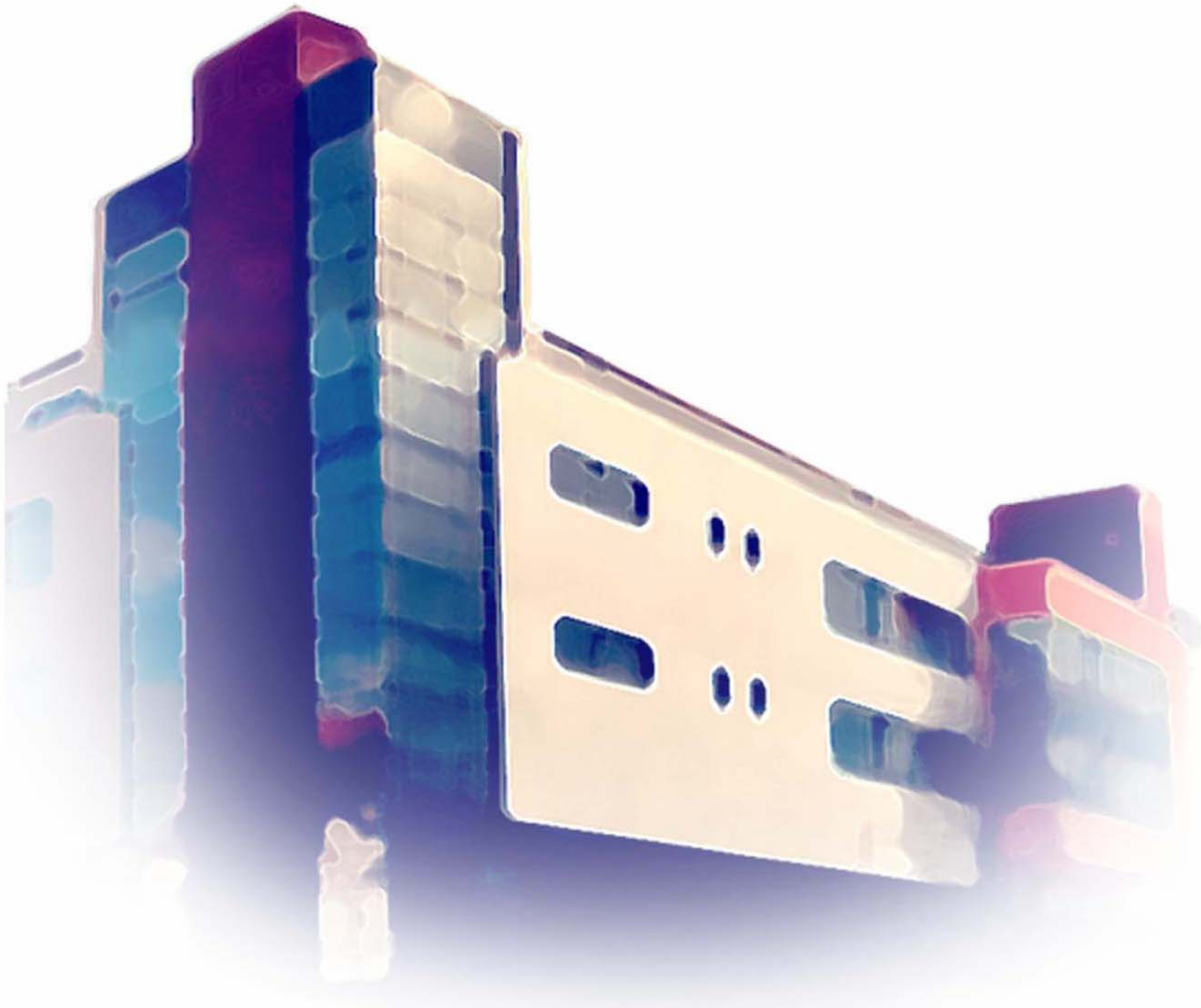


Martin Schmidt

*Restricted Higher-Order Anti-Unification for
Heuristic-Driven Theory Projection*



PICS

Publications of the Institute of Cognitive Science

Volume 31-2010

ISSN: 1610-5389

Series title: PICS
Publications of the Institute of Cognitive Science

Volume: 31-2010

Place of publication: Osnabrück, Germany

Date: September 2010

Editors: Kai-Uwe Kühnberger
Peter König
Sven Walter

Cover design: Thorsten Hinrichs



Institute of Cognitive Science

Bachelor's Thesis

**Restricted Higher-Order
Anti-Unification for Heuristic-Driven
Theory Projection**

Martin Schmidt
martisch@uos.de

October 27, 2008

Supervisors:

Dr. Angela Schwering
Artificial Intelligence, Institute of Cognitive Science
University of Osnabrück

Prof. Dr. Kai-Uwe Kühnberger
Artificial Intelligence, Institute of Cognitive Science
University of Osnabrück

Acknowledgements

First, i would like to thank my supervisors Angela Schwering and Kai-Uwe Kühnberger for their valuable advice and support. Furthermore, i credit Helmar Gust and Ulf Krumnack for helpful discussions about the formal topics involved. Additionally, my gratitude goes to Egon Stemle and Jackie Griego for their help to typeset and proof-read this thesis.

The work was supported by the German Research Foundation (DFG) through the project "Modeling of predictive analogies by Heuristic-Driven Theory Projection" (grant KU 1949/2-1).

Contents

1	Introduction	6
2	Background	7
2.1	Heuristic-Driven Theory Projection	7
2.1.1	Principles	7
2.1.2	Language	8
2.1.3	Phases	11
2.2	Anti-Unification	14
2.2.1	General Concepts	14
2.2.2	First-Order Anti-Unification	17
2.2.3	Higher-Order Anti-Unification	19
3	Anti-Unification of Theories	21
3.1	Restricted Higher-Order Anti-Unification	21
3.1.1	Higher-Order Terms	21
3.1.2	Basic Substitutions	22
3.1.3	Higher-Order Substitutions	25
3.2	Complexity Measures	26
3.2.1	Information Load	26
3.2.2	Complexity of Substitutions	28
3.2.3	Preferred Generalizations	30
3.3	Reuse of Substitutions	32
4	Computation of Preferred Generalizations	35
4.1	Algorithm	35
4.2	Optimizations	37
4.3	Implementation	41
5	Conclusions and Future Work	44
	Bibliography	46

List of Figures

2.1	Formalization of the Rutherford Analogy	8
2.2	Overview of HDTP's Phases	11
2.3	Generalized Theory of the Solar System and the Rutherford Atom . . .	13
2.4	Examples for Generalizations in First-Order Anti-Unification.	18
2.5	Examples for Plotkin's First-Order Anti-Unification	19
3.1	Examples for all Basic Substitutions	23
3.2	Example for Composition of Basic Substitutions	25
3.3	Three Basic Substitution Chains	26
3.4	Example with multiple Least General Generalizations	28
3.5	Examples for $f(X, Y) \rightarrow f(d, h(a))$	30
3.6	Example for $f(X, Y) \rightarrow f(g(a, b, c), d)$	30
3.7	Examples for Generalizations in the Rutherford Analogy	32
3.8	Example for use of Insertion as Renaming	33
4.1	Algorithm for Computing Generalizations	36
4.2	Algorithm for Computing all Preferred Generalizations up to Renaming	38
4.3	Instantiation Ordering of Anti-Instance Equivalence Classes	39
4.4	Improved Instantiation Ordering of Anti-Instances.	40

List of Definitions & Theorems

Signature	9
Term	9
Subterm	10
Atomic Formulas	10
Formulas	10
Substitution	14
Instance	15
Term Equivalence	15
Unifier	15
Anti-Unifier	16
Most Specific Anti-Unifier	16
First-Order Anti-Unification Substitutions	17
Generalization	18
Least General Generalization	18
Higher-Order Terms	21
Restricted Higher-Order Anti-Unification Substitutions	22
Information Load	26
Lemma: Information Load of Anti-Instances	26
Corollary: Finite Anti-Instances	27
Proposition: Finite Anti-Unifiers	27
Complexity of Substitutions	28
Complexity of Generalizations	29
Preferred Generalization	31
Complexity of reused Substitutions	34

1 Introduction

Analogy making is a high-level cognitive process [CFH92, FGMF98, HM05] and is considered a core component of cognition [GKKS08a]. Analogies use already known information from *source domain* to acquire knowledge in new situations in the *target domain*. Empirical research supports the belief that structural commonalities between the two domains are the main guidance for the construction of analogies. This finding is incorporated in Dedre Gentner's seminal structure-mapping theory (SMT) [Gen83] on which the computational analogy model called structure mapping engine (SME) [FFG86, FFG89] is based. Altogether, many different mechanisms to analyze and extract structural commonalities have been developed [Gen89, HM95, Ind92]. However, cognitive science and artificial intelligence still lack a logical theory for analogical reasoning. The proposed theories for analogical reasoning are mostly psychologically or neurally inspired. Heuristic-driven Theory Projection (HDTP) [GKS06] is a symbolic analogy model intended to fill this void. The very flexible detection of structural commonalities between a formally specified source and target domain is achieved by using anti-unification. More precisely, HDTP uses a form of higher-order anti-unification to capture even deep structural commonalities that are not being detected by other frameworks.

This thesis presents an indepth view of the specifically optimized anti-unification for HDTP. Section 2 establishes the context, necessary to understand the restrictions and requirements that guided the development of this anti-unification. Foremost, Heuristic-driven Theory Projection as the enclosing framework, is outlined in section 2.1. Anti-unification is summarized in section 2.2 starting with an introduction to general anti-unification theory and followed by a discourse regarding the characteristic properties of first and higher-order anti-unification. Section 3 is dedicated to explain restricted higher-order anti-unification. Core concepts involved are explained in section 3.1. Afterwards more advanced topics such as complexity metrics (section 3.2) and complexity reduction (section 3.3) are explored. This is followed by section 4 which outlines the algorithmic realization (section 4.1), computational optimizations (section 4.2) and implementation details (section 4.3) of restricted higher-order anti-unification. Section 5 concludes the work with an outlook into future research topics.

2 Background

2.1 Heuristic-Driven Theory Projection

2.1.1 Principles

Heuristic-driven Theory Projection is a formal framework based on well-known standard symbolic methods of artificial intelligence for solving analogies [GKS06]. HDTP's input consists of a finite many-sorted first-order logic axiomatization of a source and target domains. The use of first-order logic makes it trivial to incorporate classical reasoning mechanisms of artificial intelligence and also distinguishes it from other often neurally inspired theories of analogical reasoning. However, HDTP still retains the goal of only drawing inferences that are cognitively inspired.

HDTP analyzes source and target domains to determine common structural patterns. These patterns are represented in a generalized axiom system, with corresponding mappings to source and target, and describe the generalized theory of both input domains. The process of finding common structural patterns is described as aligning of source and target domain and the result is called an *alignment* or *mapping*. The source domain usually contains a set of formulas that axiomatizes an informationally richer domain than the target domain. Therefore, the found alignment can be used to transfer knowledge from the source, which is better understood, to the target domain. Thereby fulfilling the goal of analogy making to gain more information about the target domain. However, transferred knowledge is not factual information, but rather hypotheses that can be tested and checked for consistency in the better understood domain. Thus, it is possible to infer new knowledge in a creative way. Depending on the domains that are compared and the chosen representations, the result can be from one to a multitude of analogies.

The following sections 2.1.2 and 2.1.3 focus on a brief overview about the inner workings and application of formal mechanisms that HDTP employs. Semantic aspects of the analogical relations that HDTP establishes will not be covered but are discussed in [GKS06] and [GKKS07].

2.1.2 Language

The well-known *Rutherford Analogy*, which compares the solar system to the Rutherford atom model, is used as a running example to show the different aspects of HDTP's language for describing domains. Figure 2.1 shows a formalization of the Rutherford Analogy. The source domain on the left side is the solar system while the target domain on the right side is the Rutherford atom model. Both theories are simplified and not considered to be complete and accurate physical theories of the described objects. Analogies where source and target domain are from different fields of knowledge or the domains only present a partial view of the underlying theory seem to be especially creative. As can be seen by the amount of formulas the source domain has a much richer formalization. The source domain describes that a planet revolves around the sun because of the differences in mass that result in different gravitational forces. At the same time, the sun and the planets do not collide with each other. This is considered in relation to the Rutherford atom model where lightweight electrons are attracted by the nucleus due to the Coulomb force. However, the electrons and the nucleus keep a distance of greater than zero, which is an abstraction of the results of the gold foil experiment of Rutherford [Rut].

Solar System	Rutherford Atom
sorts <i>real, object, time</i>	sorts <i>real, object, time</i>
constants <i>sun : object, planet : object</i>	constants <i>nucleus : object, electron : object</i>
functions <i>mass : object × time → real</i> <i>dist : object × object × time → real</i> <i>force : object × object × time → real</i> <i>gravity : object × object × time → real</i> <i>centrifugal : object × object × time → real</i>	functions <i>mass : object × time → real</i> <i>dist : object × object × time → real</i> <i>coulomb : object × object × time → real</i>
predicates <i>revolves_around : object × object</i>	facts <i>β₁ : mass(nucleus) > mass(electron)</i> <i>β₂ : mass(electron) > 0</i> <i>β₃ : ∀ (t : time) : coulomb(electron, nucleus, t) > 0</i> <i>β₄ : ∀ (t : time) : dist(electron, nucleus, t) > 0</i>
facts <i>α₁ : mass(sun) > mass(planet)</i> <i>α₂ : mass(planet) > 0</i> <i>α₃ : ∀ (t : time) : gravity(planet, sun, t) > 0</i> <i>α₄ : ∀ (t : time) : dist(planet, sun, t) > 0</i>	
laws <i>α₅ : ∀ (t : time, o₁ : object, o₂ : object) :</i> <i>dist(o₁, o₂, t) > 0 ∧ gravity(o₁, o₂, t) > 0</i> <i>→ centrifugal(o₁, o₂, t) = -gravity(o₁, o₂, t)</i> <i>α₆ : ∀ (t : time, o₁ : object, o₂ : object) :</i> <i>0 < mass(o₁) < mass(o₂) ∧</i> <i>dist(o₁, o₂, t) > 0 ∧</i> <i>centrifugal(o₁, o₂, t) < 0</i> <i>→ revolves_around(o₁, o₂)</i>	

Figure 2.1: Logic formalization of the solar system and the Rutherford atom.

