to the Anti-Unification library. The *.jar* file is called *anti_unify.jar*

In the first section we are going to give an overview of the program and its parts, the second section will cover the steps which have to be taken to run the program and

[1] `https://www.python.org`, last access Sunday 22nd March, 2015

[2] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, last access Sunday 22nd March, 2015

[3] `http://www.risc.jku.at/projects/stout/software/urauc.php`, last access Sunday 22nd March, 2015

extend it.

B.1. Overview

Python programs consist of one or more python modules (i.e. files with the *.py* file ending). In the following we will introduce each module and present its purpose as well as interesting functions it implements:

pydot/ This folder contains the Python modules of *Pydot*[4], which is an excellent Python interface for the *Dot* language, which is part of the *Graphviz*[5] Graph Visualization Software. Graphviz must be in installed on the user's machine to be able to create visual tree graphs from terms. The visualizations in appendix part A were created with Pydot.

terms/ The *terms* folder is essential for constructing terms and functions definitions which can be anti-unified and matched. Most of the modules contain a class which has the same name as the module. The most important classes are:

Constant Represents a single value (e.g. the integer value *4* or the list *[1,2,3]*)

Variable A variable has two components: a name, which is assigned to it by the constructor of the class and a value.

Operator Operators have a name (symbol) and a theoretically arbitrary long list of operands (e.g. *add(a, 2)*, where *add* is the operator name and variable *a* and constant *2* are operands).

IfThenElse Realization of the McCarthy Conditional [McC60]. Each object of the *IfThenElse* class, has an *if* part, a *then* part and most of the time but not imperatively an *else* part. To work correctly the if part must evaluate to a truth value.

RecursiveFunction Represents a call to a recursive function. Objects of this class consist of the name of a recursive function and parameters for the recursive function call (e.g. if (eq(m, 0), 0, add(m, **sum(subtract(m, 1))**)))

[4] `https://code.google.com/p/pydot/`, last access Sunday 22nd March, 2015
[5] `http://www.graphviz.org`, last access Sunday 22nd March, 2015

The *termcomp.py* module contains a superclass for all other term components. *Deletion.py*, *Placeholder.py* and *Replacement.py* are term components which are relevant for the parsing of the results of anti-unification. There are some recursive functions which are implemented for all term components. They are used to traverse the tree structures which constitute the individual terms. An example for such a function is *get_recursive_calls*, which is used to get a list of all *RecursiveFunction* objects (i.e. recursive calls) in a term.

TreeGraphUtil.py is necessary for assigning a unique name to each term component when creating a Pydot graph.

local_config.py Simple configuration file for setting which file paths are to be used and offers the option to output results as HTML. The HTML output is further introduced in appendix chapter C.

knowledge_base.py This file is mainly responsible for storing the semantics of operators which are used when evaluating and parsing terms. Among other things it manages a list of valid operator names together with operator cardinality and names of defined recursive functions. The most essential parts, however, are the operator semantics which are given as lambda terms. This module also implements a class called *VariableManager* which handles the assigning of fresh variable names when instantiating functions.

constant_util.py Offers two functions. *to_value* is used to extract the actual value (e.g. a natural number) of an instance of a *Constant* object. That's necessary when evaluating terms. Its counterpart *to_constant* takes a value and puts in a Constant object.

function.py Comprises the definitions of all functions which have been investigated so far.

generalizer.py Contains the *anti_unify_and_match* function which is the core function of the program. It implements the whole workflow of unfolding, anti-unifying and matching of two function instances. After matching the inferred recursive scheme is tested.

term_generator.py Contains the important function *get_verbose_solution_term* which takes a function object (i.e. an instance of one of the functions in *functions.py*) and produces the unfolded initial term.

term_visualize.py Offers the *create_tree_graph* function which creates a Dot graph for any term passed as parameter.

sigma.py The *Sigma* class which is defined in this module represents one substitution of one subterm (coming from the base term) with another (coming from the target term) as a result of anti-unification. Additionally this module offers functions which are used to manage so called sigma-stores, which are lists of Sigma objects.

parsing.py Handles the parsing of strings into terms, i.e. from string representations of terms which are used for example in anti-unification into instantiated tree structured terms.

matching.py Contains the functions which are used for executing the matching between base and target terms and there fore the creation of inferred recursive schemes. The essential function is *execute_matching*.

utility.py This module offers functions for various tasks: reading and writing to files, convenience functions for the creation of special HTML tags in the context of the HTML output mode and a function for initiating the Anti-Unification which is handled by the corresponding Java library. Also encompasses the function *eval_term* which evaluates the term passed as parameter. This function can be used to compute the result of a solution term.

