Comparative Study of Search Strategies for Term-Matching and Unification Based Resolution in Prolog

Yue Li

January 2017

Abstract

Motivation We explore the method of implementing structural resolution, in Prolog, in the form of a meta-interpreter. Structural resolution is the inference rule used in Coalgebraic Logic Programming (CoALP). Existing Prolog implementation of CoALP focuses on the coinductive reasoning aspect and passes all inductive predicates to Prolog system instead of processing them with structural resolution. By implementing structural resolution itself, more observation could be done on inductive behaviour of structural resolution, and the implementation can be used as a basis to incorporate future enhanced coinductive reasoning mechanism. The form of meta-interpreter suffices for current stage of research.

Structural resolution Structural resolution features reducing goals mainly by rewriting rather than general unification as done by traditional SLD resolution. When modelled by rewriting trees, structural resolution is both inductively sound and complete. Relying on the notion of observational productivity, structural resolution could be used to do coinductive reasoning and draw conclusion about both infinite cyclic and infinite acyclic terms within finite steps of inference. Existing coinductive operational semantics for logic programming is co-SLD, which uses SLD resolution with loop detection and is good at reasoning about infinite cyclic terms. The application potential of structural resolution is programming language type inference.

Result An implementation method for rewriting tree based, inductively sound and complete structural resolution has been worked out and passed test. With help from the implementation, an execution model of the structural resolution meta-interpreter is formalised, which uses left-first computation rule to replace non-deterministic goal predicate choice, and uses matching-first sequential search rule and backtracking to replace non-deterministic clause choice.

Conclusion The reported task was accomplished as result of a blend of theory study and Prolog programming exercise. The development strategy — approximation turns out to be helpful and inspiring. Future wok will be focused on exploring combining co-SLD style loop detection with structural resolution, and using this tool to solve type inference problems.

Acknowledgement

I acknowledge the support from people who are more or less directly related to the reported work. The following list is not exhaustive.

Thanks Dr. Ekaterina Komendantskaya for excellent fulfilment of her supervising role, with warm support and care, patient tutorials, and informative suggestions and advice.
Thanks Dr. Martin Schmitt for his tutorials.
Thanks EPSRC for funding the work.

Contents

List of Figures

Chapter 1

Introduction

The method of implementing structural resolution, in Prolog, in the form of a meta-interpreter, is the focus of this report. To motivate the work, we have a look at how it fits in the bigger picture of the field of logic programming. Then the researching work done by the author before writing this report is summarized, followed by discussion of what is learnt academically during the period of doing the reported task. In the end comes the summary of rest chapters of the report.

1.1 The Bigger Picture

Structural resolution is recent contribution to the theory and practice of logic programming, which is the branch of the science of computer programming languages, that deals with using first order logic as a programming language.

The motivation of logic programming, is achieving declarative, rather than imperative, programming style, where the programmer's work is to state the problem, and the computer is expected to give the solution. In other words, letting computers do automated reasoning [2, 3].

In 1970s and 1980s, fundamental work was done on the theory and implementation of logic programming. On theoretical aspect, Robert Kowalski and his colleagues specified its syntax and semantics [8, 9, 10, 11], as necessary for the creation of any computer programming language. On the other hand, Alain Colmerauer and his team implemented logic programming and gave birth to the first version of the language Prolog. The implementation of Prolog is further developed by Robert Kowalski's PhD student, David Warren, resulting in faster speed that is comparable to the functional language Lisp [2].

Based on the above mentioned initial work and exposed interesting issues, further logic programming research began and branched into diverse directions. One of the branches studies perpetual processes in logic programming [8], where logical inference does not terminate. This happens, for example, when one does arithmetic and uses the long division algorithm to calculate the decimal result of one divided by three.

Early stage exploration of perpetual processes focused on understanding the nature of the partial answers produced by some initial finite fragment of the perpetual computation [8, 12, 13].

In 2000s and early 2010s, using the latest development of Prolog system [5, 6, 7] and mathematics [33], the perpetual process study moved on with creation of new theory and software implementations, and discovering promising fields of application. These recent research runs along two threads as follows.

One thread is co-SLD, initiated by Gopal Gupta and his colleagues, extending traditional SLD resolution based logic programming with coinductive reasoning [14, 15, 17], under the mathematical theory of non-well-founded sets. Their approach exploits loop detection, so that logical conclusion could be made correctly, with finite inference, about infinite cyclic data structures, which traditionally caused non-terminating inference. The new result sees its potential in various aspects of computer science, such as model checking, modelling complex real-time systems and programming language type checking/inference [16, 22].