Domains contain not only facts, but also general laws as seen in the formalization of the Rutherford atom model. To capture all the information present in the Rutherford analogy HDTP needs an expressive formal language to specify axioms of a domain. We now give a brief overview of the language that encodes knowledge of a domain in HDTP.

Definition 1 (Signature) *A signature is a denumerable set Σ of strings of the form $f : \sigma_1 \times \sigma_2 \times \dots \times \sigma_n \rightarrow \sigma$ or $p : \sigma_1 \times \sigma_2 \times \dots \times \sigma_n$ where $\sigma_1, \dots, \sigma_n, \sigma$ with $n \in \mathbb{N}$ are sorts in the set of sorts Sort_Σ .*

- *Functions are of the form $f : \sigma_1 \times \sigma_2 \times \dots \times \sigma_n \rightarrow \sigma$ where f is a function symbol of arity n . If n is zero we call f a constant.*
- *Predicates are of the form $p : \sigma_1 \times \sigma_2 \times \dots \times \sigma_n$ where p is a predicate symbol of arity n . If n is zero we call p an atom.*

Different strings in the signature must have different leftmost symbols. A symbol can only be used as a function symbol or predicate symbol. It cannot be used twice as function and predicate symbol with different arity.

A signature provides the different building blocks that can be used for the formalization of an algebra. In this document we often write terms without their sorts as a shorthand for ease of explaining and describing examples.

Definition 2 (Term) *Let Σ be a signature. The set $\text{Term}(\Sigma, \mathcal{V})$ relative to an infinite set of variables $\mathcal{V} = \{X_1 : \sigma_1, X_2 : \sigma_2, \dots\}$ with $\sigma_1, \sigma_2, \dots \in \text{Sort}_\Sigma$, is defined as the smallest set such that the following conditions hold:*

1. *If $X : \sigma \in \mathcal{V}$ then $X : \sigma \in \text{Term}(\Sigma, \mathcal{V})$.*
2. *If $f : \sigma_1 \times \sigma_2 \times \dots \times \sigma_n \rightarrow \sigma \in \Sigma$ and $t_i : \sigma_i \in \text{Term}(\Sigma, \mathcal{V})$, then $f(t_1, \dots, t_n) : \sigma \in \text{Term}(\Sigma, \mathcal{V})$.*

We call an element $t : \sigma$ of the inductively constructed set $\text{Term}(\Sigma, \mathcal{V})$ a term with sort σ . A term is called ground if it does not contain any variables.

Symbols like *sun* and *nucleus* are constants in the respective domain because they have an arity of zero and are of sort object. *Real*, *object* and *time* are sorts that are used to tag the type of a function which can then be used as an explicit guide in the structural information mapping employed in HDTP. They can be seen as high-level concepts in an ontology. Functions with higher arity such as *mass*, which takes one argument, are used to map constants to constants. $\text{mass}(\text{sun})$ is a ground term but

$dist(planet, sun, T)$ is not because it contains the variable T . Variables are always denoted by strings with an upper case letter.

Definition 3 (Subterm) *Given a term t and a substring s of t , where s is a term, we call s a subterm of t .*

Thereby, sun is a subterm in the terms $mass(sun)$ and $dist(planet, sun, T)$, because the string "sun", which is a valid term, is a substring of "mass(sun)" and "dist(planet,sun,T)".

Definition 4 (Atomic Formulas) *Let Σ be a signature. The set of atomic formulas $Atomicform(\Sigma, \mathcal{V})$ consists of all expressions of the form $p(t_1 : \sigma_1, \dots, t_n : \sigma_n)$ where $p : \sigma_1, \dots, \sigma_n \in \Sigma$ and $t_1 : \sigma_1, \dots, t_n : \sigma_n \in Term(\Sigma, \mathcal{V})$.*

Atomic formulas are build from predicate symbols and can be used to express relations. Predicate symbols in our example formalization include $>$, $=$ and $<$. An example for an atomic formula is $mass(sun) > mass(planet)$.

Definition 5 (Formulas) *Let Σ be a signature. Define the set $Form(\Sigma, \mathcal{V})$ as the smallest set such that the following conditions hold:*

1. *If $\varphi \in Atomicform(\Sigma, \mathcal{V})$ then $\varphi \in Form(\Sigma, \mathcal{V})$.*
2. *If $\varphi \in Form(\Sigma, \mathcal{V})$ then $\neg\varphi \in Form(\Sigma, \mathcal{V})$.*
3. *If φ and $\psi \in Form(\Sigma, \mathcal{V})$, then $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$, $\varphi \leftrightarrow \psi \in Form(\Sigma, \mathcal{V})$.*
4. *If $\varphi \in Form(\Sigma, \mathcal{V})$ and $X : \sigma \in \mathcal{V}$, then $\exists X \varphi$ and $\forall X \varphi \in Form(\Sigma, \mathcal{V})$.*

Formulas are built from *atomic formulas* with the help of *logical symbols* in the form of *propositional connectives* and *quantifiers*. The *propositional connectives* \vee , \wedge , \rightarrow , \leftrightarrow and \neg are used to express complex facts. Using the aforementioned components that can express different semantic content, complex expressions can be formed by chaining them together. Furthermore, rules may be expressed by using the classical logical *quantifiers* \forall and \exists . Such a rule can be a law like "every two bodies with positive mass will attract each other".

A finite set of formulas that is consistent and consist of the defined vocabulary is referred to as an axiomatization of the corresponding domain. Formulas that can be logically inferred from these axioms constitute the *domain theory*. While the expressive power of first-order logic allows the possibility of expressing complex systems, at

the same time it confronts us with the problem of having more than one equivalent axiomatization for a given domain. However, this expressive power enables not only situation descriptions, but also general laws that are not possible in other well-known analogy models [Kur05]. Furthermore, the representation of knowledge and reasoning over formulas is possible with this logic-based formalization.

2.1.3 Phases

The formal framework of HDTP can employ the three common phases of *retrieval*, *mapping*, and *transfer* of analogy-making. However, only the latter two are currently implemented. HDTP is very modular and can be extended with more subprocesses that complement the main phases as seen in figure 2.2.

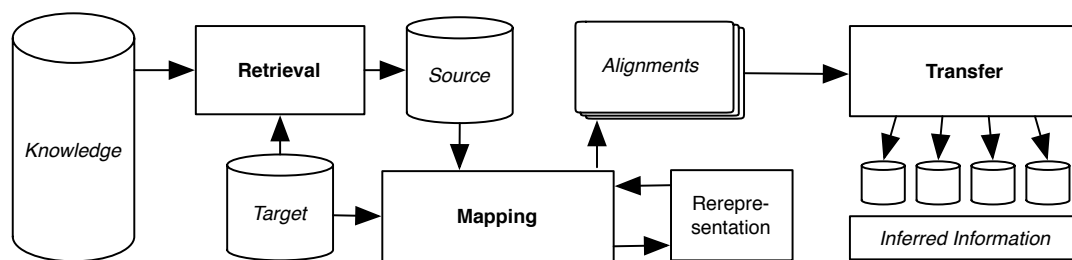


Figure 2.2: Overview of HDTP's phases.

Retrieval is the process of selecting a suitable set of formulas to represent the source domain. This source domain should be related to the given target domain in that they share a common structure that can be identified in more detail during the mapping phase. Different source domains lead to different expansions of knowledge in the target domain subsequently. The goal, therefore, is not necessarily to only find the structurally most similar domain to the target domain, but a multitude of source domains that lead to different newly discovered knowledge about the target domain. Without a retrieval phase, the source and target domain must be given explicitly to the mapping phase. The modular and sequential nature of the HDTP framework makes it possible to add an optional *retrieval* process that generates this input before the mapping phase.

During *mapping*, the analogical relation between the given domains is established by aligning formulas of the source domain with formulas of the target domain. This alignment does not necessarily use all available formulas. Identifying the knowledge

that is actually required for the alignment constitutes the *relevance problem*. Relevance of parts of the domain can change depending on the context of the analogy and desired extension of information in the target domain. Restructuring of the axiomatization while describing the same domain can also lead to different results and may be preferred to build a more plausible outcome depending again on the relative goal of the analogy. This difference in mapping possibilities is regarded as the *representation problem*.

Re-representation can be done with the help of laws of the respective domain together with deduction, an equational theory or background-knowledge. An example of background-knowledge is that *distance* is a symmetric function, which can be expressed by $\forall x \forall y \forall t : distance(x, y, t) = distance(y, x, t)$. Therefore $distance(nucleus, electron, t)$ can be rewritten as $distance(electron, nucleus, t)$ in the axiomatization of the solar system and Rutherford atom model. Another more general example of *background-knowledge* is $\forall x \forall y \forall z : (x > y \wedge y > z) \rightarrow x > z$ which states the transitivity of the *greater than* relation.

Background-knowledge is not considered for alignment because it is so general that it is applicable to both domains. However, it influences the mapping process by possibly allowing for more inferred formulas to be aligned. Re-representation aims to find pairs of formulas from the domain theories that share a common structure and lead to a good alignment. When a newly found alignment uses mappings encountered before it is mere support for the analogical relation already drawn. However, sometimes the axiomatization of domains is insufficient for building a generalized theory and without re-representation no analogy could be computed. Altogether, re-representation accounts for much of the explorative creativity in analogical inferences. The situation we have laid out in the formalization of the Rutherford analogy does not involve re-representation because the domain representations are structurally compatible already.

The structural matching within the mapping phase is created stepwise by aligning a formula from the source domain with a formula from the target. The selection criteria for both is guided by heuristics and may use information regarding already aligned parts and still to be aligned parts. Foremost the alignment depends on a metric of how well two formulas are equal in structure and semantics. A plausible pair of terms to make an alignment is $mass(sun) > mass(planet)$ and $mass(nucleus) > mass(electron)$ because they have the matching predicate symbol $>$. This is complemented by the semantic similarity that *sun*, *planet*, *nucleus* and *electron* are all of the sort *object*. The plausibility of the structural matching of two formulas can be expressed by a metric which assigns costs to mappings. Summation over the costs associated with the metric of all mappings done on formulas in the source and target

```

types
  real, object, time

constants
  X : object, Y : object

functions
  mass : object → real
  dist : object × object × time → real
  F : object × object × time → real
  centrifugal : object × object × time → real

predicates
  revolves_around : object × object × object

facts
  γ1 : mass(X) > mass(Y)
  γ2 : mass(Y) > 0
  γ3 : ∀ (t : time) : F(X, Y, t) > 0
  γ4 : ∀ (t : time) : dist(X, Y, t) > 0

laws
  γ5* : ∀ (t : time, o1 : object, o2 : object) :
    dist(o1, o2, t) > 0 ∧ F(o1, o2, t) > 0
    → centrifugal(o1, o2, t) = -F(o1, o2, t)
  γ6* : ∀ (t : time, o1 : object, o2 : object) :
    0 < mass(o1) < mass(o2) ∧
    dist(o1, o2, t) > 0 ∧
    centrifugal(o1, o2, t) < 0
    → revolves_around(o1, o2)

```

Figure 2.3: Generalization of the solar system and the Rutherford atom (including the generalizations from the transfer marked with *).

gives a relative measure of how good an overall domain mapping is regarding another alternative mapping. The operation of aligning a pair of formulas is carried out by using *anti-unification*. This core operation is very important because it constrains the possible alignment of terms and therefore shapes the generalized theory that HDTP generates. For an explanation of the form of anti-unification, which is used by HDTP, see section 3.

The generated formalization for the generalized theory of the solar system and Rutherford atom model is shown in figure 2.3. We can see that our already mentioned alignment candidates of formulas $mass(sun) > mass(planet)$ and $mass(nucleus) > mass(electron)$ were generalized to $mass(X) > mass(Y)$. Here uppercase letters are variables that must be appropriately instantiated to gain back the original terms of source and target domain. Notably *electron* in the target domain is the corresponding entity of *planet* in the source domain. The same relation holds between *nucleus* and *sun*.

In the final step of HDTP, called the *transfer* phase, the generated mapping is used to transfer information from the source domain to the target domain. Inferred knowl-

edge in the target domain can only be used as a guide or hypothesis that has to be proven using other mechanisms of logic or tests outside the theoretic realm. From a cognitive perspective, this can be viewed as the inspirational mechanism that leads the way for creative ideas that have to be put to test by trying them out in a real world setting. The information that is transferred is usually not already covered by the generalized theory. HDTP can detect inconsistent inferences and reject them by applying a theorem prover. The general usefulness of transferred information can, in contrast, not be inferred by formal processes. In our running example we can transfer $revolves_around(planet, sun)$ from the solar system domain to the Rutherford atom model and infer $revolves_around(electron, nucleus)$ by means of the object mappings between $planet$ and sun as well as $electron$ and $nucleus$.

