UNIVERSITY OF DUNDEE

SCHOOL OF SCIENCE AND ENGINEERING

MSC IN INFORMATION TECHNOLOGY AND INTERNATIONAL BUSINESS

Implementing Unification Algorithms in Haskell

Author:
Yue LI

Supervisor:
Dr. Ekaterina KOMENDANTSKAYA

September 13, 2015

Executive Summary

Motivation This project is a study of the methods involved in coding existing unification algorithms in a functional programming language Haskell. In particular, it studies how to code, in Haskell language, Robinson's Algorithm and Martelli's Algorithm, which are two of the most famous unification algorithms used in different branches of mathematics, logic and computer science . Using Haskell prepared me for future research in Haskell language.

What is unification Unification algorithm makes two different logical expressions the same by comparing and substituting components of the expressions. For example, we use "Man(X)" to represent "X is a man" in logic; particularly "Man(tom)" means "tom is a man". Given two expressions: "Man(X)" and "Man(tom)", we notice that they may be made equal (or can be unified) by substituting "tom" for "X" in "Man(X)". Hence, unification algorithm will succeed for these two expressions and will compute substitution for X. In contrast, the two sentences "Man(X)" and "Woman(mary)" cannot be unified and for them unification algorithm fails.

Results Robinson's algorithm and Martelli's algorithm were coded and tested. The test was automated by using Haskell library QuickCheck, for which I developed a pair of input generators to provide large amount of valid input, one generator only generates solvable unification problems, the other gives unsolvable unification problems. The test was designed to be able to automatically tell, for input that can be solved, whether the solution given by the algorithms is correct; and for input that has no solution, the test checks whether the algorithm reports so. The programs passed the test.

Conclusion The highlights are successful structural design of the programs, the good testing method in QuickCheck, the use of LaTeX2e instead of Word to typeset the thesis and effective communication with my academic advisors, including my supervisor and her PhD students. Further work can be done to implement more unification algorithms and compare implementations by different people or in a different language such as OCaml.

Declaration

I declare that the special study described in this dissertation has been carried out and the dissertation composed by me, and that the dissertation has not been accepted in fulfilment of the requirements of any other degree or professional qualification.

Certificate

I certify that *Yue Li* has satisfied the conditions of the Ordinance and Regulations and is qualified to submit this dissertation in application for the degree of Master of Science.

Acknowledgement

For her supervising

Dr. Ekaterina Komendantskay

For their tutoring

Mr. Frantisek Farka   Mr. Chris Warburton   Dr. Peng Fu

For their lectures

Dr. Keith Edwards   Prof. Emanuele Trucco   Prof. Annalu Waller
Dr. Paul Seaman   Dr. David Flatla   Dr. Rachel Menzies
Dr. Alison Pease   Dr. Craig Ramsay   Prof. Janet Hughes

For their IT support

Mr. Derek Brankin   Mr. Mahamadou Niakaté

For their administrative support

Miss Claire Cruickshank   Miss Jolly Kwan   Mrs. Sharon Sturrock
Dr. Norman Alm   Mrs. Anne Millar

For their parental love and financial support

Mr. Mincheng Li   Mrs. Yuhong Shan

For his friendship

Mr. Virgil lordache

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

This thesis is a study of the methods involved in implementing unification algorithms in a functional language. In particular, it introduces Haskell implementation of Robinson's Algorithm and Martelli's Algorithm. We first motivate our choice of the research topic.

1.1.1 Importance of Unification Algorithms

Several unification algorithms are common in different branches of mathematics, logic and computer science. We start with having a brief overview of the unification problems arising in these areas.

Logic Inference Requires Unification

Formal logic studies formal language in which logic reasoning can be done. Applying inference rules such as resolution and modus ponens requires deciding whether two terms match. In fact, the concept of unification was first introduced by [1] as part of a resolution theorem prover.

Syntactical unification cares about making two terms syntactically equal by variable substitution. For example, by substituting Constant Symbol "socrates" for Variable Symbol "X" in term "Man (X)", we get "Man (socrates)". In other words, "Man (X)" and "Man (socrates)" are *unified* by *unifier* {socrates/X}, which is a substitution.

More details can be found in [3, 6, 2].

Type Inference Requires Unification

The other area of computer science that uses unification is programming language design and in particular type inference. For example, as demonstrated in [8] and the OCaml code provided by it, an unification algorithm, together with other necessary actions such as annotation and constrain collection, was used in inferring the type of λ-calculus expressions. Worth being noted is that the unification algorithm used for type inference works on type expressions rather than first order logic terms, but the basic idea remains the same.

Appreciation of how unification was used in type inference was done at the last phase of the project, after all chapters but the motivation and thesis conclusion were finished. It turned out that familiarity with Robinson's unification algorithm and its Haskell implementation, as well as with a functional programming language such as Haskell were three important pre-requisites, among others, for the author's understanding the OCaml code provided by [8] that shows how unification was used in type inference.

1.1.2 Why use Haskell

To learn a new way of thinking

For people who had no previous experience with functional programming but who had some familiarity with languages such as C/C++ and Java, learning Haskell is an exciting experience that can help them develop a new way of thinking.

To prepare for future work

Haskell is popular functional programming language that is under active development from researchers around the globe. One issue is to enhance the power of Haskell's type inference engine. The author was interested this topic so he would need to prepare himself to work in this field. The preparation starts from learning Haskell and unification algorithms since Haskell will be the important working language for future work on the type inference engine and an unification algorithm is essential component of a type inference engine.

The efficient way to learn algorithms is to read about them and implement them so it was fine to implement them in any programming language. Since there is also demand for knowledge on Haskell, and the good way to learn a programming language is to read about it, work on tutorial programs and finally use the language to solve project problems, the supervisor advised that the MSc project starts from implementing unification algorithms in Haskell, which can effectively develop knowledge on both Haskell language and unification algorithm. This advice was taken by the author.

1.2 Outline of this thesis

1.2.1 What were learnt during the project

Haskell programming language The knowledge learnt about Haskell language was presented as Appendix A.

Robinson's and Martelli-Montanari unification algorithm These are two fundamental unification algorithms being widely used.

Using $\LaTeX\,2_\varepsilon$ to typeset the thesis $\LaTeX\,2_\varepsilon$ provide good quality typesetting to the thesis, making it neat and easier to write and read.

Uncertainty management Working on the project was enjoyable, because learning new knowledge and using them to solve problems were fun.

On the other hand, the quality of the project work also matters, so there was uncertainty and risk conceived in the effort to learn new things and move on in the project.

Such risk and uncertainty must be suffered, as the practically optimum response, together with some tactics to remedy the loss, such as active and detailed communication with the supervisor and detailed recording of the work done.

Communication in academic context Frequent communication with the supervisor and managed relationship with other members of the same research group enhanced sense of well being and study experience.

1.2.2 Achievements from the project

- Some working knowledge about Haskell
- Understanding of Robinson's and Martelli-Montanari unification algorithm and their application.
- Haskell implementation of the aforementioned algorithms.
- Appreciation of the thesis typeset by $\LaTeX\,2_\varepsilon$, compared with that made with Microsoft Word.
- Good experience and memory about the interaction with the project supervisor and her helpful PhD students.
- Understanding of my interest in learning knowledge.

1.2.3 Content of other chapters

Chapter 2 introduces the subset of syntax of first order logic that is used in the thesis, on which the unification algorithms work. The concept of substitution and occurs check are also introduced.

Chapter 3 introduces Robinson's unification algorithm and the considerations in concern with its Haskell implementation, such as the Haskell data type for first order logic terms and the mutually recursive structure of the Haskell code for Robinson's algorithm.

Chapter 4 introduces Martelli-Montanari (Martelli's, for short) unification algorithm and the considerations in concern with its Haskell implementation, such as how the transforms were implemented, how the need raised for termination check, and how the application of transforms were arranged.

Chapter 5 introduces the development of test method for the implementation of unification algorithms and the design of generators for random unification problems.

Appendix A is an introduction to Haskell language, which was the amount learnt by the author during he worked on the project.

Appendix B provides the source code for the implementation and testing.

Appendix C records the time spent on the project each day during the project and brief summary of daily work.

Appendix D is meeting record.

Appendix E helps the examiners to browse and run the source code.

Appendix F tells users how to use the software to unify their first order logic unification problems.

Chapter 2

Definition of First Order Terms

2.1 Subset of syntax of first order predicate calculus which is used in this thesis

The subset of syntax of first order predicate calculus used in this thesis is given below.

Digit *One of 0, 1, 2, 3 . . . 9*

Upper case letter *An upper case English character (A, B, C, D . . . Z)*

Lower case letter *A lower case English character (a, b, c, d . . . z)*

Letter *Either an Upper case letter or a Lower case letter*

Tail *Any arbitrary combination of Digits and Letters, may be empty.*

Constant Symbol *Lower case letter Tail.* Lower case letter followed by Tail. For example:

a
b1
foo
sonOf

Variable Symbol *Upper case letter Tail.* Upper case letter followed by Tail. For example:

X
Y2
Food
BBQ

Function Symbol *Either a Constant Symbol or a Variable Symbol.* When the Function Symbol is a Variable Symbol, the Function is called a Predicate.

Function *Function Symbol* $(FOT_1, FOT_2, \ldots, FOT_n)$ *where* $0 \leq n \leq m$. Function Symbol followed by a comma delimited (maybe empty but finite) list of FOTs enclosed by a pair of parenthesis. When the FOT list is empty, a Function is treated as a Constant Symbol. For example:

> UseTogether (beer , pie)
> sonOf (adam , X)
> f ()
> like (fatherOf (X),motherOf (Y))

First Order Term (Term or FOT) *One of* {*a Constant Symbol , a Variable Symbol , a Function*}. "Term" and "FOT" are used interchangeably with "First Order Term" in this thesis.

FOTE Member *Term*

First Order Term Equation (FOTE) *FOTE Member = FOTE Member.* A First Order Term Equation is a pair of FOTE Members with an equal sign in between. FOTE is also called an unification problem or an equation in this thesis.

FOTE set $\{FOTE_1, FOTE_2, \ldots, FOTE_n\}$ *where* $0 \leq n \leq m$. FOTE set is comma delimited (maybe empty but finite) list of FOTEs enclosed by a pair of braces. It is possible that $FOTE_i = FOTE_j$ where $i \neq j$.

Chapter 3

Robinson's Algorithm and its Haskell Implementation

This chapter introduces Haskell implementation of Robinson's unification algorithm, which is considered as the first unification algorithm in history and is widely used in theorem provers due to the easiness for its implementation and the rareness of the situation when its exponential time and space expense really cause problem. [9, 2]

3.1 Robinson's Unification Algorithm

This section starts from introducing the concepts of substitution component, substitution, composition and occurs check. Afterwards, the pseudo-code of Robinson's unification algorithm is presented. We then see examples of running the pseudo-code. Discussion comes at last.

3.1.1 Substitution component

Substitution Component is the notation of replacement of a Variable Symbol by a Term which is distinct from the Variable Symbol, denoted by

$$Term/Variable\ Symbol$$

where *Variable Symbol* is called the variable of the substitution component and *Term* is called the term of the substitution component. (Example 3.1.1)

Example 3.1.1 *foo(Z)/X is a substitution component whose variable is X and whose term is foo(Z). This substitution component denotes the action of substituting foo(Z) for all occurrence of X.*

3.1.2 Substitution

Substitution is a finite but maybe empty set of substitution components none of the variables of which are the same. A substitution has the form

$$\{T_1/V_1, T_2/V_2 \dots T_n/V_n\}$$

where $0 \leq n < \infty$ and if $i \neq j$ then $V_i \neq V_j$. Any lower case Greek letter can be used to denote a substitution and specifically ϵ denotes empty substitution.[1] Applying a Substitution σ to a Term T is done by applying all the substitution components to T, this action is denoted by $T\sigma$. (Example 3.1.2)

Example 3.1.2 *σ ={ Y/X , foo/Z , bar(R)/W} is a substitution with three components: Y/X , foo/Z and bar(R)/W. Applying σ to term g(X , Z , W) yields g(Y , foo , bar(R)).*

3.1.3 Substitution composition

If $\theta = \{T_1/V_1, \ldots, T_k/V_k\}$ and λ are any two substitutions, then the set $\theta' \cup \lambda'$, where λ' is the set of all components of λ whose variables are not among $V_1, \ldots, V_k$, and θ' is the set of all components $T_i\lambda/V_i \quad 1 \leq i \leq k$, such that $T_i\lambda$ is different from V_i, is called the composition of θ and λ, and is denoted by $\theta\lambda$. [1] (Example 3.1.3)

Example 3.1.3 *Let*

$$\theta = \{f(X)/X, Z/Y, g(Y)/W\}$$

and

$$\lambda = \{X/U, a/X, b/Y\}$$

so

$$\lambda' = \{X/U\}$$

$$\theta' = \{f(a)/X, Z/Y, g(b)/W\}$$

and

$$\theta\lambda = \theta' \cup \lambda' = \{f(a)/X, Z/Y, g(b)/W, X/U\}$$

3.1.4 Occurs check

A Variable Symbol occurs in a Function if you can see that Variable Symbol in the Function. A Variable Symbol cannot be unified with a Function if the argument list of the Function has any occurrence of the Variable Symbol. [3, 4, 1, 2] (Example 3.1.4)

Example 3.1.4 *Variable symbol X occurs in Function f(X). They can't be unified because substituting any thing other than f(X) for X can't make the two terms same. But if we substitute f(X) for X then the two terms would be unified under infinite term f(f(f(...))), which is not desired in the context that we want the term resulted from unification be finite.*

The mechanism for finding out whether a Variable Symbol occurs in a Function is called *occurs check*, which is required in both Robinson's Algorithm [3, 1] and Martelli's Algorithm [4], which we will see in later chapter.

