Solving Proportional Analogies by Application of Anti-Unification modulo Equational Theory

Bachelor's Thesis

Stephan Weller <stephan.weller@mi.uni-erlangen.de>
Cognitive Science Program
University of Osnabrück

Supervised by Prof. Dr. Ute Schmid, University of Bamberg
Co-Supervised by PD Dr. Helmar Gust, University of Osnabrück

June 28, 2005

Abstract This thesis applies the theory of Anti-Unification modulo Equational Theory (E-Anti-Unification, E-Generalization) to proportional analogies in the string domain. Human solving of proportional analogies of the form $A : B :: C : D$ (read: A is to B like C to D, where A, B and C are given) is modelled by an approach originally developed by Heinz [1996], refined in Burghardt [2005]. The goal is to generalize the common structure of the terms given and infer *every* possible value for D. This is made possible by the use of regular tree grammars instead of single terms. Thus, a formally sound and powerful approach to modelling human solving in this domain is achieved. Some heuristics' for selecting terms from the computed regular tree grammar are proposed. The selection of a heuristics which comes closest to human solving is discussed, however a final answer to this question can only be given by empirical investigations.

Contents

List of Figures

1 Introduction

The role of analogy-making in human reasoning is not well understood. However, it is generally assumed, that analogies do play an important role in human cognition and knowledge acquisition.

Finding an analogy is a process that requires abstraction, which is one of the core features that constitute intelligence. Thus, reasoning by analogies is one of the fields that need to be investigated to understand human intelligence as such.

Analogical reasoning has been a subject to intense study in psychology, Artificial Intelligence, Cognitive Science and other fields. A general introduction to Analogical Reasoning and various types of analogies can be found in section 2.

Some of the classical approaches include analogies in verbal settings (*Lungs are to humans as gills are to fish*, cf. Evans [1968] and O'Hara [1992]). I will describe some classical approaches in more detail in section 2. The main focus will be on proportional analogies, as the main topic of this thesis is solving proportional analogies algorithmically.

Hofstadter was the first one to investigate proportional analogies in string domains (Hofstadter and the Fluid Analogies Research Group [1995]). The type of analogies he investigated is exactly the type of analogies used in my thesis. The typical form of such a "string analogy" would be $abc : abd :: ghi : ?$, with ghj being the most probable answer. Hofstadter's *Copycat*-Model will be described in section 2.2.

To compute an answer to a proportional string analogy, following Schmid et al. [2004], it is necessary to find a description d for each part A, B, C, D of the analogy $A : B :: C : D$, such that $\mu(f(d(A))) = \mu(d(B)) = d(D) = f'(d(C)) = f'(\mu(d(A)))$, where μ is a mapping from one domain to another and f and f' are functions to transform (in our case) strings to new strings, i.e. they are representing the ':'-operator.

To achieve this aim, I follow the approach of Schmid et al. [2004] and apply the method of Anti-Unification modulo Equational theory, developed by Burghardt and Heinz [1996] and Heinz [1996]. I use the refined and more elegant version of their work found in Burghardt [2005]. By this, I achieve a method of extracting from the common structure between A and C a mapping that can be applied to transform B to D and thereby solve the analogy task.

The idea behind E-Generalization and Anti-Unification in general will be described in section 3. Afterwards I apply the method of E-Generalization to proportional analogies (section 4). Finally, I describe my implementation used to solve some exemplary proportional analogies (section 5).

Section 6 contains a conclusion of the thesis as well as a list of further work.

2 Analogical Reasoning - some classical approaches

The basic idea of an analogy is the use of old, known information to explain new facts or experiences, or to transfer knowledge to a new, similar situation. In human cognition, this is an every-day phenomenon, that occurs all of the time.

Much research has been done in psychology, Artificial Intelligence, and other fields concerning various kinds of analogies, reaching from examples like analogies at court used in legal reasoning to analogies in poems, or even metaphors, which some authors consider to be a form of analogy.

Analogies often occur in natural language ("Gills are to fish as lungs are to mammals"), but can also be used in completely different fields. For example, Davies and Goel [2001] use analogies in problem solving. Subjects were presented with one solved problem about a fortress. All roads to the fortress were mined in a way such that the mines would go off if there were too many people on the same road. Thus, a general attacking the fortress split up his army into small groups to attack the fortress. Afterwards, the subjects were presented with a new problem, that showed some analogous features to the old problem. A brain tumor was to be removed by radiation treatment, and the radiation would harm healthy tissue on its way, when it was too strong. The subjects were asked to solve this problem and were usually able to use the fortress problem to produce an analogous solution, i.e. to target the tumor with many low-level rays from various directions.

At a more formal level, two types of analogies are of special interest. The probably most basic type of an analogy is a so-called *proportional analogy*. This kind of analogy has the form "A is to B, like C is to D". The example in natural language quoted before is of this form. To make formal treatment easier, a special type of proportional analogies has been investigated, namely proportional analogies in the string domain. In this case, A, B, C, and D are strings. The analogy is usually given as a problem that is to be solved. This is done by specifying A, B, and C and asking for D. The main focus of this thesis is to show an algebraic method to compute D from A, B, and C.

Our running example will be the following:

$$abc : abd :: ghi :?$$

It is to be read like "abc is to abd like ghi is to what?". The solution most likely given by humans would be ghj, however, other solutions are possible, for example one could argue that ghd is the answer, as the last letter is replaced by d on both sides. It is typical for proportional analogies in general, that multiple possibilities exist rather than one unique solution.

The probably most important classical approach to proportional analogies in the string domain is Hofstadter's Copycat model, which will be described in section 2.2.

Another important form of analogies are *predictive* analogies. Such analogies occur for example in teaching situations and are the most important form of analogical reasoning. The most prominent example is the so-called *Rutherford-analogy*. In the model of the atom from Ernest Rutherford, the atom is seen like a planetary system. The electrons are revolving around the nucleus, which is analogous to the planets revolving around the sun. This example was investigated by Gentner [1983].

There is an approach to solve predictive analogies in an algebraic way, which uses the same method, namely Anti-Unification, as this thesis uses to solve proportional analogies. It is called *heuristic-driven theory projection* and can be found in Gust et al. [2004].

2.1 ANALOGY

One of the very early approaches to model analogies is Thomas Evans' program ANALOGY (cf. Evans [1968]). It used a microdomain of geometrical figures found in some intelligence tests. All tasks were of the form of a proportional analogy. The answer was not to be given but to be selected from a list of some possibilities. A task would contain pictures A, B, C and $1-5$ and the question would be *A is to B like C to 1,2,3,4 or 5?*. The figure B was always derived from A by some transformation like moving, deleting or replacing some object or the like. ANALOGY then selected the answer from the given list of five candidates. The selection was based upon an ordering of the five possibilities.

The pictures were given to ANALOGY in the form of Lisp data-structures describing geometrical units like lines, curves and dots. ANALOGY built its *own* representations from the given pictures.

The processes of building up the representations and computing the rank of the five possibilities were in separate stages, which might be seen as a disadvantage, as it was impossible to review decisions in the representation stage while computing the solution.