2.2 Anti-Unification

2.2.1 General Concepts

The concept of anti-unification was first outlined by Reynolds [Rey70] and Plotkin [Plo70]. It is the formal counterpart to the widely known *unification*, but in contrast, has less research specifically dedicated to it. However, many results in research of unification have a counterpart in anti-unification [Rey70]. Where unification is one of the core concepts used in logic programming, best known through the programming language Prolog, anti-unification is mostly used for inductive learning and proof generalization.

The following definitions give a vocabulary that is used to describe anti-unification while, at the same time, make the relation to unification apparent [SA91]. All definitions in this general anti-unification section are not incorporating sorts. By introducing conditions to match sorts where needed, definitions can be easily extended to take sorts of terms into account. Furthermore, we only notate function symbols without sorts for ease of explaining and describing examples.

Definition 6 (Substitution) *A substitution on terms is a partial function, mapping variables to terms, formally represented by $\tau = \{X_1/t_1, \dots, X_n/t_n\}$ (provided $X_i \neq X_j$ for $i, j \in \{1, \dots, n\}$, $i \neq j$). We say τ acts on variables X_1, \dots, X_n . An application of a substitution τ on a term is defined by simultaneously replacing all occurrences of the variables X_1, \dots, X_n and thus:*

$$\bullet \text{ apply}(X, \tau) = \begin{cases} t' & \text{if } X/t' \in \tau \\ X & \text{otherwise} \end{cases}$$

$$\bullet \text{ apply}(f(t_1, \dots, t_m), \tau) = f(\text{apply}(t_1, \tau), \dots, \text{apply}(t_m, \tau))$$

A simple *substitution* such as $\tau = \{X/a\}$ applied to a term $t = f(X, b)$ yields a new term $t' = f(a, b)$. If the to be replaced variable X occurs more than once within t , all occurrences of X within t are replaced by the constant a . Substitutions are not restricted to constants but can replace variables with complex terms such as $g(Y, e)$. The application of substitution $\tau = \{X/g(Y, e)\}$ to $f(X, b)$ results in the term $f(g(Y, e), b)$. The extended substitution $\tau = \{X/g(Y, e), Y/b\}$ applied to $f(X, Y)$ results in $f(g(Y, e), b)$ and not $f(g(b, e), b)$.

Different sets of valid substitutions result in different anti-unification forms. In this general section about anti-unifications we allow variables $V \in \mathcal{V}$ to be replaced by terms $t \in \text{Term}(\Sigma, \mathcal{V})$ to gain examples for each concept introduced.

Definition 7 (Instance & Anti-Instance) *A term t' is an instance of t and t is an anti-instance of t' , if there is a substitution τ such that application of τ on t' results in t . In this case we write $t \xrightarrow{\tau} t'$ or simply $t \rightarrow t'$.*

Hence in our example from above $t' = f(a, b)$ is called an *instance* of $t = f(X, b)$ and t is an *anti-instance* of t' because $t \xrightarrow{\tau} t'$ where τ is the substitution $\{X/a\}$. Note that $f(a, b)$ cannot be an *instance* of $f(X, X)$ because a and b are different terms and the variable X cannot be substituted context dependent into one or the other.

Definition 8 (Term Equivalence) *Two Terms t and t' are considered equivalent if $t \rightarrow t'$ and $t' \rightarrow t$. In this case we write $t \equiv t'$. If t and t' are not equivalent we write $t \not\equiv t'$. We say the t and t' are strongly equivalent if $t \xrightarrow{\tau} t'$ and $t' \xrightarrow{\nu} t$ where τ and ν are substitutions of the form $X_1/X'_1, \dots, X_n/X'_n$ and thereby only replace variables by variables.*

$f(X, X)$ and $f(Y, Y)$ are hence considered equivalent. $f(X, Y)$ and $f(Y, Y)$ are not equivalent because $f(X, Y) \xrightarrow{\tau} f(Y, Y)$ with $\tau = \{X/Y\}$ but there is no ν that makes $f(Y, Y) \xrightarrow{\nu} f(X, Y)$ possible. The equivalence relation has the usual properties reflexivity, symmetry and transitivity. An equivalence class of a term t is the set of all terms that are equivalent to t . Because the equivalence relation is reflexive such a set contains t itself. If we restrict the substitutions used in the term equivalence definition to those that only replace variables by variables, instead of arbitrary substitutions, we can gain a stronger equivalence definition. Using this stronger equivalence definition, we say *unique up to variable renaming* for a criterion if all terms that fulfill the criterion are in one equivalence class and no term of this equivalence class does not fulfill the criterion.

Definition 9 (Unifier) A unifier for a given set of terms \mathcal{T} is a term that is an instance of every term in \mathcal{T} .

The term $f(a, b)$ is a unifier for the set that consists of the terms $f(X, b)$, $f(X, Y)$ and $f(a, Y)$. This is because there exist substitutions for each of these terms ($\tau_1 = \{X/a\}$, $\tau_2 = \{X/a, Y/b\}$, $\tau_3 = \{Y/b\}$) so that by corresponding substitution application they are transformed into the given unifier $f(a, b)$.

Definition 10 (Anti-Unifier) An anti-unifier for a given set of terms \mathcal{T} is a term that is an anti-instance of every term of \mathcal{T} . We call the set \mathcal{T} the anti-unified terms in context of the anti-unifier.

By this definition the term $f(X, Y)$ is a possible anti-unifier for the set of terms $f(a, b)$, $f(c, d)$ and $f(X, c)$. But not for any set that for example contains $g(X, Y)$ or $h(X, Y, Z)$. The substitution $\tau = \{X/e, Y/e\}$ applied to $f(X, Y)$ yields $f(e, e)$ therefore $f(X, Y)$ is an anti-instance of $f(e, e)$ and one possible anti-unifier for the set just containing $f(e, e)$. An anti-unifier for a set of terms is not uniquely defined. Because $f(X, e)$ is an anti-instance of $f(e, e)$ as well, two anti-unifiers for $f(e, e)$ are already found. We observe that $f(X, Y)$ itself is an anti-instance of $f(X, e)$. This shows that the anti-instance relation induces a partial-ordering on the set of anti-unifiers of a set of terms. This ordering, which operates on all terms, is denoted *substitution ordering* because the anti-instance relation itself is defined using substitutions. Another ordering induced by the instance relation is often referred to as *instantiation ordering*.

Definition 11 (Most Specific Anti-Unifier) A most specific anti-unifier (msa) also referred to as the least general anti-unifier (lga) of a set of terms \mathcal{T} is an anti-unifier a for \mathcal{T} such that there exist no anti-unifier a' for \mathcal{T} with $a \rightarrow a'$ and $a \neq a'$.

The process of finding the most specific anti-unifier is called anti-unification. The search for the most specific anti-unifiers means that the number of basic substitution components is kept minimal and only those substitutions necessary to fulfill the conditions for a valid anti-unifier are used. Found anti-instances therefore carry maximal information about the common structure of the set of terms anti-unified. By changing the definition of possible substitutions the ordering governed by the anti-instance relation can be changed and therefore the notion of what common structure is can be varied. Which in turn effects the found most specific anti-unifiers.

The construction of a most specific anti-unifier is dual to the notion of a most general unifier (mgu) in unification theory. Both classes of terms play a central role and are a major subject of research. Whether they are well defined and unique is analyzed

in context of different forms of unification and anti-unification.

The notion of *substitution* we used to give examples for concepts in this section resembles Plotkin's first-order anti-unification. Which has some sought after properties of its most specific anti-unifier which we discuss in the following section.

2.2.2 First-Order Anti-Unification

First-order anti-unification, which was introduced by Plotkin [Plo70], restricts substitutions to replace variables $V \in \mathcal{V}$ by terms $t \in Term(\Sigma, \mathcal{V})$. This is the same restriction we already used for examples of substitutions up till now. Some more examples for first-order anti-unification are given in figure 2.5.

Definition 12 (First-Order Substitutions) *A first-order substitution $\tau = \{X_1/t_1, \dots, X_n/t_n\}$ replaces variables $X_i \in \mathcal{V}$ by terms $t_i \in Term(\Sigma, \mathcal{V})$ where $i \in \{1, \dots, n\}$.*

We can call substitutions of the form $X \rightarrow t$ with a variable $X \in Term(\Sigma, \mathcal{V})$ and a term $t \in Term(\Sigma, \mathcal{V})$ first-order, because all variables in \mathcal{V} are first-order variables.

It has been proven in [Plo70] that, with first-order substitutions, an anti-unifier for a pair of terms always exists. From this the existence of an anti-unifier for any finite set of terms results. First, the anti-unifier of two arbitrary terms of the set is generated. Then, this anti-unifier is anti-unified with another term of the set to produce a new anti-unifier for all three terms chosen. This continues until an anti-unifier for the complete set is found. Because we always have a substitution ordering between two terms, the transitivity of the ordering can be used by this process to generate a solution.

The first-order anti-unification substitution ordering forms a semi-lattice on the equivalence classes of terms. The bottom element in this lattice is the equivalence class of terms consisting of only a variable. Hence, a term consisting of only a variable is always a trivial anti-unifier for any collection of terms in first-order anti-unification. As an example see figure 2.5(a). Here the trivial anti-unifier is also the only valid one and therefore, also the most specific anti-unifier.

In the context of solving analogies we are interested in the mapping between a source domain term and a target domain term. Therefore, in analogical mapping the anti-unifier of two terms is sought. The existence of an anti-unifier is guaranteed and all most specific anti-unifiers together form an equivalence class in first-order

anti-unification. Furthermore, the equivalence relation can be restated as (compare definition 8): two terms are equivalent if we can derive them from each other by renaming of variables. Therefore, we say that the most specific anti-unifier in first-order anti-unification is unique up to renaming. (cf. [Plo70]).

An anti-unifier together with the corresponding substitutions for a set of two terms is called a *generalization* in the context of analogies.

Definition 13 (Generalization) A generalization for a pair of terms $\langle s, t \rangle$ is a triple $\langle g, \tau, \nu \rangle$ with a term g and substitutions τ, ν such that $g \xrightarrow{\tau} s$ and $g \xrightarrow{\nu} t$. The term g is therefore the anti-unifier in a generalization for s and t . It is explicitly allowed that τ and ν may be mappings that leave the term in its original form or, in other words, consist of an empty substitution.

Because the anti-unifier is not required to be a most specific anti-unifier for the pair of terms given, we introduce the notion of least general generalization where this is the case.

Definition 14 (Least General Generalization) A generalization $\langle g, \tau, \nu \rangle$ for a pair of terms $\langle s, t \rangle$ is called least general generalization if g is a most specific anti-unifier for the terms s and t .

Least general generalizations are unique up to variable renaming in first-order anti-unification because the included most specific anti-unifier is unique up to variable renaming in first-order anti-unification.

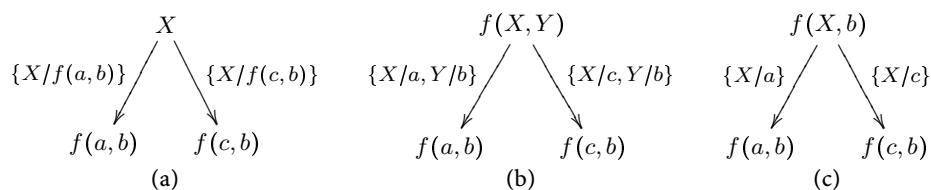


Figure 2.4: Examples for generalizations in first-order anti-unification.

Figure 2.4 gives all three possible generalizations with first-order substitutions for the terms $f(a,b)$ and $f(c,b)$. The pair of anti-unified terms at the bottom, of each generalization depicted, is included additionally to the anti-unifier placed at the top and the two corresponding substitutions, which are represented by labeled arrows. Figure 2.4(a) includes the most general anti-unifier. (c) is the least general generalization

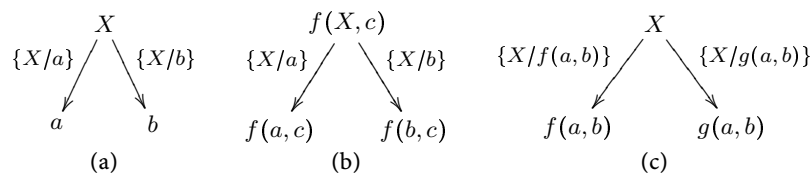


Figure 2.5: Examples for Plotkin's first-order anti-unification.

with a most specific anti-unifier. (b) is less general than (c), but not a least general generalization because it does not include a most specific anti-unifier.

In figure 2.5 we see further examples for generalizations of different pairs of terms, which we now analyze in the context of analogies. In (a), the constants a and b are anti-unified to the Variable X . In this case b can be seen as an analogon to a . In (b), these constants are used again as arguments, but each is in the same enclosing context. Still, the same substitutions are used and therefore the same analogical relation between a and b is established. In (c) the terms differ in function symbols, but the arguments supplied are the same. A most specific anti-unifier from first-order anti-unification cannot reflect this, and thus, only the most general anti-unifier in the form of a variable is found by first-order anti-unification. In turn, this leads to complex substitutions to reproduce the anti-unified terms. Plotkin's anti-unification is not powerful enough to capture complex structural equivalences in the anti-unifier like commonalities that are embedded in different functions or predicates. In extreme cases like figure 2.5(c), where the topmost function or predicate symbols differ, the result is the inability to capture any similarities of subterms.