3.1.5 Robinson's algorithm

Robinson's algorithm presented in this section is for unification of a pair of first order logic terms. To simplify manipulation of first order logic terms, in particular Functions, we represent Functions as lists. [3].

	Predicate Calculus Syntax	List Syntax
1	P(a,b)	(P a b)
2	p(f(a),g(X,Y))	(p (f a) (g X Y))

Table 3.1: Example: representing terms as lists. In row 2, sub-lists, such as (f a) and (g X Y) has equal membership to p as a member of their immediate super-list (p (f a) (g X Y))which has three members.

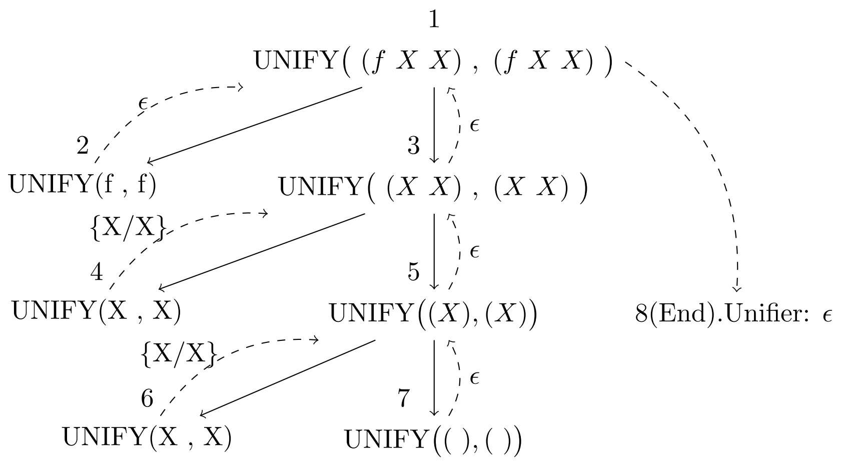

Figure 3.1: The unification problem is f (X , X) = f (X , X);list representation of terms is (f X X) = (f X X). The function UNIFY calls itself recursively for 7 times.In each function call, UNIFY first tries to unify the head of the input list by initiating one recursive call, then apply the unifier to the rest of the list(tail), and then initiate another recursive call to unify the tail. The return value from 4th and 6th call are both {X/X} which is the trivial case covered by the first "else if" in Algorithm 3.1 . Composition of {X/X}and ϵ is ϵ according to Section 3.1.3

Representing Functions as lists

The Function is simplified to be a list of symbols, starting from the function symbol, followed by the arguments of the Function. Lists are delimited by a pair of parenthesis and list elements are separated by spaces. When any arguments are also Functions, they are represented as sub-lists. (Table 3.1, from [3],page 64)

Robinson's unification algorithm

The algorithm is shown as Algorithm 3.1 on page 10 of this thesis, where the pseudo-code presented is transcribed from [3], page 65.

3.1.6 Examples for Robinson's algorithm

Examples of running Algorithm 3.1 are given in Figure 3.1 on page 9 and Figure 3.2 on page 11.

Algorithm 3.1 Robinson's unification algorithm: unify two terms E1 and E2 where both terms are represented as lists

```
Require: Both E1 and E2 are well formed.
  function UNIFY(E1,E2)
      if both E1 and E2 are constants or the empty list then
          if E1 = E2 then return {}
          else return Fail
          end if
      else if E1 is a variable then
          if E1 occurs in E2 then return Fail
          else return {E2/E1}
          end if
      else if E2 is a variable then
          if E2 occurs in E1 then return Fail
          else return {E1/E2}
          end if
      else
          HE1:= first element of E1
          HE2:= first element of E2
          SUBS1:= UNIFY(HE1,HE2)
          if SUBS1= Fail then return Fail
          end if
          TE1:= APPLY(SUBS1,rest of E1)
          TE2:= APPLY(SUBS1,rest of E2)
          SUBS2:= UNIFY(TE1,TE2);
          if SUBS2=Fail then return Fail
          else return  COMPOSITION(SUBS1,SUBS2)
          end if
      end if
  end function
```

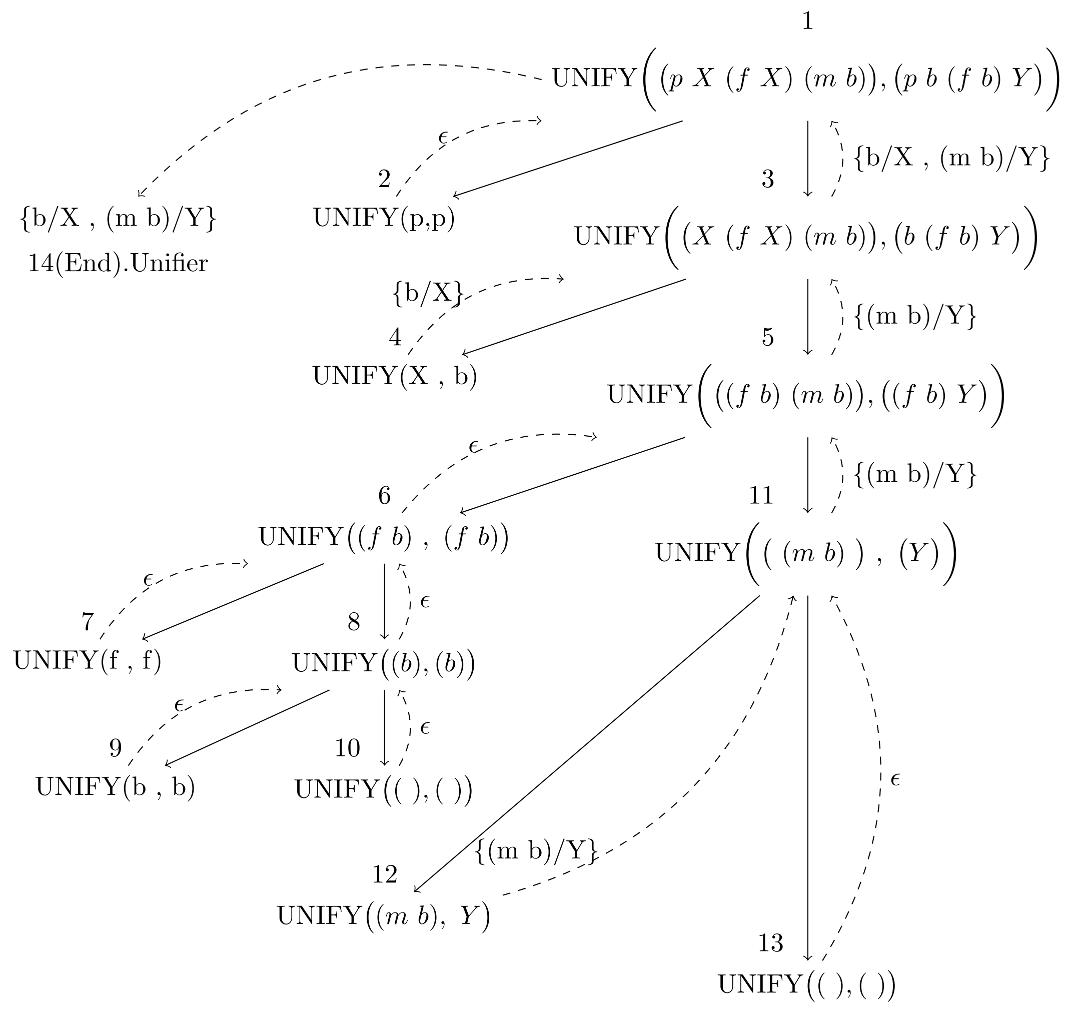

Figure 3.2: Unification Problem: p (X , f (X) , m (b)) = p (b , f (b) , Y). This example is adapted from Section 2.3.3 of [3].

```
data Term = Constant String
          | Variable String
          | Function String Int [Term]
       deriving (Read, Eq)
```

Figure 3.3: Haskell data type for Term; there is no dedicated mechanism to ensure the value in the String fields comply with syntax as defined in section 2.1

3.1.7 Discussion on Algorithm 3.1

As shown by Figure 3.1, the algorithm allows substitution component of the form V/V be generated, which doesn't agree with the definition of substitution component. However, such trivial case can be eliminated during composition of substitution.

UNIFY function is not strictly typed: sometimes it take a list as its input type, the other time it takes member of the list as input, which may not be lists, unless the list and its non-list members are all declared as belonging to the same data type which can be called "Term".

The situation of unifying a constant 'f' and a function '(f a)' is not well handled. If the constant 'f' was not wrapped in '()' as '(f)' when being fed into the UNIFY, the mechanism of getting first element of list would complain since the input was not a list; otherwise, when the constant 'f' was wrapped in a list as '(f)', the first function call would be *UNIFY ((f) , (f a))*; the second call would be *UNIFY (f , f)* and the third function call would be *UNIFY (( ) , (a))* where the empty list would challenge the mechanism for getting first element of the list.How to deal with such challenge was not addressed by the pseudo-code in Algorithm 3.1, by, for example, returning Fail when cannot get the first element from an empty list or non-list.

3.2 Haskell Implementation of Robinson's Algorithm

Implementing the algorithm started from considering what shall be its input and output. We first look at possible input to the algorithm, then we see the considerations over the data type for terms, the occurs check, and the data type for substitution component and substitution, which shall be the output. The ideas backing the Haskell implementation of the Algorithm 3.1 and substitution application and composition related concern come at last.

3.2.1 Enumeration of all possible input cases

The unification algorithm is supposed to unify two terms; as defined in Chapter 2, a term has only three forms, so there are limited($3 \times 3 = 9$) input cases for an unification algorithm, as shown in Table 3.2.

3.2.2 The data type for terms

The data type is defined in Figure 3.3 on page 12.

	Left member of FOTE=Right member of FOTE
1	Constant Symbol=Constant Symbol
2	Variable Symbol=Variable Symbol
3	Function=Function
4	Constant Symbol=Variable Symbol
5	Variable Symbol=Constant Symbol
6	Constant Symbol=Function
7	Function=Constant Symbol
8	Variable Symbol=Function
9	Function=Variable Symbol

Table 3.2: Input cases for an unification algorithm that unifies two terms

I added arity number for a function since Haskell list doesn't know its own length.

Functions with arity 0 is treated as a constant, but I thought there is still need to distinguish a 0-arity function from a constant of the same name since a function can map to another term while a constant can't. So I used a value constructor "Constant" to explicitly construct constants rather than using 0-arity functions to represent constants implicitly.

The above consideration occurred after I implemented the algorithm. At the time when I defined the data type before implementing any algorithm, I distinguished constant and 0-arity function because I was yet impressed by the idea that they are considered equal by some people [4, 1, 8], and it was straightforward to use three value constructors rather than two, in order to comply with the syntax.

As mentioned in the caption of Figure 3.3, there is no syntax check to ensure name validity of symbols, so I rely on users to provide valid names and algorithm was designed with the assumption that all terms has valid names. There are multiple reasons for doing so:

1. At the time of defining the data type, I didn't know how to introduce a name check in Haskell; when I discussed this with my academic advisers, they shown that name check is not needed without giving detailed explanation.

2. After I implemented the algorithm , I realised that the valid name is indeed immaterial for the correct running of the algorithm, since:

   (a) The lexical structure of the name doesn't determine the kind of term represented by the name. Each value of type "Term" has its own value constructor. To know whether it is dealing with a constant, a variable, or a function, the algorithm only needs to read the value constructors without considering whether the name starts with a capital letter or not.

   (b) After it knows the kind of term it is dealing with, the only cases in which the algorithm cares about names are when it needs to unify two constants or two variables, so it checks whether names of interest are equal. Such equality check doesn't rely on the lexical structure of the names neither.

3.2.3 Implementing occurs check

Occurs check cares about existence of a Variable Symbol in a Function. A Variable Symbol is simple, we need to observe the property of the Function in order to come up with an idea on how to detect existence of a Variable Symbol in it.

Properties of the Function against which a Variable Symbol is checked for occurrence

There may be Constant Symbols, Variable Symbols and Functions appear in a Function's argument list. We can compare a Variable Symbol of interest with all arguments in the list one by one. If a Variable Symbol occurs in the arguments list of a Function *f*, it can be an argument of f or be an argument of another function g which is nested in *f*.