Although Evans' model was rather primitive compared to more recent approaches, it is still an interesting approach to analogical reasoning, which was unfortunately not really continued by any follow-up projects.

2.2 Copycat

The Copycat model was developed by Hofstadter already in 1983 (cf. Hofstadter and the Fluid Analogies Research Group [1995]). Copycat is a computer program which was supposed to find "insightful analogies, and do so in a psychologically realistic way". Hofstadter calls it a "model of mental fluidity and analogy making". The domain used for the Copycat project is the string domain, however Hofstadter claims that the architecture models fluid concepts in general and that the microdomain of Copycat was designed

to model other domains (e.g. a successor relation in the string domain for *any* non-identity relationship). The goal of Copycat is clearly not research in the string domain but rather exploration of general issues of cognition.

At the core of the Copycat architecture lies the idea of "conceptual slippage". Hofstadter uses this term to describe that concepts relatively close to each other may swap their roles when under conceptual pressure. The concept "predecessor" might for example slip to "successor" when the context forces it. This might for example be used to explain that a result might slip from *abc* to *cba*.

The Copycat architecture consists of three major components called slipnet, workspace and coderack.

The slipnet can be thought of as something like a long-term memory of Copycat. It consists of all permanent concept types, but no instances of those. The distances between those concepts in the slipnet are variable, and determine how likely conceptual slippage is.

The workspace is something like a short-term or working memory containing instances of the concepts stored in the slipnet, which are combined into temporary perceptual structures.

The Coderack is something like an agenda storing agents that want to carry out tasks in the workspace. It is now very important that agents are selected *stochastically* from the coderack, not in a determinate order.

Hofstadter claims that Copycat has one main advantage over most other models. Whereas most models concentrate on a mapping from a source problem to a target problem. The representation of this source and target is usually given to this models. In contrast to this, Copycat focuses on the construction of representations for source and target. Thereby, Copycat even allows for interaction between this construction and the mapping process. Thus, Hofstadter claims, Copycat integrates high-level perception and the mapping of concepts.

However, in my point of view, there is one big disadvantage not only of Copycat, but also of other, similar models. The perception, and also the mapping between concepts may be modelled adequately by such an approach. But what is missing, is the abstraction occurring in the process of solving a proportional analogy. Basically, when Copycat, or another system, has solved an analogy and is then given a new, but similar one, the process of solving the second analogy works exactly as the one for the first analogy. This is of course not true for human solving of proportional analogies. In humans, abstraction necessarily occurs in drawing analogies. However, no model known to me incorporates this abstraction in the process of drawing analogies. As we will see later on, one big goal of this thesis is to avoid a direct mapping approach and rather use an indirect mapping *via abstraction*, such that the abstraction will be a byproduct of the process of solving analogies.

Hofstadter seems to attach great importance to his stochastic selection

procedure, however, in my opinion this is another shortcoming of Copycat. Humans, asked to solve proportional analogies, will usually be able to explain *why* they have selected a certain answer and not another one. Provided that a human subject answers seriously, he would always be able to give some kind of "rule", that leads to the selection of a result (although in some cases there might be problems to verbalize this rule). In my opinion, a stochastic procedure can in this case not contribute to our understanding of human cognition.

2.3 Structural Information Theory

Leeuwenberg [1971] first introduced *Structural Information Theory (SIT)*, a coding system for linear one-dimensional patterns. Perceptual structures are represented by three operators named *iteration, symmetry,* and *alternation.* Iteration is supposed to reflect the repetition of something, for example $Iter(xy, 3) := xyxyxy$. Symmetry may occur in two variants, odd and even, and describes the reversed repetition of a term t after a second term s. An example would be $Sym_{\text{even}}(xyz, ()) = xyzzyx$ and $Sym_{\text{odd}}(xy, z) = xyzyx$. Alternation may occur in a left and a right variant. It describes the interleaving of a term into a list of terms, such as $Alt_{\text{right}}(a, (x, y, z)) = axayaz$ and $Alt_{\text{left}}(a, (x, y, z)) = xayaza$.

On those operators, *information load* is introduced. This information load is supposed to describe how complex the application of an operator is. Leeuwenberg claims that the descriptions using the least information load correspond to perceptual gestalts (The notion "gestalt" is quite common in cognitive psychology. For an introduction, see for example Goldstein [1980]). He therefore claims that the gestalt principle can be explained by even simpler principles, such as his information load.

A more formal version of Structural Information Theory can be found in Dastani et al. [1997]. Here, an algebraic version of SIT is defined and proportional analogies are formalized upon it. Even some computational modelling of proportional analogy is done.

3 Introduction to Anti-Unification

This chapter will deal with Anti-Unification, which will constitute the core part of our approach to solving proportional analogies.

To understand the concept of Anti-Unification, I will shortly introduce the term of unification, which is widely used in Computer Science, especially in logical programming, and in other fields. I will then explain, how usual, syntactic Anti-Unification works, and finally, how Anti-Unification can be extended by an equational theory. It is then called E-generalization.

3.1 Unification

When unifying two (or more) terms, the aim is to compute the *most general unifier* (*MGU*), i.e. the most general term, such that both terms can be reduced to the *MGU* by inserting terms for variables or by renaming variables (cf. figure 1) [1] .

In this example, the terms $x + 5 \cdot 3$ and $17 + 5 \cdot z$ are unified. As x and z are variables, we can use substitutions to replace x by 17 and z by 3, in both cases the result is $17 + 5 \cdot 3$. Thus, $17 + 5 \cdot 3$ is called a unification of $x + 5 \cdot 3$ and $17 + 5 \cdot z$.

$$\begin{array}{ccc} x + 5 \cdot 3 & & 17 + 5 \cdot z \\ (x \leftarrow 17) \searrow & & \swarrow (z \leftarrow 3) \\ & 17 + 5 \cdot 3 & \end{array}$$

Fig. 1: Example for Unification

Speaking formally, we define a substitution $\Theta = \{v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n\}$ as a *unifier* of expressions $E_1, \ldots, E_n$, if it holds that applying this substitution always yields the same result, i.e. $E_1\Theta = \cdots = E_n\Theta$. The *MGU* is then defined as a unifier Θ, such that for all other unifiers σ of $E_1, \ldots, E_n$ there exists a substitution γ, such that $\sigma = \Theta \circ \gamma$. It should be noted that an *MGU* is defined uniquely except for variable renaming and that an *MGU* does not necessarily exist.

Unification has many applications, the probably most important being inference algorithms for first-order predicate logic (FOL). In this field, unification can be used to determine the correct substitution when dealing with quantified variables. Most logical programming languages like Prolog use proofs by resolution, which is based on unification.

Detailed introductions to Unification can be found in most textbooks about theoretical computer science, confer for example Schöning [1989].