2.2.3 Higher-Order Anti-Unification

Gentner et al. [Gen83] have revealed empirically that analogies are characterized by deep structural commonalities. However, first-order anti-unification fails to capture them and tends to overgeneralize as seen in figure 2.5(c). Anti-unification can be altered to recognize complex commonalities by extending the set of possible substitutions and allowing for more detailed operations on terms. Particularly, being able to express operations on arguments of functions or predicates is valuable for achieving a better coverage of similarities of terms. In the simplest case higher-order anti-unification involves a change of function symbols. A more complex case, which constitutes the operation in a higher-order logic, is the manipulation of subterm structures. Examples of this are reducing the arity of predicates or permuting function arguments.

Defining complex substitutions that operate on predicate and function structures

result in many problems. Again, higher-order anti-unification shares this property with higher-order unification. Higher-order substitution patterns may become too specific or arbitrarily complex, which can be viewed as overgeneralization. Furthermore, the most specific anti-unifier may not be well defined anymore and lose its uniqueness. This can lead to infinite chains of more and more general anti-unifiers as demonstrated by [Has95].

The coverage mechanism of substitutions in higher-order anti-unifications for analogies should therefore be kept to a minimum and be only those that are cognitive adequate to express shared structure in the applied domains.

This can be expressed with the following guidelines for anti-unifiers:

- Anti-unifiers should preserve as much of the structure of both domain terms as possible. In other words, substitutions should not contain terms that occur in both anti-unified terms.
- While preserving structure, anti-unifiers should only contain elements that are present in the original anti-unified terms.
- In general, the complexity of the anti-unifier should not be more complex than the complexity of the anti-unified terms.

Following these design goals higher-order anti-unification should not fall short of supplying us with a mechanism to capture similarities in dissimilar context.

Different restrictions for higher-order anti-unification have been proposed to obtain a tractable generation of anti-unifiers. These use a variety of different formalisms to achieve their goal. [Pfe91] uses higher-order patterns in the form of λ -terms to restrict free variables. Typed λ -calculus is the standard notation for second-order logic or in general a higher-order logic. Most general unifiers, which are called *least common anti-instances* in [Pfe91], are shown to exist and are unique up to equivalence. The application of the algorithm developed there is for proof generalization. [Has95] uses combinator terms that are further restricted to relevant combinators. It is claimed that a problem of λ -terms is the missing control of how function-arguments are used in terms. In contrast, combinator terms that are built as compositions of basic functions allow for the control of how arguments are passed from one function to the next. It is shown that a useful notion of generalization can be developed and an algorithm to compute those is presented. [Wag02] uses the same formalism in the context of analogical programming.

3 Anti-Unification of Theories

3.1 Restricted Higher-Order Anti-Unification

In this section, a formal framework for higher-order anti-unification is defined. Here, we only introduce operations on terms. The extended application of this framework, from terms to atomic formulas can be made without additional complexity. However, this extension needs additional rules specialized for handling predicate symbols, which are analogs of the ones specialized on function symbols. Furthermore, formula anti-unification is not covered here because they can be decomposed into atomic formulas by HDTP and processed as such by anti-unification. For a discussion of how this framework can be extended to anti-unification of theories see [SKKG08, GKKS08b].

This framework is guided by the requirements for a usable higher-order form of anti-unification as laid out in section 2.2.3. Most notably as much structure as possible of the anti-unified terms should be preserved where as the anti-unifier should not be more complex than these terms. The presented formalism has been referred to as *restricted higher-order anti-unification* in [KSGK07, SKKG08]. Restricted higher-order anti-unification is a weak type of higher-order anti-unification and can be reduced to first-order anti-unification modulo, a suitably restricted equational theory [GKS06, Hei96]. This sets it apart from other frameworks of higher-order anti-unification, which are discussed in section 2.2.3.

3.1.1 Higher-Order Terms

In order to define higher-order terms we extend our classical first-order term-algebra by introducing variables that can take arguments and therefore have an arity. We retain the convention that variables are denoted by uppercase letters while function symbols are written as lower case letters.

Definition 15 (Higher-order Terms) *Let Σ be a signature. The set of variables \mathcal{V}^* is the union of infinite sets of variables \mathcal{V}_n for every $n \in \mathbb{N}_0$. We say that a variable has arity n if it is an element of the set \mathcal{V}_n . Every variable $V \in \mathcal{V}$ has the form $V : \sigma_1 \times \sigma_2 \times \dots \times \sigma_n \rightarrow \sigma$ where $\sigma_1, \dots, \sigma_n, \sigma \in \text{Sort}_\Sigma$. The set $\text{Term}^*(\Sigma, \mathcal{V}^*)$ relative to \mathcal{V}^* is defined as the smallest set such that the following conditions hold:*

1. If $X : \sigma \in \mathcal{V}_0$ then $X : \sigma \in \text{Term}^*(\Sigma, \mathcal{V}^*)$.
2. If $f : \sigma_1 \times \sigma_2 \times \dots \times \sigma_n \rightarrow \sigma \in \Sigma$ and $t_i : \sigma_i \in \text{Term}^*(\Sigma, \mathcal{V}^*)$, then $f(t_1, \dots, t_n) : \sigma \in \text{Term}^*(\Sigma, \mathcal{V}^*)$.
3. If $F : \sigma_1 \times \sigma_2 \times \dots \times \sigma_n \rightarrow \sigma \in \mathcal{V}^*$ and $t_i : \sigma_i \in \text{Term}^*(\Sigma, \mathcal{V}^*)$, then $F(t_1, \dots, t_n) : \sigma \in \text{Term}^*(\Sigma, \mathcal{V}^*)$.

We call an element of this constructed set a term of the term algebra $\text{Term}^*(\Sigma, \mathcal{V}^*)$ and a term that contains any variable with arity higher than zero a higher-order term.

The set \mathcal{V}_0 contains all first-order variables. One can see that definition 15 is an extended version of definition 2 and therefore the set of terms $\text{Term}(\Sigma, \mathcal{V})$ is a subset of $\text{Term}^*(\Sigma, \mathcal{V}^*)$. Examples for first-order terms that are elements of both term algebras are X , c , $f(a, b)$, $h(a, g(X))$, $f(X, X)$. In contrast, $F(a, b)$, $F(X, c)$, $f(G(d))$ and $F(X, G(Y))$ are higher-order terms and only elements of $\text{Term}^*(\Sigma, \mathcal{V}^*)$.

3.1.2 Basic Substitutions

Using the new higher-order term-algebra we can define the notion of substitutions for restricted higher-order anti-unification. The modifier *restricted* stems directly from the fact that we do not allow arbitrary substitutions on higher-order terms, but only a few restricted types. These types were chosen manually to gain a useful form of anti-unification according to the outlined requirements in section 2.2.3 and has proven to be sufficient for the generation of generalizations in analogical reasoning with HDTP [SKKG08].

Definition 16 (Basic Substitutions) Given the term algebra $\text{Term}^*(\Sigma, \mathcal{V}^*)$, we define the following set of basic substitutions for restricted higher-order anti-unification:

1. A renaming $\rho^{F, F'}$ replaces a variable $F : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma \in \mathcal{V}_n$ by a variable $F' : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma \in \mathcal{V}_n$:

$$F(t_1 : \sigma_1, \dots, t_n : \sigma_n) : \sigma \xrightarrow{\rho^{F, F'}} F'(t_1 : \sigma_1, \dots, t_n : \sigma_n) : \sigma$$

2. A fixation ϕ_f^F replaces a variable $F : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma \in \mathcal{V}_n$ by a function symbol $f : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma \in \text{Func}_\Sigma$:

$$F(t_1 : \sigma_1, \dots, t_n : \sigma_n) : \sigma \xrightarrow{\phi_f^F} f(t_1 : \sigma_1, \dots, t_n : \sigma_n) : \sigma$$

3. An argument insertion $\iota_{G,i}^{F,F'}$ with $0 \leq i \leq n$, $F : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma \in \mathcal{V}_n$, $G : \sigma_i \times \dots \times \sigma_{i+k-1} \rightarrow \sigma_g \in \mathcal{V}_k$ with $k \leq n-i$ and $F' : \sigma_1 \times \dots \times \sigma_{i-1} \times \sigma_g \times \sigma_{i+k} \times \dots \times \sigma_n \rightarrow \sigma \in \mathcal{V}_{n-k+1}$ is defined as:

$$F(t_1 : \sigma_1, \dots, t_n : \sigma_n) : \sigma \xrightarrow{\iota_{G,i}^{F,F'}}$$

$$F'(t_1 : \sigma_1, \dots, t_{i-1} : \sigma_{i-1}, G(t_i : \sigma_i, \dots, t_{i+k-1} : \sigma_{i+k-1}) : \sigma_g, t_{i+k} : \sigma_{i+k}, \dots, t_n : \sigma_n) : \sigma$$

4. A permutation $\pi_\alpha^{F,F'}$ with $F : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma \in \mathcal{V}_n$ and $F' : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma \in \mathcal{V}_n$ together with a bijective function $\alpha : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ which is not the identity function, rearranges the arguments of a term:

$$F(t_1 : \sigma_1, \dots, t_n : \sigma_n) : \sigma \xrightarrow{\pi_\alpha^{F,F'}} F'(t_{\alpha(1)} : \sigma_{\alpha(1)}, \dots, t_{\alpha(n)} : \sigma_{\alpha(n)}) : \sigma$$

We write the chaining of basic substitutions $\tau_1, \tau_2, \dots, \tau_n$ as $[\tau_1, \tau_2, \dots, \tau_n]$ which has the application structure $\text{apply}(\text{apply}(\text{apply}(s, \tau_1), \dots), \tau_n)$ on a term s . The application order therefore is from left to right. Every chain of basic substitutions $[\tau_1, \tau_2, \dots, \tau_n]$ is a substitution.

For the ease of explaining the framework we notate terms without their sorts in the following definitions and examples. For a look at how sorts can be further utilized in restricted higher-order anti-unification see ahead to section 5.

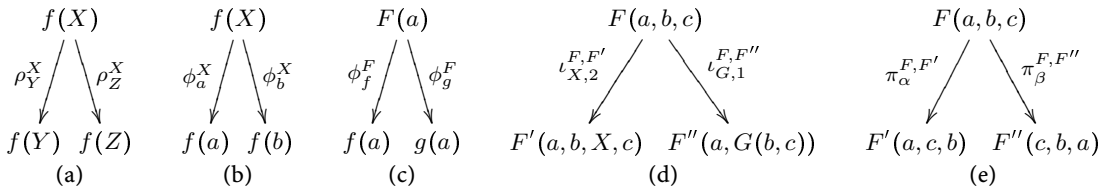


Figure 3.1: Examples for all basic substitutions from definition 16.

Examples for every basic substitution within a generalization can be found in figure 3.1 along with the resulting terms. Figure 3.1(a) shows an example of renaming that is a relatively simple substitution form. The variable X in the anti-unifier is renamed to Y to gain the term $f(Y)$ and to Z to gain the term $f(Z)$. One can note that these three terms are equivalent by definition 8. This results from the property that the variable renaming of the substitution $[\rho_Y^X]$ applied to $f(X)$ is reversible by the substitution $[\rho_X^Y]$. If the terms created by the application of renaming are in the same equivalence class, they are also strongly equivalent because renaming only replaces variables by variables of the same arity. However, renaming does not always lead to a term that is

equivalent. For example, the substitution $[\rho_Y^X]$ applied to $f(X, Y)$ results in $f(Y, Y)$, which cannot be transformed back to $f(X, Y)$ by renaming or any composition of basic substitutions.

Fixation as shown in figure 3.1(b) and (c) is used to substitute a variable by a symbol of the same arity. We can distinguish between two forms of fixation. The first form shown in (b) is the fixation of a first-order variable X . It is a first-order variable because it is an element of \mathcal{V}_0 and therefore has arity zero. The example in (c) shows fixation of a higher-order variable and represents the second case of *fixation*. The higher-order variable F involved here is an element of \mathcal{V}_1 and has arity one. This substitution is useful for anti-unification when function symbols differ in the anti-unified terms and leads to a much better preservation of structure than in first-order anti-unification. In first-order anti-unification, the anti-unifier in the generalization shown in (c) is a single first-order variable.

Argument Insertion or just *insertion* is a more challenging substitution because it can change the argument structure and arity of a term. Insertion can be used to increase and decrease the arity of a term. Inserting a variable of arity zero increases the arity of the embedding term by one. An example for this is given in figure 3.1(d) on the right side where X is inserted into $F(a, b, c)$ at position 2 to yield $F'(a, b, X, c)$. Note that insertion does only allow for insertion of a variable into an argument slot of a higher-order variable and not a function symbol. This means inserting the variable X in $F(a, b, c)$ is possible, inserting the variable X at position 2 in $f(a, b, c)$ or inserting a constant c in $F(a, b, c)$ is not valid. Inserting a term at position 0 means insertion before all current arguments, e.g. inserting d at position 0 into $F(a, b, c)$ results in $F'(d, a, b, c)$. The opposite, inserting at n where n is the number of current arguments of the term means inserting after all current arguments e.g. inserting d at position three into $F(a, b, c)$ results in $F'(a, b, c, d)$.

Inserting a variable I of arity higher than zero basically embeds, the already present subterms into a new subterm with function name I . An example for this is shown on the left side of figure 3.1(d). Whereas insertion of a variable with arity one does not change the arity of the embedding term, the insertion of a variable with arity higher than one does decrease the arity of the embedding term. For example, insertion of G with $G \in \mathcal{V}_3$ at position zero in $F(a, b, c)$ yields $F'(G(a, b, c))$, and therefore reduces the arity of the embedding term from three to one. Insertion of a variable with arity one or higher can therefore be better described as embedding than insertion.