Rules of Occurrence

We say for all Variable Symbols X:

1. X doesn't occur at any Constant Symbol;
2. X doesn't occur at any Variable Symbol whose name is not X;
3. X occurs at all Variable Symbols whose names are also X;
4. X occurs in a Function if and only if it occurs at any argument of this Function, judged using rule 1,2,3 and 4.

This is the recursion for occurs check.

3.2.4 Data type for substitution and its component

The data type of substitution component shall distinguish the variable of the substitution component from the term of the substitution component, both of which are of type `Term` as defined in Section 3.2.2, so the data type of the substitution component can be a tuple, which is a ordered pair of Terms, i.e.

```
type subsComponent = (Term , Term)
```

The left member represents the term of the substitution component, the right member represents the variable of the substitution component, so the right member of the tuple always has "Variable" as its value constructor. For example, (Constant "a" , Variable "X") stands for the substitution component "a/X".

The data type of substitution can be a list of substitution components, since a list can store multiple value of the same type, and it can be amended during composition of substitution, changing existing members as well as adding new members. Defining a recursive customised data type is not necessary at this stage since the list works satisfyingly and other existing data container, in particular, tuple, doesn't support size expansion so a tuple cannot deal with the need to add new substitution components into the substitution during substitution composition.

```
type Substitution = [(Term , Term)]
```

3.2.5 Implementing substitution application

We see the process of substitution application, from where we identify the unit action of substitution application.

Applying multiple substitution components to a list of terms

From the pseudo-code of Robinson's algorithm for Haskell implementation, we can notice that applying substitution only happens during unification of a pair of Functions and it involves applying a substitution to a list of terms, which is the argument list of the Functions.

A substitution is implemented as a list of substitution components; applying a substitution to a list of terms means applying all substitution components to every member of the list (Algorithm 3.2).

Algorithm 3.2 Two ways to apply substitution on a list of terms (with trivial difference)

```
                    ▷ Applying all substitution components to each term list member
for all Terms in the List of Terms, counter i do
    for all Substitution Components of the Substitution, counter j do
        Attempt applying Substitution Component j onto Term i.
    end for
end for
                    ▷ Applying each substitution component to all term list members
for all Substitution Components of the Substitution, counter i do
    for all Terms in the List of Terms, counter j do
        Attempt applying Substitution Component i onto Term j.
    end for
end for
```

Unit action: applying a substitution component onto a term

We saw, in Algorithm 3.2, that the unit action of applying multiple substitution components to a list of terms is to attempt applying one substitution component onto one term.A term could be a Constant Symbol, a Variable Symbol or a Function, so we would need to respond to one of these three situations when we attempt applying one substitution component onto one term, following Algorithm 3.3 on page 16.

To realize Algorithm 3.3 as a function, its input shall be a substitution component and a term onto which the substitution component will be attempted, its output shall be a term, which is the same as the input term if and only if the input term was not eligible (defined in Algorithm 3.3) for the substitution component, guaranteeing the output of this function hosts no occurrence of the variable of the input substitution component.

3.2.6 Implementing substitution composition

By inspecting the context in which composition of substitution happens, which is the "unifyTermList" function in Section 3.2.7 , we notice that the second

Algorithm 3.3 Attempting applying a substitution component onto a term

1. If the term is a Variable Symbol which is the same as the variable of the substitution component, then this term is eligible for the substitution component and we replace this term by the term of the substitution component.
2. If the term is a Variable Symbol which is different from the variable of the substitution component, then it is not eligible for this substitution component.
3. If the term is a Constant Symbol, then it is not eligible for this substitution component.
4. If the term is a Function, then we attempt the substitution component on all of its arguments, using the rules 1–4. This function is eligible for the substitution component if and only if any one of the Function's argument is eligible for the substitution component.

substitution to be composed with the first substitution is obtained after applying the first substitution to the rest of both term lists, so the variables of the first substitution do not occur in the second substitution, thus for implementation, the definition of substitution composition can be simplified as: if $\theta = \{T_1/V_1, \ldots, T_k/V_k\}$ and λ are any two substitutions, then the set $\theta' \cup \lambda$, where θ' is the set of all components $T_i\lambda/V_i \quad 1 \le i \le k$, is called the composition of θ and λ.

3.2.7 Pseudo-code of Robinson's algorithm suitable for Haskell implementation

Mutual recursion

A pair of mutually recursive functions are required to implement Robinson's unification algorithm in Haskell. The first one is "unifyTerms" which unifies a pair of FOT as its input and uses "unifyTermList" to unify the argument lists when the input is a pair of functions; the second is "unifyTermList" which unifies a pair of lists of FOTs as its input and serves as a driver for using "unifyTerms" to unify each pair of FOTs , and each pair of FOTs is unified after applying unifier of the previous pair of FOTs to the rest of the FOT lists.

There are several reason for the above structure arrangement.

1. When you need to unify a pair of functions, it is very natural for the raise of the need to unify the argument lists of the functions. However, Haskell is strongly typed, it is no longer possible to pass the argument lists back to "UnifyTerms" function as what was done in Algorithm 3.1 since input type of "UnifyTerms" is "Term" but the argument list have type "[Term]" (list of Term). So to unify the argument lists, another function is needed, and this function shall be defined outside the function "unifyTerms" and cannot be merged with "unifyTerms" to achieve a analogous structure as Algorithm 3.1.

2. Unification of a pair of lists of terms is not simply *mapping* the "unifyTerms" function on the term lists and compose the resulting list of unifiers of each pair of terms. In this way, the unification of each pair of terms are independent from the unification of other pairs of terms, which does not agree with Robinson's algorithm. (See Example 4.2.2 on page 27 for meaning of mapping) Instead, to unify a pair of list of terms,according to Robinson's algorithm, first unify the heads,then apply unifier to the tails, and obtain tail unifier by recursive function call, finally compose head unifier and tail unifier.This process is the key idea of Robinson's algorithm and it can be focused upon during programming since implementing this, is the sole task of "unifyTermList" and free from distraction from other details in "unifyTerms".

3. The spun out function for unification of a pair of lists of terms according to Robinson's algorithm can be used else where, such as in the random unifiable pair of terms generator.

When implementing this algorithm, I used Haskell's multi-clause function definition feature. For each possible input case as listed in Table 3.2, I introduced a separate clause. This is shown in the source code but not in the thesis. The pseudo-code of Robinson's algorithm for Haskell implementation is as follows.

```
function UnifyTerms($FOT_1$,$FOT_2$)
    if Both $FOT_1$ and $FOT_2$ are constant then
        if $FOT_1$ and $FOT_2$ have the same name then
            return $\epsilon$
        else
            return Fail
        end if
    else if Both $FOT_1$ and $FOT_2$ are variable then
        if $FOT_1$ and $FOT_2$ have the same name then
            return $\epsilon$
        else
            return $\{FOT_2/FOT_1\}$
        end if
    else if $FOT_1$ is constant and $FOT_2$ is variable then
        return $\{FOT_1/FOT_2\}$
    else if $FOT_1$ is variable and $FOT_2$ is constant then
        return $\{FOT_2/FOT_1\}$
    else if $FOT_1$ is constant and $FOT_2$ is function then
        if $FOT_1$ and $FOT_2$ have distinct names then
            return Fail
        else if $FOT_2$ has non-empty argument list then
            return Fail
        else
            return $\epsilon$
        end if
    else if $FOT_1$ is function and $FOT_2$ is constant then
        if $FOT_1$ and $FOT_2$ have distinct names then
            return Fail
        else if $FOT_1$ has non-empty argument list then
            return Fail
        else
            return $\epsilon$
        end if
    else if $FOT_1$ is function and $FOT_2$ is variable then
        if $FOT_2$ occurs in $FOT_1$ then
            return Fail
        else
            return $\{FOT_1/FOT_2\}$
        end if
    else if $FOT_1$ is variable and $FOT_2$ is function then
        if $FOT_1$ occurs in $FOT_2$ then
            return Fail
        else
            return $\{FOT_2/FOT_1\}$
        end if
    else                                    ▷ Both $FOT_1$ and $FOT_2$ are functions
        if $FOT_1$ and $FOT_2$ have distinct names then
            return Fail
        else if $FOT_1$ and $FOT_2$ have distinct arity numbers then
            return Fail
        else
            if Both $FOT_1$ and $FOT_2$ have empty argument list then
                return $\epsilon$
            else if Both $FOT_1$ and $FOT_2$ have singleton argument list then
                return UnifyTerms(argument of $FOT_1$,argument of $FOT_2$)
            else
                return unifyTermList(arguments of $FOT_1$ , arguments of $FOT_2$)
            end if
        end if
    end if
end function
```

```
function unifyTermList($List_1$ of FOTs , $List_2$ of FOTs)
Require: $List_1$ and $List_2$ have the same length and $2 \leq length \leq n$
    SUBS1 = UnifyTerms(head of $List_1$,head of $List_2$)
    if SUBS1 equals to Fail then
        return Fail
    end if
    T1 = applySubs(SUBS1,tail of $List_1$)
    T2 = applySubs(SUBS1,tail of $List_2$)
    SUBS2 = unifyTermList(T1 , T2)
    SUBS3 = comboSubs(SUBS1,SUBS2)
    return SUBS3          ▷ SUBS3 equals to Fail when SUBS2 equals to Fail
end function
```

3.3 Conclusion

This chapter introduces Robinson's unification algorithm and the considerations made when trying to implement this algorithm in Haskell language, which has its unique feature such as multi-clause function definition, value constructor and strict type. The restructuring of the algorithm, while keeping its key idea intact, is needed to suit the algorithm into the programming language as well as to exploit the language features.

Enumeration was used to study the input cases for Robinson's unification algorithm and it served as the framework of the algorithm implementation.

The mutually recursive structure of the implementation was also found in OCaml implementation of Robinson's algorithm in [8]. At first I thought it was a coincidence but further thought involving the feature of both languages yielded that such structural similarity shall be the necessary result of the fact that OCaml and Haskell are both strongly typed functional programming language—"essential" similarity of the languages caused structural similarity of the code when both languages are used to address the same problem.

Chapter 4

Martelli's Algorithm and its Haskell Implementation

In [4], Martelli and Montanari gave a formulation of the unification process that "has gained wide currency as a formalism for representing unification algorithms" [7]. In this chapter we first introduce Martelli's algorithm, particularly the representation of unification problems as equation sets and the four transforms of the unification process. Then in Section 4.2 we see the discussion on key considerations made for implementing Martelli's algorithm in Haskell.

4.1 Martelli's Unification Algorithm

We start from introducing the equation representation of unification problems and learning the concept of "solved form" of an equation set. Then we see the four transforms used in Martelli's algorithm and the theorem on the termination of the algorithm. The section ends up with an example of running Martelli's algorithm.

4.1.1 Input to Martelli's Algorithm

The algorithm works on a set of pairs of terms

The algorithm is supposed to unify a pair of first order terms (FOTs); since the transformations on the original pair of FOTs will make more pairs of FOTs, (as you will see later on) the algorithm is practically applicable to a set of pairs of FOTs. Strictly, the "set" shall be a *multi-set*, where duplication of members is allowed. So in this Chapter (Chapter 4) the "set" and "multi-set" are used interchangeably.

Pairs of terms and unifiers are represented as equations

An equation is an written graph where a equal sign is placed in between two items. The unification of two terms can be regarded as solving an equation where there is a term on each side of the equal sign. Such equation can be either solvable or unsolvable. (Example 4.1.1)

Example 4.1.1 *Unification of a pair of FOTs:*

$$\begin{cases} f\ (X\ ,\ a) \\ f\ (b\ ,\ Y) \end{cases}$$

can be written in the equation form: "f (X , a) = f (b , Y)"; the unifier {b/X , a/Y} can also be written as a set of equations, being regarded as the solution to the unification problem "f (X , a) = f (b , Y)":

$$\begin{cases} X{=}b \\ Y{=}a \end{cases}$$

Example input to Martelli's Algorithm

Input to Martelli's algorithm is a set of pairs of FOTs and any pair of FOTs is represented as an equation, so the input to Martelli's algorithm is a set of equations. Example 4.1.2, 4.1.3, 4.1.4 and 4.1.5 show possible input for Martelli's algorithm.

Example 4.1.2

$$p\big(X, f(X), m(b)\big) = p\big(b\ , f(b),\ Y\big)$$

The input is a singleton set of equation.

Example 4.1.3

$$\begin{cases} X{=}b \\ f(X){=}f(b) \\ m(b){=}Y \end{cases}$$

The input is a set of three equations, each can be regarded a unification problem; and Martelli's algorithm is supposed to find a solution that can solve all of these three equations at the same time.

Example 4.1.4

$$\begin{cases} X{=}b \\ m(b){=}Y \end{cases}$$

This set of two equations has obvious solution.

Example 4.1.5

$$\begin{cases} X{=}b \\ Y{=}m(b) \end{cases}$$

This set of two equations has obvious solution. This set looks similar to the one in Example 4.1.4 but Y is put on the left side of the equal sign to emphasis the value assignment to a variable. Not just Y is put on the left side, all left members of the equations in this set are variables and for any left member, it occurs only once in the equation set.