3.2 Universal Unification

The short introduction to unification was mainly given to facilitate the understanding of Anti-Unification in the following sections. As we will see, syntactic Anti-Unification, which is the dual concept to unification, will not suffice for our purposes, and it will be necessary to incorporate knowledge from equational theories in the process. Something similar can also be done to unification. It is indeed possible to use an equational theory, such that

[1] The operators $+$ and $\cdot$ are used in the usual sense, that is, $a + b \cdot c$ is to be read as $+(a, \cdot(b, c))$.

unifying two terms a and b is possible although their structure bears no resemblance, if $a =_E a'$ and $b =_E b'$[2]in the theory and b and b' can be unified in the normal way. Of course, this requires new algorithms.

As Unification is extensively used in many knowledge based systems in Artificial Intelligence, there are many applications of such "General Unification". An extensive summary of classical approaches in this field can be found in Siekmann [1984].

3.3 Syntactic Anti-Unification

Anti-unification is the dual concept to unification. Where unification looks for the most general unifier, Anti-Unification computes the most specific generalization of two (or more) terms. Anti-unification basically tries to extract the common structure between terms. The aim is to construct a new term which incorporates all common information from the anti-unified terms and marks all differences by variables. The result of the Anti-Unification of expressions $E_1, \ldots, E_n$ is therefore an expression E, such that substitutions $\sigma_1, \ldots, \sigma_n$ exist for which it holds that $E\sigma_i = E_i$ for all $i \in \{1, \ldots, n\}$.

In contrast to unification, Anti-Unification is always possible, at least the trivial Anti-Unification x, where $\sigma_i = \{x \leftarrow E_i\}$ for all $i \in \{1, \ldots, n\}$ does always exist and one can also always find one most specific one (except for variable renaming).

Consider the example given in figure 2, where the Anti-Unification of the two terms $13 + 5 \cdot 7$ and $17 + 5 \cdot 9$ is depicted. Anti-Unification replaces the numbers different on both side by variables x and y. The Substitution $\sigma = \{x \leftarrow 13, y \leftarrow 7\}$ would transform the result to the first term and another substitution $\sigma' = \{x \leftarrow 17, y \leftarrow 9\}$ would yield the second one.

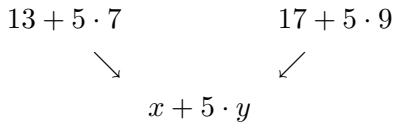

Fig. 2: Example for syntactic Anti-Unification

At this point, it is important, to note one thing: The Anti-Unification of the two terms is merely a syntactic process. I used an example from the domain of mathematics, however, the symbols + and · do not "mean" anything in this context. You could use arbitrary operators instead. Nothing about these operators needs to be known to use Anti-Unification, however on the other hand no knowledge about the operators *can* be used in syntactic Anti-Unification. Consider for example replacing the second term by 5·9+17.

[2] Here $=_E$ denotes equality in an equational theory, in contrast to syntactic identity

Assuming usual laws of commutativity, this term would not at all differ from the one used before in a mathematical sense. However, the Anti-Unification would in this case yield only the solution $x+y$, where $\sigma = \{x \leftarrow 13, y \leftarrow 5 \cdot 7\}, \sigma' = \{x \leftarrow 5 \cdot 9, y \leftarrow 17\}$. To put it in other words: The use of background knowledge of any kind is impossible in classical Anti-Unification. Of course one could consider to change the algorithm used for Anti-Unification and to give it the ability to obey laws of commutativity and associativity. This has indeed been done, cf. for example Pottier [1989]. However, approaches like this can only account for very special background knowledge. To solve other problems, namely that of solving proportional analogies by Anti-Unification, far more general background knowledge needs to be integrated. To tackle this problem, E-Generalization can be used.

Algorithms for computing the Anti-Unification for n terms effectively were written by Plotkin [1970] and Reynolds [1970] independently. Confer figure 3 for a simplified version of the algorithm for two terms, quoted freely from Reynolds [1970].

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

where φ is a newly introduced variable which maps to x under σ and to y under σ'

Fig. 3: Algorithm for syntactic Anti-Unification of two terms

3.4 E-Generalization

As we have seen in the last chapter, to use Anti-Unification for the purpose of computing proportional analogies, we need to integrate background knowledge into the Anti-Unification process. Heinz [1996] developed a way to use background knowledge in the form of canonical equational theories, i.e. theories that use a term rewrite system which is ground-confluent and noetherian (well-founded). In those theories, terms can be put to a unique normal form.

Heinz [1996] first develops a method for enumerating all generalizations to a given pair of terms. However, depending on the background theory, usually the resulting set will be infinite. Given that the theory allows for

a normal form, all rules from the background theories have to be read in only one direction, however, this still allows for infinitely many (countable) terms as a result. This method from Heinz is therefore not of interest for a practical application.

What is of more interest to us, is the question when terms and their generalization with respect to a canonical equational theory can be represented in a closed form.

The idea proposed by Heinz [1996] is to represent every term as a regular tree grammar which are defined by figure 4. The idea is then to anti-unify not terms, but complete regular tree grammars. The resulting grammar should then be the equivalence class of all terms that can be achieved anti-unifying every pair of input terms. This technique of anti-unifying complete grammars and not only terms is then referred to as *E-Generalization*.

I will not use the original algorithm from Heinz [1996] but rather an improved form from Burghardt [2005], which fulfills the same task in a far more elegant and less task-specific way.

Definition of regular tree grammars:
A regular tree grammar is a quadruple $\mathcal{G} = (\Sigma, \mathcal{N}, S, \mathcal{R})$. Σ is a signature, i.e. a set of function symbols f, where each f has a fixed arity, if the arity is 0, f is called a constant. $\mathcal{N}$ is a finite set of Nonterminals. $S \in \mathcal{N}$ is a starting symbol. $\mathcal{R}$ is a finite set of rules of the following form:

$$\mathcal{N} ::= f_1(N_{11}, \ldots, N_{1n_1}) | \ldots | f_m(N_{m1}, \ldots, N_{mn_m})$$

Fig. 4: Definition of regular tree grammars as used by Burghardt [2005], freely adopted

The class of regular grammars was developed in 1968 (Brainerd [1968] and Thatcher and Wright [1968]). It is located in the Chomsky-Hierarchy (cf. Hopcroft and Ullman [1979]) between regular and context-free languages. It is more general than regular languages, however it is closed against intersection, complement and union, like regular languages. Algorithms for determining intersection, complement and union, as well as a very deep and comprehensive introduction to regular tree grammars can be found in Comon et al. [1997].