Permutation is used for rearranging arguments of a term and does not change the arity of the term. However, the higher-order variable belonging to the arguments that

are permuted is exchanged for a new higher-order variable of the same arity. As with insertion, this operation can only operate on a higher-order variable. The permutation of arguments from (a, b, c) to (c, b, a) is shown on the right side of figure 3.1(e). Exchanging no arguments, which equals permutation with the identity function, is not regarded as a valid basic substitution.

3.1.3 Higher-Order Substitutions

As mentioned in definition 6, we call the composition of one or more of the outlined four basic substitutions types a *substitution* and specifically a *higher-order substitution*. Through composition of renaming, insertion and fixation of every first-order substitution can be described with a chain of basic substitutions. Therefore, every anti-unifier for a set of terms in first-order anti-unification is also a valid anti-unifier in this higher-order anti-unification framework. In other words, restricted higher-order anti-unification is a proper extension of first-order anti-unification. An example of the higher-order equivalent of the first-order substitution $\{X/f(a)\}$ can be seen in figure 3.2. The substitution depicted there is $[\iota_{V,0}^{X,X'}, \phi_a^V, \phi_f^{X'}]$ with $X, V \in \mathcal{V}_0$ and $X' \in \mathcal{V}_1$. We omit explicit mention of variable arity in substitutions within figures from now on, because they are obvious from the argument-structures of the terms shown.

$$X \xrightarrow{\iota_{V,0}^{X,X'}} X'(V) \xrightarrow{\phi_a^V} X'(a) \xrightarrow{\phi_f^{X'}} f(a)$$

Figure 3.2: Example for composition of basic substitutions to mimic a first-order substitution.

Three examples for higher-order substitutions applied to $F(a)$ that cannot be expressed by first-order substitutions are shown in figure 3.3. The depicted substitutions are:

- (a) $[\iota_{X,0}^{F,F'}, \iota_{Y,1}^{X,X'}, \phi_g^{X'}, \phi_b^Y]$
- (b) $[\iota_{X,0}^{F,F'}, \pi_\alpha^{F',F''}, \iota_{Y,0}^{X,X'}, \phi_g^{X'}, \phi_f^{F''}]$
- (c) $[\iota_{X,1}^{F,F'}, \rho_{F''}^{F'}, \iota_{Y,0}^{X,X'}, \phi_g^{X'}, \phi_f^{F''}]$

Note that figure 3.3(b) and (c) both express $F(a) \rightarrow f(a, g(Y))$ but differ in the first two basic substitutions applied. The basic substitution chains $[\iota_{X,0}^{F,F'}, \pi_\alpha^{F',F''}]$ and

$$\begin{aligned}
\text{(a)} \quad & F(a) \xrightarrow{\iota_{X,0}^{F,F'}} F'(X(a)) \xrightarrow{\iota_{Y,1}^{X,X'}} F'(X'(a,Y)) \xrightarrow{\phi_g^{X'}} F'(g(a,Y)) \xrightarrow{\phi_b^Y} F'(g(a,b)) \\
\text{(b)} \quad & F(a) \xrightarrow{\iota_{X,0}^{F,F'}} F'(X,a) \xrightarrow{\pi_\alpha^{F',F''}} F''(a,X) \xrightarrow{\iota_{Y,0}^{X,X'}} F''(a,X'(Y)) \xrightarrow{\phi_g^{X'}} F''(a,g(Y)) \xrightarrow{\phi_f^{F''}} f(a,g(Y)) \\
\text{(c)} \quad & F(a) \xrightarrow{\iota_{X,1}^{F,F'}} F'(a,X) \xrightarrow{\rho_{F''}^{F'}} F''(a,X) \xrightarrow{\iota_{Y,0}^{X,X'}} F''(a,X'(Y)) \xrightarrow{\phi_g^{X'}} F''(a,g(Y)) \xrightarrow{\phi_f^{F''}} f(a,g(Y))
\end{aligned}$$

Figure 3.3: Three substitutions that are depicted by chains of basic substitutions.

$[\iota_{X,1}^{F,F'}, \rho_{F''}^{F'}]$ applied to $F(a)$ both result in the term $F''(a, X)$. This shows that more than one basic substitution chain can lead to the same instance of a term. The application of $[\iota_{X,1}^{F,F'}]$, as shown in (c), already yields the term $F'(a, X)$, which is strongly equivalent to $F''(a, X)$. Therefore, the same equivalence class of terms is reached here by $[\iota_{X,0}^{F,F'}, \pi_\alpha^{F',F''}]$ in (b) and $[\iota_{X,1}^{F,F'}]$ in (c) when applied to $F(a)$.

3.2 Complexity Measures

3.2.1 Information Load

In the following it is shown that this framework, while a form of higher-order anti-unification, is still sufficiently restricted to be of practical use and adheres to the standards outlined at the beginning of this chapter. To prove that anti-instances are not more complex than the original term we introduce a complexity measure called *information load*.

Definition 17 (Information Load) *A term t is assigned the information load $\mathfrak{I}(t)$, which is recursively defined as follows:*

1. $\mathfrak{I}(F(t_1, \dots, t_n)) = n + \mathfrak{I}(t_1) + \dots + \mathfrak{I}(t_n)$ for a n -ary variable $F \in \mathcal{V}_n$ and terms $t_1, \dots, t_n \in \text{Term}^*(\Sigma, \mathcal{V}^*)$.
2. $\mathfrak{I}(f(t_1, \dots, t_n)) = 1 + n + \mathfrak{I}(t_1) + \dots + \mathfrak{I}(t_n)$ for a function symbol $f \in \text{Func}_\Sigma$ and terms $t_1, \dots, t_n \in \text{Term}^*(\Sigma, \mathcal{V}^*)$.

Basically, information load is the sum of the number of argument slots and symbols in a term. The information load of a constant is one and the information load of a variable $X \in \mathcal{V}_0$ is zero. Information load of the term $f(c, g(X))$ is six which can be written as $\mathfrak{I}(f(c, g(X))) = 6$.

Lemma 1 *Let s and t be terms. If $s \rightarrow t$, then $\mathfrak{I}(s) \leq \mathfrak{I}(t)$.*

Lemma 1 was proven by inductive decomposition in [KSGK07]. This lemma states that the application of substitutions never reduces the information load of a term. Permutation and renaming produce an instance of a term that has the same information load than the original term. Corollary 1 is derived by the observation that there is only a finite set of permutations for arguments of a given arity and that the information load of a term is higher than zero.

Corollary 1 *For a given term t there are is only a finite number of anti-instances up to renaming.*

One can rephrase this as there is only a finite number of equivalent classes of anti-instances for a given term. In figure 3.2 it was exemplified that every anti-instance produced by substitutions of first-order anti-unification can also be produced by higher-order substitutions. The most general anti-unifier, which is a variable in first-order anti-unification is therefore still a valid anti-unifier in restricted higher-order anti-unification and still the most general. From definition 10 we can see that every anti-unifier for a set of terms must be an anti-instance for each term in that set. With corollary 1 it follows that there is only a finite number of anti-unifiers (up to renaming) for a set of terms because there exists only a finite number of anti-instances (up to renaming) for each term. By lemma 1 it furthermore follows that the anti-unifier being an anti-instance has an information load that is less or equal than the anti-unified terms. The mentioned findings are documented in proposition 1.

Proposition 1 *There is only a finite number of anti-unifiers (up to renaming) for a set of terms. For every anti-unifier a and every term t of the set of anti-unified terms it holds $\mathfrak{I}(a) \leq \mathfrak{I}(t)$.*

Anti-unifiers are well defined within restricted higher-order anti-unification, because anti-unifiers always exist for a pair of terms, are finite up to the renaming of variables and are always less complex than the anti-unified terms. However, one drawback is that most specific anti-unifiers are not unique in comparison to first-order anti-unification. As a result, least general generalizations are also not unique anymore in the context of restricted higher-order anti-unification. An example of a pair of terms with more than one most specific anti-unifier is shown in figure 3.4. Here $f(X, Y)$, $F(d, G(a))$ and $F'(G(a), d)$ are most specific anti-unifiers for the pair of terms $f(g(a, b, c), d)$ and $f(d, h(a))$. All three generalizations shown are therefore least general generalizations. Note that, $F(d, G(a))$ and $F'(G(a), d)$ can be obtained by permutation from each other and are therefore, by definition 8, in the same equivalence class of terms. In contrast, $f(X, Y)$ is in a different equivalence class than $F(d, G(a))$ and $F'(G(a), d)$. Thus, we have not only multiple most specific anti-unifiers but also two separate equivalence classes of most specific anti-unifiers as

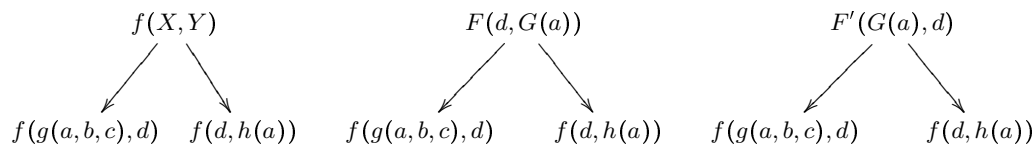


Figure 3.4: Example with multiple least general generalizations

shown in figure 3.4.

While we extended the notion of terms and substitutions from first to a higher-order version the definition of generalization does not differ from definition 13 in section 2.2.2. A generalization by our definition consists of an anti-unifier and a pair of substitutions which, when applied to the anti-unifier, yield the anti-unified terms. From proposition 1 we conclude that there exist only a finite number of classes of generalizations that have a differing anti-unifier (up to renaming) for a given pair of anti-unified terms. However, there are an infinite number of generalizations in each class. This results from the property that there are an infinite number of different chains of basic substitutions from the anti-unifier to each of the anti-unified terms. Valid chains of basic substitutions that already lead from the anti-unifier to the anti-unified terms can be further extended by renaming and permutation without changing the result of their application. These chains contain unnecessary basic substitutions but are still valid because there is no restriction on length of those chains.

There are multiple classes of least general generalizations that differ in the equivalence class of anti-unifiers used, because there exist multiple most specific anti-unifiers that are not in the same equivalence class. Figure 3.4 depicts three different classes of least general generalizations. Note that the substitutions are not explicitly stated because there exists an infinite number of valid basic substitution chains with which the arrows could be annotated in each of the classes.

3.2.2 Complexity of Substitutions

Multiple classes of least general generalizations are not necessary disadvantageous. In analogy making several different mappings with different degrees of plausibility may coexist. Thus, we need a complexity measure for ranking generalizations and therefore have a criterion to rank for alternative least general generalizations. This ranking may be based on the already defined information load. Information load of an anti-unifier, which is part of a generalization, does not take into consideration the complexity of the substitutions involved. However, substitutions need to be considered in the complexity measure because we want to promote their reuse and want to anti-unify full sets of terms.

Definition 18 (Complexity of Substitutions) *The complexity of a basic substitution τ is defined as:*

$$\mathcal{C}(\tau) = \begin{cases} 0 & \text{if } \tau = \rho & \text{(renaming)} \\ 1 & \text{if } \tau = \phi_f & \text{(fixation)} \\ k + 1 & \text{if } \tau = \iota_{V,i} \text{ and } V \in \mathcal{V}_k & \text{(argument insertion)} \\ 1 & \text{if } \tau = \pi_\alpha & \text{(permutation)} \end{cases}$$

For a composition of basic substitutions we define $\mathcal{C}([\tau_1, \dots, \tau_m]) = \sum_{i=1}^m \mathcal{C}(\tau_i)$.

The complexity of a substitution is meant to reflect its computational processing effort. Therefore permutations have a non-zero complexity even though they do not change the information load of a term. Argument insertion restructures the term, and the higher the arity of the inserted variable, the more arguments are moved and therefore more complexity is assigned to the argument insertion operation. We choose to linearly scale the complexity of insertion according to the arity of the inserted higher-order variable. The renaming substitution does not change the equivalence class and structure of a term and therefore has a complexity measure of zero. A substitution that is composed of basic higher-order substitutions has the complexity of the sum of those basic substitutions. The complexity values for basic substitutions have proven to generate plausible analogies when used by HDTP. The analysis of different complexity values is subject of future work. The complexity of a generalization can be defined in a straightforward manner by basing it on the complexity of its substitutions.

Definition 19 (Complexity of Generalizations) *Let $\langle g, \tau, \nu \rangle$ be a generalization for a pair of terms $\langle s, t \rangle$. Define the complexity of the generalization by $\mathcal{C}(\langle g, \tau, \nu \rangle) = \mathcal{C}(\tau) + \mathcal{C}(\nu)$.*

Figure 3.5 shows three possibilities for substitutions that transform $f(X, Y)$ into $f(d, h(a))$. They all have a complexity of 4, which is the minimum achievable for $f(X, Y) \rightarrow f(d, h(a))$. Therefore they are equal candidates to be substitutions used in the right side of the generalization template in figure 3.4(a). We say template here because the substitutions used must to be explicitly stated for it to be a fully specified generalization.

