The Application of Reformation to Repair Faulty Analogical Blends

Author: Chenghao Cai

UUN: s1518076

Supervisor: Prof. Alan Bundy

Second Supervisor: Dr. Ewen Maclean

Master of Science

Artificial Intelligence

School of Informatics

University of Edinburgh

2016

Abstract

This project aims at using reformation, which is a general-propose algorithm for theory repair, to repair faulty analogical blends produced by using Heuristic-Driven Theory Projection. An analogical blend may be faulty due to insufficient alignments between a source theory and a target theory, inconsistencies, or incompleteness. To solve these problems, we have implemented three algorithms *align*, *redirect* and *upgrade* which are based on the unblocking function of reformation: The *align* algorithm can adjust the alignments between the source theory and the target theory. The *redirect* algorithm can repair inconsistent rewrite rules. The *upgrade* algorithm can repair incomplete theories. Also, to extend the use of reformation to more fields, we have revised the implementation of the unblocking function, so that it can suggest more kinds of repairs. These approaches have been evaluated on many examples of analogical blending, such as the blend of natural number and list theories, the blend of trigonometric functions and the blend of gravity and electrostatic force.

Acknowledgements

I would like to express my appreciation to my supervisor, Prof. Alan Bundy, for his kind help and guidance in the last few months. I often received numerous annotations from his feedback for my drafts, and I think they always give me inspirations.

I would like to thank my second supervisor, Dr. Ewen Maclean, for his technological support and guidance about analogical blending, HETS, HDTP and Ontohub.

Also, thanks to Boris Mitrovic, Jovita Tang and Xue Li for many times of discussions about reformation.

Finally, thanks to my computer for his hundreds of working hours in summer.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Chenghao Cai*)

Table of Contents

Chapter 1

Introduction

1.1 The objective of this project

The objective of this project is to use reformation to repair faulty analogical blends. Given an analogical blend of two theories, reformation is used to adjust alignments between the two theories and process inconsistencies and incompleteness of the blend.

1.2 Motivation

Analogical reasoning has been widely used in human thinking. Recently, such a way of thinking has been formalised as a framework named Heuristic-Driven Theory Projection (HDTP) which enables machines to reason analogically (Gust et al., 2006; Schmidt et al., 2014; Schwering et al., 2009). Specifically, given a source theory and a target theory, HDTP is able to align the two theories and find analogous functions and axioms (Schwering et al., 2009). Since some analogous functions can be used to transfer axioms from the source theory to the target theory, it is possible to invent new axioms for the target theory (Kutz et al., 2014). Hence, an analogical blend of the two theories can be created. However, there exist some potential problems: Firstly, alignments between the two theories may be insufficient. This is because the alignments require that two aligned axioms have the same logical structure. Although some alignments are intuitively possible, they are not allowed by HDTP as their logical structures are not rigorously the same. Secondly, it is possible that the resulting blend is inconsistent. This is because the source and target theories themselves may be inconsistent, or the blending of them brings about inconsistencies. Thirdly, it is possible that the resulting blend is incomplete. This is because the source and target

theories themselves may be incomplete. The existence of these problems means that we need some methods to deal with faults with analogical blends. In Section 3.1 of Chapter 3, we will use more examples to explain why these are the cases.

Reformation is a new algorithm for theory repair and conceptual change (Bundy and Mitrovic, 2016). It is a general-purpose algorithm that it can be used to repair theories in different domains. Especially, it repairs a theory by changing the *language* of axioms, rather than simply adding and deleting axioms (Bundy, 2013). Hence, this project explores the possibilities of using reformation to change the *language* of faulty analogical blends and repair these blends. Specifically, in order to solve the problems of insufficient alignments, inconsistencies and incompleteness, the use of reformation includes two aspects: (1) The preprocessing use to adjust alignments between the source and target theories; (2) The postprocessing use to processing inconsistencies and incompleteness with the analogical blends.

1.3 Research contributions

This project produced the following research outputs:

- An *align* algorithm for adjusting alignments of analogical blends has been proposed and implemented. Details of this algorithm are described by Section 3.2.3 of Chapter 3.
- A *redirect* algorithm for repair inconsistent rewrite rules has been proposed and implemented. Details of this algorithm are described by Section 3.3.1.2 of Chapter 3.
- The *upgrade* algorithm, which is used to repair incomplete theories, has been implemented. This algorithm has been proposed by previous work (Bundy and Mitrovic, 2016). Details of this algorithm are described by Section 3.3.2 of Chapter 3.
- A revision of the unblocking function, which enables reformation to process more kinds of faulty theories, have been completed based on a previous implementation of reformation (Bundy and Mitrovic, 2016). Details about this are described by Section 3.2.2 of Chapter 3.
- The above algorithms and functions have been evaluated on many examples of analogical blends, such as the blend of natural numbers and lists, the blend

of the trigonometric sine and cosine functions, and the blend of gravity and electrostatic force. These examples are described by Chapter 4 and Chapter 5.

1.4 The document structure

This document is organised as the following chapters: In Chapter 2, we describe the background of this project, including reformation, analogical blending, programming languages and tools. In Chapter 3, we describe the actual work undertaken. Firstly, problems with analogical blends, including insufficient alignments, inconsistencies and incompleteness, are identified. Then we discuss the preprocessing use of reformation to adjust alignments. Next, we discuss the postprocessing use of reformation to repair inconsistent theories and incomplete theories. In the final section of this chapter, we provide details about the implementations of the algorithms. In Chapter 4, some worked examples are used to explain practically and analyse how the algorithms work. In Chapter 5, some other examples are used to evaluate the algorithms. In Chapter 6, we summarise our work and outline potential future work, as a conclusion of this project.

Chapter 2

Background

2.1 Reformation

Reformation is a general-propose algorithm for automated theory repair (Bundy, 2013; Bundy and Mitrovic, 2016). It is able to repair a theory by changing the *language* of axioms, rather than simply adding desired axioms and deleting undesired axioms. Conceptually, reformation is adapted from a non-standard unification algorithm, which is equivalent to the standard unification algorithm, but more suitable for the adaption of reformation. For details of the non-standard unification algorithm, readers could refer to the article above.

Reformation usually works with unification: It can unblock failed unifications if they are wanted, or block successful unifications if they are unwanted (Bundy and Mitrovic, 2016). In the article above, unblocking of reformation is realised by:

- **Make $F(\vec{s}^m) = G(\vec{t}^n)$.** If the wanted unification of two compound terms $F(\vec{s}^m)$ and $G(\vec{t}^n)$ fails, reformation will try to make them unifiable by making $F = G$ and $\vec{s}^m = \vec{t}^n$. This may require changes to both functors and arguments.

- **Make $x \notin \mathbf{V}(t)$.** If x is a variable of a term t which is different from x, they cannot be unified because they cannot pass the occurs-check. In this case, reformation will remove x from t.

On the other hand, blocking of reformation is realised by (Bundy and Mitrovic, 2016):

- **Make $F(\vec{s}^m) \neq F(\vec{t}^m)$.** If the unwanted unification of two compound terms $F(\vec{s}^m) \equiv F(\vec{t}^m)$ succeeds, reformation will try to make it fail by splitting the functor F or making $\vec{s}^m$ and $\vec{t}^m$ be two different terms which cannot be unified.

- **Make $x \in \mathcal{V}(t)$.** If the unification $x \equiv t$ is successful, where x is a variable and t is a term different from x, reformation will try to make x be a variable of t, so that they cannot pass the occurs-check.

The original unsorted first-order reformation has been implemented by Bundy and Mitrovic (2016). Also, Mitrovic (2013) has extended the implementation of reformation from unsorted first-order logics to multi-sorted first-order logics, and Tsialos (2014) has extended the implementation to description logics. Further, Mitrovic (2013) has implemented an algorithm named *revise*. It can repair inconsistent theories by blocking proofs leading to contradictions. The counterpart of the *revise* algorithm, which is named *upgrade*, has also been proposed by (Bundy and Mitrovic, 2016) recently, and it is used to repair incomplete theories by unblocking wanted proofs. However, not much work has been done on unblocking. It is possible that unblocking will have more significant applications.

2.2 Analogical blending

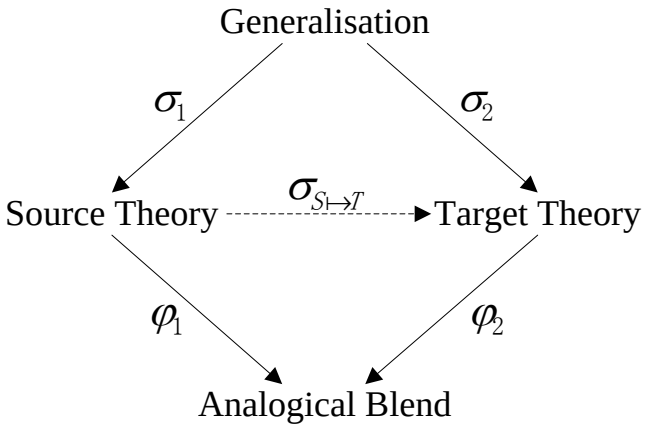

Figure 2.1: HDTP produces an analogical blend.

Analogical blending, or conceptual blending, has been formalised as a mathematically sound framework HDTP (Schmidt et al., 2014). Given a source theory and a target theory, HDTP is able to compare their logical structures and discover possible analogies by using first-order anti-unification (Plotkin, 1970) or restricted higher-order anti-unification (Krumnack et al., 2007; Schmidt et al., 2011). Also, the derived signature morphism method has been used by HDTP (Mossakowski

et al., 2015), where λ-calculus is used to map symbols instead of the conventional signature morphism method described by Codescu and Mossakowski (2008). Figure 2.1 reveals the basic concept of analogical blending (Bou et al., 2015): Given a source theory and a target theory, HDTP can compute a generalisation and two morphisms σ_1 and σ_2. The generalisation contains abstractions of aligned axioms in both theories, and σ_1 and σ_2 can map the generalisation to the source and target theories respectively. Comparing σ_1 and σ_2, it can further compute a morphism $\sigma_{S\mapsto T}$ which realises a cross-domain mapping from the source theory to the target theory. Then it can compute two morphisms φ_1 and φ_2 via the heterogeneous colimit method (Codescu and Mossakowski, 2008) combined with the Heterogeneous Tool Set (HETS) (Mossakowski et al., 2007), transferring the source and target theories to an analogical blend. Recently, analogical blending has been used to realise mathematical discovery (Guhe et al., 2011) and mathematical invention (Bou et al., 2015).

2.3 Programming languages and tools

Some programming languages and tools are relevant to this project: Firstly, HDTP (Schmidt et al., 2014) has been used to produce analogical blends. In particular, the Common Algebraic Specification Language (CASL) (Mossakowski et al., 2003) has been used to represent first-order theories in HDTP. Secondly, Nitpick (Blanchette, 2010), which is combined with Isabelle (Nipkow et al., 2002), has been used to detect inconsistent theories. Thirdly, SWI-Prolog (Wielemaker et al., 2010) has been used to develop essential programs for this project, because the previous implementation of reformation, which is the basis of this project, has also been developed on this platform (Bundy and Mitrovic, 2016). Finally, two ontologies on Ontohub (Kutz et al., 2014) has been used to develop our system. Details about the two ontologies are described by Section 4.2 in Chapter 4.

Chapter 3

Description of the work undertaken

3.1 Problems with analogical blends

In this section, we identify some potential problems with analogical blends. These problems include insufficient alignments, inconsistencies and incompleteness. They may be causes of faults with the analogical blends, and they are what this project focuses on.

Before starting the discussion, we explain briefly the style of representations: Constants are represented as numbers, arithmetic symbols or words starting with uppercase. Variables are represented as words starting with lowercase. For instance, 0, $Plus$ and $\times$ are constants, whereas x is a variable. An axiom is represented as a combination of constants and variables. For instance, $\forall x.Plus(0,x) = x$ is represented as $Plus(0,x) = x$. In particular, second-order functions are represented as lists. For instance, $Cos'(x)$, which is the derivative of $Cos(x)$, can be represented as $List(List(Der,Cos),x)$. For readability, $List$ is omitted, and it is further represented as $((Der\ Cos)\ x)$.

3.1.1 Insufficient alignments

In some cases, HDTP (Schmidt et al., 2014) produces an unexpected analogical blend because of insufficient alignments between two axioms. An example of natural numbers and lists can be such a case: Let $Succ : Nat \rightarrow Nat$ denote the successor of natural numbers and $Cons : El \times List \rightarrow List$ denote the constructor of lists, where Nat, $List$ and El are types of natural numbers, lists and elements respectively. Usually, 0 is the minimal unit of natural numbers, and a natural number n can be represented

as $Succ(Succ(\cdots Succ(0)))$, where *Succ* repeats n times. Also, *Nil* is the minimal unit of lists which represents an empty list, and a list $[H_1, H_2, \cdots, H_n]$, which has n elements, can be represented as $Cons(H_1, Cons(H_2, \cdots Cons(H_n, Nil)))$, where *Cons* repeats n times. Intuitively, 0 and Nil are analogous, because both of them are the minimal units, and *Succ* and *Cons* are analogous, because both of them are recursively defined. However, HDTP fails to align *Succ* and *Cons*, because *Succ* only has one argument, whereas *Cons* has two arguments. This results in the fact that some lemmas fail to be transferred from one theory to another. For instance, $Plus(Succ(0), Succ(0)) = Succ(Succ(0))$ cannot be transferred to from the theory of natural numbers to the theory of lists.

3.1.2 Inconsistencies

Although an analogical blend can be produced successfully, it is still possible that there exist inconsistencies in the blend. For the example of natural numbers and lists, let $Plus : Nat \times Nat \rightarrow Nat$ denote the addition operation of natural numbers and $App : List \times List \rightarrow List$ denote the appending operation of lists. Assume that the following three axioms of natural numbers are given[1]:

$$Plus(0, x) = x \tag{3.1}$$

$$Plus(Succ(x), y) = Succ(Plus(x, y)) \tag{3.2}$$

$$Plus(x, y) = Plus(y, x) \tag{3.3}$$

They can be used to simplify some expressions. For instance, $Plus(Succ(0), 0)$ can be simplified to $Succ(Plus(0, 0))$ and then to $Succ(0)$. Also, assume that the following axioms of lists are given:

$$App(Nil, l) = l \tag{3.4}$$

$$App(Cons(h, l), m) = Cons(h, App(l, m)) \tag{3.5}$$

HDTP (Schmidt et al., 2014) can align Axiom (3.1) and Axiom (3.4) by aligning 0 with *Nil* and *Plus* with App, which means that 0 is analogous to *Nil*, and *Plus* is analogous App. The alignment between *Plus* and App will transfer Axiom (3.3) to the domain of lists, yielding a new axiom:

$$App(l, m) = App(m, l) \tag{3.6}$$

[1] Axiom (3.1) and Axiom (3.2) are two Peano axioms.

However, there exist some counterexamples which refute this axiom. For instance, let $l = Cons(H_1, Nil)$ and $m = Cons(H_2, Nil)$. In this case, $App(l,m)$ can be simplified as $Cons(H_1, Cons(H_2, Nil))$, and $App(m,l)$ can be simplified as $Cons(H_2, Cons(H_1, Nil))$, but they are not equal. This contradiction means that the blend is inconsistent.