4.1.2 The solved form of an equation set

The *solved form* of an equation set is used to represent the unifier for that equation set, if any. Any equation set may *have* a solved form, and for any equation set, *being in* solved form means: for all equations in the set, the left member is a variable that occurs only once in the set.

(Example 4.1.6/ 4.1.7/ 4.1.8/ 4.1.9)

Example 4.1.6 *The FOTE set in Example 4.1.5 is in solved form and it is the solved form of the FOTE set in Example 4.1.2.*

Example 4.1.7

$$\begin{cases} X_1{=}X_2 \\ X_3{=}X_2 \\ X_4{=}a \\ X_5{=}f(X_2,a) \end{cases}$$

This equation set is in solved form according to the definition. Although X_2 occurs three times, it doesn't matter since X_2 is not left member of any equation. All left members must be variables but not all variables must be left members.

Example 4.1.8

$$\begin{cases} X_1{=}X_1 \\ X_3{=}X_2 \\ X_4{=}a \\ X_5{=}f(X_2,a) \end{cases}$$

This equation set is not in solved form because X_1 occurs twice in the set.

Example 4.1.9 *The FOTE set in Example 4.1.4 is* not *in solved form because not all left members of FOTEs are variable.*

All unifiers, when written as FOTE set, shall be in solved form. Thus the process of obtaining the unifier of an unification problem can be regarded as *transforming* the FOTE representation of the problem into its solved form. We see the ways to transform in Section 4.1.3 .

4.1.3 Four transformations in Martelli's algorithm

There are four transforms defined by Martelli and Montanari in order to transform any FOTE set into its solved form. In this section we introduce them.

Trivial Removal

Find any FOTE of the form

Variable Symbol = Variable Symbol

where the two members of the FOTE are the same Variable Symbol, remove such a FOTE (trivial FOTE) from the FOTE set.

Orientation

Find any FOTE of the form

Function = Variable Symbol *or* Constant Symbol = Variable Symbol

and swap the two members of such a FOTE.

Term Reduction

Find any FOTE of the form

Function = Function

where by syntax in Section 2.1 such FOTE must have an internal structure[1] as:

Function Symbol $\{FOT_1, FOT_2 \ldots FOT_n\}$ =
Function Symbol $\{\mathcal{FOT}_1, \mathcal{FOT}_2 \ldots \mathcal{FOT}_m\}$

If the two Function Symbols are different or $\mathbf{m \neq n}$, return Fail.

If the two Function Symbols are the same and $\mathbf{m = n}$, amend the FOTE set by removing such a FOTE from it and adding into it the following FOTEs:

$$\begin{aligned} FOT_1 &= \mathcal{FOT}_1 \\ FOT_2 &= \mathcal{FOT}_2 \\ &\ldots \\ FOT_n &= \mathcal{FOT}_n \end{aligned}$$

Alternatively, find any FOTE of the form

Constant Symbol = Constant Symbol

If the two Constant Symbols are the same , remove such a FOTE from the FOTE set.

If the two Constant Symbols are distinct , return Fail.

Alternatively, find any FOTE of the form

Constant Symbol=Function *or*
Function=Constant symbol

If they have the same name, and Function has no argument , remove such a FOTE from the FOTE set.

If they have distinct name, or Function has argument(s) , return Fail.

[1]Note the "FOT" in two members of the FOTE are in different font, this implies that the exact content of corresponding FOTs— italic FOT_i and calligraphic $\mathcal{FOT}_i$ where $i \leq min\{n, m\}$ are independent from each other.

Variable Elimination

Select any FOTE of the form[2]:

form 1: Variable Symbol=$\mathcal{V}$ariable $\mathcal{S}$ymbol
or form 2: Variable Symbol=Function
or form 3: Variable Symbol=Constant Symbol

where the Variable Symbol on the left side of the FOTE occurs *more* than once in the FOTE set.

If the selected FOTE has form 2 and the Variable Symbol occurs in the Function, then return Fail.

Otherwise amend the FOTE set by substituting $\mathcal{V}$ariable $\mathcal{S}$ymbol or Function or Constant Symbol (determined by the form 1,2 or 3 of selected FOTE) for all occurrence of the Variable Symbol (determined by the selected FOTE) in the FOTE set, but keep the selected FOTE intact. When there are more than one copies of the selected FOTE in the FOTE set, keep only one copy intact.

4.1.4 Theorem on termination of Martelli's algorithm

The four transforms are supposed to be applied when applicable and it was proved in [4] that application of these four transforms will finally terminate: if terminated with Fail then there is no solved form of the original equation set, which is the algorithm input; otherwise the FOTE set onto which none of four transforms can be further applied is in solved form and it is the solved form of the original FOTE set.

4.1.5 An Example for Martelli's algorithm

Example 4.1.10 *Solve the following unification problem using Martelli's unification algorithm:*

$$p\big(X, f(X), m(b), Z\big) = p\big(b\ , f(b),\ Y, Z\big)$$

.

[2]In form 1, the initial capital letters of the left member have Roman font while the initial capital letters of the right member have calligraphic font, this means the two members are different.

$$\mathit{1}\quad p\big(X, f(X), m(b), Z\big) = p\big(b\ , f(b),\ Y, Z\big)$$

$$\Big\downarrow\ \textit{Term Reduction on } p$$

$$\mathit{2}\ \begin{cases} X=b \\ f(X)=f(b) \\ m(b)=Y \\ Z=Z \end{cases}$$

$$\Big\downarrow\ \textit{Variable Elimination on } X$$

$$\mathit{3}\ \begin{cases} X=b \\ f(b)=f(b) \\ m(b)=Y \\ Z=Z \end{cases}$$

$$\Big\downarrow\ \textit{Term Reduction on } f$$

$$\mathit{4}\ \begin{cases} X=b \\ b=b \\ m(b)=Y \\ Z=Z \end{cases}$$

$$\Big\downarrow\ \textit{Term Reduction on } b$$

$$\mathit{5}\ \begin{cases} X=b \\ m(b)=Y \\ Z=Z \end{cases}$$

$$\Big\downarrow\ \begin{array}{l}\textit{Orientation of } Y \textit{ then} \\ \textit{Trivial Removal on } Z\end{array}$$

$$\mathit{6}\ \begin{cases} X=b \\ Y=m(b) \end{cases}\quad \textit{Solved Form}$$

4.2 Haskell Implementation of Martelli's Algorithm

We are going to discuss about the Haskell representation of the FOTE set, implementation of the transforms and termination check, which are components of the algorithm. Finally we see how these components were integrated to perform the unification.

The order in which the subsections are arranged also reveals the order in which the relevant considerations were made during the implementation. As you will see in this section, implementation of termination check was make after implementing the four transforms, since termination is determined during performing the transforms.

4.2.1 Representation of FOTE set

The FOTE is a pair of terms separated by an equal sign. In order to represent the FOTE, the equal sign does not need to present in the data structure, since it is the common part of all FOTEs. Only the pair of terms and their left-right order in the FOTE shall be stored. All of these can be achieved by using a Haskell tuple, which is an ordered pair of value of any type. So the FOTE was defined as:

```
type FOTE = (Term, Term)
```

The multi-set nature of the FOTE set can be represented by a Haskell list, which contains the same type of value but allows value duplication. A list also supports amendment of itself during transformations. So the FOTE set was defined as:

```
type FOTEset = [FOTE]
```

4.2.2 Implementing Trivial Removal: filtering the equation set against triviality

Filtering against triviality

Haskell provides a list manipulator called "filter" to make a sub-list by selecting from a list for elements that satisfy a predicate. (Example 4.2.1)

Example 4.2.1
```
ghci> filter odd [1,2,3,4,5,6]
[1,3,5]
ghci> filter (<0) [1 , -1 , 2 , -2]
[-1,-2]
ghci> filter even [1,3,5]
[]
ghci> :type filter
filter :: (a -> Bool) -> [a] -> [a]
```

The FOTE set is represented as a list of FOTEs so the trivial removal can be regarded as filtering the list of FOTEs and keeping all but the FOTEs that has the trivial form. So this process is somewhat looks like:

`filter notTrivial` [$\texttt{FOTE}_1$,$\texttt{FOTE}_2$...$\texttt{FOTE}_n$]

Triviality Check

The `notTrivial` function shall work as a predicate having type `notTrivial :: FOTE -> Bool`. So it shall take a tuple of terms and judge whether the FOTE represented by this tuple is in trivial form. This work involves telling the kind of terms this FOTE has and if both are Variable Symbols, then further compare their names, same name means trivial otherwise means not trivial; if at least one term is not a Variable Symbol then this FOTE must not be in an trivial form. This process of judging the triviality can be presented as pseudo-code in Algorithm 4.1.

Algorithm 4.1 tell whether a FOTE has or does not have a trivial form

```
function TRIVIAL($term_1, term_2$)
    if Both $term_1$ and $term_2$ are Variable Symbol and they have the same name then
        return True
    else
        return False
    end if
end function
function NOTTRIVIAL($term_1, term_2$)
    return $\neg$ TRIVIAL($term_1, term_2$)        ▷ $\neg$ means logical negation
end function
```

4.2.3 Implementing Orientation: mapping a conditional swap function onto the equation set

Overview

The unit operation for orientation is to inspect a FOTE for its eligibility for orientation, and if it was eligible, then perform orientation; otherwise do nothing to it. So such unit operation for orientation is a conditional swap of the left and right member in the FOTE: when the swap condition is met, i.e. the FOTE shall be oriented, swap; otherwise, not swap. The orientation transform on the FOTE set can be regarded as mapping such unit conditional swap on all FOTEs in the FOTEset, which is a list of FOTEs. To do this we need:

1. a mapping mechanism that can map any function onto a list of values, so the function can be applied to every individual value of the list.
2. a conditional swap mechanism that takes a FOTE and swap it when appropriate. This mechanism is to be mapped onto the list of FOTEs. And to build such mechanism, it requires:
    (a) a swap mechanism that can perform swap on a FOTE.
    (b) a predicate that check whether the swap condition is met.

Mapping

Haskell has already provided a mapping mechanism, which takes a function and a list of values and applies the function to each value in the list separately. (Example 4.2.2)

Example 4.2.2
```
ghci> map (+1) [1,2,3,4]
[2,3,4,5]
ghci> map (<5) [1,4,8]
[True,True,False]
ghci> :type map
map :: (a -> b) -> [a] -> [b]
```

Swapping

The swap mechanism is also provided by Haskell in its standard library Data.Tuple. (Example 4.2.3)

Example 4.2.3
```
ghci> :module +Data.Tuple
ghci> :type swap
swap :: (a, b) -> (b, a)
ghci> swap ("Hi",True)
(True,"Hi")
ghci> swap ("Adam","Smith")
("Smith","Adam")
```

Condition of swap

The predicate for swap condition checking and the conditional swap mechanism is defined in Algorithm 4.2 on page 29, where the comprehensive mechanism for performing orientation was provided by the end.

4.2.4 Implementing Term Reduction: removing equations or zipping argument lists after Fail check, make sure the return type is correct

Checking for Term Reduction Fail

Term Reduction can be attempted on *all* FOTEs in the FOTE set: if the FOTE has the form onto which term reduction is applicable, perform term reduction; otherwise perform nothing. Since Fail maybe returned during term reduction, it would be useful to check all FOTEs in the set before any attempt of term reduction to ensure all of them would not cause Fail be returned during term reduction, otherwise the effort made on reducing terms would turned out to be a waste of time if later on a term that can cause Fail during term reduction was discovered, and this term could have been discovered before the term reduction that has been attempted and some terms reduced. This is demonstrated in Example 4.2.4 on page 28.

Example 4.2.4 *Say term reduction is attempted along term 1, 2, 3, 4, 5. Only term 5 causes Fail be returned for term reduction, before which term reduction has been attempted on term 1,2 and performed on term 3,4. These effort turned out to be wasted when term 5 is discovered as cannot be reduced, since such a discovery doesn't rely on all previous attempt of term reduction. If before any term reduction was performed, all terms were inspected for whether any of them directly causes term reduction fail, then attempt of term reduction would not start on term 1–4 and the algorithm can probably terminate earlier without wasting our effort.*

1	*X=a*
2	*f (X)=Y*
3	*g (Z)=g (a)*
4	*h (a, b, c)=h (X, a, c)*
5	*h (X, d, e)=k (m)*

Algorithm 4.2 The predicate for swap condition check,the conditional swap mechanism and the comprehensive orientation function.

```
function SwapConditionMet(($term_1, term_2$))
            ▷ This is the predicate for swap condition check. ($term_1, term_2$) represents $term_1 = term_2$

    if $term_2$ is a Variable Symbol and $term_1$ is not a Variable Symbol then
        return True
    else
        return False
    end if
end function

function SwapIf(condition,($term_1, term_2$))
            ▷ This is the framework of the conditional swap mechanism. To use it a predicate shall be provided. Haskell fascilitates passing the function "condition" as an input argument to the function "swap if"

    if condition(($term_1, term_2$)) returns True then
        return ($term_2, term_1$)                    ▷ Swap
    else
        return ($term_1, term_2$)                    ▷ Do nothing
    end if
end function

            ▷ Finally, provide a predicate to the framework of conditional swap mechanism to get the finished conditional swap mechanism and map this mechnism onto a list of FOTEs
```

```
orientation :: FOTEset -> FOTEset
orientation = map (swapIf swapConditionMet)
```

Thus the first step in term reduction implementation is developing the mechanism to look through all FOTEs in the set to tell if any of them could cause Fail be returned during term reduction. There are two sub-mechanisms required to build this *Term Reduction Fail Case Detection* mechanism:

1. a mechanism to decide whether an individual term could cause Fail be returned during term reduction. We can call such a term a "single fail" and this mechanism "Single Fail Check"
2. a mechanism to decide existence of a single fail in the whole FOTE set. This mechanism basically drives the application of the "Single Fail Check" across the whole set and return a True value when it meets the first "single fail".