A substitution yielding $f(g(a, b, c))$ when applied to $f(X, Y)$, has a minimum complexity of 8. Such a substitution must be at least composed of the five fixations for g , a , b , c and d . Furthermore, three insertions are needed to produce the argument structure for g . Summing the complexities for fixations and insertions we gain a complexity of 8 for the complete chain of basic substitutions needed.

$$\begin{aligned}
\text{(a)} \quad & f(X, Y) \xrightarrow{\phi_d^X} f(d, Y) \xrightarrow{\iota_{Z,0}^{Y,Y'}} f(d, Y'(Z)) \xrightarrow{\phi_h^{Y'}} f(d, h(Z)) \xrightarrow{\phi_a^Z} f(d, h(a)) \\
\text{(b)} \quad & f(X, Y) \xrightarrow{\iota_{Z,0}^{Y,Y'}} f(X, Y'(Z)) \xrightarrow{\phi_a^Z} f(X, Y'(a)) \xrightarrow{\phi_d^X} f(d, Y'(Z)) \xrightarrow{\phi_h^{Y'}} f(d, h(a)) \\
\text{(c)} \quad & f(X, Y) \xrightarrow{\iota_{Z,0}^{Y,Y'}} f(X, Y'(Z)) \xrightarrow{\phi_h^{Y'}} f(X, h(Z)) \xrightarrow{\rho_{Z'}^Z} f(X, h(Z')) \xrightarrow{\phi_d^X} f(d, h(Z')) \xrightarrow{\phi_a^{Z'}} f(d, h(a))
\end{aligned}$$

Figure 3.5: Examples for $f(X, Y) \rightarrow f(d, h(a))$.

An example for a substitution with complexity 8 for $f(X, Y) \rightarrow f(g(a, b, c), d)$ is $[\iota_{Z,0}^{X,X'}, \iota_{U,1}^{X',X''}, \iota_{V,2}^{X'',X'''}, \rho_g^{X'''}, \rho_a^Z, \rho_b^U, \rho_c^V, \rho_d^Y]$ which is depicted in figure 3.6. Therefore, we have a minimal complexity of 12 for a generalization conforming to figure 3.4(a), consisting of a minimal complexity of 8 for $f(X, Y) \rightarrow f(g(a, b, c), d)$ and of a minimal complexity of 4 for $f(X, Y) \rightarrow f(d, h(a))$.

$$\begin{aligned}
& f(X, Y) \xrightarrow{\iota_{Z,0}^{X,X'}} f(X'(Z), Y) \xrightarrow{\iota_{U,1}^{X',X''}} f(X''(Z, U), Y) \xrightarrow{\iota_{V,2}^{X'',X'''}} f(X'''(Z, U, V), Y) \\
& \xrightarrow{\rho_g^{X'''}} f(g(Z, U, V), Y) \xrightarrow{\rho_a^Z} f(g(a, U, V), Y) \xrightarrow{\rho_b^U} f(g(a, b, V), Y) \xrightarrow{\rho_c^V} f(g(a, b, c), Y) \xrightarrow{\rho_d^Y} f(g(a, b, c), d)
\end{aligned}$$

Figure 3.6: Example basic substitution chain for $f(X, Y) \rightarrow f(g(a, b, c), d)$.

However, the minimal complexity is 9 for generalizations where the terms $f(g(a, b, c), d)$ and $f(d, h(a))$ are anti-unified. The composition of the anti-unifier $F(d, G(a))$ as seen in figure 3.4(b), together with a substitution of complexity 7 for $F(d, G(a)) \rightarrow f(g(a, b, c), d)$, and a substitution of complexity 2 for $F(d, G(a)) \rightarrow f(d, h(a))$, is an example of this. The anti-unifier $F'(G(a), d)$ in figure 3.4(c), with corresponding substitutions, can also be made into a generalization that has the minimal complexity of 9.

3.2.3 Preferred Generalizations

Before we define generalizations, we have to observe that there is only a slight difference in form of a renaming between the substitutions in figure 3.5(b) and (c). This renaming neither alters the structure nor does it help to produce the instance $f(d, h(a))$ in any fewer basic substitutions. In-fact, it only delays it. However, this is not reflected in the substitution complexity measure because renaming has a complexity of zero.

The only situation in which renaming is needed to produce an anti-unifier is when the variables of the two anti-unified terms have variables in the same argument positions and are unifiable. Therefore, we can take an altered view on which substitutions

we prefer in generalizations. For two terms s and t we search for substitutions τ and ν with $s' \xrightarrow{\tau} s$ and $t' \xrightarrow{\nu} t$ such that for s' and t' there exists an anti-unifier a with $a \xrightarrow{\Theta_1} s'$ or $a \xrightarrow{\Theta_2} t'$ where Θ_1 and Θ_2 are chains composed of only basic renaming substitutions. For example if the terms are $F(a)$ and $G(c)$ we can find $F(X) \xrightarrow{\phi_a^X} F(a)$ and $G(Y) \xrightarrow{\phi_c^Y} G(c)$ where $F(X)$ and $G(Y)$ are strongly equivalent terms. However, the found anti-instances need not be always strongly equivalent as in this case. To anti-unify $F(X)$ and $G(Y)$ we only need two basic renaming substitutions. One possibility is $[\rho_G^F]$ and $[\rho_Y^X]$. Because of this, our chain of basic substitutions should only have all basic renaming substitutions before all other basic substitutions. A chain as in figure 3.5(c) is therefore discouraged for use in generalizations because $[\rho_{Z'}^Z]$ is preceded by basic substitutions that are not of the type renaming.

Not only should basic renaming substitutions be first, but also restricted to the minimum amount needed. For our example of renaming $F(X)$ to $G(Y)$ we could also use the substitution $[\rho_H^F, \rho_G^H, \rho_Y^X]$ which is unnecessary long, because $[\rho_H^F, \rho_G^H]$ can be shortened to just $[\rho_G^F]$. To gain a minimal amount of substitutions, we require that renaming in a substitution only renames a variable that is not further renamed. Note that this does not rule out the possibility of an anti-unifier that is completely free of variables occurring in the anti-unified terms. The anti-unifier $H(Z)$ for $F(a)$ and $G(c)$ together with the basic substitution chains $H(Z) \xrightarrow{\rho_F^H} F(Z) \xrightarrow{\phi_a^Z} F(a)$ and $H(Z) \xrightarrow{\rho_G^H} G(Z) \xrightarrow{\phi_c^Z} G(c)$ do adhere to the minimality of renaming, as it is defined on the individual substitutions used. The construction of anti-unifiers that do not contain variables from the original terms is necessary to build terms in which variables have no other direct relation than renaming to other variables. Such variables, which are not associated with the anti-unified pair of terms, except in renaming, are used in the terms of the generalized theory constructed in HDTP and are produced by the algorithm for preferred generalizations in 4.1.

Given the restrictions on the use of renaming we can now define the notion of preferred generalizations:

Definition 20 (Preferred Generalization) *A preferred generalization $\langle g, \tau, \nu \rangle$ is a least general generalization for the pair of terms s and t such that there is no other least general generalization that has less complexity and τ and ν contain only the minimal amount of basic renaming substitutions required at their beginning.*

In contrast to least general generalizations there are only a finite number of preferred generalizations (up to renaming of the anti-unifier and the basic substitution chains) for a pair of terms.

3.3 Reuse of Substitutions

Anti-unification within HDTP is used to produce a set of anti-unifiers for a set of terms of source and target domains by building generalizations. Subterms within the source and target domains can appear more than once. The constants *sun* and *planet*, for example are used within the terms $mass(sun) > mass(planet)$ and $dist(planet, sun, T)$ on the source side within the Rutherford analogy formalized in figure 2.1. Furthermore, in the target domain the terms $mass(nucleus) > mass(electron)$ and $dist(electron, nucleus, T)$ have the same subterms *electron* and *nucleus*. A generalization for $mass(sun) > mass(planet)$ and $mass(nucleus) > mass(electron)$ is shown in figure 3.7(a). In (b) the depicted generalization is between $dist(planet, sun, T)$ and $dist(electron, nucleus, T)$.

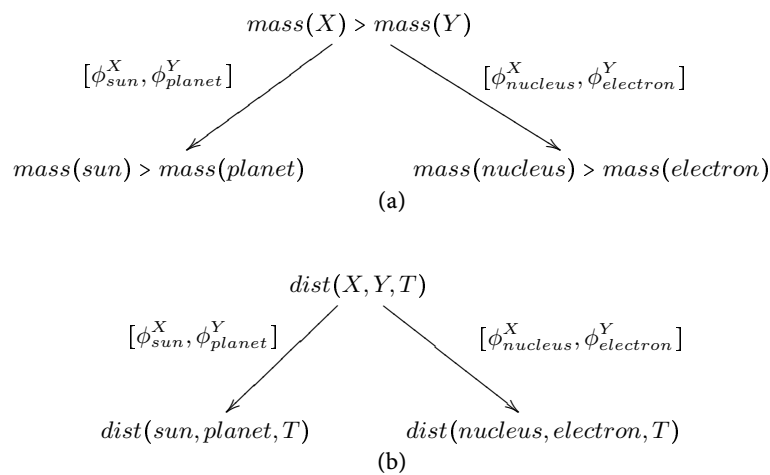


Figure 3.7: Examples for generalizations in the Rutherford analogy.

Both generalizations use two fixations for each substitution and therefore have a complexity of 4. Both generalizations combined have a complexity of 8. However, the fixations used in these two generalizations are the same. It is cognitively plausible if already used substitutions and constructed mappings do not add complexity on reuse. Because HDTP builds generalizations sequentially one by one, the desired process should be that when the generalization in (a) is already made and a new generalization as in (b) is constructed, the complexity assigned to (b) should be 0.

In order to define the desired mechanism of reuse for substitutions, we first examine a few problems that arise from a simple definition of reuse. We could simply define reuse as follows: the complexity of an already used basic substitution during the mapping process of HDTP is zero. This yields a form of reuse that is not desired. The root of the problem is that the variable F in the basic substitutions ϕ_f^F , $\iota_1^{F, F'}$ and $\phi_\alpha^{F, F'}$ is

not in any direct relation to the other parameters such as F' , I or α . Therefore, these basic substitutions can be used as if they did an additional basic renaming substitution. This is a problem because renaming should only be at the beginning of a basic substitution chain, as discussed in section 3.2.3.

$$\begin{aligned}
 \text{(a)} \quad & F \xrightarrow{\iota_{X,0}^{F,F'}} F'(X) \xrightarrow{\iota_{Y,1}^{F',F''}} F''(X,Y) \xrightarrow{\iota_{Z,2}^{F'',F'''}} F'''(X,Y,Z) \\
 \text{(b)} \quad & G \xrightarrow{\iota_{X,0}^{G,G'}} G'(X) \xrightarrow{\iota_{Y,1}^{G',G''}} G''(X,Y) \xrightarrow{\iota_{Z,2}^{G'',G'''}} G'''(X,Y,Z) \\
 \text{(c)} \quad & G \xrightarrow{\rho_F^G} F \xrightarrow{\iota_{X,0}^{F,F'}} F'(X) \xrightarrow{\iota_{Y,1}^{F',F''}} F''(X,Y) \xrightarrow{\iota_{Z,2}^{F'',G'''}} G'''(X,Y,Z)
 \end{aligned}$$

Figure 3.8: Example for use of insertion as renaming.

In figure 3.8 we see three substitutions that illustrate how insertion can be used as a replacement for a basic renaming substitution. Let us assume that the basic substitutions made in (a) do not add more complexity on the next reuse. Now we want to find a substitution with minimal complexity for $G \rightarrow G'''(X,Y,Z)$. One possibility is given in (b) and the basic substitution chain shown there has a complexity of 3, because no basic substitution from (a) is used again. The substitution $[\rho_F^G, \iota_{X,0}^{F,F'}, \iota_{Y,1}^{F',F''}, \iota_{Z,2}^{F'',G'''}]$, which corresponds to (c), reuses the basic substitutions $\iota_{X,0}^{F,F'}$ and $\iota_{Y,1}^{F',F''}$ and therefore only has complexity 1. This is certainly not the intended case of reuse as F and G share no connection here, but rather are just variables. This problem is not restricted to insertion because permutations and fixations can be used to the same effect. The basic renaming substitution is already restricted to the minimal amount of applications needed and can only be used in the beginning of a basic substitution chain. We therefore require no further restriction on its use.

We observe that F'' and F''' have an explicit relation once $\iota_{Z,2}^{F'',F'''}$ is used to connect them. The basic substitution $\iota_{Z,2}^{F'',F'''}$ states that a term with variable F'' as function symbol is obtained if we insert the variable Z at position 2 into a term with variable F''' as function symbol. It is only plausible that no other variable than F'' has this connection with F''' using the parameters Z and 2 for the insertion. The basic substitution $\iota_{Z,2}^{F'',G'''}$ from figure 3.8(c) therefore violates this restriction. We can require the same for the connections established between variables and constants by fixation. If we used ϕ_f^F then F should not be renamed into any other symbol than f . If there exists a permutation $\pi_\alpha^{F,F'}$, then there should not be a permutation $\pi_\alpha^{F,F''}$, because F' and F'' differ while using the same permutation function α .

We can therefore leave out the parameter F' in $\iota_{V,P}^{F,F'}$ and $\pi_{\alpha}^{F,F'}$, as well as the function symbol f in ϕ_f^F , because they can be inferred from the other parameters. To achieve the mentioned restrictions, we introduce the function \mathcal{S} , which takes these basic substitutions with the inferable parameter missing and maps them to a variable in case of inserting and permutation and to a function symbol in case of fixation. Because $\iota_{Z,2}^{F'',F''}$ is valid in figure 3.8(a), $\mathcal{S}(\iota_{Z,2}^{F''}) = F'''$ holds. This prevents the usage of $\iota_{Z,2}^{F'',G''}$ in (c) because $\mathcal{S}(\iota_{Z,2}^{F''}) = G'''$ cannot be true at the same time as $\mathcal{S}(\iota_{Z,2}^{F''}) = F'''$ is and therefore (c) is not a valid substitution chain anymore.