In particular, if an analogical blend consists of rewrite rules (Bundy, 1983), it is possible that some rules are inconsistent. In Section 3.3.1.2, we uses an example about trigonometric functions to explain it in detail. Briefly, that example is an analogical blend of the sine and cosine functions. It contains a rule $((Der\ Cos)\ x) \Rightarrow (Sin\ x)$ which means that the derivative of $Cos(x)$ is $Sin(x)$, and this rule is analogous to $((Der\ Sin)\ x) \Rightarrow (Cos\ x)$. However, this rule is inconsistent, because the derivative of $Cos(x)$ should be $-Sin(x)$.

3.1.3 Incompleteness

A theory may be incomplete, and this is also the case for an analogical blend. Assume that a theory T_1 has two axioms:

$$IsCapOf(Tokyo, Japan) \tag{3.7}$$

$$IsCapOf(x,y) \implies Flourishing(x) \tag{3.8}$$

where $IsCapOf(x,y)$ means that x is the capital of y, and $Flourishing(x)$ means that x is a flourishing city. Assume that another theory T_2 also has two axioms:

$$Capital(London, UK) \tag{3.9}$$

$$EconomicallyDeveloped(London) \tag{3.10}$$

which means that London is the capital of the UK and is economically developed. Also, a goal is expected to be proved:

$$Prosperous(London) \implies \tag{3.11}$$

which means that *London* is a prosperous city. T_2 is incomplete, because the goal cannot be proved. If $Capital$, *London*, UK and $EconomicallyDeveloped$ are aligned with $IsCapOf$, *Tokyo*, *Japan* and $Flourishing$ respectively, then an analogical blend can be created:

$$Capital(London, UK) \tag{3.12}$$

$$Capital(x,y) \implies EconomicallyDeveloped(x) \tag{3.13}$$

$$EconomicallyDeveloped(London) \tag{3.14}$$

where $Capital(x,y) \implies EconomicallyDeveloped(x)$ is analogous to $IsCapOf(x,y) \implies Flourishing(x)$. However, the goal $Prosperous(London) \implies$ still cannot be proved. Therefore, the blend is still incomplete.

3.2 The preprocessing use of reformation

In this section, we discuss the preprocessing use of reformation to enable further alignments when alignments computed via HDTP (Schmidt et al., 2014) are insufficient. In Section 3.2.1, we introduce the basic concept of using reformation to enable alignments. In Section 3.2.2, we look closely at unblocking in reformation. In Section 3.2.3, we introduce a greedy algorithm which is based on reformation and used to adjust alignments given by HDTP. In Section 3.2.4, we discuss how to make use of information about types to influence alignments.

3.2.1 The basic concept

The basic concept of using reformation to enable alignments is to unblock the unification between two terms which need to be aligned. Given two terms T_1 and T_2 which are failed to be aligned, reformation can suggest possible repairs which change T_1 or T_2 and make the unification $T_1 \equiv T_2$ succeed.

For instance, if T_1 is $Succ(x)$ and T_2 is $Cons(h,l)$, where $Succ(x)$ is the successor of natural numbers, $Cons(h,l)$ is the constructor of lists, x is a natural number, h is an element and l is a list, reformation will try to unblock the following unification:

$$Succ(x) \equiv Cons(h,l) \tag{3.15}$$

Reformation can suggest many possible ways of repair. A possible way is to add a new argument arg_{new} to the first position of *Succ* and merge *Succ* with *Cons*. Specifically,

this can be done via the following steps:

$$\begin{gathered} Succ(x) \equiv Cons(h,l) \\ \Downarrow \\ Succ(arg_{new},x) \equiv Cons(h,l) \\ \Downarrow \\ Cons(arg_{new},x) \equiv Cons(h,l) \\ \Downarrow \\ arg_{new} \equiv h \wedge x \equiv h \\ \Downarrow \\ True \end{gathered} \tag{3.16}$$

At the beginning, since the $Succ(x)$ only has an argument, but $Cons(h,l)$ has two arguments, reformation suggests adding the new argument arg_{new} to $Succ$. If it is added to the first position of $Succ$, $Succ(x)$ will be changed to $Succ(arg_{new},x)$. Then $Succ$ and $Cons$ are merged, and $Succ(arg_{new},x)$ is changed to $Cons(arg_{new},x)$. Hence, $Cons(arg_{new},x)$ can be unified with $Cons(h,l)$ by substituting arg_{new} for h and substituting x for l. Here, the merging of $Succ$ and $Cons$ can be considered as an alignment between the functors of $Succ(x)$ and $Cons(h,l)$.

3.2.2 A closer look at unblocking in reformation

As discussed by Section 3.2.1, unblocking is the core of enabling alignments. Usually, unblocking uses two rules, including CC_f and VC_f, to enable the unification between two terms when a conventional unification algorithm cannot unify them (Bundy and Mitrovic, 2016). What reformation does for the unification algorithm are twofold: Firstly, if an unification $F(\vec{s}^n) \equiv G(\vec{t}^m)$ fails, reformation will try to make $F(\vec{s}^n) = G(\vec{t}^m)$. Secondly, if an unification $x \equiv t$ fails due to the fact that x does not pass the occurs-check, reformation will try to change t and make x pass the occurs-check.

3.2.2.1 Make $F(\vec{s}^n) = G(\vec{t}^m)$

A main work package of this project is for refining or implementing functions which "make $F(\vec{s}^n) = G(\vec{t}^m)$". Generally, two terms can be represented as $F(\vec{s}^n)$ and $G(\vec{t}^m)$, where F and G are functors, $\vec{s}^n$ and $\vec{t}^m$ are sequences of arguments which have n and m arguments respectively, and an argument is either a term or a variable. In different cases of an unification pair $F(\vec{s}^n) \equiv G(\vec{t}^m)$, different operations can be used, including:

- **Merging**. When reformation unblocks $F(\vec{s}^n) \equiv G(\vec{t}^n)$, if $F \neq G$, but they have the same arities, then F can be merged with G. After merging, F can be replaced by G, and vice versa. If F is replaced by G, the unification pair will become $G(\vec{s}^n) \equiv G(\vec{t}^n)$. In particular, if $n = 0$, then F and G are terms without any arguments. In this case, F and G can still be merged.

- **Adding a constant as a new argument**. When reformation unblocks $F(\vec{s}^n) \equiv G(\vec{t}^m)$, if $n < m$, then $\vec{s}^n$ can be changed to $\vec{s}^{n+1}$ by adding a new argument to $\vec{s}^n$, and the unification pair can become $F(\vec{s}^{n+1}) \equiv G(\vec{t}^m)$. The new argument is a constant. On the other hand, if $n > m$, then $\vec{t}^m$ can be changed to $\vec{t}^{m+1}$ by adding a new argument to $\vec{t}^m$, and the unification pair can become $F(\vec{s}^n) \equiv G(\vec{t}^{m+1})$.

- **Adding a variable as a new argument**. This operation is analogous to the one above, but the new argument is a variable.

- **Removing an argument**. When reformation unblocks $F(\vec{s}^n) \equiv G(\vec{t}^m)$, if $n > m$, then $\vec{s}^n$ can be changed to $\vec{s}^{n-1}$ by removing an argument from $\vec{s}^n$, and the unification pair can become $F(\vec{s}^{n-1}) \equiv G(\vec{t}^m)$. On the other hand, if $n < m$, then $\vec{t}^m$ can be changed to $\vec{t}^{m-1}$ by removing an argument from $\vec{t}^m$, and the unification pair can become $F(\vec{s}^n) \equiv G(\vec{t}^{m-1})$.

- **Permuting arguments**. When reformation unblocks $F(\vec{s}^n) \equiv G(\vec{t}^n)$, where $n > 1$, the sequence $\vec{s}^n$ or $\vec{t}^n$ can be permuted. For $\vec{s}^n$, since there are n arguments in the sequence, it has $n! - 1$ permutations different from the original sequence. For $\vec{t}^n$, this is also the case.

- **Adding a functor**. When reformation unblocks $F(\vec{s}^n) \equiv G(\vec{t}^m)$, if the depth[2] of $F(\vec{s}^n)$ is smaller than that of $G(\vec{t}^m)$, then the functor G can be added to $F(\vec{s}^n)$. The unification pair then becomes $G(F(\vec{s}^n)) \equiv G(\vec{t}^m)$. Also, if the depth of $G(\vec{t}^m)$ is smaller than that of $F(\vec{s}^n)$, then the functor F can be added to $G(\vec{t}^m)$. The unification pair then becomes $F(\vec{s}^n) \equiv F(G(\vec{t}^m))$.

- **Removing a functor**. When reformation unblocks $F(\vec{s}^n) \equiv G(\vec{t}^m)$, if the depth of $F(\vec{s}^n)$ is larger than that of $G(\vec{t}^m)$, then the functor F can be removed from $F(\vec{s}^n)$. The unification pair then becomes $s_i \equiv G(\vec{t}^m)$, where s_i is the ith member of $\vec{s}^n$. Also, if the depth of $G(\vec{t}^m)$ is larger than that of $F(\vec{s}^n)$, then the functor

[2]The depth of a function/predicate is the depth of the tree when the function/predicate is written in the tree format.

G can be removed from $G(\vec{t}^m)$. The unification pair then becomes $F(\vec{s}^n) \equiv t_i$, where t_i is the ith member of $\vec{t}^m$.

- **Unifying arguments**. When reformation unblocks $F(\vec{s}^n) \equiv F(\vec{t}^n)$, where $n > 0$, it is required that the arguments are unified correspondingly.

In Chapter 4 and Chapter 5, we will use some examples to discuss how these functions are used in different cases.

3.2.2.2 Make $x \notin \mathcal{V}(t)$

The unification $x \equiv t$ fails when x is a variable of t. This is because it cannot pass the occurs-check. In this case, reformation will try to change t and make it pass the occurs-check. This can be realised by detecting the position of x in t and removing x. In order to remove x, the operation "**removing an argument**" is used.

3.2.2.3 The cost of unblocking

In many cases, we need to evaluate the cost of unblocking. Practically, the cost of unblocking is counted by the following rules:

- The total cost is equal to the sum of the cost for each atomic operation.
- The cost of substitution is 0.
- The cost of **merging** is 1.
- The cost of **adding a constant as a new argument** is 1.
- The cost of **adding a variable as a new argument** is 1.
- The cost of **removing an argument** is 1.
- The cost of **permuting arguments** is $(p-1)!$ if p arguments are located at positions different from their original positions. For instance, if $F(1,2,3,4)$ is changed to $F(3,1,2,4)$, then the cost is $(3-1)!$ because 1, 2 and 3 are not located at their original positions.
- The cost of **adding a functor** is 1.
- The cost of **removing a functor** is equal to the arity of the repaired function.
- The cost of **unifying arguments** is 0.

3.2.3 Adjusting insufficient alignments

We use an algorithm named *align* to adjust insufficient alignments.

In Section 3.2.1, we have discussed how to use reformation to align $Succ(x)$ and $Cons(h,l)$. Sometimes, aligning only a pair of functions is not enough, because it is possible that there exist many functions or axioms failed to be aligned, and there exist many possibilities of alignment. In this case, some alignments are preferable and should be chosen, whereas some others are not. For instance, aligning $Plus(Succ(x),y) = Succ(Plus(x,y))$ with $App(Cons(h,l),m) = Cons(h,App(l,m))$ is preferable to aligning $Plus(x,y) = Plus(y,x)$ with $App(Cons(h,l),m) = Cons(h,App(l,m))$, because the first way requires less change to the axioms than the second way.

To choose the preferable alignments, a greedy algorithm based on reformation is used in this project. It is named *align*. Assume that T_1 and T_2 are two parent theories, $\{A_1,A_2,\cdots,A_n\}$ and $\{B_1,B_2,\cdots,B_m\}$ are axioms in T_1 and T_2 respectively, C_{rep} is the cost of repairs, and C_{max} is an upper limit of the cost. The algorithm has the following steps:

- Step 1: HDTP (Schmidt et al., 2014) is used to compute a generalisation G and two morphisms σ_1 and σ_2 which map symbols from G to T_1 and T_2 respectively, so that successfully aligned axioms can be recognised. All successfully aligned axioms are labelled as pairs, and the remaining axioms are unlabelled. C_{rep} is initialised to 0.

- Step 2: Reformation takes two unlabelled axioms A_i and B_j, unblocks $A_i \equiv B_j$ and suggests a set of repairs $R_{i,j}$. After reformation tries all possible pairs of $A_i \equiv B_j$, the pair with the minimal cost of repairs is chosen. If there exist different pairs or sets of repairs which have the minimal cost, then backtracking is required. Assume that this pair is $A_p \equiv B_q$, its cost is C_{min}, and the set of repairs is R_{min}.

- Step 3: If $C_{rep}+C_{min} \leq C_{max}$, then C_{rep} is updated to $C_{rep}+C_{min}$, R_{min} is applied to all members of $\{A_1,A_2,\cdots,A_n\}$ and $\{B_1,B_2,\cdots,B_m\}$, and A_p and B_q are labelled as a pair. After that, if further alignments are possible, then go to Step 2. Otherwise, if $C_{rep}+C_{min} > C_{max}$, or no further alignment is possible, then all labelled pairs of axioms, all unlabelled axioms and all suggested repairs are output as results, and the algorithm terminates. The labelled pairs are considered

as the preferable alignments.

This algorithm, for instance, can work for the theories of natural numbers and lists described by Section 3.1.2: Assume that C_{max} is 2, or another value larger than 2. Firstly, HDTP can compute a generalisation G and two morphisms σ_1 and σ_2 for them. Here, $A(B,x) = x$, which is the abstraction of $Plus(0,x) = x$ and $App(Nil,l) = l$, is the only axiom in G, σ_1 is $\{A \mapsto Plus, B \mapsto 0\}$, and σ_2 is $\{A \mapsto App, B \mapsto Nil\}$. This means that $Plus(0,x) = x$ and $App(Nil,l) = l$ have been aligned by aligning 0 with Nil and $Plus$ with App, and they are labelled as a pair. The other axioms are unlabelled, and the initial value of C_{rep} is set to 0. Then reformation tries to align the remaining axioms. Here, the minimal cost C_{min} is 2, because when aligning $Plus(Succ(x),y) = Succ(Plus(x,y))$ with $App(Cons(h,l),m) = Cons(h,App(l,m))$, reformation only suggests adding a new argument arg_{new} to $Succ$ and merging $Succ$ with $Cons$. After that, since $C_{rep} + C_{min} \leq C_{max}$ is satisfied, these repairs are adopted, and the two axioms are labelled as a pair. Finally, since no further alignment can be found, the algorithm terminates, and the final results of the aligned axioms are:

$$\begin{cases} Plus(0,x) = x \\ App(Nil,l) = l \end{cases} \tag{3.17}$$

and

$$\begin{cases} Plus(Succ(arg_{new},x),y) = Succ(arg_{new},Plus(x,y)) \\ App(Cons(h,l),m) = Cons(h,App(l,m)) \end{cases} \tag{3.18}$$

3.2.4 The use of types

In some cases, the use of types can help improve the quality of alignment. Consider the following two axioms:

$$Plus(Succ(0),0) = Succ(0) \tag{3.19}$$

$$App(Cons(h,Nil),Nil) = Cons(h,Nil) \tag{3.20}$$

Unfortunately, reformation can suggest two ways of repair when aligning them. Specifically, it can suggest adding arg_{new} as the first argument of $Succ$, so that Axiom (3.19) becomes $Plus(Succ(arg_{new},0),0) = Succ(arg_{new},0)$. On the other hand, it can also suggest adding arg_{new} as the second argument of $Succ$, so that Axiom (3.19) becomes $Plus(Succ(0,arg_{new}),0) = Succ(0,arg_{new})$. Intuitively, the first way is expected by using our knowledge about natural numbers and lists. Reformation,

of course, does not have this intuition, but it can make use of some information about types to obtain the result expected. Recall that 0 has the type Nat, Nil has the type $List$, and the types of the functions are defined as $Succ : Nat \to Nat$, $Plus : Nat \times Nat \to Nat$, $Cons : El \times List \to List$ and $App : List \times List \to List$. They can be rewritten as axioms. Generally, let $T(x)$ denotes that x has a type T. The above information of types can be rewritten as axioms for natural numbers:

$$Nat(0) \tag{3.21}$$

$$Nat(Succ(Nat(v_1))) \tag{3.22}$$

$$Nat(Plus(Nat(v_2), Nat(v_3))) \tag{3.23}$$

and those for lists:

$$List(Nil) \tag{3.24}$$

$$List(Cons(El(v_4), List(v_5))) \tag{3.25}$$

$$List(App(List(v_6), List(v_7))) \tag{3.26}$$

where $v_1, v_2, \cdots, v_7$ are variables which are independent of each other. When reformation aligns Axiom (3.19) and Axiom (3.20), the above axioms about types can also be taken into account. Specifically, if reformation suggests adding arg_{new} as the first argument of $Succ$, Axiom (3.22) will be changed to $Nat(Succ(arg_{new}, Nat(v_1)))$, and it can be aligned with Axiom (3.25) by aligning Nat exactly with $List$. In this case, reformation only needs to merge Nat and $List$. On the other hand, if reformation suggests adding arg_{new} as the second argument of $Succ$, Axiom (3.22) will be changed to $Nat(Succ(Nat(v_1), arg_{new}))$. To align it with Axiom (3.25), Nat will be aligned with both $List$ and El. In this case, reformation not only needs to merge Nat with $List$, but also needs to merge Nat with El. Obviously, this way costs more than the first way, so the first way is preferable.