3.5 An Example

For the sake of brevity, let us consider the very simple equational theory given in figure 5. It contains just rules for the commutativity of addition

and multiplication.

$$E_1 : x + y = y + x$$
$$x \cdot y = y \cdot x$$

Fig. 5: Equational theory for Example 7

If we now consider the example given before, namely the two terms $5 \cdot 9 + 17$ and $13 + 5 \cdot 7$, we have to find regular tree grammars $\mathcal{G}_1$ and $\mathcal{G}_2$ for those terms with respect to the given theory. Such tree grammars are given in figure 6[3].

$$\begin{aligned}
\Sigma &= \{+/2, \cdot/2, 5/0, 7/0, 13/0, 17/0\} \\
\mathcal{N} &= \{t, t_{5\cdot 9}, t', t_{5\cdot 9}\} \\
\mathcal{R} &= \{ \\
t &::= +(t_{5\cdot 9}, 17) | + (17, t_{5\cdot 9}) \\
t_{5\cdot 9} &::= \cdot(5, 9) | \cdot (9, 5) \\
t' &::= +(13, t_{5\cdot 7}) | + (t_{5\cdot 7}, 13) \\
t_{5\cdot 7} &::= \cdot(5, 7) | \cdot (7, 5) \\
&\} \\
S &= t && \text{for the term } 5 \cdot 9 + 17 (\mathcal{G}_1) \\
S &= t' && \text{for the term } 13 + 5 \cdot 7 (\mathcal{G}_2)
\end{aligned}$$

Fig. 6: Regular tree grammars for Example 7 wrt. to the theory 5

Our aim is now to compute the Anti-Unification of the two given terms modulo our theory. Burghardt [2005] states a suitable algorithm for doing so. I will describe it in section 4.1 in detail. We can thereby compute a regular tree grammar describing exactly the Anti-Unification of our given terms. Following our example, the Anti-Unification of our two terms now delivers a grammar containing among others the term $x + 5 \cdot y$ (cf. figure 7). We have therefore achieved our goal and are able to account for our background knowledge and, what is important, did *not* incorporate it in our algorithm.

[3] Here φ/n denotes a function symbol with name φ and arity n

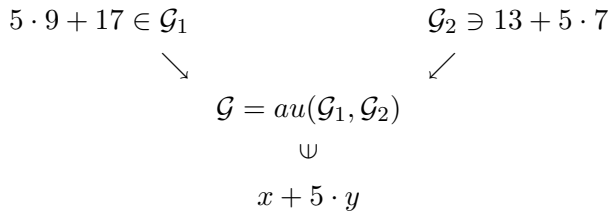

Fig. 7: Example for E-Generalization

3.6 Drawbacks

In the description of the procedure given before, two steps are problematic. The first problem occurs when the regular grammars need to be generated from the background theory. For some special cases, algorithms are known, however, a generic method is not yet available.

Emmelmann [1991] gives criteria for a certain class of canonical equational theories which can be used to generate regular tree grammars. The same work also states an algorithm for the generation of those grammars, however, an application of those grammars cannot be done easily and does also not solve the problem for *all* canonical equational theories. About the problem of grammar generation, confer also my comments in v. Thaden and Weller [2003].

Currently, for some cases automatic or at least semi-automatic methods for generating tree grammars can be given, for example for some part of arithmetics (see v. Thaden and Weller [2003]). For all other cases, unfortunately the grammars have to be generated by hand.

The second problem occurring in our procedure, is the enumeration of our resulting grammar. Regular grammars do usually describe an infinite set of terms. If a result giving a concrete term is desired, it is not a priori clear, which term has to be selected from the regular grammar. This question becomes extremely important when applying E-Generalization to solve analogies. In section 4.4 we will see that in this case this is exactly the point where human cognition is concerned most. As long as no deep understanding of human representation of analogies is available, we can then answer this question only empirically.

3.7 Other applications of E-Generalization

In this work it is intended to use E-Generalization to compute a solution of proportional analogies. However, there are plenty other applications possible.

Heinz [1996] mentions especially the application in lemma generation. In automatically generated proofs often situations occur, where lemmata are needed to complete a proof. In such situations a proof can only be completed when a given problem is generalized and then solved. E-Generalization can be used in this case to generate lemma candidates.

Another possible application, which is closer to the solution of proportional analogies, is the completion of number series. This approach, as undertaken first by Hofstadter and the Fluid Analogies Research Group [1995] (cf. chapter 2.2), also makes use of analogies. The application of E-Generalization to this problem has been done in v. Thaden and Weller [2003].

4 Applying E-Generalization to solve proportional analogies

In this section I will describe how we can use E-generalization to solve proportional analogies. This will be done in several steps. I will first describe how to incorporate knowledge about a substitution into a grammar, also called "lifting". I will then show how we can use this method to obtain an Anti-Unification of two grammars in the case that the substitutions are already known. If suitable substitutions are not known beforehand, I will describe how to obtain *universal* substitutions that can take their place, such that an algorithm for anti-unifying grammars also without prior knowledge is derived.

In section 4.2 I will then describe how to apply all those steps to solve a proportional string analogy.

4.1 Algorithms for E-Generalization

4.1.1 Constrained E-Generalization

If we allow grammars to contain also variables, which are treated like constants (i.e. nullary functions), we can apply substitutions to grammars. I will use the notation $G\sigma$ to describe the result of applying the substitution σ to the regular tree grammar G.

To anti-unify two grammars G and H, our goal is to find two substitutions σ_1 and σ_2 and a grammar Q, such that $Q\sigma_1 = G$ and $Q\sigma_2 = H$. Constrained E-Generalization is constrained in the sense that the substitutions σ_1 and σ_2 have to be known beforehand. In some applications this is the case, cf. Burghardt [2005] or Heinz [1996] for examples. One example of such an application would be "number series guessing", see v. Thaden and Weller [2003] for details.

The idea of constrained E-Generalization is now to apply the substitutions "inversely" and intersect the resulting grammars.

Now suppose, two substitutions σ_1 and σ_2 and a variable set V with dom σ_1[4] = dom $\sigma_2 = V$ is given. In the first step we therefore compute two grammars G^{σ_1} and H^{σ_2}, such that $G^{\sigma_1}\sigma_1 = G, H^{\sigma_2}\sigma_2 = H$ and every "suitable" variable is introduced into the grammar. The algorithm is given in figure 8.

The second step is to intersect the two new grammars. This is done in a straightforward way, the algorithm is shown in figure 9. A more general version of this algorithm can be found in Comon et al. [1997].

> For a regular tree grammar $\mathcal{G} = (\Sigma, \mathcal{N}, S, \mathcal{R})$ and a substitution σ we define a new grammar $\mathcal{G}^\sigma = (\Sigma \cup \text{ dom } \sigma, \{N^\sigma | N \in \mathcal{N}\}, S^\sigma, \mathcal{R}^\sigma)$, where N^σ is a new nonterminal, one distinct nonterminal is introduced for each old nonterminal. The same is done to S, and the rules $\mathcal{R}^\sigma$ are derived from $\mathcal{R}$ as follows:
> For every rule
>
> $$N ::= \Big|_{i=1}^{m} f_i(N_{i1}, \ldots, N_{in_i})$$
>
> from $\mathcal{R}$ we introduce a new rule
>
> $$N^\sigma ::= \Big|_{i=1}^{m} f_i(N_{i1}, \ldots, N_{in_i}) \Big|_{x \in \text{ dom } \sigma, x\sigma \in \mathcal{L}_\mathcal{G}(N)} x$$
>
> Where $\mathcal{L}_\mathcal{G}(N)$ describes all terms in the grammar $\mathcal{G}$, that can be reached when using N as a starting symbol.

Fig. 8: Lifting algorithm, from Burghardt [2005]

With those steps, we can now anti-unify two grammars, provided that we already have the substitutions σ_1 and σ_2.