"Single fail Check" was implemented as Algorithm 4.3 on page 30.

Algorithm 4.3 The mechanism to decide whether an individual term could cause Fail be returned during term reduction

```
function singleFail(($term1, term2$))
    if $term1, term2$ are both Constant Symbols then
        if $term1, term2$ has distinct names then
            return True
        else                                  ▷ $term1, term2$ has the same name
            return False
        end if
    else if $term_1$ is Constant Symbol and $term_2$ is Function then
        if $term_1, term_2$ have distinct name or $term_2$ has argument(s) then
            return True
        else ▷ $term_1, term_2$ have the same name and $term_2$ has no argument
            return False
        end if
    else if $term_2$ is Constant Symbol and $term_1$ is Function then
        if $term_1, term_2$ have distinct name or $term_1$ has argument(s) then
            return True
        else ▷ $term_1, term_2$ have the same name and $term_1$ has no argument
            return False
        end if
    else if $term_1, term_2$ are both Functions then
        if $term_1, term_2$ have distinct name or distinct arity then
            return True
        else              ▷ $term_1, term_2$ have the same name and the same arity
            return False
        end if
    else ▷ ($term_1, term_2$) represents a FOTE for which term reduction is not applicable
        return False
    end if
end function
```

The mechanism for driving the "Single Fail Check" can be achieved using the Haskell list manipulator "`any`" which tells whether in a list there exist a member that satisfies a given predicate. (Example 4.2.5)

Example 4.2.5
```
ghci> :type any
any :: (a -> Bool) -> [a] -> Bool
ghci> any odd [2,4,6,8]
False
ghci> any odd [2,4,1,8]
True
```

The "Term Reduction Fail Case Detection" mechanism can then be achieved as in Algorithm 4.4 on page 31.

Algorithm 4.4 term reduction fail case detection: driving the single fail check by "any"

```
termReduceFail ::  FOTEset -> Bool

-- check-of-fail for term reduction;
-- point-free definition of function.
-- True case:
--      there is any FOTE makes singleFail return True
-- False case:
--      no FOTE makes singleFail return True

termReduceFail = any singleFail
```

The return type of the Term Reduction function

After the FOTE set passes the "Term Reduction Fail Case Detection", it is sure that in current FOTE set, there is no "single fail", even though some "single fail"s might be released after term reduction is performed (Example 4.2.6).

Example 4.2.6 *For example of the situation where there is no "single fail" in the FOTE set but some "single fail"s are released after term reduction, assume* term 5 *doesn't exist in Example 4.2.4, then this FOTE set can pass the "Term Reduction Fail Case Detection", but a "single fail" $b = a$ will be released after term reduction is performed on* term 4. *Martelli's algorithm cannot foresee such hidden "single fail" before they are released.*

Anyway, after the FOTE set passes the "Term Reduction Fail Case Detection",term reduction can be attempted on *all* FOTEs in the set. Since none of the FOTEs would cause Fail be returned, according to term reduction rule, the result of any attempt on any FOTE will be:

1. Keeping the FOTE intact, since the FOTE doesn't has the form onto which term reduction is applicable. Or
2. Removing the FOTE, nothing more to be done. Or
3. Removing the FOTE, adding more FOTEs into the set.

However, when the FOTE is kept intact, the return type of the attempt is `FOTE`; when more FOTEs shall be added into the set, the return type of the attempt is `[FOTE]`. The original FOTE set has type `[FOTE]`, attempting term reduction on its members will cause some of its new members have type `[FOTE]`, the other members have type `FOTE`, such type diversity is not allowed within a Haskell list. So the return type of all attempts of term reduction on all terms, whether the term can be reduced or not, giving that the set have passed the "Term Reduction Fail Case Detection" so none of the FOTE will cause Fail be returned, shall be made the same: `[FOTE]`. If any FOTE shall be kept intact, then wrap it in a pair if brackets to make a singleton list. If the FOTE shall be removed, the return type is `[]` which can also be regarded as `[FOTE]`. In other cases the return type is fine. Thus after term reduction, the type of the resulting FOTE set would be `[[FOTE]]`, and it can be converted into having type `[FOTE]` by using a sub-list concatenation function provided by Haskell: `concat` (Example 4.2.7).

Example 4.2.7
```
ghci> concat [[],[1],[2,3]]
[1,2,3]
```

Zipping function argument lists

When both sides of the FOTE to be reduced are Functions, and the Functions has non-empty argument list, the way to make new FOTE set using their corresponding arguments is made possible by the somewhat coincidence of four pieces of facts:

1. The data structure of FOTE is a tuple of Terms: `(Term , Term)`. For instance, (t, τ) represents $t = \tau$.
2. The data structure of FOTE set is a list of tuple of Terms:`[(Term , Term)]`. For instance:
$$[(t_1, \tau_1), (t_2, \tau_2), (t_3, \tau_3), (t_4, \tau_4), (t_5, \tau_5)]$$
3. The data structure of the Function argument list is a list of terms: `[Term]`. For instance: function f on left side of the FOTE has argument list
$$[t_1, t_2, t_3, t_4, t_5]$$
function f on right side of the FOTE has argument list
$$[\tau_1, \tau_2, \tau_3, \tau_4, \tau_5]$$
4. There is a Haskell function can "zip" two lists into one list of tuples, each tuple gets its members from corresponding members of the lists. (Example 4.2.8)

Example 4.2.8
```
ghci> zip [1,2,3] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]
ghci> zip ['a','b','c'] [1,2,3]
[('a',1),('b',2),('c',3)]
ghci> :type zip
zip :: [a] -> [b] -> [(a, b)]
```

Thus using the "zip" function, the two lists of arguments of functions to be term reduced can be turned into a new FOTE set:

$$\begin{tikzcd} {[t_1,t_2,t_3,t_4,t_5]} \\ {[\tau_1,\tau_2,\tau_3,\tau_4,\tau_5]} \arrow[d, "\text{zip}"] \\ {[(t_1,\tau_1),(t_2,\tau_2),(t_3,\tau_3),(t_4,\tau_4),(t_5,\tau_5)]} \end{tikzcd}$$

4.2.5 Implementing Variable Elimination

The "good form"

In section 4.1.3 we defined the form that a FOTE shall have in order to be eligible for variable elimination We call these forms "good form", which you will see later in Figure 4.1 on page 33 that plans our discussion on implementation of variable elimination.

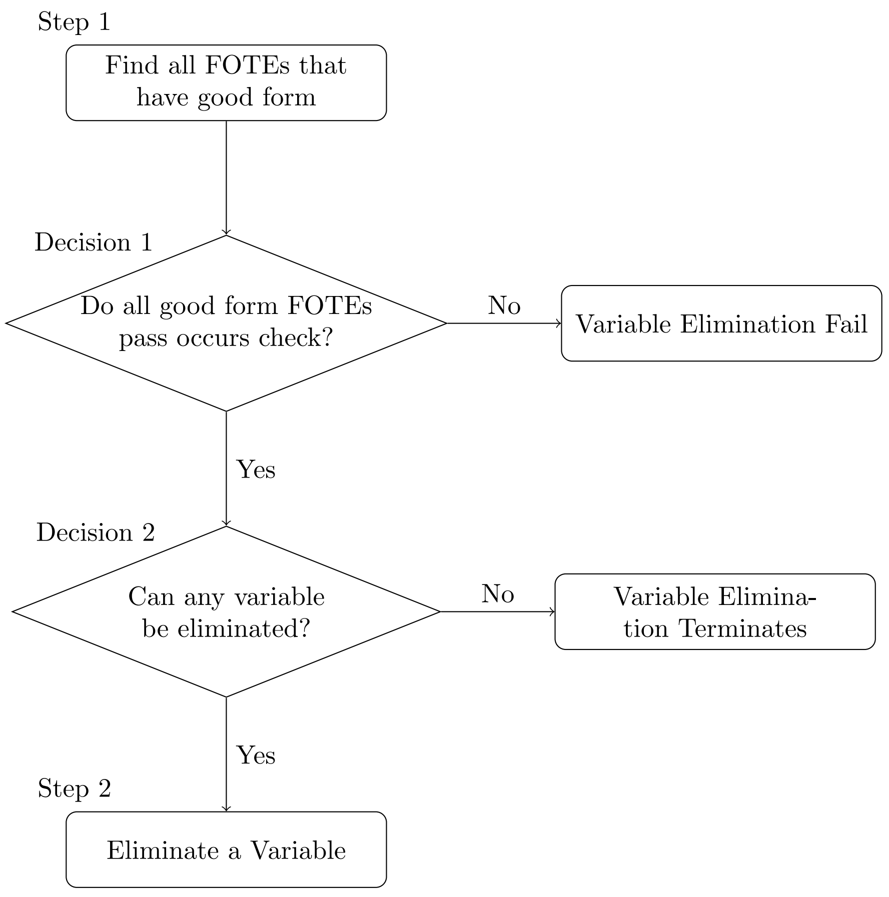
Step 1
Find all FOTEs that have good form
Decision 1
Do all good form FOTEs pass occurs check?
No
Variable Elimination Fail
Yes
Decision 2
Can any variable be eliminated?
No
Variable Elimination Terminates
Yes
Step 2
Eliminate a Variable

Figure 4.1: Actions and decisions made during variable elimination.

Check for Fail before eliminating any variable

As required by the algorithm (Section 4.1.3 on page 22), before eliminating the Variable Symbol on the left side of any good form FOTE, we perform occurs check.

Variable elimination returns Fail when any good form FOTE has its Variable Symbol occurs in the right side of the FOTE when the right side is a Function. Consider the rules of occurrence introduced in section 3.2.3, when the right side of the good form FOTE is a Function and the Variable Symbol on the left side occurs in this Function, we say the good form FOTE *doesn't* pass occurs check. When the right side of the good form FOTE is not a Function, or it a Function but it has no occurrence of the Variable Symbol on the left of the FOTE, we say this FOTE passes occurs check. So variable elimination returns Fail when any good form FOTE doesn't pass the occurs check.

It is good to go through all good form FOTEs in the FOTE set to make sure they all pass occurs check before we perform variable elimination on one of the good form FOTEs that passes occurs check because this, we call it *Fail Check for Variable Elimination*, may prevent waste of time and effort.(Example 4.2.9)

Example 4.2.9 *If we started from the first good form FOTE, which is 1, and eliminate all Ys,then we would find that 2 causes variable elimination return Fail, so the effort on elimination of Y was wasted since it doesn't rely on elimination of Y to find out that 2 causes variable elimination return Fail. By looking through all good form FOTEs which are 1 and 2 , and checking whether they would pass occurs check before doing any variable elimination, we can prevent the waste of effort made on eliminating Ys.*

1 $Y=g\ (a\ ,\ b)$
2 $X=f\ (X)$
3 $h\ (X\ ,\ Y)=h(b\ ,\ g\ (a\ ,\ b))$
4 $k\ (Y)=Z$

The Fail Check for Variable Elimination has two steps:

1. Filter the FOTE set for good form FOTEs.This corresponds to Step 1 in Figure 4.1.
2. Decide whether any of the good form FOTEs doesn't pass occurs check. This corresponds to Decision 1 in Figure 4.1.

We have already seen the filtering mechanism during introducing implementation of trivial removal. The filter works on a FOTE set and takes a predicate ("good form predicate") to judge individual FOTEs for the goodness of its form and returns all good form FOTEs in the FOTE set or an empty list when there is no good form FOTE in the FOTE set.

In concern with the "good form predicate",since we have seen for many times how to tell whether a FOTE satisfies a particular predicate, for example in Algorithm 4.1, 4.2 and 4.3, the algorithm for checking whether a FOTE has a "good form" is a similar procedure, thus not being shown in the thesis.

We also have already seen how to use Haskell function "`any`" in Algorithm 4.4 for existential check, so we now only need to provide a mechanism to tell whether an individual good form FOTE passes or doesn't pass an occurs check. This is again a predicate over a FOTE so we omit account of it provided we have already known the rules of occurrence.

Now we can write the variable elimination fail check as Algorithm 4.5 on page 35.