Generally, axioms about types can be created by the following methods:

- For a constant C which has a type T, create $T(C)$.
- For a function $F : T_1 \times T_2 \times \cdots \times T_n \to T$, create $T(F(T_1(v_1), T_2(v_2), \cdots, T_n(v_n)))$, where $v_1, v_2, \cdots, v_n$ are variables which are independent of each other.
- For a predicate $P : T_1 \times T_2 \times \cdots \times T_n$, create $Bool(P(T_1(v_1), T_2(v_2), \cdots, T_n(v_n)))$, where $v_1, v_2, \cdots, v_n$ are variables which are independent of each other. The type $Bool$ is used because the predicate is treated as a function to the boolean type.

3.3 The postprocessing use of reformation

In this section, we describe the postprocessing use of reformation. In Section 3.3.1, we describe the use of reformation for repairing inconsistent theories. In particular, Section 3.3.1.1 describes briefly the *revise* algorithm implemented by Mitrovic (2013), and Section 3.3.1.2 describes the *redirect* algorithm developed in this project. In Section 3.3.2, we describe the *upgrade* algorithm (Bundy and Mitrovic, 2016) for repairing incomplete theories.

3.3.1 Repairing inconsistent blends

3.3.1.1 Repairing inconsistent blends via the *revise* algorithm

It is possible that inconsistencies exist in an analogicla blend. Let B denote a blend. If both $B \vdash P$ and $B \vdash \neg P$ are true, then B is considered inconsistent, as P contradicts $\neg P$, and vice versa. For the example of natural numbers and lists described by Section 3.1.2, since we can prove both $App(Cons(H_1,Nil),Cons(H_2,Nil)) = App(Cons(H_2,Nil),Cons(H_1,Nil))$ and $App(Cons(H_1,Nil),Cons(H_2,Nil)) \neq App(Cons(H_2,Nil),Cons(H_1,Nil))$, the blend is inconsistent.

Reformation is able to repair such inconsistent blends via blocking. Since P and $\neg P$ are not allowed to be true at the same time, one of the proofs of P or $\neg P$ should be blocked. This algorithm has been implemented by Mitrovic (2013), and it is named *revise*. The implementation is based on reformation and resolution: Given a theory T, it infers everything which can be inferred via resolution, detects whether or not contradictions exist and blocks proofs resulting in the contradictions. Practically, in some cases, the original theory is not enough, and users need to provide more hints for the implementation. For the instance in Section 3.1.2, such hint can be found by using Nitpick (Blanchette, 2010) combined with Isabelle (Nipkow et al., 2002): It refutes $App(l,m) = App(m,l)$ because it finds that when $l = Cons(H_1,Nil)$ and $m = Cons(H_2,Nil)$, both $App(l,m) = App(m,l)$ and $App(l,m) \neq App(m,l)$ can be proved. Given this counterexample, the implementation of *revise* can suggest some possible repairs. For instance, it can suggest changing $App(l,m) = App(m,l)$ to $App(l,m) \simeq App(m,l)$ by renaming "$=$" to "$\simeq$", so that no further counterexample can be found. Here, "$\simeq$" has NOT been defined earlier, so it is a new symbol. Sometimes, the new symbol could have a realistic meaning. For instance, it can be

considered as a permutation operation for lists such that permutations of $[1,2,3,4]$ are $[1,2,3,4],[1,2,4,3],\cdots,[4,3,2,1]$, and $App([1],[2,3,4]) \simeq App([2,3,4],[1])$ holds because a permutation of $[1,2,3,4]$ can be $[2,3,4,1]$.

3.3.1.2 Repairing rewrite rules via the *redirect* algorithm

Rewrite rules (Bundy, 1983) are used to simplify expressions. Usually, a rewrite rule is applied to an expression via rewriting instead of implication. This is because rewriting uses matching, whereas implication uses two-way unification. Specifically, when matching a rewrite rule $l \Rightarrow r$ with a term t, rewriting allows t itself or a subterm of t to be unified with l. On the other hand, when applying a rule $p \Longrightarrow q$ to a condition u, implication requires that p can be fully unified with u, but not any subterms of u.

For a set of rewrite rules, local confluence is highly desirable, which means that an expression is expected to be rewritten to one and only one normal form (Bundy, 1983). In most cases, if a set of rewrite rules is not locally confluent, it will be considered incomplete and be repaired via the Knuth-Bendix completion algorithm (Knuth and Bendix, 1983). The premise of using the Knuth-Bendix completion algorithm is that the rewrite rules are consistent. If consistencies are not guaranteed, it will be possible that inconsistencies result in non-confluence. Consider the following set of rules about trigonometric functions (represented by the list format):

$$((Der\ Cos)\ x) \ \Rightarrow\ (Sin\ x) \tag{3.27}$$

$$((Der\ Cos)\ (Divi\ \pi\ 2)) \ \Rightarrow\ (Neg\ 1) \tag{3.28}$$

$$(Sin\ (Div\ \pi\ 2)) \ \Rightarrow\ 1 \tag{3.29}$$

Here, Rule (3.27) means that the derivative of $Cos(x)$, which is denoted by $Cos'(x)$, is $Sin(x)$. This rule has been created by making an analogy with another rule $((Der\ Sin)\ x) \ \Rightarrow\ (Cos\ x)$ which means that the derivative of $Sin(x)$ is $Cos(x)$. However, this rule is faulty, because $Cos'(x)$ should be $-Sin(x)$. Rule (3.28) means that $Cos'(\pi/2) = -1$. Rule (3.28) means that $Sin(\pi/2) = 1$. This set of rules is not locally confluent, because $((Der\ Cos)\ (Divi\ \pi\ 2))$ can be rewritten to $(Sin\ (Divi\ \pi\ 2))$ and then to 1, but it can also be directly rewritten to $(Neg\ 1)$. If the Knuth-Bendix completion algorithm is used to process this issue, then the following new rule will be added to this set of rules:

$$(Neg\ 1) \Rightarrow 1 \tag{3.30}$$

This rule means that $-1 = 1$, which is false under the usual interpretation. It is created on the premise that all given rules are consistent. Instead of using the completion algorithm, on the other hand, if we doubt the consistency of Rule (3.27) and change it to

$$((Der\ Cos)\ x)\ \Rightarrow\ (Neg\ (Sin\ x)) \tag{3.31}$$

which means that $Cos'(x) = -Sin(x)$, then this set of rules can also be locally confluent. This example reveals that inconsistent rules may also be the cause of non-confluence, and we need an algorithm to repair them.

In this project, an algorithm named *redirect* is designed to repair inconsistent rewrite rules[3]. The basic concept of this algorithm is to block an unwanted path of rewriting by unblocking a wanted path. Unblocking, conceptually, should be used to unblock wanted proofs, while blocking should be used to block unwanted proofs. However, unblocking has side effects which block some other proofs. The side effects mean: If proving a goal G_1 requires an axiom A, but unblocking the proof of a goal G_2 requires changing A to A', it will be possible that the proof of G_1 is blocked, because proving G_1 requires A, but A has been changed to A'. In Section 3.3.1.3, we will discuss the side effects of unblocking in detail. Here, we make use of the side effects of unblocking to design the *redirect* algorithm: We apply unblocking to the unification between an unwanted expression and a wanted expression. If an expression s can be written to two different normal forms t and t', then the unwanted expression may exist on the path from s to t', whereas the wanted expression may exist on all paths to t. For the example of trigonometric functions, s is $((Der\ Cos)\ (Divi\ \pi\ 2))$, t can be $(Neg\ 1)$, t' can be 1, the path from s to t' is $((Der\ Cos)\ (Divi\ \pi\ 2)) \Rightarrow (Sin\ (Divi\ \pi\ 2)) \Rightarrow 1$, a path to t is $(Neg\ ((Der\ Cos)\ (Divi\ \pi\ 2))) \Rightarrow (Neg\ (Sin\ (Divi\ \pi\ 2))) \Rightarrow (Neg\ 1)$, the unwanted expression can be $(Sin\ (Divi\ \pi\ 2))$, and the wanted expression can be $(Neg\ (Sin\ (Divi\ \pi\ 2)))$. In this case, reformation will try to unblock the unification between the unwanted expression and the wanted one. If unblocking is successful, the path to t' will be redirected to t. As discussed before, unblocking has the side effects which block some other proofs. Here, the proof of t' will be blocked, so that the redirected path only rewrites to t, but it does not rewrite to t'. For the example above, $(Sin\ (Divi\ \pi\ 2)) \equiv (Neg\ (Sin\ (Divi\ \pi\ 2)))$ will be unblocked. This means that $(Sin\ (Divi\ \pi\ 2))$ is changed to $(Neg\ (Sin\ (Divi\ \pi\ 2)))$, and the path will become $((Der\ Cos)\ (Divi\ \pi\ 2)) \Rightarrow (Neg\ (Sin\ (Divi\ \pi\ 2))) \Rightarrow (Neg\ 1)$.

[3]This algorithm is similar to a method whereby Jovita Tang uses traces to direct unblocking. Her method uses resolution, while this algorithm uses rewriting.

Specifically, the *redirect* algorithm has the following steps (assuming termination):

- **Step 1: Check whether or not a set of rewrite rules R is locally confluent.** This can be done by the conventional method which finds all critical pairs of R and checks whether or not they can be conflated. If R is not locally confluent, then go to Step 2. Otherwise, R is output as a result, and the algorithm terminates.

- **Step 2: Find a possibly faulty path and a possibly correct path.** This can be done by identifying a critical pair $< s_1, s_2 >$ which starts from a term s and ends in two different normal forms $< t_1, t_2 >$. Let P_1 denote the path from s to t_1 and P_2 denote the path from s to t_2. Since either P_1 or P_2 is possibly faulty, backtracking search is needed here: Firstly, assume that P_1 is the faulty path and P_2 is the correct path, and go to Step 3. Secondly, assume that P_2 is the faulty path and P_1 is the correct path, and go to Step 3.

- **Step 3: Find an unwanted expression.** Assume that the faulty path found by Step 2 is $s \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_n$. Let the unwanted expression u be u_i, where $i = 1, 2, \cdots, n$, and then go to Step 4. Here, backtracking search is used, because there are n different choices of u.

- **Step 4: Find a wanted expression.** Assume that the correct path found by Step 2 is $s \Rightarrow v_1 \Rightarrow \cdots \Rightarrow v_l$. The wanted expression can be not only any expressions on the path, but also any other expressions that can be rewritten to v_l. To find such expressions, backward rewriting is applied to v_l. Assume that these expressions are $w_1, w_2, \cdots, w_m$. Let the wanted expression w be w_j, where $j = 1, 2, \cdots, m$, and then go to Step 5. Here, backtracking search is used again, because there are m different choices of w.

- **Step 5: Redirect the faulty path.** This can be done by unblocking the unification between the unwanted expression u and the wanted expression w. Assume that u is obtained by applying a rule $l \Rightarrow r$. Firstly, reformation is used to unblock $u \equiv w$, and it suggests a set of repairs ψ. Then ψ is applied to r, and $l \Rightarrow r$ is changed to $l \Rightarrow r'$. After that, $l \Rightarrow r$ is removed from R, and $l \Rightarrow r'$ is added to R. Finally, we remove all paths from the search space, build new paths by using the repaired R and go to Step 1 to check if it is locally confluent. If it passes the check, then this means that all faulty paths have been redirected to correct paths.

Now we use the *redirect* algorithm to repair the faulty rule about trigonometric functions: Firstly, the path to $(Neg\ 1)$ is chosen as the correct path, and the path to 1 is chosen as the faulty path. Then candidates of wanted expressions can be worked out by backward rewriting. They are $(Neg\ 1)$ itself, $(Neg\ (Sin\ (Div\ \pi\ 2)))$ and $(Neg\ (Der\ Cos)\ (Div\ \pi\ 2))$. Also, candidates of unwanted expressions are $(Sin\ (Div\ \pi\ 2))$ and 1. Next, reformation tries to unblock the unification between an unwanted expression $(Sin\ (Div\ \pi\ 2))$ and a wanted expression $(Neg\ (Sin\ (Div\ \pi\ 2)))$. Reformation will suggest adding a *Neg* to the unwanted expression. Applying this repair to Rule (3.27), the rule is changed to $((Der\ Cos)\ x)\ \Rightarrow\ (Neg\ (Sin\ x))$. Finally, the algorithm checks if the repaired rules are locally confluent. Since $((Der\ Cos)\ (Divi\ \pi\ 2))$ can always be rewritten to $(Neg\ 1)$, they have satisfied the requirement of local confluence. The resulting rules are:

$$((Der\ Cos)\ x)\ \Rightarrow\ (Neg\ (Sin\ x)) \tag{3.32}$$

$$((Der\ Cos)\ (Divi\ \pi\ 2))\ \Rightarrow\ (Neg\ 1) \tag{3.33}$$

$$(Sin\ (Div\ \pi\ 2))\ \Rightarrow\ 1 \tag{3.34}$$

Rule (3.32) has become a correct rule, because it indicates $Cos'(x) = -Sin(x)$.