If we extend the definition of basic substitutions to incorporate this function to restrict the possible basic substitutions further, we can define reuse of substitutions as originally intended:

Definition 21 (Complexity of reused Substitutions) *Let g_1, g_2, \dots, g_n be generalizations constructed after each other in this order. Any basic substitution τ that is used in g_j with the same parameters as used in g_i has a complexity of 0 if $1 \leq i < j \leq n$.*

With this defining the combined complexity of the generalizations in figure 3.7 becomes 4. The complexity is 8 when reuse of substitutions is not allowed.

4 Computation of Preferred Generalizations

4.1 Algorithm

In this section we present the development of a bottom up algorithm to compute preferred generalizations in the context of restricted higher-order anti-unification, with complexity free reuse. It is called *bottom up* because the anti-unified terms are used as a starting point for constructing a suitable anti-unifier. By definition, preferred generalizations are a subset of generalizations. Therefore, we start with a general algorithm that produces all generalizations and refine it to only generate preferred generalizations. In section 4.2 we outline some further changes to the algorithm to reduce its runtime. Finally, in section 4.3 some details about a reference implementation in Prolog are outlined.

Shown in figure 4.1 is pseudo code for an algorithm that generates all generalizations together and computes their complexity. For each of the two input terms the algorithm generates chains of basic substitutions. Each chain starts from an anti-instance of the corresponding input term. An application of a chain to the associated anti-instance results in the corresponding input term. While generating chains it is checked whether a chain was generated from an anti-instance that is also the starting point of a chain from the other input term. Therefore, it is checked whether or not the found anti-instance is an anti-unifier for the two input terms. The anti-unifier, together with the found chains corresponding to opposite input terms, is thereby a valid generalization.

Note that this algorithm as such is not usable since it never terminates. This is grounded in the fact that while there is only a finite number of anti-instances and anti-unifier up to renaming (corollary 1 & proposition 1), there can be an infinite number of anti-instances for a given term. For example $f(X)$ has an infinite number of anti-instances $f(X')$, $f(X'')$, \dots due to renaming. Another source of the problem of infinite basic-substitution chains is permutation. Permutations of terms like $F(X, Y)$ produce anti-instances that are equivalent up to renaming within the statement in line

26. These created equivalent terms can be permuted and renamed further into equivalent terms again. This chain never breaks up and by adding the newly created terms to the open list in line 27 the open list never becomes empty.

The swap in line 23 is needed to generate generalizations that first state the basic substitution chain for the left term and then for the right term, as compared to having them in the reverse order in cases where an anti-instance of a right term is matched against the closed list in line 21. The complexity reduction for reuse is assumed to be handled directly within the function \mathcal{C} in line 26.

```

01 function anti_unify(left_term, right_term) returns generalizations
02   // anti_instance is a quadruple  $\langle c, s, g, i \rangle$  where
03   //   c is the sum of complexities of the substitutions in s
04   //   s is a basic substitution chain
05   //   g is the generalized term to which s can be applied
06   //   i is the index  $\in \{\text{"left"}, \text{"right"}\}$  with
07   //     op(“left”) = “right” and op(“right”) = “left”
08   // generalization is a quadruple  $\langle c, g, s\_left, s\_right \rangle$  where
09   //   c is the sum of complexities of the substitutions from s_left and s_right
10   //   s_left and s_right are basic substitution chains
11   //   g is the generalized term to which s_left and s_right can be applied
12   variables:
13     Open: list of anti_instance sorted by complexity
14     Closed: list of anti_instance
15     Generalizations: list of generalization sorted by complexity
16   initialize:
17     Open =  $\{(0, [], left\_term, \text{"left"}), (0, [], right\_term, \text{"right"})\}$ 
18     Closed =  $\{\}$ 
19   while Open  $\neq \{\}$  do
20      $\langle c, s, g, i \rangle = \text{first}(\textit{Open})$ ; Open = rest(Open)
21     for each  $\langle c\_match, s\_match, g, op(i) \rangle \in \textit{Closed}$ 
22       c_sum = c + c_match
23       if i == “right” do swap(s, s_match)
24       Generalizations = merge(Generalizations,  $\langle c\_sum, g, s, s\_match \rangle$ )
25     end for each
26     Ai =  $\{\langle c', s', g', i \rangle \mid \exists \tau \in \textit{basic\_subst} : g' \xrightarrow{[\tau]} g, c' = c + \mathcal{C}(\tau), s' = [\tau] + s\}$ 
27     Open = merge(Open, Ai)
28     Closed = Closed  $\cup \{\langle c, s, g, i \rangle\}$ 
29   end while
30   return Generalizations
31 end function

```

Figure 4.1: Algorithm for computing generalizations.

We can use the pseudo code for generating generalizations and rewrite it to only generate preferred generalizations. We use the scheme outlined in section 3.2.3 and first generate basic substitution chains without renaming, then add renaming to the beginning of those chains to get a matching anti-unifier from the generated two basic substitution chains for the generalization. We have to make two changes to convert the algorithm. First, we have to restrict the basic substitutions that are used to generate new terms in line 26 to only use permutation, fixation and insertion. Because the complexity for these basic substitutions is higher than zero, the complexity of the newly generated terms becomes higher than the instance, from which they are generated from.

Next, the removed renaming substitutions with complexity zero have to be reintegrated elsewhere into the basic substitution chains. We exchange the condition in line 21 which matches a quadruple with the same generalization term and opposite index to a new condition. The new condition states that for the generalized term with the opposite index and for the generalized term that was in the first quadruple of the open list there has to exist an anti-unifier by which these two terms can be derived using only renaming. These additional renaming substitutions are generated and then added at the beginning of the corresponding already built up basic substitution chains.

Still, because of the basic permutation substitution the algorithm does not terminate with these modifications. To make the algorithm terminate we can use preferred generalizations that have a minimal combined complexity of the basic substitution chains involved. If we find a preferred generalizations with cost C , we can stop searching for new preferred generalization once we know that the ordered open list has as first element a quadruple with complexity higher than C . Basic substitutions only add complexity and therefore there cannot be a basic substitution chain that has negative complexity. If we know that a substitution chain already has a complexity higher than C , adding the complexity of another basic substitution chain (see line 22) does not make the complexity of the generated generalization equal or less than C . We therefore change the `while` condition in line 19 to reflect this additional termination condition. The created algorithm is a terminating, complete, optimal and sound search for preferred generalizations.

4.2 Optimizations

The algorithm for preferred generalizations from figure 4.2 gives a platform for further improvements. The graph in figure 4.3 shows us all of the different basic substitution paths that lead to $G(X, a)$ from its equivalence classes of anti-instances. In

```

01 function anti_unify(left_term, right_term) returns generalizations
02   initialize:
03     Open = {{0, [], left_term, "left"}, {0, [], right_term, "right"}}
04     Closed = {}
05   while Open ≠ {} and  $\mathcal{C}(\text{first}(\textit{Open})) = \mathcal{C}(\text{first}(\textit{Generalizations}))$  do
06      $\langle c, s, g, i \rangle = \text{first}(\textit{Open})$ ; Open = rest(Open)
07     for each  $\langle c_m, s_m, g_m, \text{op}(i) \rangle \in \textit{Closed}$  with  $(g \xleftarrow{[\rho, \dots]} a \xrightarrow{[\rho, \dots]} g_m)$ 
08       Au, r, rm = anti-unifier_by_renaming(g, gm)
09       csum = c + cm; s = r + s; sm = rm + sm
10       if i == "right" do swap(s, sm)
11       Generalizations = merge(Generalizations,  $\langle c\_sum, Au, s, s\_m \rangle$ )
12     end for each
13     Ai = {{c', s', g', i} |  $\exists \tau \in \{\pi, \iota, \phi\} : g' \xrightarrow{[\tau]} g, c' = c + \mathcal{C}(\tau), s' = [\tau] + s$ }
14     Open = merge(Open, Ai)
15     Closed = Closed ∪ {{c, s, g, i} }
16   end while
17   return least_complex(Generalizations)
18 end function

```

Figure 4.2: Algorithm for computing all preferred generalizations up to renaming.

section 3.2.2 it was outlined that there can be multiple substitution chains that lead from a term to an instance of that term. Note that for the least specific anti-unifier, which is the class of single variables depicted here by the term G , there are many different ways though the graph to reach $G(X, a)$. Our bottom up algorithm traverses the graph in the opposite direction starting from $G(X, a)$, and thus finding all the chains leading to G . In the set of paths there are a finite set of paths with minimal combined complexity of the basic substitutions used. One can observe that these are usually permutations of each other as seen in the paths from $G(X)$ to $G(X, a)$ via $G(a)$ and $G(X, Y)$. One can either first make an insertion of a variable Y as second argument and then fixate the second argument to a , or first make the fixation to a and then insert a variable as argument before a . Both have the same complexity because complexity is defined by the basic substitutions that are used and not by the order in which basic substitutions are applied.

The algorithm thereby does find many paths that have the same complexity and lead to an equivalent anti-instance. It is unnecessary overhead traversing a instantiation ordering graph that has loops of permutations that are guaranteed not to be in a basic substitution chain within a preferred generalization. Furthermore, finding one canonical representation of a basic substitution chain is good enough, as all other variations with the same complexity can be generated from it.

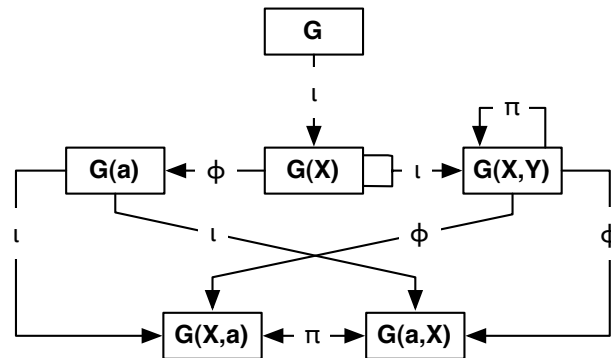


Figure 4.3: Instantiation ordering of anti-instance equivalence classes.

It is therefore proposed that we follow a standard order in which basic substitutions can be made. It already has been established in section 3.2.3 that renaming substitutions have to be made first. The order that has to be followed for the rest of the basic substitution chain is permutation, insertion, fixation. Whereas a proof that there always exist a minimal path in this order is subject to future work, there are rationale as to why no other order can achieve this. First, fixations cannot be made before insertions or permutations, because this would prevent fixations of variables that are a function symbol of a permuted term. Fixation is needed when anti-unifying $f(a, b)$ with $g(b, a)$ to gain the anti-unifier $F(a, b)$ or $F'(b, a)$. These two anti-unifiers need basic substitutions with total complexity of 3 to be generated. Permutation should be before insertion, as insertion can introduce subterms that are functions with arity higher than one. If permutation occurred after insertion and two of these new subterms needed to be permuted, it would be necessary to have two permutations with a combined complexity of 2. The same result can be achieved by one permutation, followed by insertion of the new variables at the right places. For example, $F(G(a, b), H(a, b))$ can be obtained from $F'(b, a, b, a)$ with a permutation to $F''(a, b, a, b)$ and then two insertions. If permutation was after insertion, two insertions would be required from $F'(b, a, b, a)$ to $F(G'(b, a), H'(b, a))$ and then two permutations to $F(G(a, b), H(a, b))$.

There still exists the possibility for permutations of basic substitutions within a chain of the same type of basic substitutions. This can be seen in figure 4.3 between the equivalence classes of terms $G(X)$ and $G(X, Y)$. To get to the equivalence class $G(X, Y)$ one can either insert a variable Y at position 0 into $G(X)$ to gain $G(Y, X)$ or at position 1 to gain $G(X, Y)$.

For fixation of $F(A, B)$ to $F(a, b)$ we can choose to first fixate the variable A to a and then the variable B to b or the other way around. A similar process holds for insertion. To generate $F(A, B)$ from G we can first insert the variable A into G and then B into the thereby generated term at position 1. The other possibility is to first insert the variable B into G and then A at position 1. With permutation we have the problem that two permutations after each other can produce the same argument structure as does one permutation alone. This can even lead to a combined permutation that is identical to the original argument structure of the twice permuted term.

To counter these cases, we only allow a specific order within a chain of basic substitutions of the same type. We propose to allow only fixations from left to right within a term. In the case where $F(A, B)$ is instantiated to $F(A, b)$, further fixation of either F or A is not possible, because they are left of b . However, when having fixated A it is still possible to fixate B in the next application of a basic substitution. Regarding the position of the to-be-inserted variable in the case of insertion where the left-to-right principle holds too, there is the slight modification that more than one variable can be inserted at the same position. Inserting A at position 0 into H in $F(G, H)$ leads to $F(G, H'(A))$. Continuation by inserting a variable B at position 0 or 1 into F or at position 0 into G is not allowed. In contrast, inserting into F at position 2 or into H' at position 0 or 1 is allowed. Finally, permutations also should be applied from left to right. Permutation of the same position twice in a term is not allowed. This is to prevent loops that are not part of basic substitution chains within preferred generalizations.

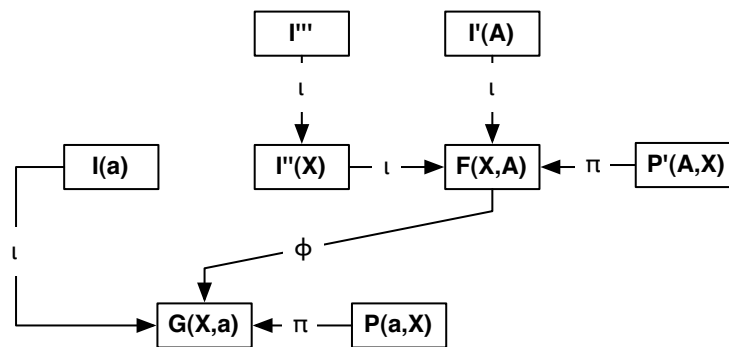


Figure 4.4: Improved instantiation ordering of anti-instances.