4.1.2 Unconstrained E-Generalization

We now consider a situation where nothing is known about substitutions, only two grammars $\mathcal{G}_1$ and $\mathcal{G}_2$ are given. In Heinz [1996], a new, monolithic algorithm for this situation was used for this purpose, however Burghardt [2005] proposes another, more slim alternative. His idea is to compute two universal substitutions τ_1 and τ_2 and then apply the algorithm for constrained E-Generalization, using those substitutions. Figure 11 shows how to compute those substitutions.

Burghardt [2005] proves the existence of universal substitutions, where "universal" is defined as follows: For any two substitutions σ_1 and σ_2 we can find a substitution σ, such that it holds that for all terms both from the domain of σ_1 and the domain of σ_2 and for all nonterminals from the

[4] dom σ denotes the domain of σ, i.e. the set of all terms occurring on the left-hand side of a substitution

Let two grammars $\mathcal{G}_1 = (\Sigma_1, \mathcal{N}_1, S_1, \mathcal{R}_1)$ and $\mathcal{G}_2 = (\Sigma_2, \mathcal{N}_2, S_2, \mathcal{R}_2)$ be given. We presuppose $\Sigma_1 = \Sigma_2, \mathcal{N}_1 = \mathcal{N}_2, \mathcal{R}_1 = \mathcal{R}_2$. This can always be achieved by renaming and then using the disjoint union of the parts of the grammar. We now define a new grammar $\mathcal{G} := \mathcal{G}_1 \cap \mathcal{G}_2 := (\Sigma_1, \mathcal{N}, S, \mathcal{R})$. $\mathcal{N}$ is a set of new nonterminals and $\mathcal{R}$ a new set of rules, that are defined as follows:
We run the following algorithm `intersect` on the nonterminals and rules, starting with S_1 and S_2 as arguments and S as the first newly introduced nonterminal.[5]
Let s, s' be nonterminals and θ a newly introduced nonterminal. Rules are added by the following algorithm `intersect`. To compute `intersect`, the following steps are applied:

1. If `intersect`(s, s') has been called before, no new rule needs to be added.
2. If $\mathcal{R}_1$ contains a rule $s ::= s_1 | \ldots | s_n$,
   add the rule
   $\theta ::=$ `intersect`$(s_1, s') | \ldots |$ `intersect`(s_n, s')
   to $\mathcal{R}$
3. If $\mathcal{R}_1$ contains a rule $s' ::= s'_1 | \ldots | s'_n$,
   add the rule
   $\theta ::=$ `intersect`$(s, s'_1) | \ldots |$ `intersect`(s, s'_n)
   to $\mathcal{R}$
4. If $\mathcal{R}_1$ contains rules $s ::= \varphi(s_1, \ldots, s_n)$ and $s' ::= \varphi(s'_1, \ldots, s'_n)$,
   add the rule
   $\theta ::= \varphi($`intersect`$(s_1, s'_1), \ldots,$ `intersect`$(s_n, s'_n))$
   to $\mathcal{R}$
5. Else θ is $\bot$, eliminate θ from all rules and for every rule that is now empty, remove its head.

Fig. 9: Algorithm for grammar intersection

grammar we have $t\sigma_i \in \mathcal{L}(N) \Rightarrow t\sigma\tau_i \in \mathcal{L}(N)$ for $i = 1, 2$.

We therefore have a sound method to compute the Anti-Unification of two grammars.

It is important to note one thing at this point. The "lifting"-algorithm and the grammar intersection are relatively efficient. The lifting-step can be done in time $O(ns)$, where n is the number of occurring nonterminals and s

[5] Note, that to use such rules we have to widen the definition from Fig. 4 to allow for nonterminals on the right-hand-side of the rules. This comes closer to the "old" definition in Heinz [1996]. This algorithm for intersecting was chosen for the sake of simplicity in the implementation. An alternative to this procedure is the use of the product automaton construction, found in Comon et al. [1997].
We also have to presuppose that the grammar is in a kind of normal form, such that the right hand side of all rules either consists only of nonterminals or of one function. This can be done by renaming easily, the algorithm can be found in Heinz [1996].

Let $\mathcal{N}$ be the set of all nonterminals.

1. Set $N = \emptyset$ and $\mathcal{N}_{\max} = \emptyset$.
2. For each Nonterminal $n \in \mathcal{N}$, compute $(\bigcap_{x \in N} \mathcal{L}(x)) \cap \mathcal{L}(n)$ and if the result is not empty, add n to N.
3. Add N to $\mathcal{N}_{\max}$, remove all elements in N from $\mathcal{N}$, and if $\mathcal{N} \neq \emptyset$, set $N = \emptyset$ and continue with step 2.

Fig. 10: Algorithmic computation of $\mathcal{N}_{\max}$

Define $\mathcal{N}_{\max}$ as in figure 10.
For a nonterminal N, define $t(N)$ as an arbitrary term from $\mathcal{L}(N)$.
For each pair $(N_1, N_2) \in \mathcal{N}_{\max} \times \mathcal{N}_{\max}$ do:
Introduce a new variable $v(N_1, N_2)$. Define $\tau_i = \{v(N_1, N_2) \leftarrow t(N_i)\}$ for $i = 1, 2$.

Fig. 11: Computation of Universal Substitutions, from Burghardt [2005]

is the total number of function symbols in the substitution. The grammar intersection is possible in time $O(n_1 n_2)$, where n_i are the numbers of nonterminals in both grammars respectively (cf. Comon et al. [1997] for details). However, the computation of the universal substitutions is very inefficient. As it requires the use of *every* subset of $\mathcal{N}_{\max}$, its computation time depends exponentially on the size of $\mathcal{N}_{\max}$. Burghardt [2005] gives an upper bound in the case that the grammar is deterministic. His upper time bound is a polynomial of $n_0 + 1$-th degree, where n_0 is the number of nonterminals representing one congruence class in the grammar. Burghardt states that in most applications n_0 does not exceed 1 and the E-Generalization can then be done in $O(n^2)$. However, for the general case, no such upper time bound can be given.

Thus, whenever it is possible to avoid the computation of universal substitution, this should be done. This is of course only possible if the application allows for some a priori knowledge about the substitution and, unfortunately, that is not always the case.

4.2 Solving proportional analogies

I will now describe how exactly to apply E-Generalization to solve proportional analogies. I will assume in this section, that the analogy to solve is of the form $A : B :: C : D$, i.e. A is to B, as C to D, where A, B, C are given and D is to be computed.

Figure 12 shows the basic steps of the procedure. First the algorithm from section 4.1.2 is used to compute the common grammar G_{AC} of A and C and, more important, the substitutions τ_1 and τ_2. Now, we compute a

grammar Q, such that $Q\tau_1 = B$, i.e. we apply the inverse of τ_1, we will call this σ_1^{-1}. This is done by the "lifting"-algorithm described in section 4.1.1. The substitution τ_2 is then applied to Q and yields a grammar describing possible candidates for D. The question of how to extract terms from this grammar will be discussed in section 4.4.