This concludes the overview of the general program architecture, the next section will briefly state how the program can be run and extended with new functions.

B.2. Execution and Extension

B.2.1. Execution

For separating the code for algorithms from the code which actually runs the program, it is a good idea to create a new file for the code which actually runs the anti-unify and match process. As part of the program code which is included on the CD, there are already python modules which provide examples for execution. The *main.py* module contains a very simple example for anti-unifying and matching two functions:

```
from function import *
from generalizer import anti_unify_and_match

anti_unify_and_match(FacFunction(), SumFunction())
```

The first two statements import relevant classes and functions for running the program. As was stated in the previous section, the *function* module in file *function.py* contains all defined functions which are currently available. The *anti_unify_and_match* function, which is imported from the *generalizer* module, takes two functions as arguments.

When instantiating function objects, the user can add parameters to the function. For reasons of universal applicability the parameters are given as a list, even when a function takes only one parameter, like the *FacFunction* as well as the *SumFunction* which both take only one argument. When a function is instantiated without passing a list of parameters as an argument, default parameters are used which are defined for the individual functions.

The following are examples for passing parameters when instantiating functions, function parameters must be given as Constant objects:

```
# natural numbers domain
SumFunction([Constant(1)])
PowerFunction([Constant(3), Constant(3)])

# list domain
UnpackFunction([Constant([1, 2, 3, 4])])
MemberFunction([Constant(2), Constant([1,2,3])])
```

Further examples can be found in other modules. The module *main_rec_schemes* instantiates all functions sequentially and creates tree graphs from those functions. The *worker_<FUN>* modules are used to automatically anti-unify and match all other functions with a certain function. In the *worker_sum* module, we try to infer the sum function from all other functions. The worker modules are especially useful for creating the HTML output which is introduced in chapter C.

B.2.2. Extension

The most natural possibility to extend the project is by including new recursive functions to investigate. This can easily be done in the *function* module. Existing functions can be used as a template for creating new functions. It is important to consistently rename all occurrences of all function short names which are used in the definition of a function class.

The most important method is the *__init__* method, which initializes all relevant data structures. In the following some important code statements are explained.

```
first_var = VariableManager.get_unused_variable_name()
second_var = VariableManager.get_unused_variable_name()
```

In order to get new and unused variables to use in a function, the *VariableManager* class is used. For each actual parameter a recursive function uses (e.g. one for the sum function and two for the member function), we create a new variable name.

```
self.initial_params = {
    first_var: params[0],
    second_var: params[1]
}
```

The variable names and parameter values are stored in the *initial_params* attribute of the function.

```
self.recursion['newfun'] = {}
self.recursion['newfun']['check_for_base_case'] = lambda self, xs: ...
# add scheme with given variable names
self.recursion['newfun']['scheme'] = IfThenElse(...)
```

The recursive *scheme* and *check_for_base_case* is put in the *recursion* attribute. The check for base case must be a lambda expression which has one input: a list of values. This lambda expression is always called with the parameters of the current recursive call. It must return true when the base case is reached for the given parameters.

```
self.recursion['newfun']['test_info'] = {
    'rec_vars': [first_var, second_var],
    'values': {first_var: copy.deepcopy(params[0]),
```

```
                second_var: copy.deepcopy(params[1])
    }
}
```

For testing the resulting inferred functions, the assigned variable names and copies of the initial parameters must be stored separately.

```
self.variable_names = []
self.variable_names.append(first_var)
self.variable_names.append(second_var)
```

As *dictionary* objects in Python do not save the order of key and value pairs, the names of assigned variables must also be put into a list.

The other methods which are present in a function class are all straightforward and implemented identically for each function.

For testing a newly defined function and its implementation some of the following functions are useful:

```
fun = NewFunction()
rec_def = fun.recursive_definition()
print(rec_def)
verbose_sol = get_verbose_solution_term(fun)
print(verbose_sol)
print(eval_term(verbose_sol)
```

This snippet creates an instance of the *NewFunction* function with its default parameters. Then the recursive definition of the function is printed to the terminal. The solution term for the corresponding parameters is created and printed. The last line evaluates the verbose solution term and prints the result to the terminal. Therefore, we can check whether the semantics of the function are correct or more fine-tuning is required.