Algorithm 4.5 Fail Check for Variable Elimination,where "`variableEliminationFail`" is composition of two partial functions "`any (not.goodFormFOTEPassesOccursCheck)`" and "`filter foteHasGoodForm`". Note that when there is no good form FOTE in the FOTE set, "`filter`" will return an empty list; the "`any`" function, working on the empty list,will return *False*, as defined by Haskell—this *False* is just the correct answer to the question "Does variable elimination fail when there is no good form FOTE in the FOTE set?".

```
variableEliminationFail :: FOTEset -> Bool
variableEliminationFail =
  any (not.goodFormFOTEPassesOccursCheck) . filter foteHasGoodForm
```

Alternatively, using the inference rule

$$\exists\ X\ \neg\ p(X) \Leftrightarrow \neg\ \forall\ X\ p(X)$$

, we can also write the variable elimination fail check as Algorithm 4.6 on page 35.[3]

Algorithm 4.6 Alternative fail check for variable elimination, which is composition of three functions: the function "`not`" and the partial functions "`all goodFormFOTEPassesOccursCheck`" and "`filter foteHasGoodForm`".

```
variableEliminationFail :: FOTEset -> Bool
variableEliminationFail =
  not . all goodFormFOTEPassesOccursCheck . filter foteHasGoodForm
```

Deciding whether there are any variable that can be eliminated

After Step 1 and Decision 1 in Figure 4.1, and we have made sure that variable elimination would not return Fail when it is applied to the current FOTE set, we need to find out are there any good form FOTE whose Variable Symbol on the left can be used to eliminate all other occurrence of the same Variable Symbol in the FOTE set. This corresponds to Decision 2 in Figure 4.1.

When we try to find such a FOTE, we don't pay attention to which good form FOTE has Variable Symbol that can be eliminated; we only pay attention to the *existence* of such a good form FOTE, because whether such a good form FOTE exists determines what action we are going to take, choosing from two

[3]The feature that the "`any`" function, when working on an empty list,will return *False* while the "`all`" function, when working on an empty list,will return *True* caused me misunderstood the behavior of "`any`" function so I adopted Algorithm 4.6 in the program. Later on I realised that this is how Haskell takes care of the inference rule $\exists\ X\ \neg\ p(X) \Leftrightarrow \neg\ \forall\ X\ p(X)$ when the domain of the variable X is $\emptyset$.

choices—if there was such a good form FOTE, we can use it to do variable elimination; otherwise we know that in current FOTE set there are no Variable Symbol that can be eliminated, so we may try to decide whether the FOTE set is already in solved form and if it was not in solved form we may try some other transforms. (Example 4.2.10-Part 1)

Example 4.2.10 *This example has two parts (Part 1 and Part 2) to demonstrate two different statements. The equation set is shared by both parts.*

Part 1 : All good form FOTEs passed occurs check. In Case 1, X can be eliminated so we can eliminate X. In Case 2, only 1 and 2 have good form but neither X nor Y has more than one occurrence in the FOTE set, so variable elimination can't be done further on the current phase of this FOTE set, though it can be done later on either after term reduction on f in 3 or after orientation of 4. So the different results of existential check for variables that can be eliminated determine different action as next step.

	Case 1	*Case 2*
1	*X=a*	*X=a*
2	*Y=f (b)*	*Y=f (b)*
3	*g (X)=g (Z)*	*f (Z)=f (b)*
4	*X=a*	*g=Z*

Part 2 : In case 1, candidate variable X for variable elimination occurs more than once in the entire FOTE set. Its first occurrence is as the left member of a good form FOTE, which is 1, the other occurrences are scattered across the rest of FOTEs other than the FOTE that hosts its first occurrence. So X also occurs at least once in the rest of the FOTE set (in 2, 3 and 4).

Under the condition that all good form FOTEs pass occurs check, as shown in Figure 4.1, to check for the existence of Variable Symbol that can be eliminated, we need to:

Choice 4.2.1 *Check through all good form FOTEs in the FOTE set to see if any of the good form FOTE's left member, which is a Variable Symbol, occurs* more than *once in the entire FOTE set.*

Since a good form FOTE has already passed occurs check, having its left member occur more than once in the entire FOTE set means the first occurrence is at the left side of the good form FOTE as the left FOTE Member, all other occurrences are not on the right side of the same FOTE and they are in the rest of the FOTE set, occurring at least once. (Example 4.2.10-Part 2)

So we may alternatively choose to:

Choice 4.2.2 *Check through all good form FOTEs in the FOTE set to see if any of the good form FOTE's left member, which is a Variable Symbol, occurs* at least *once in the rest FOTEs of the FOTE set.*

Note that the statement "if a variable as the left member of a good form FOTE occurred more than once in the entire FOTE set, then this variable occurred at least once in the rest of the FOTE set" may not hold before we make sure that all good form FOTEs pass occurs check. (Example 4.2.11)

Example 4.2.11 *FOTE 1 doesn't pass occurs check, its Variable Symbol X on left side also occurs on the right sides of it so X occurs more than once in the entire FOTE set but it is still possible that X doesn't occur in other FOTEs (2, 3 and 4) so occurring more than once in entire FOTE set doesn't necessarily means occurring at least once in the rest FOTEs of the FOTE set.*

1	*X=f (X)*
2	*Y=a*
3	*g (Y , Z)=g (a , b)*
4	*h (Z)=h (b)*

Occurrence Count No mater which of the Choices 4.2.1 or 4.2.2 of action we take to check for the existence of Variable Symbol that can be eliminated, we always need a Variable Symbol occurrence counting mechanism to decide how many times of occurrence does a given Variable Symbol have in a given FOTE set— if we took Choice 4.2.1, we provide the Variable Symbol and the entire FOTE set to the counting mechanism; if we took Choice 4.2.2, we provide the Variable Symbol, and the entire FOTE set excluding one instance of the good form FOTE that the Variable Symbols belongs to, to the counting mechanism.

We can realise occurrence counting by comparing the Variable Symbol to all FOTE Members in the FOTE set and increment an accumulator by 1 every time we find a FOTE Member where the Variable Symbol occurs, according to the Rules of Occurrence introduced in section 3.2.3.

Elimination of a Variable Symbol

When the time comes to eliminate a Variable Symbol, let us first review the current status of the FOTE set that we are about to work on, to inform our action; then we see the reuse of code for applying substitution; after that, we present the algorithm for eliminating a given variable, followed by an example.

Current status of the FOTE set After we go through Step 1 and Decision 1, 2 we may reach the situation where we found Condition 4.2.1 holds.

Condition 4.2.1
- *There are good form FOTEs in the FOTE set.*
- *For all good form FOTEs in current FOTE set the variable elimination would not return Fail.*
- *There are Variable Symbols that can be eliminated.*

So we can start to eliminate one Variable Symbol.

Reusing code for Substitution by unzipping the FOTE set Given a good form FOTE, performing variable elimination means substituting its right member for all occurrence of its left member in the FOTE set but the occurrence as the left member of the given good form FOTE.

In implementation, variable elimination involves choosing the tuple that represents the good form FOTE whose left member is to be eliminated, then substitute the right member for all occurrence of the left member in the list of tuples excluding the chosen tuple. The chosen tuple can be regarded as a substitution component.

In Section 3.2.5 it was introduced that a substitution component could be applied on a single term, and a substitution, as a list of substitution components, were applied on a list of terms. Now we need to apply substitution component onto a list of tuples (a list of tuples is also called an associate list) of terms.

Instead of trying to devise a mechanism to map application of individual substitution component to an associate list, the Haskell standard "unzip" and "zip" mechanism can be used to turn an associate list into a pair of lists of terms, and we treat the substitution component as a singleton substitution that contains only one component, so all the existing code of applying substitution can be reused easily, indifferent from any new mechanism mapping a substitution component onto an associate list.

Algorithm for elimination an eligible variable The whole process of eliminating a variable is given in Algorithm 4.7 on page 38 (see also Example 4.2.12); this corresponds to Step 2 in Figure 4.1 on page 33.

Algorithm 4.7 Elimination procedure of a Variable Symbol

```
1. Find from all good form FOTEs the first good form FOTE $ft$ that contains
   a Variable Symbol that can be eliminated.
2. Remove $ft$ from the FOTE set but keep a copy of $ft$ elsewhere; yielding
   a new FOTE set $fts$ from where one occurrence of $ft$ has been removed.
3. Regard $ft$ as substitution $\sigma_{ft}$ by tuple swap.
4. Unzip $fts$ to get a tuple $t$ of type ([Term],[Term]).
5. Apply $\sigma_{ft}$ to both lists of terms in $t$, yielding $t'$.
6. Zip the two lists in $t'$ where there is no occurrence of the Variable Symbol
   to be eliminated, yielding FOTE set $fts'$.
7. Add $ft$ back to $fts'$ to get the result of variable elimination on $ft$.
```

Example 4.2.12 *Let's apply Algorithm 4.7 on the FOTE set*

$$\begin{aligned} X&=a \\ Y&=f\ (b) \\ g\ (X)&=g\ (Z) \\ X&=a \end{aligned}$$

This set is stored in a "list of tuple" form:

$$[\ (X\ ,\ a)\ ,\ (Y\ ,\ f\ (b))\ ,\ (g\ (X)\ ,\ g\ (Z))\ ,\ (X\ ,\ a)\ ]$$

. Assume we have checked that Condition 4.2.1 holds.

1. *The good form FOTEs are:*

$$[\ (X\ ,\ a)\ ,\ (Y\ ,\ f\ (b))\ ,\ (X\ ,\ a)\ ]$$

, of which the first one $(X\ ,\ a)$ contains a Variable Symbol X that can be eliminated. Let $ft = (X\ ,\ a)$.

2. *The FOTE set with ft removed from it, is:*

$$fts = [\ (Y , f\ (b)) , (g\ (X) , g\ (Z)) , (X , a)\ ]$$

.

3. *Let* $\sigma_{ft} = swap\ (\ ft\ ) = (a , X)$*, representing substitution* $\{a\ /\ X\}$.

4. *Unzip* fts.

$$\begin{tikzcd} {[\ (Y , f\ (b)) , (g\ (X) , g\ (Z)) , (X , a)\ ]} \arrow[d, "unzip"'] \\ t = (\ [\ Y , g\ (X) , X\ ] , [\ f\ (b) , g\ (Z) , a\ ]\ ) \end{tikzcd}$$

5. *Apply* σ_{ft} *to both lists in* t*, yielding* t'.

$$\begin{tikzcd} t = (\ [\ Y , g\ (X) , X\ ] , [\ f\ (b) , g\ (Z) , a\ ]\ ) \arrow[d, "Apply\ \sigma_{ft} = \{a/X\}"'] \\ t' = (\ [\ Y , g\ (a) , a\ ] , [\ f\ (b) , g\ (Z) , a\ ]\ ) \end{tikzcd}$$

6. *Zip two lists in* t'.

$$\begin{tikzcd} zip\ [\ Y , g\ (a) , a\ ]\ and\ [\ f\ (b) , g\ (Z) , a\ ] \arrow[d] \\ fts' = [\ (Y , f\ (b)) , (g\ (a) , g\ (Z)) , (a , a)\ ] \end{tikzcd}$$

7. *Add* ft *back to* fts' *yielding*

$$[\ (X , a) , (Y , f\ (b)) , (g\ (a) , g\ (Z)) , (a , a)\ ]$$

which represents FOTE set

$$\begin{aligned} X&=a \\ Y&=f\ (b) \\ g\ (a)&=g\ (Z) \\ a&=a \end{aligned}$$

where all but one particular occurrence of X *have been eliminated.*

4.2.6 Algorithm termination check: deciding when the algorithm can terminate

The algorithm was designed to be able to terminate either with success or Fail. Assume the unification problem passed to the algorithm can be transformed into a solved form, the application of the four transforms will finally terminate, when no more transform can be applied to the FOTE set and turn it to a different FOTE set. At this time the FOTE set is in solved form. Assume the unification problem passed to the algorithm cannot be transformed into a solved form, the Fail must be returned during application of either term reduction or variable elimination. [4]

Termination of the algorithm is made after inspection of the current state of the FOTE set. By inspecting the current FOTE set one can make a decision, choosing among Decision 4.2.1, 4.2.2 and 4.2.3.

Decision 4.2.1 *No more transforms can be applied and the FOTE set is already in the solved form.*

Decision 4.2.2 *Fail shall be returned.*

Decision 4.2.3 *Transforms can be applied.*

Decision 4.2.1 and 4.2.2 define the termination; moreover, if both of them were not made then Decision 4.2.3 must be made.

There are two aspects of the termination check, from which we can make Decision 4.2.1 and 4.2.2 respectively.For Decision 4.2.1 to be made, it requires all FOTEs have "good form" and all left FOTE Members shall have only one occurrence in the FOTE set. (See also Section 4.1.2) For Decision 4.2.2 to be made, it requires the FOTE set fail to pass Term Reduction Fail Check or Variable Elimination Fail Check.

Thus we devise checking mechanism for Decision 4.2.1 and Decision 4.2.2 respectively: "Is In solved Form" and "Is Unsolvable".

Is In Solved Form

During introduction of implementing variable elimination in Section 4.2.5 we have already seen the mechanism for "good form" check and the mechanism for Variable Symbol occurrence counting.