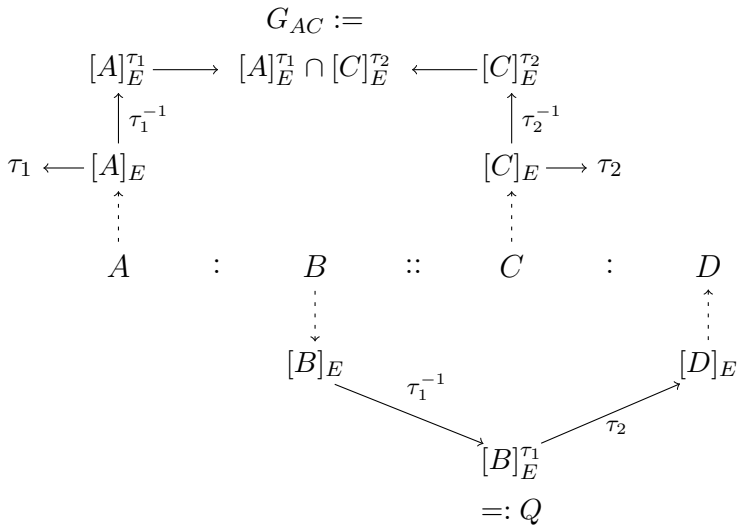

Fig. 12: Solving a proportional string analogy

I will now explain why this approach will yield a reasonable result.

In the first step, we anti-unify grammars describing the terms A and C. As a result, we get a grammar G_{AC} describing the "common structure" of A and C. We also get substitutions τ_1, τ_2, such that $G_{AC}\tau_1 = A$ and $G_{AC}\tau_2 = C$. Consider for example a variable (which would actually be contained in G_{AC} when using the example grammar from the Appendix) v_{ag} with $v_{ag}\tau_1 = a$ and $v_{ag}\tau_2 = g$. In some way this variable describes the role that a plays in abc and g play in ghi. We now want to find the "corresponding" variable in the grammar B describing abd. We do this by "lifting" the grammar B, such that it also contains for example a variable v_{a*} with $v_{a*}\tau_1 = a$. We now compare the variables in G_{AC} and those in Q. For every two variables x, y occuring in G_{AC} and Q respectively, with $x\tau_1 = y\tau_1$, we can apply τ_2 to y and will in thereby "do the same thing" to y that we did to x, i.e. we transform B in the same way we transformed A to get C – which is exactly what proportional analogies are supposed to do.

4.3 Serial solving of similar analogies

Within our approach, so far, it is important to notice, that the first step, namely to compute the universal substitutions of A and C, is a computationally very expensive one. However, it is also important to notice, that this step does *not* depend on B. Thus, to solve for example some tasks like $abc : abd :: ghi :?$, $abc : bcd :: ghi :?$, and $abc : cba :: ghi :?$, the substitutions have to be computed only once. The main reason for this is, that by anti-unifying A and C, we compute the common structure of both. This step of abstraction needs to be done only once. Actually, abstraction occurs in our method as a byproduct, which distinguishes our approach from most other approaches.

4.4 Enumerating regular tree grammars

We are now able to solve proportional analogies like $A : B :: C : D$ in a sense that we can get a grammar describing all possibilities for D. However, to compare our results to human problem solving, it is desirable to get just a few terms as a result.

Most humans would be able to give several different solutions, when asked to solve a proportional string analogy, and it would also be possible to compare the answers from several humans. Consider for example the analogy $abc : abd :: ghi :?$. The most common answer would probably be ghj, which can be explained by the fact that in abc c is the successor of b and in abd d is the successor of the successor of b, so we replace the successor of h in ghi by the successor of the successor of it, j. However, there are more possibilities, for example ghd (by the rule: replace the last letter by d), or even abd (by the very straightforward rule: replace anything by abd). Other analogies are even more ambiguous.

It is not completely clear, *why* humans prefer certain kinds of rules to others. The explanations for the selection of a certain possibility might include attributes like “simplicity” or “elegance”. Clearly, some kind of heuristics is used to select a term from several possibilities.

The grammar computed in my approach should contain every solution of the analogy that can be grasped by the – relatively powerful – terms of E-generalization. Of course some rules will be never captured by this approach, because they depend on personal experience or the like (“My grandmother told me always to answer abc in analogy tasks...”), however, every solution that can be grasped in algebraic terms, can be captured.

So the only thing missing for a model of human problem solving, is a selection function for our resulting grammar. As our grammar has a tree structure, we can traverse it with any standard algorithm. The most sensible approach seems to be to use an algorithm like best first search and put the knowledge about humans in the weight of the tree edges.

The most simple approach would be to assign a constant weight of 1, which would yield a breadth-first search and therefore select terms first that have the simplest algebraic description.

Weights might also be determined heuristically by running a series of experiments and probably yield a result closer to human problem solving than mere depth of a solution as a criterion.

However, even very good adjusted weights will have one problem: The information about which part of the substitutions τ_1 and τ_2 was used to yield the resulting term is not used here. In the next part I will suggest another variation of the algorithm that may incorporate such knowledge and is therefore better suited to model human solving of proportional analogies.

4.5 Extended result selection

The point where human cognition is concerned most, is clearly the selection of a term from the resulting tree grammar. To improve this method, several possibilities can be considered.

The simplest extension of the algorithm could be a kind of serialization. One might notice, that the algorithm for grammar intersection (Fig. 9) contains qualitatively different steps. Steps 2 and 3 traverse alternatives in rules, whereas step 4 descends into a term. Steps 1 and 5 do not change our "position" in the tree. Thus, starting at the root of each tree grammar, as long as step 4 is not applied, every symbol that occurs as an argument of the algorithm, is still a *complete* description of our term. The overall algorithm could therefore be varied in the following way:

1. Do the grammar lifting as described before
2. Apply the `intersect` algorithm as long as this is possible without using step 4. Store all occurring nonterminal pairs (s_i, s'_i) from the grammars $\mathcal{G}_1 \times \mathcal{G}_2$ (i.e. whenever `intersect`(s_i, s'_i) is called, remember this pair).
3. Use an heuristics to select the best symbol pair.
4. Use the selected pair instead of the start symbols of the grammars and proceed with the algorithm as before

This approach allows for an extended version of the selection of the terms. We can not only account for the structure of the term in our resulting grammar, but we can also use criteria derived from the nonterminal pair (s_i, s'_i) used to compute this solution. Using the symbols s_i and s'_i as starting symbols, and using $\Sigma, \mathcal{N}$, and $\mathcal{R}$ from the original grammars, we can interpret them as descriptions of our terms A and C respectively. And, as rule 4 was *not* used in the `intersect`-algorithm, our new grammar still

describe the *complete* terms A and C, although some alternatives are now ruled out.

However, a nonterminal s_i will contain only part of the information that was originally contained in $\mathcal{G}_1$. Some alternatives to describe the structure of A (or C respectively) will not be contained in the new grammar. And exactly this information can be used to select a term in the resulting grammar.