For extending or adjusting the matching process, changes should be done in the *matching* module. Promising starting points are the *match_recursive_functions* and the *execute_matching* functions. For tracking and understanding the program flow, the generalizer module's anti_unify_and_match is the best bet.

C

HTML Output

By setting the corresponding in the local_config module to *True*, our program will output the processing information enriched with html tags which allow for a neat and more comprehensible representation. The HTML files are created according to a canonical naming scheme.

- The entry point is the *index.html* file. It is basically a list of links to all investigated function.
- When clicking a link to a function in the index file, we get to a detail view for a that function as target function, i.e. after clicking on the *Sum* function in the index file, the corresponding detail view offers a list of all other functions which have been anti-unified and matched with the *Sum* function. Each list entry contains the resulting inferred recursive scheme and states whether the inferred result is correct.
- Clicking on a link in the detail view, loads a page which offers the anti-unification and matching process for the selected pair of functions in detail. This content is basically the same as the output of the *anti_unify_and_match* for these two functions.

Figure C.1 depicts snippets from those three types of output files. The detail view for a function and the result of the anti-unification and matching view offer backward links to the next higher level of the hierarchy to facilitate navigation.

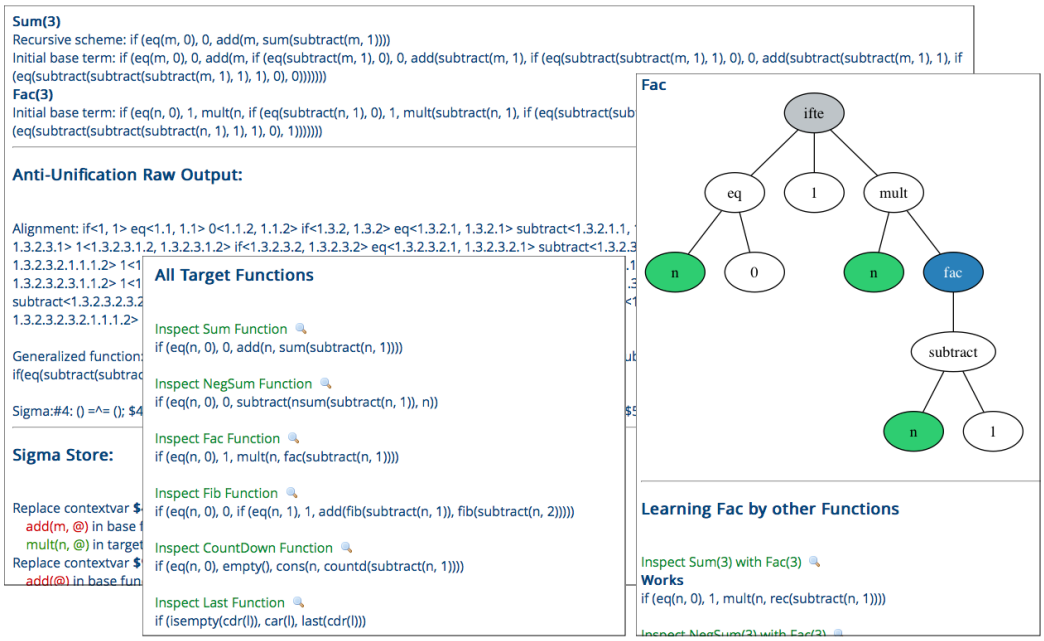


Figure C.1.: Three different kinds of content of the HTML output: index, view for faculty function and detail view for faculty and sum functions

All HTML files for all pairs of investigated functions, as well as function detail and index files are included on the enclosed CD in directory *html_ output*.

For recreating all HTML files from scratch, the *main_ rec_ schemes* and *worker* modules must be executed. For running all worker modules in parallel the shell script *worker.sh* in the program directory can be employed.

D

Content of the CD

The CD attached to the inner side of the back cover contains the following artifacts:

- This document:
  - Compiled PDF file
  - LaTeX source files, bibliography file and pictures (main .tex file: *main.tex*)
  - PDF files of most bibliography items and captures of URLs mentioned in footnotes.
- Java source code
  - `at.jku.risk.stout.urauc` package containing the source files of the anti-unification library version used
  - `de.uniba.kogsys.boosz.au` package containing the java files for the interface to the AU library we built.
- Python source code
  - Contains the executable anti-unification executable (*anti_unify.jar*)
- HTML output files for all function pairs
  - Entry point: *index.html*

Ich erkläre hiermit gemäß § 17 Abs. 2 APO, dass ich die vorstehende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Datum Unterschrift