3.3.1.3 The side effects of unblocking

Unblocking, conceptually, should be used to process incompleteness, while blocking should be used to process inconsistencies. In other words, unblocking can make a theory complete by unblocking wanted proofs, and blocking can make the theory consistent by blocking unwanted proofs. However, the *redirect* algorithm uses unblocking, rather than blocking, to repair inconsistent rewrite rules. This is because unblocking has a side effect which blocks unwanted proofs. Specifically, assume that when unblocking unblocks a wanted proof, it changes an existing axiom. If an unwanted proof can be derived by using this axiom, it will be possible that it can no longer be derived after this axiom is changed. For the instance of trigonometric functions, the wanted proof is $((Der\ Cos)\ (Div\ \pi\ 2))\ \Rightarrow\ (Neg\ 1)$, while the unwanted proof is $((Der\ Cos)\ (Div\ \pi\ 2))\ \Rightarrow\ 1$. The unwanted proof can be derived by using the rule $((Der\ Cos)\ x)\ \Rightarrow\ (Sin\ x)$. When unblocking the wanted proof, this rule is changed to $((Der\ Cos)\ x)\ \Rightarrow\ Neg(\ (Sin\ x))$, so that the unwanted proof is blocked. This means that unblocking can have side effects which block other proofs.

The side effects are twofold: Firstly, an unwanted proof may be blocked. This is a positive effect and has been used by the *redirect* algorithm. Secondly, a wanted proof may be blocked. This is a negative effect. We will discuss it by Section 5.3 in Chapter 5.

3.3.2 Repairing incomplete blends

In this project, the *upgrade* algorithm is implemented and used to repair incomplete analogical blends. The *upgrade* algorithm has been proposed by Bundy and Mitrovic (2016)[4]. It is the counterpart of the *revise* algorithm, because *upgrade* is based on unblocking and used to enable a wanted proof, while *revise* is based on blocking and used to disable an unwanted proof.

The *upgrade* algorithm is combined with linear resolution with selection function (SL-resolution) (Kowalski and Kuehner, 1971): Generally, SL-resolution starts from a negated goal clause and derives the goal clause until it becomes an empty clause. If it cannot be derived to an empty clause, then the resolution fails. For instance, assume that the goal clause is $\neg G_1 \vee \cdots \neg G_i \vee \cdots \vee \neg G_n$ with a selected literal $\neg G_i$, which means that the goal is $G_1 \wedge \cdots G_i \wedge \cdots \wedge G_n$, and an axiom represented by a definite clause is $\neg P_1 \vee \cdots \vee \neg P_m \vee Q$, which means $P_1 \wedge \cdots \wedge P_m \implies Q$. SL-resolution derives the goal clause by unifying G_i and Q with the most general unifier θ and replacing $\neg G_i[\theta]$ with $\neg P_1[\theta] \vee \cdots \vee \neg P_m[\theta]$, so that goal clause becomes $\neg G_1[\theta] \vee \cdots \neg P_1[\theta] \vee \cdots \vee \neg P_m[\theta] \vee \cdots \vee \neg G_n[\theta]$. If the unification $G_i \equiv Q$ fails, the derivation will fail. In this case, reformation unblocks this unification and suggests a set of repairs ψ. ψ can be applied to Q and change the axiom to $\neg P_1 \vee \cdots \vee \neg P_m \vee Q'$, where Q' can be unified with G_i, so that the derivation can succeed.

Consider the incomplete theory described by Section 3.1.3. They can be represented by the following definite clauses:

$$Capital(London, UK) \tag{3.35}$$

$$\neg Capital(x, y) \vee EconomicallyDeveloped(x) \tag{3.36}$$

$$EconomicallyDeveloped(London) \tag{3.37}$$

and the negated goal clause is:

$$\neg Prosperous(London) \tag{3.38}$$

[4]Also, the need of the *upgrade* algorithm arose when Jovita Tang required an algorithm which uses reformation to repair faulty proofs of arithmetic.

The *upgrade* algorithm can make it complete. Firstly, starting from the goal clause $\neg Prosperous(London)$, SL-resolution will try to derive it to an empty clause. However, this attempt fails, because $Prosperous(London)$ cannot be unified with $Capital(London, UK)$, $EconomicallyDeveloped(x)$ or $EconomicallyDeveloped(London)$. Then reformation will try to unblock an unification. Assume that $Prosperous(London) \equiv EconomicallyDeveloped(x)$ is unblocked. Reformation can suggests merging $EconomicallyDeveloped$ and $Prosperous$, so that the unification pair becomes $Prosperous(London) \equiv Prosperous(x)$ and can succeed by substituting x with $London$. Next, the repair is applied to Clause (3.36), changing it to $\neg Capital(x,y) \vee Prosperous(x)$. Now the goal clause can be derived to $\neg Capital(London, y)$ via the changed axiom. Finally, $\neg Capital(London, y)$ can be derived to an empty clause by unifying $Capital(London, y)$ with $Capital(London, UK)$. Since the result is an empty clause, $Prosperous(London)$ has been proved. This means that the theory has been partially completed. In the resulting theory, $Capital(x,y) \implies EconomicallyDeveloped(x)$ has been changed to $Capital(x,y) \implies Prosperous(x)$, which means that if x is the capital of a country, then x is prosperous.

Also, there exists another way of repair: If reformation tries to unblock $Prosperous(London) \equiv EconomicallyDeveloped(London)$, it will also suggest merging $EconomicallyDeveloped$ and $Prosperous$. However, this repair will not be applied to Clause 3.36, but to Clause 3.37. The result is that $EconomicallyDeveloped(London)$ is changed to $Prosperous(London)$, and the goal can be directly proved via this axiom.

3.4 Implementations

The implementations of the methods described before are based on the unsorted reformation adapted from the original unsorted reformation program (Bundy and Mitrovic, 2016) and adapted by Mitrovic (2013)[5]. These implementations are compiled with $SWI-Prolog$ (Wielemaker et al., 2010). The following are descriptions about them:

A revision of the unblocking function. This revision enables reformation to suggest and apply more kinds of repairs when using the unblocking function. In the previous version of reformation implementation, unblocking is able to suggest

[5]Thanks to Boris Mitrovic for his help on discussing the algorithm and managing the code.

and apply the following repairs to functions: (a) Merge two functors; (b) Add new constants to the last positions of arguments; (c) Delete arguments in the last positions; (d) Make $x \notin \mathbf{V}(t)$. However, to deal with more complex situations, more kinds of repairs are needed. Section 3.2.2 has outlined all kinds of repairs implemented in this project conceptually. Practically, they are implemented as repairs with a "$left|right$" option indicating that a repair is applied to the left/right of an unification pair. They are listed as follows (in the Prolog style):

- $merge(F,G,left|right)$ — Merging a functor F with another functor G.
- $addargc(F,Ar,N,C,left|right)$ — Adding a new constant C as a new argument to the Nth position of a function F/Ar, where F is its functor and Ar is its arity. N must satisfies $0 \leq N \leq Ar$.
- $addargv(F,Ar,N,V,left|right)$ — Adding a new variable V as a new argument to the Nth position of a function F/Ar, where F is its functor and Ar is its arity. N must satisfies $0 \leq N \leq Ar$.
- $delarg(F,Ar,N,left|right)$ — Deleting the Nth argument from a function F/Ar, where F is its functor and Ar is its arity. N must satisfies $0 \leq N \leq Ar-1$.
- $addfunc(F,Ar,P,left|right)$ — Adding a functor P to a function F/Ar, where F is its functor and Ar is its arity. After adding the functor, the function becomes $P(F/Ar)$.
- $delfunc(F,Ar,I,left|right)$ — Deleting the functor of a function F/Ar, where F is its functor and Ar is its arity. After adding the functor, the function becomes X_I, where X_I is the Ith argument of the original F/Ar.
- $permute(F,P,left|right)$ — Permuting the arguments of a function F via a permutation P, where F is the functor, P is a list and the length of P must be equal to the arity of the function. If the arity is Ar, then there are $Ar!-1$ different possibilities of P. For instance, if the arity is 3, then P can be $[1,3,2]$, $[2,1,3]$, $[2,3,1]$, $[3,1,2]$ or $[3,2,1]$, but cannot be $[1,2,3]$.

These repairs has been combined with the *diagnose* function and the *repair* function of reformation (Bundy and Mitrovic, 2016; Mitrovic, 2013). They cover all repairs in the previous version of unblocking. In particular, the "make $x \notin \mathbf{V}(t)$" repair is realised via *delarg* and a special procedure which detects the position of x in t.

Filters. Filters are used to prohibit certain repairs. Usually, a filter is a list of repairs defined by users. For instance, it can be $[addfunc(\_,\_,\_,\_), permute(\_,\_,left)]$, which means that "$addfunc$" is not allowed and "$permute$" is not allowed to be applied to the left. The filters can be used in many cases. For instance, the *align* algorithm can use a filter to prohibit all *right* repairs, so that when unblocking $A_i \equiv B_j$, all suggested repairs are for A_i, but B_j remains unchanged. In fact, in most cases, we only need either $left$ or *right* repairs. In a few cases we need both, which will be discussed by Section 5.3 in Chapter 5.

An implementation of the *align* algorithm. The details of this algorithm have been described by Section 3.2.3. This implementation is based on the revision of unblocking. It is a Prolog function[6] $align(+S,+T,+N,-B,-R)$, where S is the source theory, T is the target theory, N is the maximum number of repairs allowed, B is the resulting analogical blend, and R is the set of suggested repairs.

An implementation of the *redirect* algorithm. The details of the *redirect* algorithm have been described by Section 3.3.1.2. This implementation is also based on the revision of unblocking. It is a function $redirect(+RuleIn,+DF,+DB,+N,-RuleOut,-R)$, where $RuleIn$ is the original set of rewrite rules, DF and DB are the search depths of forward rewriting and backward rewriting respectively (They are restricted because this algorithm cannot deal with non-terminating rewrite rules), N is the maximum number of repairs allowed, $RuleOut$ is the repaired set of rewrite rules, and R is the set of suggested repairs.

An implementation of the *upgrade* algorithm. The details of the *upgrade* algorithm have been described by Section 3.3.2. This implementation is also based on the revision of unblocking. It is a function $upgrade(+G,+TIn,+N,+D,-TOut,-R)$, where G is the goal clause, TIn is the original theory, N is the maximum number of repairs allowed, D is the maximum depth of the resolution tree allowed, $TOut$ is the repaired theory, and R is the set of suggested repairs.

[6]The terms with "+" are considered as inputs, whereas the terms with "−" are considered as outputs.

Chapter 4

Worked examples and analysis

In this chapter, some worked examples, which have been used to develop the algorithms, are used to analyse and explain how reformation repairs faulty analogical blends.

4.1 Merging two ontologies of food chains

Assume that there are two ontologies O_1 and O_2 which use different predicates to represent food chains: O_1 uses $Eat(x,y)$ to represent that x eats y, whereas O_2 uses $IsEatenBy(p,q,u)$ to represent that p is eaten by q, and the knowledge is created by an user u. Intuitively, if the user u is not taken into account, then $Eat(x,y)$ has the same meaning as $IsEatenBy(y,x,u)$, and vice versa. Now the ontologies are expected to be merged. This problem can be considered as a problem of producing an analogical blend: We need to produce an analogical blend for O_1 and O_2, in which $Eat(x,y)$ is analogous to $IsEatenBy(p,q,u)$, x is analogous to q, and y is analogous to p.

To make the problem clear, consider the following ontologies:

$$
\begin{array}{c}
O_1 \\
\hline
\underline{\textbf{< Type Declaration >}} \quad \underline{\textbf{< Axiom >}} \\
\begin{array}{rr}
Eagle : Life & Eat(Eagle, Mouse) \\
Mouse : Life & Eat(Eagle, Snake) \\
Dove : Life & Eat(Snake, Mouse) \\
Snake : Life & Eat(Cat, Mouse) \\
Cat : Life & Eat(Cat, Dove) \\
Eat : Life \times Life &
\end{array}
\end{array}
\tag{4.1}
$$

$$
\begin{array}{c}
O_2 \\
\hline
\begin{array}{c|c|c}
\textbf{< Type Declaration >} & & \textbf{< Axiom >} \\
\hline
Mouse : Animal & Cat : Animal & IsEatenBy(Mouse, Cat, Alice) \\
Dove : Animal & Owl : Animal & IsEatenBy(Dove, Cat, Bob) \\
Alice : User & Bob : User & IsEatenBy(Mouse, Owl, Bob) \\
\multicolumn{3}{c}{IsEatenBy : Animal \times Animal \times User}
\end{array}
\end{array}
\tag{4.2}
$$

For O_1, $Life$ is a type of all lives, and Eat is a predicate with two arguments which have the type of $Life$. For O_2, $Animal$ is a type of all animals, $User$ is a type of all users, and $IsEatenBy$ is a predicate with two arguments which have the type of $Animal$ and an argument which have the type of $User$. Both ontologies have three individuals which are the same, and they are Cat, $Mouse$ and $Dove$. Regardless of the user information, $Eat(Cat, Mouse)$ and $Eat(Cat, Dove)$ indicate the same knowledge as $IsEatenBy(Mouse, Cat, Alice)$ and $IsEatenBy(Dove, Cat, Bob)$, and they are expected to be aligned when merging O_1 and O_2. The other axioms in O_1, however, cannot find their counterparts in O_2, so they are not expected to be aligned with the others.

Unfortunately, HDTP (Schmidt et al., 2014) cannot produce a generalisation for the two ontologies, because it fails to align $Eat(x, y)$ and $IsEatenBy(p, q, u)$. To align them, their arities should be the same, and their arguments should also be reasonably aligned. For instance, when aligning $Eat(Cat, Mouse)$ and $IsEatenBy(Mouse, Cat, Alice)$, Eat is expected to have an extra argument which can be aligned with $Alice$, and Cat and $Mouse$ are expected to be aligned exactly with Cat and $Mouse$ respectively. In this case, we use the *align* algorithm to adjust the alignments. It will unblocks the following unification:

$$
Eat(Cat, Mouse) \equiv IsEatenBy(Mouse, Cat, Alice) \tag{4.3}
$$

There exist two directions of unblocking: (a) Unblocking from left to right; (b) Unblocking from right to left. The first direction means that repairs will only be applied to O_1, whereas the second direction means that repairs will only be applied to O_2. The different directions of unblocking can be realised by using different filters.

If the axioms are expected to be transferred from O_1 to O_2, then unblocking should be from left to right: Firstly, a new argument is added to the last position of $Eat(Cat, Mouse)$, changing it to $Eat(Cat, Mouse, arg_{new})$. Then the arguments of $Eat(Cat, Mouse, arg_{new})$ are permuted, and this changes it

to $Eat(Mouse, Cat, arg_{new})$. Finally, Eat and $IsEatenBy$ are merged, and $Eat(Mouse, Cat, arg_{new})$ is changed to $IsEatenBy(Mouse, Cat, arg_{new})$, so that it can be unified with $IsEatenBy(Mouse, Cat, Alice)$. The above repairs can be applied to all axioms in O_1, and this will transfer them from O_1 to O_2. Thus, the blend $O_{1\mapsto 2}$ becomes:

$O_{1\mapsto 2}$

$<$ **Type Declaration** $>$	$<$ **Axiom** $>$
$Eagle : Animal \mid Mouse : Animal$	$IsEatenBy(Mouse, Eagle, arg_{new})$
$Dove : Animal \mid Snake : Animal$	$IsEatenBy(Snake, Eagle, arg_{new})$
$Cat : Animal \mid Owl : Animal$	$IsEatenBy(Mouse, Snake, arg_{new})$
$Alice : User \mid Bob : User$	$IsEatenBy(Mouse, Cat, Alice)$
$IsEatenBy : Animal \times Animal \times User$	$IsEatenBy(Dove, Cat, Bob)$
	$IsEatenBy(Mouse, Owl, Bob)$

(4.4)

It is noticeable that the axioms transferred from O_1 may contain a variable arg_{new}, and the variable remains unknown. This result is expected because O_1 does not indicate any information about users. In particular, since $IsEatenBy(Mouse, Cat, arg_{new})$ can be unified with $IsEatenBy(Mouse, Cat, Alice)$ which is already in O_2, the blend does not include $IsEatenBy(Mouse, Cat, arg_{new})$. For the same reason, the blend does not include $IsEatenBy(Dove, Cat, arg_{new})$. Here, arg_{new} does not need to be instantiated in the same way. In other words, different instances can be instantiated to *Alice*, *Bob*, or someone else.