The other thread is CoALP, initiated by Ekaterina Komendantskaya and her colleagues, where a new machine oriented logical inference rule for logic programming, called structural resolution, is proposed as an alternative to SLD resolution, and coinductive reasoning is used to make correct conclusions about both infinite cyclic and infinite acyclic data structures within finite inference, under the extra condition of productivity [19, 18, 20]. CoALP is under the mathematical theory of coalgebra. Currently its application in programming language type checking and type inference is under active investigation [21, 22].

In the recent CoALP workshop, i.e. Workshop on Coalgebra, Horn Clause Logic Programming and Types, held on 28-29 November 2016 in Edinburgh, U.K., Martin Schmidt demonstrated the latest version of implementation of CoALP in Prolog in the form of a meta-interpreter [23]. To help focusing their attention on developing the coinductive reasoning aspect, the developers decided to pass all inductive predicates to the SLD meta-interpreter, instead of processing them with structural resolution, and only let coinductive predicates be processed by structural resolution. Since structural resolution is designed as an inference rule for both inductive and coinductive reasoning [19, 18], it would be helpful to implement structural resolution itself as a stand alone inference engine, then more observation could be made on its behaviour, and decision on further development of the structural resolution inference engine could be informed.

On 11 November 2016, Ekaterina Komendantskaya formally commissioned Yue, who is her first year PhD student and also the author of this report, to implement structural resolution and explore combining it with co-SLD style loop detection. Nevertheless, this direction had been communicated to Yue as early as before Yue started his PhD in September 2016. On 9 December 2016 Yue introduced his implementation to Ekaterina Komendantskaya, the latter then suggested that Yue could start writing a report for this part of the task.

1.2 Overview of Researching Activity

The work that leads to this report spanned about three months, starting on 14 September 2016, until entering report writing stage on 10 December 2016. The period could be further divided into 3 phases according to different focus of the work at different time.

The first phase, 14 September 2016 – 14 October 2016, was the time when the author learnt introductory level Prolog programming using [1].

The second phase, 15 October 2016 – 11 November 2016, was used on reading relevant background theories of traditional and coinductive logic programming. The author consulted, beneficially, [8, 9, 10, 11, 12, 13, 24, 25, 26, 27, 28, 29] for traditional logic programming semantics, lattice theory and fixed point theory. Attempt was also made to understand the theory on non-well-founded sets and coinductive reasoning, and [30, 31, 32, 33] were consulted but only increased the author's knowledge on standard set theory.

In the third phase, 12 November 2016 – 09 December 2016, effort was devoted for improving the author's understanding of structural resolution and derivation process, and skill of Prolog programming. [2, 4, 18, 19] are main resource used in this phase. This phase culminated in successful implementation of structural resolution in Prolog as a meta-interpreter. Exploration of combining structural resolution with coSLD style loop detection also started.

1.3 Things Learnt from Doing the Task

There are several academic things I learnt from doing the task. They are summarized as follows.

Prolog programming From basic skills to deeper understanding of Prolog's behaviour, particularly *variable value sharing*, which underlies Prolog's ability to systematically instantiate variables in the search tree, corresponding to systematic application of unifier/matcher in theory of resolution.

Semantics of logic programming This refers to operational and declarative semantics of inductive and co-inductive logic programming. I appreciated the complete Herbrand universe which exhausts the expressive power of first order language, but also leaving a lot to be solved by computing researchers in order to find a good operational semantics to navigate in that universe. I acknowledged that sequential search and backtracking are elegant response to non-deterministic choice in SLD resolution.

CoALP Which is current research project on co-algebraic logic programming. It is new and provides many research opportunities such as exploring co-inductive operational semantics and application in programming language type inference.

1.4 Rest of the Report

Chapter-2 There are several preliminary first order logic and Prolog concepts that readers should familiarise themselves with, in order to understand the work described in this report. Here we see some exemplified definitions with their relevance to the reported work explained.

Chapter-3 We define inductively incomplete structural resolution and replace the non-deterministic choices involved by a computation rule, a search rule and backtracking method, as our step towards its implementation, and the eventual implementation of inductively complete structural resolution.

Chapter-4 We define rewriting tree and use it to model *inductively complete structural resolution*. Then we explore extending the computation rule, search rule and backtracking method we developed for inductively incomplete structural resolution to inductively complete structural resolution.

Chapter-5 We see how structural resolution is implemented in Prolog. Each section of this chapter introduces a meta-interpreter that represents a major step of approximation. We indicate how to learn these meta-interpreters, and how they came into being. The detailed codes are attached in appendix.

Chapter-6 We conclude the report by talking about the achievement by the author from doing the reported work, the highlight of the work, what can be improved, and further work.

Chapter 2

Preliminaries

There are several preliminary first order logic and Prolog concepts that readers should familiarise themselves with, in order to understand the work described in this report. Here we see some exemplified definitions with their relevance to the reported work explained. If in any doubt, readers are suggested to consult [24, 8, 9, 11, 1, 2] for more details.