These two mechanisms can be used together to tell whether a FOTE set is already in solved form. But before further discussion, let's handle a boundary situation first: when the FOTE set is $\emptyset$. Say $\emptyset$ is in solved form. (Example 4.2.13)

Example 4.2.13 *Consider the singleton FOTE set*

$$\{ \ a \ = \ a \ \}$$

It is not in solved form nor does it cause term reduction or variable elimination fail. So we can apply transforms on it and the only applicable transform is term reduction, which removes the only member of the singleton set and yields a $\emptyset$. Now the algorithm has to terminate since no transforms can be applied to it; and Fail was not returned during application of transforms so now the FOTE set, which is $\emptyset$, is in solved form. From another perspective,since the FOTE set representation of any unifier is in solved form, this $\emptyset$ corresponds to the empty unifier for the unification problem, so this $\emptyset$ is in solved form.

When the FOTEs set is not empty, it is in solved form if and only if

1. All FOTEs are in "good form" , and
2. There is no Variable Symbol, as a left FOTE Member, occurs more than once in the FOTE set.

This is the equivalent expression of "For all FOTEs, it's left FOTE Member is a Variable Symbol which occurs only once in the FOTE set." These two criteria were checked one by one, starting from the "good form check", then if all FOTEs in the FOTE set has good form, check the times of occurrence of each left FOTE Member, using the already developed occurrence counting mechanism.

Is Unsolvable

The FOTE set is unsolvable if and only if

- The FOTE set cause Term Reduction fail, or
- It causes variable Elimination fail.

Since we have already had mechanisms to look through all FOTEs in the FOTE set to decide whether term reduction or variable elimination will return Fail, we can use these mechanisms to judge the termination of algorithm with Fail.

4.2.7 Loop of decisions and actions

When we pass a FOTE set to Martelli's algorithm, we consider Decision 4.2.1, Decision 4.2.2 and Decision 4.2.3 in a row and may go through these considerations for several rounds until one of Decision 4.2.1 and Decision 4.2.2 was made. So our function for Martelli's algorithm shall have a recursive structure to go through this loop of decisions and actions, as depicted in Figure 4.2 on page 42.

4.2.8 Planning application of transforms

When neither of Decision 4.2.1 and 4.2.2 is made, it is appropriate to perform transforms. In order to use the transforms in a rational rather than random way, we start from analysing the properties of the target of the transforms, which is the FOTE set at this stage.

The properties of the target of the transforms

The FOTE set we are dealing with at this stage has predictable properties:

Property (4.2.1 $\vee$ 4.2.2) $\wedge$ 4.2.3.

Property 4.2.1 *It has FOTEs that are not in good form.*

Property 4.2.2 *There are multiple occurrence of one Variable Symbol.*

Property 4.2.3 *Neither term reduction nor variable elimination would fail when being applied to the current FOTE set.*

Aim of the transforms

To change this FOTE set towards solved form, we need to cure its two problems: Property 4.2.1 and 4.2.2.

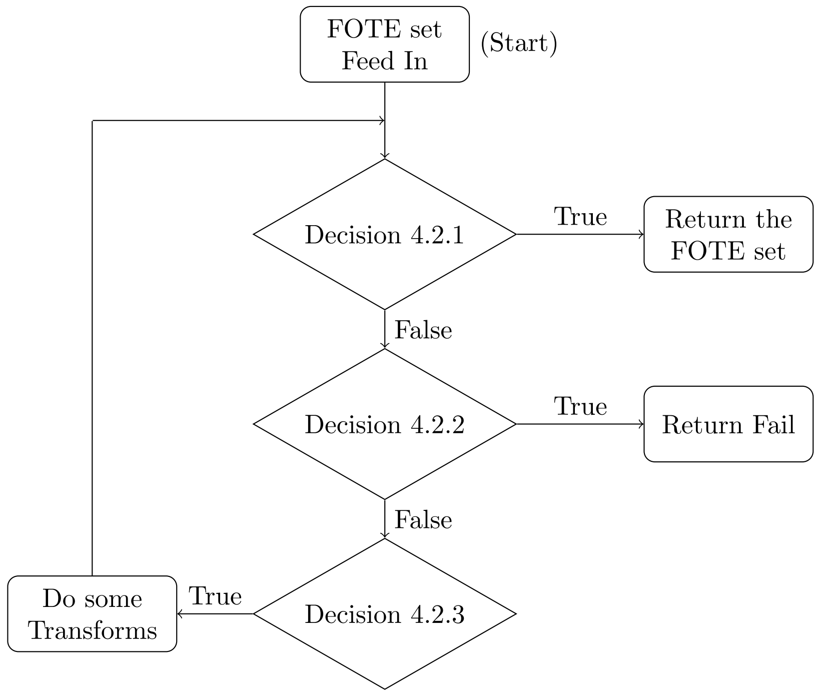

Figure 4.2: Implementing Martelli's algorithm: loop of decisions and actions

Handling double aims: prioritization

Since there are two problems we need to solve: Property 4.2.1 and 4.2.2, We can deal with them by assigning an *priority* to one of them. Let's see if we prioritize solving Property 4.2.1 over Property 4.2.2, which means every time we need to perform transformation, we first focus on making FOTEs in the FOTE set have good form until all FOTEs are in good form, then we start eliminating duplicated Variable Symbols.

Responding to Property 4.2.1 at first

Problem details There are various, but limited reasons for a FOTE not to be in good form, for

1. being "disoriented" (not being oriented with Variable Symbol on the left), or

2. being trivial, or

3. having only Functions or Constant Symbols as its FOTE Members.

Choice of transforms to solve the problem The above atomic problems can be cured by Orientation, Trivial Removal and Term Reduction, respectively.

Planning the order of transform application Trivial Removal and Orientation can always be done at any time to push the FOTE set towards a solved form. Term Reduction helps elimination of FOTEs whose FOTE Members are only Constant Symbols or Functions, and it may generate new FOTEs that are not in good form (Example 4.2.14).

Example 4.2.14 *Performing term reduction on FOTE*

$$p\big(X, f(X), m(b), Z\big) = p\big(b\ , f(b),\ Y, Z\big)$$

yields FOTE set

$$\begin{cases} X{=}b \\ f(X){=}f(b) \\ m(b){=}Y \\ Z{=}Z \end{cases}$$

where all but the first FOTE are not in good form.

Since the FOTEs liable to Orientation and Trivial Removal will not be influenced by Term Reduction, we can leave them there and apply Term Reduction first, then apply Orientation and Trivial Removal to deal with the original disoriented and trivial FOTEs (which existed before Term Reduction and remained intact after Term Reduction) and newly generated (generated by Term Reduction) disoriented and trivial FOTEs altogether.

Responding to Property 4.2.2 in second

Exhaustive pattern Once we checked that all FOTEs in the set have good form, but the FOTE set is still not in solved form nor being unsolvable, we can be confident that there are Variable Symbols that can be eliminated; non-existence of these variables that are subject to elimination is impossible. This is the exhaustive pattern of or algorithm. So we can start applying Variable Elimination as the transform we do at the current round of recursion. Then we start the next round under our prioritisation.

Doing a little bit cleaning Since it is possible to generate disoriented or trivial FOTEs after Variable Elimination(Example 4.2.15), we can apply Orientation and Trivial Removal immediately after the Variable Elimination, before we leave this round of loop.

Example 4.2.15 *For FOTE set*

$$\begin{cases} X{=}Z \\ f(X){=}Y \\ X{=}Z \end{cases}$$

Taking the first FOTE $X = Z$ and eliminate X, we get FOTE set

$$\begin{cases} X{=}Z \\ f(Z){=}Y \\ Z{=}Z \end{cases}$$

where the second FOTE is disoriented and the third FOTE is trivial, both of which were resulted from variable elimination on X.

4.2.9 The comprehensive flow chart of implementing Martelli's algorithm

After arranging all the decisions and actions, we get the action plan for implementing Martelli's algorithm ready, as in Figure 4.3 on page 45.

4.3 Conclusion

During implementing Martelli's algorithm, a concise action plan was gradually developed. Such a plan was not made complete from the very beginning. Instead, it started from being incomplete and lacking a global view: starting from implementing individual transforms and their own fail check, then as transforms and fail checks were implemented, further actions, such as termination check, had their necessity discovered and implemented. After all these component parts were ready, a clearer perception on how the algorithm was performed was formed, which informed the configuration of the component parts,particularly the loop structure. The order of transform application was not explicitly stated in the original paper. The attempt to implement the algorithm raised the need to inspect the running of the algorithm and to find out what a person exactly need to do when he uses this algorithm to solve problems. Such detailed perception was then gradually converted to code to instruct a computer.

When planning transform application, the priority was put on turning FOTEs into good form rather than eliminating variables. It is possible to relocate the priority to the latter, and such choice could be explored to help seeing the difference between the two prioritization options.

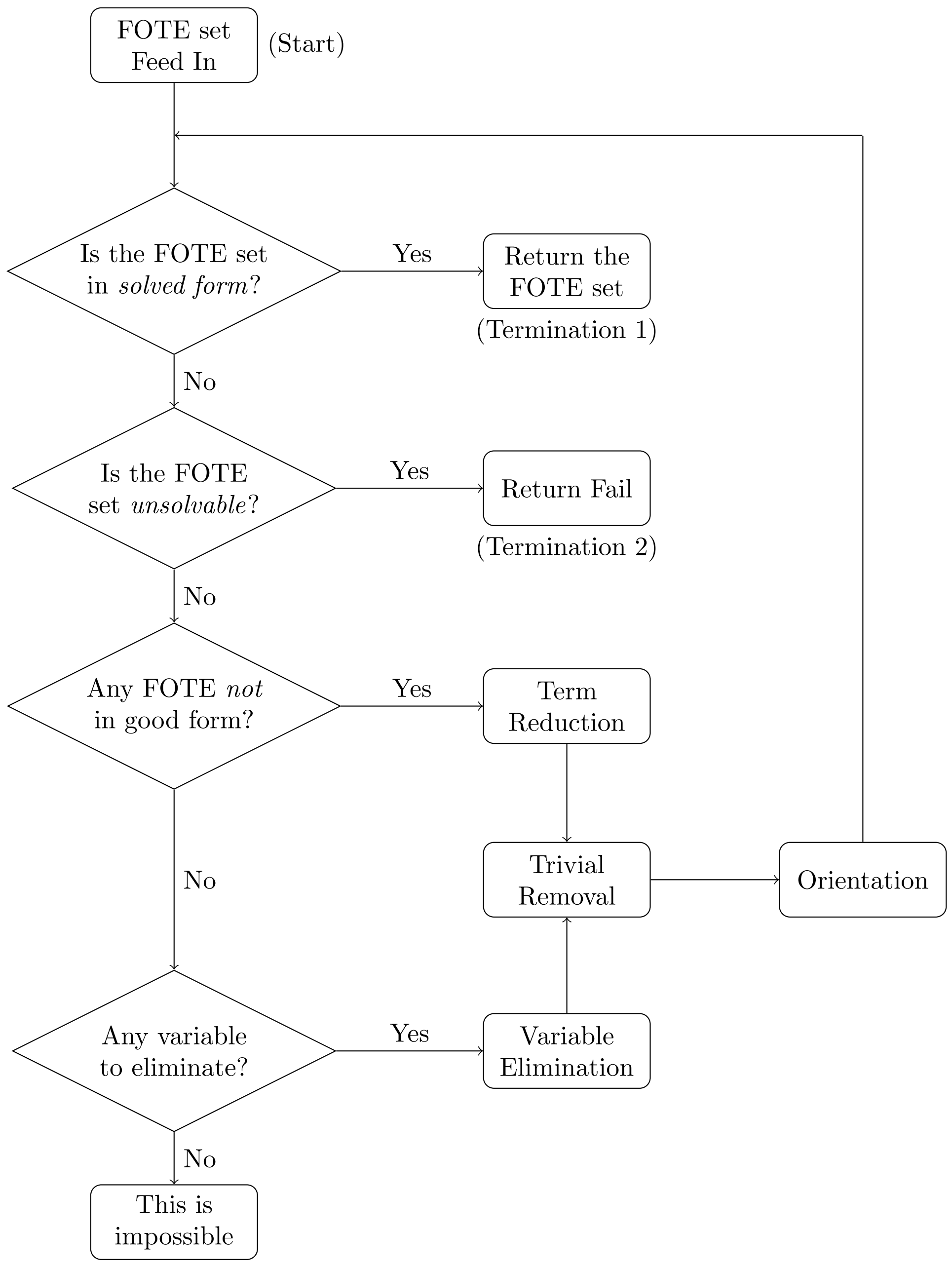

Figure 4.3: Martelli's algorithm: recursive flow chart of decisions, transforms and termination

Chapter 5

Testing The Algorithms

5.1 Self Project Quality Control

5.1.1 Initial testing method

During the development of implementations I relied on hand made test data. For example, if I wanted to test the algorithms, I would first come up with a unification problem, such as

$$f(a, X) = f(Y, b)$$

then I would need to type in the problem as

```
[( Function ‘‘f’’ 2 [Constant ‘‘a’’, Variable ‘‘X’’] , Function
‘‘f’’ 2 [Variable ‘‘Y’’, Constant ‘‘b’’] )]
```

It was very slow to type and when the size of the problem became larger it was tedious to type in all the value constructors and double quotes. I initially didn’t have clear methodology to come up with unification problems, neither. I made them in an intuitive way based on examples I have seen in textbooks and papers.