Reformation, alternatively, can also suggest adding a new constant C_{new} as the new argument, instead of adding a new variable. In this case, to unblock the unification, reformation needs to further merge C_{new} with *Alice*. This results in the fact that all axioms transferred from O_1 have the form of $IsEatenBy(p, q, Alice)$, which means their users are all forced to be *Alice*. Then it will further merge *Alice* and *Bob*, as $IsEatenBy(Dove, Cat, Alice)$ and $IsEatenBy(Dove, Cat, Bob)$ are expected to be unified. This is an unexpected result. Fortunately, this result can be refuted, if we always require the cost to be minimal. In other words, when merging C_{new} with *Alice* and merging *Alice* with *Bob*, the cost is 2. On the other hand, when substituting arg_{new} for *Alice* or *Bob*, the cost is 0. Therefore, adding arg_{new} is better than adding C_{new}.

On the other hand, if the axioms are expected to be transferred from O_2 to O_1, then unblocking should be from right to left: Firstly, the third argument of $IsEatenBy(Mouse, Cat, Alice)$ is removed, and it is changed to

$IsEatenBy(Mouse, Cat)$. Then the arguments of $IsEatenBy(Mouse, Cat)$ are permuted, and this changes it to $IsEatenBy(Cat, Mouse)$. Finally, $IsEatenBy$ and Eat are merged, and $IsEatenBy(Cat, Mouse)$ is changed to $Eat(Cat, Mouse)$. The above repairs can be applied to all axioms in O_2, and this will transfer them from O_2 to O_1. Thus, the blend $O_{2\mapsto 1}$ becomes:

$$\begin{array}{c} O_{2\mapsto 1} \\ \hline \begin{array}{|c|c|} \hline \textbf{< Type Declaration >} & \textbf{< Axiom >} \\ \hline Eagle : Life \mid Mouse : Life & Eat(Eagle, Mouse) \\ Dove : Life \mid Snake : Life & Eat(Eagle, Snake) \\ Cat : Life \mid Owl : Life & Eat(Snake, Mouse) \\ Eat : Life \times Life & Eat(Cat, Mouse) \\ & Eat(Cat, Dove) \\ & Eat(Owl, Mouse) \\ \hline \end{array} \end{array} \tag{4.5}$$

It is noticeable that the axioms from O_2 have lost their user information. This is because reformation has suggest deleting the third argument of $IsEatenBy(p, q, u)$, in order to transfer them from O_2 to O_1.

4.2 Natural numbers and lists

This example is about natural numbers and lists[1]. Assume that two parent theories T_{nat} and T_{list} are:

T_{nat}

$<$ Type Declaration $>$	$<$ Axiom $>$
$0 : Nat$	$Sum(0) = 0$
$Succ : Nat \to Nat$	$Sum(Succ(x)) = Plus(Succ(x), Sum(x))$
$Sum : Nat \times Nat \to Nat$	$Qsum(Succ(x), y) = Qsum(x, Plus(Succ(x), y))$
$Qsum : Nat \times Nat \to Nat$	$Qsum(0, x) = x$
$Plus : Nat \times Nat \to Nat$	$Plus(0, x) = x$
	$Plus(Succ(x), y) = Succ(Plus(x, y))$
	$Sum(x) = Qsum(x, 0)$
	$Plus(Sum(x), y) = Qsum(x, y)$

(4.6)

T_{list}

$<$ Type Declaration $>$	$<$ Axiom $>$
$Nil : List$	$App(Nil, l) = l$
$Cons : El \times List \to List$	$App(Cons(h, l), m) = Cons(h, App(l, m))$
$App : List \times List \to List$	$Rev(Nil) = Nil$
$Rev : List \to List$	$Rev(Cons(h, l)) = App(Rev(l), Cons(h, Nil))$
$Qrev : List \times List \to List$	$Qrev(Nil, l) = l$
	$Qrev(Cons(h, l), m) = Qrev(l, Cons(h, m))$
	$Rev(l) = Qrev(l, Nil)$

(4.7)

[1] These examples can be downloaded from Ontohub (Kutz et al., 2014). Their URLs are `https://ontohub.org/lemma-examples/smaill_nat.dol` and `https://ontohub.org/lemma-examples/smaill_list.dol`.

HDTP (Schmidt et al., 2014) can compute a possible generalisation $G_{nat,list}$ and two morphisms σ_{nat} and σ_{list} for them. For instance, they can be:

$$\begin{array}{c} G_{nat,list} \\ \hline \begin{array}{ll} \underline{\boldsymbol{< Type\ Declaration >}} & \underline{\boldsymbol{< Axiom >}} \\ A : NatList & D(A) = A \\ B : NatList \times NatList \to NatList & C(A,x) = x \\ C : NatList \times NatList \to NatList & B(A,x) = x \\ D : NatList \to NatList & D(x) = C(x,A) \end{array} \end{array} \tag{4.8}$$

$$\sigma_{nat} = \{A \mapsto 0, B \mapsto Plus, C \mapsto Qsum, D \mapsto Sum\} \tag{4.9}$$

$$\sigma_{list} = \{A \mapsto Nil, B \mapsto App, C \mapsto Qrev, D \mapsto Rev\} \tag{4.10}$$

Here, the generalisation indicates some alignments of types, functions and axioms between two theories. Specifically, *NatList* indicates that the type *Nat* has been aligned with the type *List*, A, B, C and D are abstractions of constants in the two parent theories, and the axioms are abstractions of axioms in the two parent theories. The morphisms indicate alignments of constants between the generalisation and the parent theories. By using the generalisation and the morphisms, the axioms in T_{nat} can be transferred to those in T_{list} by applying the following morphisms:

$$\sigma_{nat,list} = \{0 \mapsto Nil, Plus \mapsto App, Qsum \mapsto Qrev, Sum \mapsto Rev\} \tag{4.11}$$

However, the axioms, which use the *Succ* function, cannot been completely transferred. This is because *Succ* is not aligned with any function of T_{list}. Here, using the *align* algorithm, reformation can try to enable the alignment between them. The following is the alignment which has "the minimal cost of repairs": Firstly, $Plus(Succ(x),y) = Succ(Plus(x,y))$ is transferred to $App(Succ(x),y) = Succ(App(x,y))$ by applying the morphism $\sigma_{nat,list}$. Then reformation is used to unblock the following unification:

$$\begin{array}{c} Equal(App(Succ(x),y), Succ(App(x,y))) \\ \equiv \\ Equal(App(Cons(h,l),m), Cons(h,App(l,m))) \end{array} \tag{4.12}$$

where "$Equal(a,b)$" denotes "$a = b$". Reformation will suggest adding a new argument to $Succ(x)$ and merging *Succ* with *Cons*, so that it is changed to $Cons(arg_{new},x)$, and the unification can succeed. Applying these repairs to the

remaining axioms and transferring them to T_{list}, we can obtain an analogical blend:

$$
\begin{array}{c}
B_{nat\mapsto list} \\
\hline
\boldsymbol{< Type\ Declaration >} \\
\hline
Nil : List \quad | \quad App : List \times List \to List \quad | \quad Rev : List \to List \\
Qrev : List \times List \to List \quad | \quad Cons : El \times List \to List \\
\hline
\boldsymbol{< Original\ Axiom >} \\
\hline
App(Nil,l) = l \quad | \quad Rev(Nil) = Nil \\
Qrev(Nil,l) = l \quad | \quad Rev(l) = Qrev(l,Nil) \\
App(Cons(h,l),m) = Cons(h,App(l,m)) \\
Rev(Cons(h,l)) = App(Rev(l),Cons(h,Nil)) \\
Qrev(Cons(h,l),m) = Qrev(l,Cons(h,m)) \\
\hline
\boldsymbol{< New\ Axiom >} \\
\hline
Rev(Cons(arg_{new},x)) = App(Cons(arg_{new},x),Rev(x)) \\
Qrev(Cons(arg_{new},x),y) = Qrev(x,App(Cons(arg_{new},x),y)) \\
App(Rev(x),y) = Qrev(x,y)
\end{array}
\tag{4.13}
$$

The next step is to process inconsistencies in this blend. For this example, Nitpick (Blanchette, 2010), which is combined with Isabelle (Nipkow et al., 2002), is used to find counterexamples of axioms. It can find that if $arg_{new} = H_1$ and $x = Cons(H_1, Nil)$, then $Rev(Cons(arg_{new},x)) = App(Cons(arg_{new},x),Rev(x))$ is false. Based on this counterexample, the *revise* algorithm can suggest different ways of repair. For instance, it can suggest renaming App to App_2, so that the axiom is changed to $Rev(Cons(arg_{new},x)) = App_2(Cons(arg_{new},x),Rev(x))$. Also, Nitpick can find that if $arg_{new} = H_1$, $x = Cons(H_1,Nil)$ and $y = Nil$, then $Qrev(Cons(arg_{new},x),y) = Qrev(x,App(Cons(arg_{new},x),y))$ is false. Again, the *revise* algorithm can suggest different ways of repair, such as adding a new argument C_{new} to $Qrev$, so that the axiom is changed to $Qrev(Cons(arg_{new},x),y,C_{new}) = Qrev(x,App(Cons(arg_{new},x),y),C_{new})$.

For $App(Rev(x),y) = Qrev(x,y)$, however, since no counterexample can be found, it will not be changed. In fact, it is true and can be proved via mathematical induction.

4.3 Trigonometric functions

HDTP (Schmidt et al., 2014) can produce an analogical blend for two sets of rewrite rules. Assume that S_1 and S_2 are sets of rewrite rules. If S_1 and S_2 are analogous, HDTP will compute a generalisation G of rules and two morphisms σ_1 and σ_2 which

indicate possible alignments of symbols. Based on the morphisms, some rules can be transferred from S_1 to S_2, so that a blend of rules can be produced. However, this blend may be faulty, because local confluence is not guaranteed.

The following is an example where HDTP produces a faulty analogical blend of rewrite rules. In this example, we use lists to represent second-order functions, as HDTP and reformation only support first-order logic now. For instance, $Sin'(x)$, which is the derivative of $Sin(x)$, can be represented as $((Der\ Sin)\ x)$. Assume that S_1 is a set of rewrite rules about trigonometric functions

$$(Plus\ (Squ\ (Sin\ x))\ (Squ\ (Cos\ x)))\ \Rightarrow\ 1 \tag{4.14}$$

$$((Der\ Sin)\ x)\ \Rightarrow\ (Cos\ x) \tag{4.15}$$

and S_2 is another set of rewrite rules

$$(Plus\ (Squ\ (Cos\ x))\ (Squ\ (Sin\ x)))\ \Rightarrow\ 1 \tag{4.16}$$

$$((Der\ Cos)\ (Div\ \pi\ 2))\ \Rightarrow\ (Neg\ 1) \tag{4.17}$$

$$(Sin\ (Div\ \pi\ 2))\ \Rightarrow\ 1 \tag{4.18}$$

$$(Cos\ (Div\ \pi\ 2))\ \Rightarrow\ 0 \tag{4.19}$$

$$(Plus\ (Squ\ 0)\ (Squ\ 1))\ \Rightarrow\ 1 \tag{4.20}$$

where π is a constant, x is a variable, *Plus* means addition, *Div* means division, *Sin* means the sine function, *Cos* means the cosine function, *Squ* means taking a square, *Neg* means taking a negative value, and *Der* means taking the derivative of a function. Translated to conventional representations in mathematics, they are $Sin^2(x) + Cos^2(x) = 1$, $Sin'(x) = Cos(x)$, $Cos^2(x) + Sin^2(x) = 1$, $Cos'(\pi/2) = -1$, $Sin(\pi/2) = 1$, $Cos(\pi/2) = 0$ and $0^2 + 1^2 = 1$ respectively. These rules can be used to simplify terms about trigonometric functions. Given S_1 and S_2, HDTP will output a generalisation by aligning Rule (4.14) and Rule (4.16):

$$G = \{(A\ (B\ (C\ x))\ (B\ (D\ x)))\ \Rightarrow\ 1\} \tag{4.21}$$

and two morphisms

$$\sigma_1 = \{A \to Plus, B \to Squ, C \to Sin, D \to Cos, E \to 1\} \tag{4.22}$$

and

$$\sigma_2 = \{A \to Plus, B \to Squ, C \to Cos, D \to Sin, E \to 1\} \tag{4.23}$$

It is noticeable that $C \rightarrow Sin$ in σ_1 corresponds to $C \rightarrow Cos$ in σ_2, and $D \rightarrow Cos$ in σ_1 corresponds to $D \rightarrow Sin$ in σ_2. Therefore, the rules in S_1 can be transferred into S_2 by applying the following morphism:

$$\sigma' = \{Sin \rightarrow Cos, Cos \rightarrow Sin\} \tag{4.24}$$

This yields an analogical blend B which contains the following rules:

$$(Plus\ (Squ\ (Cos\ x))\ (Squ\ (Sin\ x))) \Rightarrow 1 \tag{4.25}$$

$$^{*}\quad ((Der\ Cos)\ x) \Rightarrow (Sin\ x) \tag{4.26}$$

$$^{*}\quad ((Der\ Cos)\ (Div\ \pi\ 2)) \Rightarrow (Neg\ 1) \tag{4.27}$$

$$^{*}\quad (Sin\ (Div\ \pi\ 2)) \Rightarrow 1 \tag{4.28}$$

$$(Cos\ (Div\ \pi\ 2)) \Rightarrow 0 \tag{4.29}$$

$$(Plus\ (Squ\ 0)\ (Squ\ 1)) \Rightarrow 1 \tag{4.30}$$

In this blend, $((Der\ Cos)\ x) \Rightarrow (Sin\ x)$ is a new rule transferred from $((Der\ Sin)\ x) \Rightarrow (Cos\ x)$ in S_1, while the other rules are all from S_2. This blend is not locally confluent, and the rules violating local confluence have been marked by "$*$". Specifically, $((Der\ Cos)\ (Div\ \pi\ 2))$ can be rewritten in two ways

$$((Der\ Cos)\ (Div\ \pi\ 2)) \overset{(4.27)}{\Rightarrow} (Neg\ 1) \tag{4.31}$$

and

$$((Der\ Cos)\ (Div\ \pi\ 2)) \overset{(4.26)}{\Rightarrow} (Sin\ (Div\ \pi\ 2)) \overset{(4.28)}{\Rightarrow} 1 \tag{4.32}$$

where the numbers on "$\Rightarrow$" indicate the rules which are used. It is noticeable that the use of the new rule (4.26) yields 1 which is not equivalent to $(Neg\ 1)$. This means that the blend is not locally confluent. Also, it is noticeable that the parent theories are locally confluent. This means that local confluence is not guaranteed for an analogical blend of rewrite rules, even though two parent theories are locally confluent.