2.1 Definite Programs

The computer programs that structural resolution works on are *definite programs*. If we make an analogy between a logic program and an essay, then *first order terms* of the definite program corresponds to the vocabulary of the essay, *definite clauses* and *definite goals* corresponds to the sentences. The emphasized concepts are formally defined as follows.

Definition 1 (First Order Terms)**.** A constant symbol, such as a,b,c,..., is a first order term (we use "term" for short). A variable symbol, such as A,B,C,..., is a term. $f(t_1, \ldots, t_n)$ $(n > 0)$ is a term if $t_1, \ldots, t_n$ are terms and f is a constant symbol, called functor The arity of a functor means the number of parameters that that functor can have. A functor f with arity n is denoted as f/n. A term is a complex term if it is of the form $f(t_1, \ldots, t_n)$ $(n > 0)$. If a term is a constant symbol or a complex term, then it is non-variable.

Terms are used to refer to objects and relations in the language of logic. Mnemonic names are usually chosen for terms in Prolog programming to enhance code readability. For example, `N` is used as a variable that ranges over natural numbers; constant `nil` refers to an empty list; complex term `cons(a, cons(b,nil))` denotes a list `[a,b]`. The arity of functor `cons` is 2. Term `sum(N1,N2,N3)` means the relation that `N3` is the sum of `N2, N3`.

Definition 2 (Definite Clause)**.** If term h and $t_1, \ldots, t_n$ are non-variables, then $h \leftarrow t_1, \ldots, t_n$ $(n > 0)$ is a definite clause, h is the head (or conclusion) of the clause, $t_1, \ldots, t_n$ is the body (or premises) of the clause, $h, t_1, \ldots, t_n$ are called predicates, each of which is a predicate. The commas delimiting the body of the clause is right associative. A single non-variable term t is a definite clause whose head is t, body is term "true" in Prolog or empty in logic, and t is called a predicate. Variables in definite clauses are implicitly universally quantified.

A definite clause expresses the meaning that if conjunctively all predicates in the body of the clause holds, then the head holds.

Definition 3 (Definite Goal)**.** If $t_1, \ldots, t_n$ are non-variable terms, then $\leftarrow t_1, \ldots, t_n$ $(n > 1)$ is a conjunctive definite goal. $\leftarrow t_1$ is an atomic definite goal. Removing t_1 from atomic goal $\leftarrow t_1$ gives an empty goal denoted by $\perp$ (bottom symbol). Variables in definite goals are implicitly universally quantified. The body of a goal is the right hand side of the left arrow ($\leftarrow$).

A definite goal expresses the meaning that all predicates in the body of the goal does not hold conjunctively.

Definition 4 (Definite Program)**.** A definite program is a non-empty set of definite clauses, where for every predicate, there shall be at least one definite clause whose head shares the same functor and arity as the predicate.

The definition of definite program requires that all predicates in a definite program be defined [11].

Example 5. The following is a small definite program written in Prolog. We name it as P_{sum}.

```
1| sum(0, N, N).
2| sum(s(A), B, s(C)) :- sum(A, B, C).
```

This program has two definite clauses, `sum(0,N,N).` and `sum(s(A), B, s(C)) :- sum(A, B, C).` The first clause says: for all N, the sum of 0 and N is N. The second clause says: for all A, B and C, if the sum of A and B is C, then the sum of successor of A, and B, is successor of C. Successor is defined for all natural numbers, e.g. 1 is 0's successor, 2 is 1's successor, and so on. In the second clause, `sum(s(A), B, s(C))` is head, `sum(A, B, C)` is body, colon dash (`:-`) is Prolog's notation for the left arrow ($\leftarrow$) in a definite clause.

A definite goal for P_{sum} could be `?- sum(s(0), s(0), N).` This goal says: for all N, N is not the sum of 1 and 1. Question mark dash (`?-`) is Prolog's notation for left arrow in a definite goal. Using Prolog's operational semantics, which we will introduce later as *SLD resolution*, the goal could be refuted by Prolog interpreter and a counter example `N=s(s(0))` be given.

Instantiation is a logic inference rule, whose special case called substitution reduction is used in structural resolution.

Definition 6 (Instantiation)**.** Given a definite clause or goal G and any substitution θ, it logically follows, by instantiation, that $G\theta$.

Example 7. Given the clause `sum(0,N,N)` and a substitution {`N/s(0)`}, we could use instantiation rule to derive a logical conclusion that `sum(0,s(0),s(0))`.