We are now able to use criteria that were formerly inaccessible. On the one hand, criteria similar to those described in section 4.4 can be used in this case as well. It might also be possible, to use a heuristics that prefers "high level" functions to simpler ones, for example our *Iter* function to the concatenation function $\cdot$. But what is far more important, we gain the possibility to base a heuristics on the *number of variables* used to compute a result term. This gives us the possibility to choose a result term, such that its computation uses only a minimal subset of the substitutions. For example, a rule mapping a description of *abc* like $Iter(a, succ, 3)$ to one of *ghi* like $Iter(g, succ, 3)$, using only one variable term, can be given precedence over a mapping of $a \cdot b \cdot c$ to $g \cdot h \cdot i$ (i.e. the concatenation of the three constants), which would need three variables with the corresponding substitutions.

It is probably a little far-fetched to describe such solutions as more "elegant", however it gives us a possibility to prefer solutions using more abstract descriptions to those using only basic replacement of constants.

In some sense, this enforces a more abstract level of reasoning, and it may even be one kind of criteria employed by humans. Using the terminology of Leeuwenberg [1971], the information load might be lower using this approach.

One completely different approach to support the result selection might be to incorporate knowledge about B in the selection process. The relationship of A to B is only indirectly used in the algorithm, and may be necessary to make the selection step more adequate. However, such steps would require a quite fundamental re-design of the algorithm, and it is also not clear, whether the feature of obtaining the abstraction as a by-product of the process could be kept up in such a re-designed algorithm.

5 Implementation

Together with this thesis, a proof-of-concept implementation in Moscow-ML was done. Moscow-ML is an implementation of Standard ML, which is a strictly functional language. Because ML contains pattern-matching algorithms, it can be used to interact with objects like trees and terms very easily. The majority of the implementation is therefore very straightforward.

The implementation delivered with this thesis consists of seven different files:

conventions.ml contains conventions about terms etc. It defines a term, that can be used in regular tree grammars, as Bottom, a Nonterminal, a Variable (to be used in a substitution) or a Function, consisting of a name and a list of arguments. All definitions are generic in the sense, that they use the basic type *symbol*, which for this application is defined as string, but can be changed easily to whatever type an application might require (as long as it allows for an equality check).

A regular tree grammar is then defined as a start symbol and a list of rules, the signature and the set of nonterminals are not represented explicitly.

convenience.ml contains all convenience functions that are not part of any of the algorithms. Those are functions like parsing terms from text files, printing terms in a sensible way, and others.

synt_au.ml contains a sample implementation of syntactic Anti-Unification. This file is not needed for the E-Generalization-programs to work, it is contained only to give a possibility to compare E-Generalization with syntactic Anti-Unification.

normalize.ml features an algorithm to convert regular tree grammars into half-normalized form, i.e. to a form where at the right-hand-side of all rules either only nonterminals appear or only one function. Half-normalized form of grammars is needed to facilitate the grammar-intersection process (note that the grammar-lifting algorithm does not disturb the normal form).

cegen.ml is the implementation of constrained E-Generalization. It contains therefore the grammar-lifting algorithm as well as a function to intersect two regular-tree grammars.

uegen.ml contains everything needed to allow for unconstrained E-Generalization, namely, a function that computes two universal substitutions given two regular tree grammars. It then just calls the function from `cegen.ml` with those substitutions.

prop_analogy.ml uses the programs from the other files to give the possibility to compute the D of the proportional string analogy $A : B :: C : D$, where A, B, and C are given as regular tree grammars.

Besides those files, some sample grammar files (`*.gram`) are given to facilitate the testing of the programs.

6 Conclusion and further work

As we have seen, we have found an approach to solving proportional analogies, that differs from most other approaches in some aspects.

On the one hand, we aim not only at computing one or several terms serving as propositions for solutions. We want to describe *all* possible solutions in a closed form. We have seen, that this goal can be achieved by using regular tree grammars. This gives us a possibility to describe the (potentially infinitely many) possible solutions of an analogy.

On the other hand, we do not wish to use a direct mapping (which would also not compute all solutions), but an indirect one via abstraction. The common structure of our terms is extracted as a byproduct of the process of solving the analogy.

As the algorithms are all proven to be correct (most of them in Burghardt [2005]), as long as our background knowledge correctly describes the domain, the result will also be correct in the sense, that an analogy between $A : B$ and $C : D$ does really exist in an algebraic sense.

An equational theory for strings can be worked out relatively easy. Following Leeuwenberg [1971], this holds also for other domains, e.g. geometrical figures, which can be represented as strings, and the equations equations can be given accordingly.

The problem is, that what is missing is an automated way to generate the regular tree grammars needed for our algorithm from the equational theory describing the background knowledge. Some ideas about such an algorithm can be found in Emmelmann [1991] and Burghardt [2005], however, a concrete description of an algorithm and an implementation of this is yet to be done.

Another piece of further work to be done is the serialization step described in chapter 4.5. The implementation of this step should be a rather straightforward step, but nevertheless it might give a lot more possibilities concerning the heuristics for result selection.

These heuristics are the final piece of further research that is necessary. I proposed only very basic heuristics, that seem to be reasonable to me, however, to come closer to human problem solving, empirical investigations are needed. Statistics about the proposed solutions from human subjects needs to be done, and a mapping of those results to positions in the corresponding regular tree grammar might then lead to the development of a heuristics that resembles human cognition closer.

Bibliography

W.S. Brainerd. The minimalization of tree automata. *Information and Control*, 13:484–491, 1968.

Jochen Burghardt. *E*-generalization using grammars. *Artificial Intelligence Journal*, 165 (1):1–35, 2005.

Jochen Burghardt and Birgit Heinz. Implementing Anti-Unification Modulo Equational Theory. Technical report, GMD - Forschungszentrum Informationstechnick GmbH, 1996.

H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 1997. release October, 1rst 2002.

Mehdi Dastani, Bipin Indurkhya, and Remko Scha. An Algebraic Approach to Modeling Analogical Projection in Pattern Perception. In *Proceedings of Mind II*, 1997.

Jim Davies and Ashok K. Goel. Visual analogy in problem solving. In *IJCAI*, pages 377–384, 2001. URL citeseer.ist.psu.edu/davies01visual.html.

H. Emmelmann. Code Selection by Regularly Controlled Term Rewriting. In *Proc. of Int. Workshop on Code Generation*, 1991.

Thomas G. Evans. A Program for the Solution of a Class of Gemetric-Analogy Intelligence-Test Questions. In Marvin Minsky, editor, *Semantic Information Processing*, chapter 5, pages 271–353. MIT Press, 1968.

Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2):155–170, April–June 1983.

E. Bruce Goldstein. *Sensation and Perception*. Wadsworth Publishing Co., Belmont, California, 1980.

Helmar Gust, Kai-Uwe Kühnberger, and Ute Schmid. Ontological aspects of computing analogies. In *ICCM 2004, Procedings of the International Conference on Cognitive Modelling*, 2004.

Birgit Heinz. Anti-Unifikation modulo Gleichungstheorie und deren Anwendung zur Lemmagenerierung. Technical report, GMD - Forschungszentrum Informationstechnick GmbH, 1996.