The above analogical blend is faulty because Rule (4.26) indicates that $Cos'(x) = Sin(x)$, but actually $Cos'(x)$ should be $-Sin(x)$. If Rule (4.26) is changed to $((Der\ Cos)\ x) \Rightarrow (Neg\ (Sin\ x))$ which indicates that $Cos(x) = -Sin(x)$, $((Der\ Cos)\ (Div\ \pi\ 2))$ will always be rewritten to $(Neg\ 1)$, and the blend will be locally confluent. This change can be done via the *redirect* algorithm, as discussed in Section 3.3.1.2.

Chapter 5

Evaluation

In this chapter, some new examples are used to evaluate the algorithms developed by this project.

5.1 Pascal and Python

Consider the following scene: A programmer was asked to translate a program from Pascal to Python for some reasons. However, the program was so large that he/she was considering using a program to help himself/herself to some extent. He/She found that HDTP (Schmidt et al., 2014) might help.

Pascal Code

Line	*Program*	*Predicate Form*
1 :	$read(a);$	$Read(A)$
2 :	$read(b);$	$Read(B)$
3 :	$writeln(a+b);$	$Writeln(+(A,B))$
4 :	$s := 0;$	$:=(S,0)$
5 :	$for\ i := a\ to\ b\ do\ s := s+i;$	$ForToDo(I,To(A,B),Do(:=(S,+(S,I))))$
6 :	$writeln(s);$	$Writeln(S)$
7 :	$writeln(a*b);$	$Writeln(*(A,B))$
8 :	$t := 1;$	$:=(T,1)$
9 :	$for\ i := a\ to\ b\ do\ t := t*i;$	$ForToDo(I,To(a,b),Do(:=(T,+(T,I))))$
10 :	$writeln(s);$	$Writeln(T)$

(5.1)

Program (5.1) is a Pascal program, which can read two integers a and b and output the values of $a+b$, $a+(a+1)+\cdots+(b-1)+b$, $a\times b$ and $a\times(a+1)\times\cdots\times(b-1)\times b$.

Python Code (5.2)

Line	*Program*	*Predicate Form*
1 :	$a \ = \ input()$	$=(A,Input)$
2 :	$b \ = \ input()$	$=(B,Input)$
3 :	$print(a+b)$	$Print(+(A,B))$
4 :	$s \ = \ 0$	$=(S,0)$
5 :	$for\ i\ in\ xrange(a,b+1):$	$ForIn(I,Xrange(A,+(B,1)),$
	$s \ = \ s+i$	$Execute(=(S,+(S,I))))$
6 :	$print(s)$	$Print(S)$

Program (5.2) is a Python program, which can read two numbers a and b and output the values of $a+b$ and $a+(a+1)+\cdots+(b-1)+b$:

HDTP is able to produce an analogical blend for them by aligning $Writeln(+(A,B))$ (Pascal Line 3) with $Print(+(A,B))$ (Python Line 3) and aligning $:=(S,0)$ (Pascal Line 4) with $=(S,0)$ (Python Line 4). This means that any *writeln* sentences in Pascal can be transferred to *print* sentences in Python, and any $:=$ sentences can be transferred to $=$ sentences. It is also expected that $Read(A)$ (Pascal Line 1) can be aligned with $=(A,Input)$ (Python Line 1). In this case, we use the *align* algorithm to enable the alignment. Unfortunately, it will align $:=(T,1)$ (Pascal Line 8) with $=(A,Input)$, because this yields the minimal cost of repair. If we prohibit this alignment via a filter, then it can align $Read(A)$ and $=(A,Input)$ by unblocking:

$$Read(A) \equiv= (A,Input) \tag{5.3}$$

To unblock it, reformation suggests adding a new constant C_1 to *Read*, merging *Read* with $=$, and merging C_1 with *Input*. This process is $Read(A) \Rightarrow Read(A,C_1) \Rightarrow= (A,C_1) \Rightarrow= (A,Input)$. The repairs can also be applied to $Read(B)$ (Pascal Line 2) and transfer it to $=(B,Input)$ (Python Line 2). This means that $read(a)$ (Pascal Line 1) and $read(b)$ (Pascal Line 2) now can be transferred to $a \ = \ input()$ (Python Line 1) and $b \ = \ input()$ (Python Line 2). Now we can use these alignments and repairs to produce new sentences for Python. For instance, $writeln(a*b)$ (Pascal Line 7) can be transferred to $print(a*b)$, $t \ := \ 1$ (Pascal Line 7) can be transferred to $t \ = \ 1$, and $writeln(s)$ (Pascal Line 6 and Line 10) can be transferred to $print(s)$ (Python Line 6) and another new line which is also $print(s)$. If we have a Pascal sentence $read(u)$, then we can transfer it to a Python sentence $u = input()$.

A glance at the programs reveals that the $for-to-do$ (Pascal Line 5) sentence can also be aligned with the $for-in$ sentence (Python Line 5). To align them, reformation needs to unblock the following unification:

$$\begin{gathered} ForToDo(I,To(A,B),Do(:=(S,+(S,I)))) \\ \equiv \\ ForIn(I,Xrange(A,+(B,1)),Execute(=(S,+(S,I)))) \end{gathered} \tag{5.4}$$

Reformation can suggest the following alignments via merging:

$$\{ForToDo \mapsto ForIn, To \mapsto Xrange, Do \mapsto Execute, :=\mapsto=\} \tag{5.5}$$

Also, to make the unification $B \equiv +(B,1)$ succeed, reformation can suggest adding a new functor $+$ to B, adding a new argument C_2 to $+$ and merging C_2 with 1. This process is $B \Rightarrow +(B) \Rightarrow +(B,C_2) \Rightarrow +(B,1)$. These repairs can also be applied to $ForToDo(I,To(A,B),Do(:=(T,+(T,I))))$ (Pascal Line 10) and can transfer it to $ForIn(I,Xrange(A,+(B,1)),Execute(=(T,+(T,I))))$. At this moment, we have obtained the following Python program where all sentences can be aligned with the original Pascal sentences, and it can be considered as a blend of the two original programs:

$Blend_{Pascal \mapsto Python}$

Line	*Program*	*Predicate Form*
1 :	$a = input()$	$=(A,Input)$
2 :	$b = input()$	$=(B,Input)$
3 :	$print(a+b)$	$Print(+(A,B))$
4 :	$s = 0$	$=(S,0)$
5 :	$for\ i\ in\ xrange(a,b+1):$	$ForIn(I,Xrange(A,+(B,1)),$
	$\quad s = s+i$	$\quad Execute(=(S,+(S,I))))$
6 :	$print(s)$	$Print(S)$
7 :	$print(a*b)$	$Print(*(A,B))$
8 :	$t = 1$	$=(T,1)$
9 :	$for\ i\ in\ xrange(a,b+1):$	$ForIn(I,Xrange(A,+(B,1)),$
	$\quad t = t+i$	$\quad Execute(=(T,+(T,I))))$
10 :	$print(t)$	$Print(T)$

(5.6)

This Python program can read two numbers a and b and output the values of $a+b$, $a+(a+1)+\cdots+(b-1)+b$, $a \times b$ and $a \times (a+1) \times \cdots \times (b-1) \times b$. These functions

are nearly the same as those of the original Pascal program, which mean that the translation from the Pascal program to the Python program has been successful. Also, the alignments and repairs obtained can be used to translate more Python sentences, which might help the programmer finish his/her task. However, to obtain this result, the alignment between $:= (T,1)$ and $= (A, Input)$ needs to be prohibited by using the filter, as discussed before. This means that interactions between the *align* algorithm and users may be helpful. Also, this reveals that the greedy strategy may not be sufficient or suitable, and we may need some heuristic methods to improve the quality of alignment.

5.2 Gravity and electrostatic force

Newton's law of universal gravitation reveals that two particles attract each other via gravity. Assume that m_1 and m_2 are the two particles, $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ are 3-dimensional vectors which denote the coordinates of the two particles, and $G = 6.67 \times 10^{-11}$ is the gravitational constant. Newton's law of universal gravitation reveals that m_2 attracts m_1 via gravity, and the gravity is a 3-dimensional vector $\boldsymbol{F_G}(m_1, m_2, \boldsymbol{p_1}, \boldsymbol{p_2}) = Gm_1m_2(\boldsymbol{p_2} - \boldsymbol{p_1})/\|\boldsymbol{p_2} - \boldsymbol{p_1}\|^3$. This can be formalised as a theory T_G which contains an implication rule:

$$\begin{aligned} &Particle_1(m_1,x_1,y_1,z_1) \wedge Particle_2(m_2,x_2,y_2,z_2) \implies \\ &Gravity(F_G(m_1,m_2,Vec(x_1,y_1,z_1),Vec(x_2,y_2,z_2))) \end{aligned} \tag{5.7}$$

and two rewrite rules:

$$\begin{aligned} &F_G(m_1,m_2,Vec(x_1,y_1,z_1),Vec(x_2,y_2,z_2)) \\ \Rightarrow\ &G \times m_1 \times m_2 \times \frac{Vec(x_2,y_2,z_2) - Vec(x_1,y_1,z_1)}{\|Vec(x_2,y_2,z_2) - Vec(x_1,y_1,z_1)\|^3} \end{aligned} \tag{5.8}$$

$$G \Rightarrow 6.67 \times 10^{-11} \tag{5.9}$$

Here, " $\implies$ " is the symbol for implication, whereas "$\Rightarrow$" is the symbol for rewriting. The implication rule (5.7) means that if there exist two particles $Particle_1$ and $Particle_2$ such that the mass of $Particle_1$ is m_1, the coordinates of $Particle_1$ are (x_1,y_1,z_1), the mass of $Particle_2$ is m_2, and the coordinates of $Particle_2$ are (x_2,y_2,z_2), then the gravity applied to $Particle_1$ is $F_G(m_1,m_2,Vec(x_1,y_1,z_1),Vec(x_2,y_2,z_2))$. The rewrite rule (5.8) is the formalisation of Newton's law. The rewrite rule (5.9) means that the gravitational constant G is 6.67×10^{-11}.

Similarly, Coulomb's law reveals that two electric charges attract or repel each other via electrostatic forces. Assume that q_1 and q_2 are the two electric charges, $\boldsymbol{r_1}$ and $\boldsymbol{r_2}$ are 3-dimensional vectors which denote the coordinates of the two electric charges, and $K_e = 8.99 \times 10^9$ is the electrostatic constant. Coulomb's law reveals that if both q_1 and q_2 are positive (or negative), q_2 repels q_1 via an electrostatic force which is a 3-dimensional vector $\boldsymbol{F_E}(q_1, q_2, \boldsymbol{r_1}, \boldsymbol{r_2}) = K_e q_1 q_2 (\boldsymbol{r_1} - \boldsymbol{r_2}) / \|\boldsymbol{r_1} - \boldsymbol{r_2}\|^3$. It is noticeable that the structure of $\boldsymbol{F_G}(m_1, m_2, \boldsymbol{p_1}, \boldsymbol{p_2}) = G m_1 m_2 (\boldsymbol{p_2} - \boldsymbol{p_1}) / \|\boldsymbol{p_2} - \boldsymbol{p_1}\|^3$ is similar to that of $\boldsymbol{F_E}(q_1, q_2, \boldsymbol{r_1}, \boldsymbol{r_2}) = K_e q_1 q_2 (\boldsymbol{r_1} - \boldsymbol{r_2}) / \|\boldsymbol{r_1} - \boldsymbol{r_2}\|^3$, except the fact that the positions of $\boldsymbol{p_1}$, $\boldsymbol{p_2}$, $\boldsymbol{r_1}$ and $\boldsymbol{r_2}$ are different. The similarity means that it is possible for HDTP (Schmidt et al., 2014) to invent Coulomb's law by transferring Newton's law from T_G. Assume that we have another theory T_E which contains an implication rule:

$$\begin{aligned} &Charge_1(q_1, x_1, y_1, z_1) \wedge Charge_2(q_2, x_2, y_2, z_2) \implies \\ &EleForce(F_E(q_1, q_2, Vec(x_1, y_1, z_1), Vec(x_2, y_2, z_2))) \end{aligned} \tag{5.10}$$

and three rewrite rules:

$$\begin{aligned} &F_E(4 \times 10^{-8}, 4 \times 10^{-8}, Vec(0.1, 0, 0), Vec(0, 0, 0)) \\ &\Rightarrow Vec(1.44 \times 10^{-2}, 0, 0) \end{aligned} \tag{5.11}$$

$$(8.99 \times 10^9) \times (4 \times 10^{-8}) \times (4 \times 10^{-8}) \times \frac{Vec(0,0,0) - Vec(0.1,0,0)}{\|Vec(0,0,0) - Vec(0.1,0,0)\|^3}$$
$$\Rightarrow -((8.99 \times 10^9) \times (4 \times 10^{-8}) \times (4 \times 10^{-8}) \times \frac{Vec(0.1,0,0) - Vec(0,0,0)}{\|Vec(0.1,0,0) - Vec(0,0,0)\|^3}) \tag{5.12}$$

$$\begin{aligned} &(8.99 \times 10^9) \times (4 \times 10^{-8}) \times (4 \times 10^{-8}) \times \frac{Vec(0.1,0,0) - Vec(0,0,0)}{\|Vec(0.1,0,0) - Vec(0,0,0)\|^3} \\ &\Rightarrow Vec(1.44 \times 10^{-2}, 0, 0) \end{aligned} \tag{5.13}$$

Here, " $\implies$ " is the symbol for implication, whereas "$\Rightarrow$" is the symbol for rewriting. The implication rule (5.10) means that if there exist two matters $Charge_1$ and $Charge_2$ such that the quantity of electric charge of $Charge_1$ is q_1, the coordinates of $Charge_1$ are (x_1, y_1, z_1), the mass of $Charge_2$ is q_2, and the coordinates of $Charge_2$ are (x_2, y_2, z_2), then the electrostatic force applied to $Charge_1$ is $F_E(q_1, q_2, Vec(x_1, y_1, z_1), Vec(x_2, y_2, z_2))$. The rewrite rule (5.11) indicates that if $q_1 = q_2 = 4 \times 10^{-8}$, $(x_1, y_1, z_1) = (0.1, 0, 0)$ and $(x_2, y_2, z_2) = (0, 0, 0)$, then the electrostatic force applied to $Charge_1$ will is $Vec(1.44 \times 10^{-2}, 0, 0)$. This can be considered as data from the real world. The other rewrite rules are used to simplify vector expressions.