Using this new more restricted ordering of basic substitutions we gain a new instantiation ordering that is depicted in the graph in figure 4.4. Note that different nodes in the graph may represent terms that are strongly equivalent but differ in variables and

constants and their applicable substitutions. One can see that there is one and only one path from an anti-instance to $G(X, a)$. Furthermore, one can see that this structure is not an arbitrary graph but a weighted tree structure from the viewpoint of the algorithm. This in turn makes traversing the tree structure from figure 4.4, starting from $G(X, a)$ in the opposite direction, by our algorithm much more efficient than in the cyclic graph from figure 4.3, which does not use the new ordering restrictions. However, overall runtime complexity to search for anti-instances of a term is still similar to that of uniform-cost search, which is a variant of breath-first search. Only the branching factor was considerably reduced. The worst case amount of branches is at the nodes representing terms that can be permuted. Those terms have $n! - 1$ anti-instances for each subterm with n arguments that can be permuted. Additionally those terms have anti-instances by insertion and fixation. In figure 4.4 such a term is $F(X, A)$. The depth limit here for the reverse search in each of the instantiation ordering graphs of the two input terms is the complexity of the preferred generalization for the two input terms.

In analogy making we differentiate between one-to-one and many-to-many mappings of constants. One-to-one means that one constant is mapped to maximally one constant in the other domain, whereas many-to-many allows for multiple mappings of one constant to constants within the other domain. Our current algorithm however allows only for many-to-many mappings and cannot enforce one-to-one mappings. Let us have a look at the anti-instance $f(A, B, B, A)$ of $f(a, b, b, a)$ and the anti-instance $f(C, C, D, D)$ of $f(c, c, d, d)$. The term $f(V, X, Y, Z)$ is an anti-unifier from which these two terms can be derived only by renaming. Here, C and D are renamed into A , which is then fixated to a . However, C and D are fixated into c and d . Therefore, a stands in relation to c and d , which is not possible in a one-to-one mapping, and may not be desired [VO89]. We can easily modify the algorithm to create only one-to-one mappings by changing the condition in line 7 of the pseudo code. The condition that an anti-unifier for the compared terms exists through renaming has to be changed to the condition of strong equivalence between the compared terms. Because strong equivalence means a symmetrical relation between variables by renaming it only allows for one-to-one mappings. There exists no renaming from $f(A, B, B, A)$ to $f(C, C, D, D)$ because A cannot be renamed into C at position 0 and D at position 3 within $f(A, B, B, A)$. This comes from the convention that all variables on which a substitution acts on must be replaced simultaneously.

4.3 Implementation

The outlined restricted higher-order anti-unification was implemented as part of this bachelor thesis. A lightweight implementation of the mapping phase of HDTP was

added to test the complexity reduction in the reuse case when anti-unifying sets of terms. The program is supplied under the terms of the GNU General Public License [GPL] version 2 and can be download from the webpage [CODE] of the analogy research group, which is part of the Institute of Cognitive Science at the University of Osnabrück. How to use the program is described in the README file supplied with the source code. Notable features are:

- option to do one-to-one or many-to-many mappings
- configuration of allowed basic substitutions
- background-knowledge integration in the form of cost declaration of basic substitutions
- tracer to view the path from an anti-instance in the instantiation ordering graph
- complexity reduction when reusing basic substitutions
- loading of files with formalizations of analogies
- support to directly specify higher-order terms with uppercase letters as variable names
- sorts of terms are used to determine valid anti-unifiers

Two types of background-knowledge can be used. One can either specify the cost of a specific basic substitution to be *free* or specify that the function symbol of a term is not to be altered when insertion or permutation is applied. This suffices to encode that the term $distance(X, Y)$ can be changed into $distance(Y, X)$ without any cost. An example of the use for this is within the Rutherford analogy, for which a file with a formalization is supplied as part of the program. For two terms to be the same, the current implementation requires that the sort of each subterm needs to match its counterpart subterm.

The implementation was done in SWI-Prolog [SWI] a feature rich dialect of Prolog which is available as open source and free of costs. The choice of Prolog was done for a number of reasons:

- A parser for first order structures is already implemented. It's therefore easy to read strings of the form 'G(a,X)' and convert them into a term $G(a, X)$.
 - It incorporates the mechanism of unification of terms, which makes operations on terms like subterm extraction easy, and can be used as a precondition to check for term equivalence.
-

- Prolog implements backtracking and recursion very well, which is helpful for traversing graphs.
- Many implementations of theorem provers are available in Prolog and can therefore be integrated without much effort. These are especially needed for a consistency check of domain axioms within a to-be-added transfer phase for a full implementation of HDTP.
- Deep copies of terms are easy to make.

The program makes use of modules, where possible, to separate name spaces of predicates with different functionalities from each other. This makes rewriting parts of the implementation easier and generates abstractions in the form of interfaces between the modules. Attributed variables are often used within data structures of the program. These allow Prolog variables to have attributes much like instance variables of objects in object orientated languages. One attribute of a variable within a term for example, stores substitutions that are already applied to this variable. This information is then used to determine if a basic substitution can be applied cost free. Attributed variables ensure that information about a variable can be directly looked up as the variable is encountered. Data about terms are not kept in the program database and are always passed around in arguments which makes garbage collection and handling multiple terms at the same time easier. For more details about the implementation, for example which data-structures are used see the documentation and comments within the source code.

5 Conclusions and Future Work

We presented a restricted form of higher-order anti-unification that is suitable for anti-unification of sets of formulas. We showed that the definition of most specific anti-unifiers within this framework yields anti-unifiers that fulfill the guidelines to obtain a suitable higher-order anti-unification for use in analogy making. These guidelines were defined to include that anti-unifiers should preserve as much structure as possible. Furthermore, anti-unifiers should not be more complex and not contain symbols that are not part of the anti-unified terms. Using a metric called information load, it was shown that with restricted higher-order anti-unification only a finite number of anti-instances (up to renaming) of a term exist. A measure for the complexity of substitutions was presented, and with additional restrictions, a usable notion of preferred generalizations could be defined. The restrictions for complexity free reuse of substitutions were outlined. This enabled restricted higher-order anti-unification to anti-unify sets of formulas, which is useful in context of anti-unification of theories. An algorithm to find preferred generalizations for a pair of terms was developed. We discovered that by finding only a canonical representation of substitution chains, this algorithm can be optimized to yield results faster without compromising the optimality of the found solutions. Additionally, the algorithm was put to a practical test by a full implementation in Prolog. This implementation was made publicly available as an open source project to ease further external research of restricted higher-order anti-unification.

Whereas this thesis presents a thorough explanation of restricted higher-order anti-unification, it also uncovered not-yet-solved problems, leaving research to be done.

- The complexity measure for substitutions has yielded good and consistent generalizations in practice tests with the implemented algorithm. However, alternative definitions of complexity and their impact on preferred generalizations should be subject of further research.
- Complexity and reuse of substitutions is, at the moment, defined globally without taking into account the context of the terms on which the substitutions operate. This could be refined to gain a more detailed computation of complexity and reuse that takes into account what other symbols were used in the formula operated on and how often substitutions have been made to determine their complexity cost.

- It was shown that if a restricted ordering for substitution chains, having minimal complexity exists, it must be fixation, insertion, permutation and last renaming. However, it is not known yet whether for every preferred generalization another preferred generalization exists that incorporates the same anti-unifier and only uses such ordered basic substitution chains. While there is no evidence to the contrary, it should be formally proven that it is indeed always the case.
 - The complexity of the presented optimized algorithm can be further reduced by investigating better cutoff heuristics for the search in the instantiation ordering graph. The current depth limit to traverse the instantiation ordering graph is considered to be suboptimal and is a main contributor to unnecessary runtime complexity. Also, an indepth analysis about the runtime and space complexity of the algorithm should be made.
-

Bibliography

The websites referred to below were last accessed on October 27, 2008. In case of unavailability at a later time, we recommend visiting the Internet Archive.

- [CFH92] D. J. Chalmers, R. M. French, and D. R. Hofstadter. High-level perception, representation, and analogy: A critique of artificial intelligence methodology. *Journal of Experimental and Theoretical Artificial Intelligence*, 4(3):185–211, 1992.
- [FFG86] B. Falkenhainer, K. D. Forbus, and D. Gentner. The structure-mapping engine. In *5th National Conference on Artificial Intelligence*, pages 272–277, 1986.
- [FFG89] B. Falkenhainer, K. D. Forbus, and D. Gentner. The structure-mapping engine: Algorithm and examples. *Artificial Intelligence*, 41(1):1–63, 1989.
- [FGMF98] K. D. Forbus, D. Gentner, A. B. Markman, and R. W. Ferguson. Analogy just looks like high-level perception: why a domain-general approach to analogical mapping is right. *Journal of Experimental Theoretical Artificial Intelligence*, 10:231–257, 1998.
- [Gen83] D. Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2):155–170, 1983.
- [Gen89] D. Gentner. The mechanism of analogical learning. In S. Vosniadou and A. Ortony, editors, *Similarity and analogical reasoning*, pages 199–241. Cambridge University Press, New York, USA, 1989.
- [GKKS07] H. Gust, U. Krumnack, K.-U. Kühnberger, and A. Schwering. An approach to the semantics of analogical relations. In *2nd European Cognitive Science Conference*, pages 640–645, Delphi, Greece, 2007. Lawrence Erlbaum.
- [GKKS08a] H. Gust, U. Krumnack, K.-U. Kühnberger, and A. Schwering. Analogical reasoning: A core of cognition. *Zeitschrift für Künstliche Intelligenz (KI), Themenheft KI und Kognition*, 1:8–12, 2008.
- [GKKS08b] H. Gust, U. Krumnack, K.-U. Kühnberger, and A. Schwering. Re-representation in a logic-based model for analogy making. In W.R. Wobcke M. Zhang, editor, *AI 2008: Advances in Artificial Intelligence, 21st Australasian Artificial Intelligence Conference (AI-08)*, Lecture Notes in Artificial Intelligence, pages 42–48, Auckland, New Zealand, 2008. Springer.
- [GKS06] H. Gust, K.-U. Kühnberger, and U. Schmid. Metaphors and heuristic-driven theory projection (HDTP). *Theoretical Computer Science*, 354(1):98–117, 2006.
- [Has95] R. W. Hasker. *The replay of program derivations*. Phd thesis, University of Illinois at Urbana-Champaign, 1995.
- [Hei96] Birgit Heinz. *Anti-Unifikation modulo Gleichungstheorie und deren Anwendung zur Lemmagerierung*. R. Oldenbourg Verlag, Wien, 1996.
- [HM95] D. R. Hofstadter and J.C. Mitchell. The copycat project: A model of mental fluidity and analogy-making. In D. R. Hofstadter and Fluid Analogies Research Group, editors, *Fluid Concepts and Creative Analogies*, pages 205–267. Basic Books, 1995.

-
- [HM05] K. Holyoak and R. Morrison, editors. *The cambridge handbook on thinking and reasoning*. Cambridge University Press, 2005.
- [Ind92] B. Indurkha. *Metaphor and cognition*. Kluwer, Dodrecht, 1992.
- [KSGK07] U. Krumnack, A. Schwering, H. Gust, and K.-U. Kühnberger. Restricted higher-order anti-unification for analogy making. In *20th Australian Joint Conference on Artificial Intelligence (AI07)*, volume 4830 of *Lecture Notes of Artificial Intelligence*, Gold Coast, Australia, 2007. Springer.
- [Kur05] L. M. Kurzen. *Analogy and Bisimulation: A Comparison of Indurkha's Cognitive Models and Heuristic-Driven Theory Projection*. PICS (Publications of the Institute of Cognitive Science). University of Osnabrück, Osnabrück, 2005.
- [Pfe91] F. Pfenning. Unification and anti-unification in the calculus of constructions. In *6th Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, Netherlands, 1991.
- [Plo70] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.
- [Rey70] J.C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–153, 1970.
- [Rut] E. Rutherford. The scattering of α and β particles by matter and the structure of the atom. *Philosophical Magazine, Series 6* 21:669–688.
- [SA91] V. Sperschneider and G. Antoniou. *Logic A Foundation For Computer Science*. Addison-Wesley, 1991.
- [SKKG08] Angela Schwering, Ulf Krumnack, Kai-Uwe Kühnberger, and Helmar Gust. Syntactic principles of heuristic-driven theory projection. to appear, 2008.
- [VO89] S. Vosniadou and A. Ortony. *Similarity and analogical reasoning*. Cambridge University Press, Cambridge, 1989.
- [Wag02] U. Wagner. *Combinatorically restricted higher order anti-unification. An application to programming by analogy*. Diploma thesis, Technische Universität Berlin, 2002.
- [CODE] Source code of the restricted higher-order anti-unification implementation [online, cited 10-2008]. Available from: http://www.cogsci.uni-osnabrueck.de/~ai/analogy_project.html.
- [GPL] Gnu general public license, version 2 [online, cited 10-2008]. Available from: <http://www.gnu.org/licenses/gpl-2.0.html>.
- [SWI] Swi-prolog [online, cited 10-2008]. Available from: <http://www.swi-prolog.org>.
-

Ich versichere hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und eigenhändig angefertigt habe und keine anderen Quellen oder Hilfsmittel als die angegebenen verwendet habe.

Osnabrück, den 27.Oktober 2008

Martin Schmidt (Matrikelnummer: 909586)