Douglas Hofstadter and the Fluid Analogies Research Group. *Fluid Concepts and Creative Analogies*. BasicBooks, 1995.

J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

E. Leeuwenberg. A perceptual coding language for visual and auditory patterns. *American Journal of Psychology*, 84:307–349, 1971.

S. O'Hara. A model of the redescription process in the context of geometric proportional analogy problems. In *Int. Workshop on Analogical and Inductive Inference (AII '92)*, volume 642, pages 268–293. Springer, 1992.

G.D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

L. Pottier. Generalisation De Termes En Theorie Equationnelle. Technical Report 1056, INRIA, 1989.

J. Reynolds. Transformational Systems and the Algebraic Structure of Atomic Formulas. In *Machine Intelligence*, volume 5. Edinburgh University Press, 1970.

Ute Schmid, Helmar Gust, Kai-Uwe Kühnberger, and Jochen Burghardt. An Algebraic Framework for Solving Proportional and Predictive Analogies. In Franz Schmalhofer, Richard Young, and Graham Katz, editors, *Proceedings of the European Conference on Cognitive Science, Osnabrück*, 2004.

Uwe Schöning. *Logik für Informatiker*. Number 56 in Reihe Informatik. Wissenschaftsverlag, Mannheim, 1989.

Jörg H. Siekmann. Universal Unification. In Robert E. Shostak, editor, *Proceedings of the 7th International Conference on Automated Deduction (CADE-7)*, volume 170 of *Lecture Notes in Computer Science*, pages 1–42. Springer, 1984.

J.W. Thatcher and J.B. Wright. Generalized finite automata theory with an application to a decision problem of second–order logic. *Mathematical Systems Theory*, 2(1), 1968.

Michael v. Thaden and Stephan Weller. Lösen von Intelligenztestaufgaben mit E-Generalisierung (Solving intelligence tasks by E-Generalization). In *Tagungsband der Informatiktage 2003*, pages 84–87. Gesellschaft für Informatik e.V., 2003.

Appendix

A Program installation and usage

A.1 Availability

The implementation used for this thesis can be downloaded at `http://www-lehre.inf.uos.de/~stweller/ba/`.

A.2 ML requirements

The program was developed and tested using Moscow-ML (`http://www.dina.kvl.dk/~sestoft/mosml.html`). However, every other implementation of Standard ML should also work.

A.3 Installation

The program ships as `.ml`-files containing the sourcecode. To get the system running, the only thing necessary is to unpack all files to the same directory.

A.4 Usage

To compute the string analogy $A : B :: C : D$, the following steps are necessary:

- Open an ml-session and read the file `prop_analogy.ml` by the command: `use prop_analogy.ml`.
- Build the grammars either interactively in an ml-session, or read them from textfiles by the function `readGrammar`, given the filename as an argument.
- Call the main function `prop_analogy` with the following arguments (in order):
  - The starting symbol of the grammar describing A
  - The starting symbol of the grammar describing B
  - The starting symbol of the grammar describing C
  - The (common) rule list of the grammars. If your grammars do not share the same rules, append the rules, as long as the names are not conflicting, this will not cause any problems.
- You are returned the resulting grammar for D. It can be printed nicely by the function `printGram`.

B Example Grammar for $abc : abd :: ghi :?$

$$
\begin{aligned}
N_{abc} &::= N_a \cdot N_{bc} | N_{ab} \cdot N_c | Iter(N_a, N_{succ}, N_3) \\
N_{ab} &::= N_a \cdot N_b | Iter(N_a, N_{succ}, N_2) \\
N_{bc} &::= N_b \cdot N_c | Iter(N_b, N_{succ}, N_2) \\
N_{abd} &::= N_{ab} \cdot N_d | N_a \cdot N_{bd} \\
N_{bd} &::= N_b \cdot N_d \\
N_{ghi} &::= N_{gh} \cdot N_i | N_g \cdot N_{hi} | Iter(N_g, N_{succ}, N_2) \\
N_{gh} &::= N_g \cdot N_h | Iter(N_g, N_s, N_2) \\
N_{hi} &::= N_h \cdot N_i | Iter(N_h, N_s, N_2) \\
N_a &::= a | Iter(N_a, N_{succ}, N_1) | p(N_b) \\
N_b &::= b | Iter(N_b, N_{succ}, N_1) | s(N_a) | p(N_c) \\
N_c &::= c | Iter(N_c, N_{succ}, N_1) | s(N_b) | p(N_d) \\
N_d &::= d | Iter(N_d, N_{succ}, N_1) | s(N_c) | p(N_e) \\
N_e &::= e | Iter(N_e, N_{succ}, N_1) | s(N_d) | p(N_f) \\
N_f &::= f | Iter(N_f, N_{succ}, N_1) | s(N_e) | p(N_g) \\
N_g &::= g | Iter(N_g, N_{succ}, N_1) | s(N_f) | p(N_h) \\
N_h &::= h | Iter(N_h, N_{succ}, N_1) | s(N_g) | p(N_i) \\
N_i &::= i | Iter(N_i, N_{succ}, N_1) | s(N_h) | p(N_j) \\
N_j &::= j | Iter(N_j, N_{succ}, N_1) | s(N_i) | p(N_k) \\
N_k &::= k | Iter(N_k, N_{succ}, N_1) | s(N_j) | p(N_l) \\
N_l &::= l | Iter(N_l, N_{succ}, N_1) | s(N_k) | p(N_m) \\
N_m &::= m | Iter(N_m, N_{succ}, N_1) | s(N_l) | p(N_n) \\
N_n &::= n | Iter(N_n, N_{succ}, N_1) | s(N_m) | p(N_o) \\
N_o &::= o | Iter(N_o, N_{succ}, N_1) | s(N_n) | p(N_p) \\
N_p &::= p | Iter(N_p, N_{succ}, N_1) | s(N_o) | p(N_q) \\
N_q &::= q | Iter(N_q, N_{succ}, N_1) | s(N_p) | p(N_r) \\
N_r &::= r | Iter(N_r, N_{succ}, N_1) | s(N_q) | p(N_s) \\
N_s &::= s | Iter(N_s, N_{succ}, N_1) | s(N_r) | p(N_t) \\
N_t &::= t | Iter(N_t, N_{succ}, N_1) | s(N_s) | p(N_u) \\
N_u &::= u | Iter(N_u, N_{succ}, N_1) | s(N_t) | p(N_v) \\
N_v &::= v | Iter(N_v, N_{succ}, N_1) | s(N_u) | p(N_w) \\
N_w &::= w | Iter(N_w, N_{succ}, N_1) | s(N_v) | p(N_x) \\
N_x &::= x | Iter(N_x, N_{succ}, N_1) | s(N_w) | p(N_y) \\
N_y &::= y | Iter(N_y, N_{succ}, N_1) | s(N_x) | p(N_z) \\
N_z &::= z | Iter(N_z, N_{succ}, N_1) | s(N_y) \\
N_{succ} &::= succ \\
N_1 &::= 1 \\
N_2 &::= 2 \\
N_3 &::= 3
\end{aligned}
$$