HDTP (Schmidt et al., 2014) is able to produce an analogical blend for T_G and T_E via the following alignments:

$$\begin{aligned} &Particle_1 \to Charge_1, Particle_2 \to Charge_2, \\ &Gravity \to EleForce, F_G \to F_E, m_1 \to q_1, m_2 \to q_2 \end{aligned} \tag{5.14}$$

This yields a blend $B_{G \mapsto E}$ which contains two new rules:

$$\begin{aligned} &F_E(q_1, q_2, Vec(x_1, y_1, z_1), Vec(x_2, y_2, z_2)) \\ \Rightarrow\ &G \times q_1 \times q_2 \times \frac{Vec(x_2, y_2, z_2) - Vec(x_1, y_1, z_1)}{\|Vec(x_2, y_2, z_2) - Vec(x_1, y_1, z_1)\|^3} \end{aligned} \tag{5.15}$$

$$G \Rightarrow 6.67 \times 10^{-11} \tag{5.16}$$

However, $B_{G \mapsto E}$ violates local confluence, because $F_E(4 \times 10^{-8}, 4 \times 10^{-8}, Vec(0.1, 0, 0), Vec(0, 0, 0))$ can be rewritten to $Vec(1.44 \times 10^{-2}, 0, 0)$ via Rule (5.11), but it can also be rewritten to $(6.67 \times 10^{-11}) \times (4 \times 10^{-8}) \times (4 \times 10^{-8}) \times \frac{Vec(0,0,0) - Vec(0.1,0,0)}{\|Vec(0,0,0) - Vec(0.1,0,0)\|^3}$ via Rule (5.15) and Rule (5.16) sequentially. In this case, the *redirect* algorithm can try to repair the rules. When the path to $Vec(1.44 \times 10^{-2}, 0, 0)$ is chosen as the correct path, a wanted expression can be $(8.99 \times 10^9) \times (4 \times 10^{-8}) \times (4 \times 10^{-8}) \times \frac{Vec(0.1,0,0) - Vec(0,0,0)}{\|Vec(0.1,0,0) - Vec(0,0,0)\|^3}$. On the other hand, an unwanted expression can be $(6.67 \times 10^{-11}) \times (4 \times 10^{-8}) \times (4 \times 10^{-8}) \times \frac{Vec(0,0,0) - Vec(0.1,0,0)}{\|Vec(0,0,0) - Vec(0.1,0,0)\|^3}$. Therefore, reformation will unblock the following unification:

$$\begin{aligned} &(6.67 \times 10^{-11}) \times (4 \times 10^{-8}) \times (4 \times 10^{-8}) \times \frac{Vec(0,0,0) - Vec(0.1,0,0)}{\|Vec(0,0,0) - Vec(0.1,0,0)\|^3} \\ &\equiv \\ &(8.99 \times 10^9) \times (4 \times 10^{-8}) \times (4 \times 10^{-8}) \times \frac{Vec(0.1,0,0) - Vec(0,0,0)}{\|Vec(0.1,0,0) - Vec(0,0,0)\|^3} \end{aligned} \tag{5.17}$$

Reformation will suggest merging 6.67 and 8.99, removing "$-$" from -11 and merging 11 and 9. They can change (6.67×10^{-11}) to (8.99×10^9). Also, reformation will suggest permuting the two arguments of $Vec(0,0,0) - Vec(0.1,0,0)$, so that it becomes $Vec(0.1,0,0) - Vec(0,0,0)$ and can be aligned with the wanted expression. These repairs will be applied to Rule (5.16), which is the last rule applied to the unwanted expression, changing it to:

$$G \Rightarrow 8.99 \times 10^9 \tag{5.18}$$

However, after this change, $B_{G\mapsto E}$ is still not locally confluent, because $F_E(4\times 10^{-8}, 4\times 10^{-8}, Vec(0.1,0,0), Vec(0,0,0))$ can be rewritten to $Vec(1.44\times 10^{-2},0,0)$ via Rule (5.11), but it can also be rewritten to $-Vec(1.44\times 10^{-2},0,0)$ via Rule (5.15), (5.18) (5.12) and (5.13) sequentially. In this case, the *upgrade* algorithm can repair them further: If the path to $Vec(1.44\times 10^{-2},0,0)$ is chosen as the correct path again, then a wanted expression can be $G\times(4\times 10^{-8})\times(4\times 10^{-8})\times \frac{Vec(0.1,0,0)-Vec(0,0,0)}{\|Vec(0.1,0,0)-Vec(0,0,0)\|^3}$. On the other hand, an unwanted expression can be $G\times(4\times 10^{-8})\times(4\times 10^{-8})\times \frac{Vec(0,0,0)-Vec(0.1,0,0)}{\|Vec(0,0,0)-Vec(0.1,0,0)\|^3}$. Hence, reformation will unblock the following unification:

$$\begin{aligned} G\times(4\times 10^{-8})\times(4\times 10^{-8})\times \frac{Vec(0,0,0)-Vec(0.1,0,0)}{\|Vec(0,0,0)-Vec(0.1,0,0)\|^3} \\ \equiv \\ G\times(4\times 10^{-8})\times(4\times 10^{-8})\times \frac{Vec(0.1,0,0)-Vec(0,0,0)}{\|Vec(0.1,0,0)-Vec(0,0,0)\|^3} \end{aligned} \tag{5.19}$$

Reformation will suggest permuting the two arguments of $Vec(0,0,0)-Vec(0.1,0,0)$, so that it becomes $Vec(0.1,0,0)-Vec(0,0,0)$ and can be aligned with the wanted expression. This repair will be applied to Rule (5.15), which is the last rule applied to the unwanted expression, changing it to:

$$\begin{aligned} & F_E(q_1,q_2,Vec(x_1,y_1,z_1),Vec(x_2,y_2,z_2)) \\ \Rightarrow\ & G\times q_1\times q_2\times \frac{Vec(x_1,y_1,z_1)-Vec(x_2,y_2,z_2)}{\|Vec(x_1,y_1,z_1)-Vec(x_2,y_2,z_2)\|^3} \end{aligned} \tag{5.20}$$

Now, $Vec(x_2,y_2,z_2)-Vec(x_1,y_1,z_1)$ has been changed to $Vec(x_1,y_1,z_1)-Vec(x_2,y_2$

$,z_2)$. After this repair, T_E has been locally confluent. The final version of $B_{G\mapsto E}$ is:

$$
\begin{array}{c}
B_{G\mapsto E} \\
\hline
\textbf{< Implication Rule >} \\
\hline
Charge_1(q_1,x_1,y_1,z_1) \wedge Charge_2(q_2,x_2,y_2,z_2) \implies \\
EleForce(F_E(q_1,q_2,Vec(x_1,y_1,z_1),Vec(x_2,y_2,z_2))) \\
\hline
\textbf{< Original Rewrite Rule >} \\
\hline
F_E(4\times10^{-8},4\times10^{-8},Vec(0.1,0,0),Vec(0,0,0)) \\
\Rightarrow Vec(1.44\times10^{-2},0,0) \\
(8.99\times10^{9})\times(4\times10^{-8})\times(4\times10^{-8})\times\frac{Vec(0,0,0)-Vec(0.1,0,0)}{\|Vec(0,0,0)-Vec(0.1,0,0)\|^3} \\
\Rightarrow -((8.99\times10^{9})\times(4\times10^{-8})\times(4\times10^{-8})\times\frac{Vec(0.1,0,0)-Vec(0,0,0)}{\|Vec(0.1,0,0)-Vec(0,0,0)\|^3}) \\
(8.99\times10^{9})\times(4\times10^{-8})\times(4\times10^{-8})\times\frac{Vec(0.1,0,0)-Vec(0,0,0)}{\|Vec(0.1,0,0)-Vec(0,0,0)\|^3} \\
\Rightarrow Vec(1.44\times10^{-2},0,0) \\
\hline
\textbf{< New Rewrite Rule >} \\
\hline
F_E(q_1,q_2,Vec(x_1,y_1,z_1),Vec(x_2,y_2,z_2)) \\
\Rightarrow G\times q_1\times q_2\times\frac{Vec(x_1,y_1,z_1)-Vec(x_2,y_2,z_2)}{\|Vec(x_1,y_1,z_1)-Vec(x_2,y_2,z_2)\|^3} \\
G\Rightarrow 8.99\times10^{9}
\end{array}
\tag{5.21}
$$

It is noticeable that the new rules are equivalent to Coulomb's law which indicates that $\boldsymbol{F_E}(q_1,q_2,\boldsymbol{r_1},\boldsymbol{r_2}) = K_e q_1 q_2(\boldsymbol{r_1}-\boldsymbol{r_2})/\|\boldsymbol{r_1}-\boldsymbol{r_2}\|^3$, $\boldsymbol{p_1}$ and $K_e = 8.99\times10^9$. The only flaw is that G may confuse people, because G is equivalent to the electrostatic constant, but not the gravitational constant. This result means that Coulomb's law has been invented via transferring Newton's law from the domain of gravity to the domain of electrostatic force and applying the *redirect* algorithm. In reality, electrostatic force is similar to gravity. A difference between them is that their direction is opposite: gravities always attracts two particles, whereas electrostatic forces always repels two positive charges (or two negative charges). This is the reason why for the right-hand side of $\boldsymbol{F_G}(m_1,m_2,\boldsymbol{p_1},\boldsymbol{p_2}) = Gm_1m_2(\boldsymbol{p_2}-\boldsymbol{p_1})/\|\boldsymbol{p_2}-\boldsymbol{p_1}\|^3$ and that of $\boldsymbol{F_E}(q_1,q_2,\boldsymbol{r_1},\boldsymbol{r_2}) = K_e q_1 q_2(\boldsymbol{r_1}-\boldsymbol{r_2})/\|\boldsymbol{r_1}-\boldsymbol{r_2}\|^3$, $\boldsymbol{p_1}$ needs to be aligned with $\boldsymbol{r_2}$, and $\boldsymbol{p_2}$ needs to be aligned with $\boldsymbol{r_1}$.

In this example, three different kinds of repairs are used: (a) Merging; (b) Deleting a functor; (c) Permuting arguments. The first two are used to change 6.67×10^{-11}

to 8.99×10^9, which means changing the gravitational constant to the electrostatic constant. The third is used to swap the two vectors, which means reversing the direction of the electrostatic force. These are the cases that the repairs could have realistic meanings. It is also noticeable that the *redirect* algorithm needs to repair the rules twice, because at most one rule can be repaired each time. After each time of repair, the rewriting tree needs to be rebuilt, in order to check local confluence and identify potentially faulty paths. This is the reason why *redirect* is recursive.

All repair operations mentioned above can be done automatically via the *redirect* algorithm, without any help from users.

5.3 Addition and subtraction

Assume that we have known that $+$ is analogous to $-$ and the successor *Succ* is analogous to the predecessor *Pred*. Given the following theory about addition which contains two Peano axioms:

$$\frac{T_{Add}}{\begin{array}{c} x+0=x \\ x+Succ(y)=Succ(x+y) \end{array}} \tag{5.22}$$

By mapping $+$ to $-$ and mapping the successor *Succ* to the predecessor *Pred*, the following theory about subtraction can be obtained:

$$\frac{T_{Add\mapsto Sub}}{\begin{array}{c} x-0=x \\ x-Pred(y)=Pred(x-y) \end{array}} \tag{5.23}$$

Also, we look forward to proving $0-Pred(0)=Succ(0)$ and $0-Succ(0)=Pred(0)$ by using the two axioms.

Obviously, $x-Pred(y)=Pred(x-y)$, which means that $x-(y-1)=(x-y)-1$, is faulty. The correct axiom should be $x-Pred(y)=Succ(x-y)$, which means that $x-(y-1)=(x-y)+1$, or $x-Succ(y)=Pred(x-y)$, which means that $x-(y+1)=(x-y)-1$. Fortunately, the *upgrade* algorithm is able to repair it. Firstly, $T_{Add\mapsto Sub}$ can be rewritten to the following two implication rules, which are represented by the list format described by Section 3.1:

$$(Minus\ x\ 0) \implies x \tag{5.24}$$

$$Minus\ x\ (Pred\ y) \implies (Pred\ (Minus\ x\ y)) \tag{5.25}$$

To extend the use of Rule (5.24), we add an extra rule:

$$(p\ (Minus\ x\ 0)) \implies (p\ x) \tag{5.26}$$

where p is a functor, but it is a variable. This is a second-order rule which enables the functor p to be instantiated. In other words, it enables implications such as $(Succ\ (Minus\ x\ 0)) \implies (Succ\ x)$ and $(Pred\ (Minus\ x\ 0)) \implies (Pred\ x)$. Also, $0 - Pred(0) = Succ(0)$ and $0 - Succ(0) = Pred(0)$, which are expected to be proved, can be rewritten to the following theorems:

$$(Minus\ 0\ (Pred\ 0)) \longrightarrow (Succ\ 0) \tag{5.27}$$

$$(Minus\ 0\ (Succ\ 0)) \longrightarrow (Pred\ 0) \tag{5.28}$$

Then the *upgrade* algorithm is used to enable their proofs. For Theorem (5.27), the first proof step succeeds:

$$\frac{(Succ\ 0) \implies}{(Succ\ (Minus\ 0\ 0)) \implies}(p\ (Minus\ x\ 0)) \implies (p\ x) \tag{5.29}$$

However, the second step fails:

$$\frac{(Succ\ (Minus\ 0\ 0)) \implies}{Blocked}Minus\ x\ (Pred\ y) \implies (Pred\ (Minus\ x\ y) \tag{5.30}$$

This is because the unification between $(Succ\ (Minus\ 0\ 0))$ and $(Pred\ (Minus\ x\ y)$ fails. Reformation can suggest changing *Pred* to *Succ*. Applying this repair to the right-hand side of $Minus\ x\ (Pred\ y) \implies (Pred\ (Minus\ x\ y)$ will change it to $Minus\ x\ (Pred\ y) \implies (Succ\ (Minus\ x\ y)$. Now the proof has been unblocked:

$$\frac{\dfrac{(Succ\ (Minus\ 0\ 0)) \implies}{Minus\ 0\ (Pred\ 0) \implies}Minus\ x\ (Pred\ y) \implies (Succ\ (Minus\ x\ y)}{\implies}Assumption \tag{5.31}$$

The assumption is $\implies Minus\ 0\ (Pred\ 0)$. This means that Theorem (5.27) has been proved.

However, the *upgrade* algorithm fails to unblock the proof of Theorem (5.28) further. Specifically, although the first proof step succeeds:

$$\frac{(Pred\ 0) \implies}{(Pred\ (Minus\ 0\ 0)) \implies}(p\ (Minus\ x\ 0)) \implies (p\ x) \tag{5.32}$$

the second step fails:

$$\frac{(Pred\ (Minus\ 0\ 0)) \implies}{Blocked}Minus\ x\ (Pred\ y) \implies (Succ\ (Minus\ x\ y) \tag{5.33}$$

To unblock the second step, reformation suggests changing $Minus\ x\ (Pred\ y) \implies (Succ\ (Minus\ x\ y))$ to $Minus\ x\ (Pred\ y) \implies (Pred\ (Minus\ x\ y))$. However, this change blocks the proof of Theorem (5.27)! If we ignore this accident and continue unblocking the proof, the goal will become $Minus\ 0\ (Pred\ 0) \implies$. The third step is to unblock the unification between the assumption $\implies Minus\ 0\ (Succ\ 0)$ and the goal:

$$Minus\ 0\ (Succ\ 0) \equiv Minus\ 0\ (Pred\ 0) \tag{5.34}$$

Reformation can suggest changing *Succ* to *Pred*. However, it does not change any axioms, but changes the assumption. On the other hand, reformation can also suggest changing *Pred* to *Succ*. However, it still does not change any axioms, but changes the goal.

The above example reveals two issues: The first is that unblocking can also block a wanted proof. This can be a negative side effect of unblocking. For this issue, a possible solution, which is our future work, is to keep the original axiom unchanged to prevent the wanted proof from being blocked, add a new copy of this axiom and apply unblocking to this copy[1]. The second is that for an implication rule $P \implies Q$, the *upgrade* algorithm can only repair Q, but cannot repair P. To solve this issue, some adaptation for the current *upgrade* implementation is needed.