5.1.2 Turning point during the development of testing method

The way to test was upgraded after two changes happened: firstly, in order to come up with unification problems more efficiently, even still manually, I studied and discovered how to design unification problems that can be solved, ending up with an algorithm for generating solvable unification problems; secondly, a PhD student who works on QuickCheck heard about my problem with test data during the daily meeting of our research group, he suggested that QuickCheck might help. I didn’t know QuickCheck before so we spent a total of about 4 hours together,(we had two 2-hour sessions) he made a initial version random term generator and demonstrated how these random terms can be used to test properties of unification algorithms. After his demonstration I began to believe that QuickCheck can help me for the test. After he went back home in the evening, he upgraded the term generator to generate terms with controllable size and he gave it to me as a gift.

I accepted this gift and studied it, and modified it so it can generate terms of syntactically valid names. The unexpected good performance of this term generator backed my decision that I wanted to use it, together with my knowledge on how to make solvable unification problems, to develop a random unifiable FOTE generator. (Section 5.2) The plan was successful, so later I studied how to make unsolvable unification problems, and made another generator of random unsolvable FOTE.(Section 5.3)

5.1.3 New testing method: QuickCheck

Test Properties

The two generators, one for solvable unification problem, the other for unsolvable unification problem, can work, and worked together to test Property 5.1.1 and 5.1.2 of the algorithms I implemented.

Property 5.1.1 *My unification algorithm implementation can always give correct unifier for solvable FOTE.*

Property 5.1.2 *My unification algorithm implementation always return Fail when the FOTE is unsolvable.*

Test Design

The Property 5.1.1 was verified by providing many random solvable unification problems to the algorithm, then obtaining the unifier from the algorithm, then checking that applying the unifier to the original unification problem yields a new unification problem with ϵ (empty unifier) as its unifier.

The Property 5.1.2 was verified by providing many random unsolvable unification problems to the algorithm, then see whether the algorithm always returns Fail for these unsolvable problems.

It turned out that the random solvable and unsolvable FOTE generators both require a unification algorithm to work. I used Robinson's algorithm in the generator, and used the generated FOTEs to test Martelli's algorithm, since my implementation of Robinson's algorithm provides reusable code for the implementation of the generator.

Some initial manual inspection showed that the generators worked correctly, so implementation of Robinson's algorithm was verified. Further more, if Martelli's algorithm can respond to random problems correctly, then its implementation was also verified.

Test Result Brief

Test on Property 5.1.1 Automated test shown that Property 5.1.1 holds.The huge body of auto generated problems and their solution by the algorithm was also sampled and inspected manually to ensure correctness.

Test on Property 5.1.2 A bug was discovered during test on unsolvable problems about Property 5.1.2: a Variable Symbol was unified with a Function in which the Variable Symbol occurs. This problem was investigated and solved.

A somehow Benign Bug During writing the thesis I spotted another inappropriateness in implementation of Martelli's algorithm: the criteria for existence of variables subject to elimination was occurring more than once, rather than at least once, in the rest FOTEs of the FOTE set. Somehow this piece of inappropriateness didn't cause trouble in previous automated test. After I fixed it, there was still no trouble in the automated test, and manual inspection of random unification problems and their solutions were correct.

5.1.4 Quality guarantee

After all aforementioned test, I committed the codes as tested and reliable final product of my MSc project.

5.2 Design of the Unifiable Equation Generator: What kind of equations can be unified?

A unifiable equation can be created by complying with any single one of the following rules:

Rule:

5.2.1
5.2.2
5.2.3
5.2.4
5.2.5
5.2.6
5.2.7
5.2.8
5.2.9

These rules are referred to collectively as *Rules of Solvable Unification Problem.*

Rule 5.2.1 *Variable Symbol = Variable Symbol, where they can have either the same name or different names.*

Rule 5.2.2 *Variable Symbol = Function, where the Variable Symbol doesn't occur in Function.*

Rule 5.2.3 *Variable Symbol = Constant Symbol*

Rule 5.2.4 *Constant Symbol = Variable Symbol*

Rule 5.2.5 *Constant Symbol = Constant Symbol, where the two Constant Symbols must have the same name.*

Rule 5.2.6 *Constant Symbol = Function, where the Constant Symbol has the same name as the Function, and the Function has no arguments.*

Rule 5.2.7 *Function= Variable Symbol, where the Variable Symbol doesn't occur in Function.*

Rule 5.2.8 *Function = Constant Symbol, where the Constant Symbol has the same name as the Function, and the Function has no arguments.*

Rule 5.2.9 *Function = Function, where each Function must have finite depth of function composition. The Functions must have the same name (i.e. The Function Symbols of these two Functions are the same) and their argument lists must have the same and finite length n such that $0 \leq n \leq m$. Their argument lists shall be represented by L_n and L'_n,which shall be generated in the following way:*

```
Set the value of n to an arbitrary natural number.
if $n = 0$ then
    return
        $L_0 = Nil$
        $L'_0 = Nil$
else if $n = 1$ then
    Create $t_1$ and $t'_1$ by making the equation $t_1 = t'_1$ comply with any single one of Rules of Solvable Unification Problem.
    return
        $L_1 = t_1$
        $L'_1 = t'_1$
else
    Create $t_1$ and $t'_1$ by making the equation $t_1 = t'_1$ comply with any single one of Rules of Solvable Unification Problem.
    for $k : (1 < k \leq n)$ do
        repeat
            Create $t_k$ and $t'_k$ by making the equation $t_k = t'_k$ comply with any single one of Rules of Solvable Unification Problem.
            $\sigma_k$ = unifyTerms($t_k$ , $t'_k$)
            Apply $\sigma_k$ to
                Function Symbol $(L_{k-1})$ = Function Symbol $(L'_{k-1})$
            [1] where
                $L_{k-1} = t_1, t_2, t_3 \ldots t_{k-1}$
                $L'_{k-1} = t'_1, t'_2, t'_3 \ldots t'_{k-1}$
        until The equation: $\sigma_k\big(\text{Function Symbol } (L_{k-1})\big) = \sigma_k\big(\text{Function Symbol } (L'_{k-1})\big)$ is solvable
    end for
    return
        $L_n = t_1, t_2, t_3 \ldots t_n$
        $L'_n = t'_1, t'_2, t'_3 \ldots t'_n$
end if
```

[1]The Function Symbols on both sides of the equation shall be the same

5.3 Random Unsolvable Unification Problems

To create an unsolvable unification problem, comply with any single one of the following rules:

Rule:
5.3.1
5.3.2
5.3.3
5.3.4
5.3.5
5.3.6

These rules are referred to collectively as *Rules of Unsolvable Unification Problem.*

Rule 5.3.1 *Constant Symbol = Constant Symbol ,where the FOTE Members must have different names.*

Rule 5.3.2 *Constant Symbol = Function, where the FOTE Members have different names or they have the same name but the Function has arguments.*

Rule 5.3.3 *Variable Symbol = Function, where the Variable Symbol occurs somewhere in the Function argument list.*

Rule 5.3.4 *Function = Variable Symbol, where the Variable Symbol occurs somewhere in the Function argument list.*

Rule 5.3.5 *Function = Constant Symbol , where the FOTE Members have different names or they have the same name but the Function has arguments.*

Rule 5.3.6 *Function = Function , where they have finite depth of function composition and*

1. *The two FOTE Members have different names, or*
2. *they have the same name but different arity, or*
3. *they have the same name and same non-zero arity, but there is at least one pair of corresponding arguments that can form an equation that complying the Rules of Unsolvable Unification Problem.*

Note that the Rules of Unsolvable Unification Problem do not cover all types of unsolvable unification problems, particularly the "Function = Function" type when any pair of corresponding arguments are unifiable but after applying unifier of argument pairs on the left (Example 5.3.1) to the rest of the arguments, the rest arguments become not being unifiable. The incompleteness of the rule is due to either I didn't notice such a case at the time of designing or I wanted to simplify the design of Unsolvable Unification Problem Generator.

Example 5.3.1 *The Rules of Unsolvable Unification Problem can't generate a FOTE like*

$$f(X, X) = f(a, b)$$

where corresponding pairs of arguments are unifiable but the problem as a whole is unsolvable.

5.4 Algorithm Working Demonstration

The generators were also used to create a demonstration of solving unification problems. The computer generates a random unification problem, either solvable or unsolvable, and uses an unification algorithm, as specified by the user, to solve the problem and display both the problem and the result. Currently the demonstration is only available for Martelli's algorithm, because Robinson's algorithm was implicitly demonstrated through generating the random unifiable problems.

5.5 Conclusion

Study on unification algorithm helped me understand what kind of unification problems are solvable or unsolvable, which further facilitated the creation of random equation generators.

Enumeration was an important method to reveal possible details of unification problems. The Rules of Solvable/Unsolvable Unification Problems are both devised based on enumeration.

Chapter 6

Conclusion of the thesis

6.1 Achievements

- Two unification algorithms, Robinson's and Martelli's, were implemented in Haskell. Tests showed that they worked correctly.
- Random (solvable or unsolvable) unification problem generators were devised and programmed, and used in test. They may also be used to test any unification algorithm implemented in the future.
- The process of implementing the two algorithms in Haskell was recorded in detail, and well typeset by $\LaTeX\,2_\varepsilon$.

6.2 Highlight: things that were particularly well done

- Structural design of the Haskell code for both unification algorithms.
- Make and use of random solvable and unsolvable unification problem generator in the test.
- Use of $\LaTeX\,2_\varepsilon$ in writing the thesis.
- Rich and active communication with the supervisor and good relationship with students in the same lab.

6.3 Improvement

In concern with the implementation of unification algorithms, there is nothing apparent that can be improved. The generator for unsolvable unification problems cannot generate the complete set of unsolvable problems so here lies the task to improve it to provide richer variety of test data for the algorithms.

6.4 Future work

More unification algorithms could be implemented There are many other unification algorithms that were designed intending to reduce the time and space cost. They can be implemented as a way to gain more knowledge on existing unification algorithms as well as practising Haskell programming skills.

Comparison of different implementations It would be interesting to compare implementation of the same algorithm in different functional programming languages, such as Haskell implementation vs. OCaml implementation, or to compare the implementation of the same algorithm in the same language by different people, in order to appreciate different but successful ways from different people to deal with the same problem, and to evaluate the quality of my own implementation through comparing my work with others' work.

Appendix A

Haskell Language

Submitted in electronic media.

Appendix B

Source Code

All codes were submitted in electronic media.

Appendix C

Project Log Sheet

Highlighted Statistics

- Total amount of days the project last: *131* days
- Total amount of hours spent on the project: *383* hours
- Average hours per day spent on project: *2.92* hours ($\approx$ 2 hours 55 minutes)
- Standard deviation of daily hours on project: *2.55* hours ($\approx$ 2 hours 33 minutes)
- Plot of daily working hour versus date, showing the fluctuation of daily working hours. (See electronic submission)

Appendix D

Meeting Log

Submitted in electronic media.

Appendix E

Guide for Examiners

Submitted in electronic media.

Appendix F

Guide for Users

Submitted in electronic media.

Bibliography

[1] Robinson,J.A.. (1965). *A Machine-Oriented Logic Based on the Resolution Principle.* Journal of the ACM (JACM), 12(I), 23-41.

[2] Knight, K. (1989). *Unification: A Multidisciplinary survey.* ACM Computing surveys, 93-124.

[3] Luger, G. F., & Stubblefield, W. A. (1993). Artificial Intelligence Structures and Strategies for Complex Problem Solving Seond Edition. Benjamin/Cummings Publishing Company, Inc.

[4] Martelli, A., & Montanari, U. (1976). *Unification in linear time and space: a structured presentation.* Pisa, Italy.: Istituto di Elaborazione delle lnformazione, Consiglio Nazionale delle Ricerche.

[5] Robinson, J. A. (1968). *The Generalized Resolution Principle.* (D. Michie, Ed.) Machine Intelligence 3.

[6] Winston, P. H. (1992). Artificial Intelligence Third Edition. Addison-Wesley.

[7] Baader, F. , & Snyder, W. (2001) Unification Theory. Chapter eight of Handbook of Automated Reasoning. Springer Verlag, Berlin. Retrieved from: http://www.bu.edu/cs/wayne-snyder/

[8] Lecture 26: Type Inference and Unification. Retrieved from: http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec26-type-inference/type-inference.htm

[9] Hoder, K. , & Voronkov, A. (September 15-18, 2009) *Comparing Unification Algorithms in First-Order Theorem Proving.* Paper presented at the KI 2009: Advances in Artificial Intelligence, 32nd Annual German Conference on AI, Paderborn, Germany .