5.4 Lists and binary trees

Consider a binary tree structure which is defined by $Empty$ and $Node(v,l,r)$ [2]: $Empty$ means an empty tree, and $Node(v,l,r)$ means a node, where v is a value, l is a left subtree and r is a right subtree. For instance, a binary tree can be $Node(A,Empty,Empty)$, which only has a node with a value A and without any subtrees, or $Node(A,Node(B,Empty,Empty),Empty)$, which has a node with a value A, a left subtree $Node(B,Empty,Empty)$, but without a right subtree. Formally, this binary tree structure can be defined via two functions: $Empty : BTree$ and $Node : Val \times BTree \times BTree \to BTree$, where $BTree$ is a type of binary trees and Val is a type of values. This structure can be analogous to the list structure which is defined via $Nil : List$ and $Cons : El \times List$ and has been discussed by previous sections.

[1] This idea arose after a discussion with Xue Li about the possibility of adding and deleting axioms via reformation.

[2] This example is adapted from an example of binary trees described by Bundy (2015)

Assume that a source theory of lists is:

$$\frac{\frac{\frac{T_{List}}{\textbf{< Type Declaration >}}}{\begin{array}{cc} Nil : List & 0 : Nat \\ Cons : El \times List \to List & Succ : Nat \to Nat \\ Length : List \to Nat & Plus : Nat \times Nat \to Nat \end{array}}}{\textbf{< Axiom >}}}{\begin{array}{cc} Length(Nil) = 0 & Length(Cons(h,l)) = Succ(Length(l)) \\ Plus(x,0) = x & Plus(x,Succ(y)) = Succ(Plus(x,y)) \end{array}} \tag{5.35}$$

where $Length(l)$ is a function for computing the length of a list l. Also, a target theory of binary trees is:

$$\frac{\frac{\frac{T_{BTree}}{\textbf{< Type Declaration >}}}{\begin{array}{cc} Empty : BTree & 0 : Nat \\ Node : Val \times BTree \times BTree \to BTree & Succ : Nat \to Nat \\ Size : BTree \to Nat & Plus : Nat \times Nat \to Nat \end{array}}}{\textbf{< Axiom >}}}{\begin{array}{c} Size(Node(v,Empty,Node(v,Empty,Empty))) = Succ(Succ(0)) \\ Plus(x,0) = x \quad Plus(x,Succ(y)) = Succ(Plus(x,y)) \end{array}} \tag{5.36}$$

where $Size(t)$ is a function for computing the size of a binary tree t. HDTP (Schmidt et al., 2014) is able to produce a blending of theories by aligning the axioms about natural numbers (They can be aligned because they are totally the same). However, the alignments are insufficient when lists and binary trees are also expected to be aligned. The reasons why they cannot be aligned are twofold: Firstly, the axioms do not indicate possible alignments between them. Secondly, *Cons* and *Node* have different arities. To solve the first problem, the information about types is used via the method described by Section 3.2.4. To solve the second problem, the *align* algorithm is used. After the two steps, the following alignments hold:

$$\{Nil \mapsto Empty, Cons \mapsto Node, Length \mapsto Size, List \mapsto BTree, El \mapsto Val\} \tag{5.37}$$

In particular, reformation has suggested adding a new argument *r* to *Cons*, so that *Cons*

and $Node$ have the same arity. Hence, the blend of T_{List} and T_{BTree} becomes:

$$\dfrac{\dfrac{\dfrac{T_{List \mapsto BTree}}{\boldsymbol{< Type\ Declaration >}}}{\begin{array}{ll} Empty : BTree & 0 : Nat \\ Node : Val \times BTree \times BTree \to BTree & Succ : Nat \to Nat \\ Size : BTree \to Nat & Plus : Nat \times Nat \to Nat \end{array}}}{\dfrac{\boldsymbol{< Original\ Axiom >}}{\begin{array}{l} Size(Node(v, Empty, Node(v, Empty, Empty))) = Succ(Succ(0)) \\ Plus(x,0) = x \qquad Plus(x, Succ(y)) = Succ(Plus(x,y)) \end{array}}}}{\dfrac{\boldsymbol{< New\ Axiom >}}{Size(Empty) = 0 \qquad Size(Node(v,l,r)) = Succ(Size(l))}} \tag{5.38}$$

There are two new axioms. The first axiom $Size(Empty) = 0$ is transferred from $Length(Nil) = 0$. The second axiom $Size(Node(v,l,r)) = Succ(Size(l))$ is transferred from $Length(Cons(h,l)) = Succ(Length(l))$, but it is false under our usual interpretation, because it does not take the size of the right subtree r into account. In other words, its right-hand side is expected to be $Succ(Plus(Size(l), Size(r))$.

Although these axioms can be considered as rewrite rules, it is difficult for the *redirect* algorithm to repair them. Specifically, the rules violate local confluence, because $Size(Node(v, Empty, Node(v, Empty, Empty)))$ can be rewritten to $Succ(Succ(0))$ or $Succ(0)$, but the two expressions are not equivalent. If $Succ(Succ(0))$ is chosen as the correct one, then backward rewriting will be applied to it. However, there exists a rule $Plus(x,0) \Rightarrow x$ which enables any terms and subterms s to be rewritten to $Plus(s,0)$. This means that the backward rewriting cannot terminate, and the *redirect* algorithm also cannot terminate. Being not able to process non-termination is a drawback of this algorithm. Practically, setting a maximum depth for backward rewriting could prevent non-termination to some extent. Recall the implementation of this algorithm in Section 3.4: It is a function $redirect(+RuleIn, +DF, +DB, +N, -RuleOut, -R)$, where DB denotes the maximum depth for backward rewriting. For this example, if DB is set to 2, then the *redirect* algorithm is able to repair these rules: Applying backward rewriting to $Succ(Succ(0))$, we can obtain a wanted expression $Succ(Plus(Succ(0), Size(Empty)))$. Also, there is an unwanted expression $Succ(Size(Empty))$ on the path to $Succ(0)$. Thus, reformation will unblock the following unification:

$$Succ(Size(Empty)) \equiv Succ(Plus(Succ(0), Size(Empty))) \tag{5.39}$$

It suggests adding a functor $Plus$ to $Size$ on the left and adding a new argument u to $Plus$ on the left, so that the unification pair becomes:

$$Succ(Plus(u, Size(Empty))) \equiv Succ(Plus(Succ(0), Size(Empty))) \tag{5.40}$$

The unification can succeed. Applying these repairs to the right-hand side of $Size(Node(v,l,r)) \Rightarrow Succ(Size(l))$, it becomes:

$$Size(Node(v,l,r)) \Rightarrow Succ(Plus(u, Size(l))) \tag{5.41}$$

An issue of this rule is that u is independent of any other variables. This is because, when reformation adds a new variable, the variable is independent of the existing variables. This way of repair is suggested by reformation because this variable can be substituted for any terms, so that further repair is not needed.

The above results reveal that there are two problems which need to be solved: The first problem is that it is difficult for the *redirect* algorithm to deal with non-termination. In other words, it can only work in the case that both forward rewriting and backward rewriting are terminating. The second problem is that the operation of adding new variables is not sufficiently intelligent. Specifically, a new variable is always independent of the existing variables, and it can be substituted for any terms, so that it always enables unification. This means that the new variable is just a variable without any realistic meaning. To make it have a realistic meaning, we may need an evolution mechanism which can change it to some other functions and variables. For instance, if u can be changed to $Size(r)$, then $Size(Node(v,l,r)) \Rightarrow Succ(Plus(u, Size(l)))$ can evolve to $Size(Node(v,l,r)) \Rightarrow Succ(Plus(Size(r), Size(l)))$, which is the expected result. The evolution mechanism [3], however, may be complex, because it needs to recognise that u has been substituted for $Succ(0)$, and it needs to infer that $Succ(0)$ is equivalent to $Size(r)$ when $r = Node(v, Empty, Empty)$. The evolution mechanism can be a topic to be explored in the future.

[3] The idea of the evolution mechanism arose from a discussion with Boris Mitrovic. He used "interactions between repairs" to fix a bug in his reformation implementation. This means that it is possible to improve the repairs by analysing the relationships between these repairs.

Chapter 6

Conclusion

6.1 Remarks and observations

Reformation is able to deal with different issues with analogical blends, such as insufficient alignments, inconsistencies and incompleteness. Firstly, the *align* algorithm can process the insufficient alignments. It achieves this via unblocking of reformation and the greedy strategy. Secondly, the *revise* algorithm can process the inconsistencies by applying blocking to unwanted proofs. In particular, the *redirect* algorithm can repair inconsistent rewrite rules. Thirdly, the *upgrade* algorithm can process the incompleteness by applying unblocking to wanted proofs. It is often the case that reformation can suggest repairs to solve these issues, and some repairs even have realistic meanings.

In this project, the *align*, *redirect* and *upgrade* algorithms have been implemented. Since these implementations are based on the unblocking function of reformation, in order to deal with more complicated problems, the unblocking function has also been revised and refined. These implementations have been used to repair some analogical blends: The *align* algorithm has been used to adjust the alignments of the blend of two food chain ontologies, the blend of natural numbers and lists, the blend of lists and binary trees and the blend of Pascal and Python programs. The *redirect* algorithm has been used to repair inconsistent rewrite rules in the blend of lists and binary trees, the blend of the trigonometric sine and cosine functions and the blend of gravity and electrostatic force. The *upgrade* algorithm has been used to process some incomplete analogical blends, such as the blend of two capital ontologies and the blend of addition and subtraction. In addition, the previous implementation of the *revise* algorithm (Bundy and Mitrovic, 2016) has been used to process the inconsistent blend of natural

numbers and lists.

6.2 Unsolved problems and future work

There are some unsolved problems:

- **The side effects of unblocking.** The side effects of unblocking are twofold: Firstly, an unwanted proof may be blocked when reformation is used to unblock a wanted proof, as discussed by Section 3.3.1.3. This is a positive effect. Secondly, a wanted proof may be blocked, as discussed by Section 5.3. This is a negative effect. Currently, the *redirect* and *upgrade* implementations are not able to predict and deal with the negative effect.

- **The need of adding new axioms.** The example in Section 5.3 reveals that we not only need to repair existing axioms, but also need to add new axioms. Unfortunately, the current implementation of reformation is not able to add new axioms.

- **Independent variables.** All new variables added by reformation are independent of existing variables, as discussed by Section 5.4. This may reduce the quality of repair. Therefore, more intelligent algorithms are needed to solve this problem.

In the future, we will try to solve these problems. Firstly, a protection mechanism is need to prevent a wanted proof from being blocked by reformation. Secondly, we need to consider how to use reformation to add new axioms (or delete existing axioms), as discussed by Section 5.3[1]. Thirdly, an evolution mechanism is needed to process the independent variables, as discussed by Section 5.4[2]. Also, in the future, we will try to extend the use of reformation to more fields. For instance, it is possible to combine it with a lemma discovery method based on statistical proof-pattern recognition (Heras et al., 2013), or analogy driven lemma discovery (Johansson and Maclean, 2013) for completing proofs via discovering new lemmas.

[1] Again, thanks to Xue Li for the discussion about adding and deleting axioms via reformation.
[2] Again, thanks to Boris Mitrovic for the discussion about "interactions between repairs".

Bibliography

Blanchette, J. C. (2010). Nitpick: A counterexample generator for Isabelle/HOL based on the relational model finder kodkod. In *Short papers for 17th International Conference on Logic for Programming, Artificial intelligence, and Reasoning, LPAR-17-short, Yogyakarta, Indonesia, October 10-15, 2010*, pages 20–25.

Bou, F., Schorlemmer, M., Corneli, J., Gómez-Ramírez, D., Maclean, E., Smaill, A., and Pease, A. (2015). The role of blending in mathematical invention. In *Proceedings of the Sixth International Conference on Computational Creativity, Park City, Utah, USA, June 29 - July 2, 2015.*, pages 55–62.

Bundy, A. (1983). *The Computer Modelling of Mathematical Reasoning*. Academic Press London.

Bundy, A. (2013). The interaction of representation and reasoning. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 469(2157).

Bundy, A. (2015). Possible applications of repairing faulty analogical blends using reformation. Technical report, the University of Edinburgh.

Bundy, A. and Mitrovic, B. (2016). Reformation: A domain-independent algorithm for theory repair. Technical report, the University of Edinburgh.

Codescu, M. and Mossakowski, T. (2008). Heterogeneous colimits. In *First International Conference on Software Testing Verification and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008, Workshops Proceedings*, pages 131–140.

Guhe, M., Pease, A., Smaill, A., Martínez, M., Schmidt, M., Gust, H., Kühnberger, K., and Krumnack, U. (2011). A computational account of conceptual blending in basic mathematics. *Cognitive Systems Research*, 12(3-4):249–265.

Gust, H., Kühnberger, K., and Schmid, U. (2006). Metaphors and heuristic-driven theory projection (HDTP). *Theoretical Computer Science*, 354(1):98–117.

Heras, J., Komendantskaya, E., Johansson, M., and Maclean, E. (2013). Proof-pattern recognition and lemma discovery in ACL2. In *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, pages 389–406.

Johansson, M. and Maclean, E. (2013). Analogy driven lemma discovery. Technical report, the University of Edinburgh.

Knuth, D. E. and Bendix, P. B. (1983). Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer.

Kowalski, R. A. and Kuehner, D. (1971). Linear resolution with selection function. *Artificial Intelligence*, 2(3/4):227–260.

Krumnack, U., Schwering, A., Gust, H., and Kühnberger, K. (2007). Restricted higher-order anti-unification for analogy making. In *AI 2007: Advances in Artificial Intelligence, 20th Australian Joint Conference on Artificial Intelligence, Gold Coast, Australia, December 2-6, 2007, Proceedings*, pages 273–282.

Kutz, O., Neuhaus, F., Mossakowski, T., and Codescu, M. (2014). Blending in the hub. In *Proceedings of the Fifth International Conference on Computational Creativity, Ljubljana, Slovenia, June 10-13, 2014.*, pages 297–305.

Mitrovic, B. (2013). Repairing inconsistent ontologies using adapted reformation algorithm for sorted logics. *UG4 Project in University of Edinburgh*.

Mossakowski, T., Haxthausen, A. E., Sannella, D., and Tarlecki, A. (2003). CASL - the common algebraic specification language: Semantics and proof theory. *Computers and Artificial Intelligence*, 22(3-4):285–321.

Mossakowski, T., Krumnack, U., and Maibaum, T. (2015). What is a derived signature morphism? In *Recent Trends in Algebraic Development Techniques*, pages 90–109. Springer.

Mossakowski, T., Maeder, C., and Lüttich, K. (2007). The heterogeneous tool set, hets. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European*

Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings, pages 519–522.

Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer.

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

Schmidt, M., Gust, H., Kühnberger, K., and Krumnack, U. (2011). Refinements of restricted higher-order anti-unification for heuristic-driven theory projection. In *KI 2011: Advances in Artificial Intelligence, 34th Annual German Conference on AI, Berlin, Germany, October 4-7,2011. Proceedings*, pages 289–300.

Schmidt, M., Krumnack, U., Gust, H., and Kühnberger, K. (2014). Heuristic-driven theory projection: An overview. In *Computational Approaches to Analogical Reasoning: Current Trends*, pages 163–194.

Schwering, A., Krumnack, U., Kühnberger, K., and Gust, H. (2009). Syntactic principles of heuristic-driven theory projection. *Cognitive Systems Research*, 10(3):251–269.

Tsialos, A. (2014). Repairing inconsistent description logic ontologies using reformation. *UG4 Project in University of Edinburgh*.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. (2010). SWI-Prolog. *CoRR*, abs/1011.5332.