2.2 SLD Resolution

Given a definite program and a definite goal, the logic program interpreter resolves the goal using some inference rule and the information provided in the form of the definite program. *SLD resolution* is the most common inference rule used by modern logic program interpreters [24]. Structural resolution is an

alternative inference rule, but we will see later that good understanding of SLD resolution helps understanding structural resolution. *Unification* plays a key role in both SLD resolution and structural resolution as the mechanism to choose clauses to reduce a goal, and to pass data between the goal and program clauses. We see some preparatory definitions: *substitution* and *substitution composition*, then we define unification and SLD resolution. A special case of unification—*term matching* is also defined, due to its use in structural resolution.

Definition 8 (Substitution)**.** X to t, denoted as X/t, is a variable binding, if X is a variable, t is a term. The set $\{V_1/t_1, \ldots, V_n/t_n\}(n \geq 0)$ of variable bindings is a substitution if $V_1, \ldots, V_n$ are pairwise distinct, and for any $1 \leq i, j \leq n$, t_i does not contain symbol V_j. Applying substitution σ to t, where t is either a term , a definite goal or a definite clause, is denoted by $t\sigma$, meaning replacing variables in t according to variable bindings in σ. When n=0, the substitution is empty, denoted by ϵ.

Definition 9 (Substitution Composition)**.** Given two substitutions $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$ and $\sigma = \{X_1/r_1, \ldots, X_m/r_m\}$, their composition δ, which is also a substitution denoted as $\delta = \theta\sigma$, is given by $\delta = \theta' \cup \sigma'$, where $\theta' = \{V_1/(t_1\sigma), \ldots, V_n/(t_n\sigma)\}$, and $\sigma' = \{(X/r) \mid (X/r) \in \sigma, X \notin \{V_1, \ldots, V_n\}\}$.

Definition 10 (Unification)**.** Given two terms $t_1,\ t_2$, unification is the process of finding a substitution σ such that $t_1\sigma$ is identical to $t_2\sigma$, noted as $t_1\sigma = t_2\sigma$. Then σ is called the unifier for $t_1,\ t_2$. Further, σ is the *most general unifier* if for any unifier γ of $t_1,\ t_2$, there exist a substitution δ such that $\gamma = \sigma\delta$.

Definition 11 (Term Matching)**.** Given two terms $t_1,\ t_2$, if there exist substitution θ, such that $t_1 = t_1\theta = t_2\theta$, then term t_2 matches against t_1 with matcher θ.

Example 12. Term $(A + B)$ unifies with term $(X + 2)$ with unifier $\{A/X, B/2\}$. Since the unifier does not instantiate variables in $(X + 2)$, we could say

- term $(A + B)$ *matches against* term $(X + 2)$ with matcher $\{A/X, B/2\}$, and
- term $(A + B)$ *subsumes* $(X + 2)$, and
- term $(A + B)$ is *more general* than $(X + 2)$.

Definition 13 (SLD Resolution)**.** Given goal G: $\leftarrow t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_n$ $(n > 0)$ and definite clause H: $h \leftarrow s_1, \ldots, s_n$ $(n \geq 0)$ where $t_i\sigma = h\sigma$, assuming that variables in G and H have been renamed apart, using SLD resolution it logically follows that G_1: $\leftarrow (t_1, \ldots, t_{i-1}, s_1, \ldots, s_n, t_{i+1}, \ldots, t_n)\sigma$ $(n > 0)$. We say G is *reduced* by clause H to G_1, and G_1 is *derived* from G by clause H. The symbolic notation for this, is $G \overset{H}{\rightsquigarrow} G_1$.

Renaming variables from clause H and goal G apart achieves clarity of the meaning of the logical expressions when they are considered together.

Example 14. The P_{sum} program used previously is copied as follows. We pose a query G_0: `?-sum(s(0), s(0), N)` and observe the sequence of SLD reduction. The concepts of determinism and non-determinism are informally introduced.

```
1| sum(0, N, N).
2| sum(s(A), B, s(C)) :- sum(A, B, C).
```

G_0: `?-sum(s(0), s(0), N)`

$\rightsquigarrow$ G_1: `?-sum(0, s(0), C1)`

$\rightsquigarrow$ G_2: $\bot$

The chosen clause and computed unifier in each reduction are as follows.

$G_0 \rightsquigarrow G_1$:

Clause 2: `sum(s(A1), B1, s(C1)) :- sum(A1, B1, C1).`

Unifier: $\sigma_1 = \{$`A1/0,B1/s(0),N/s(C1)`$\}$.

$G_1 \rightsquigarrow G_2$:

Clause 1: `sum(0, N2, N2).`

Unifier: $\sigma_2 = \{$`N2/s(0),C1/s(0)`$\}$.

Whenever a program clause is chosen, its variables are renamed to avoid potential confusion. The empty goal indicates contradiction, so the original goal G_0, saying that for all N, N is not sum of 1 and 1, is successfully refuted. The counter example is given by composition of σ_1 and σ_2: σ_1 binds N to s(C1), σ_2 binds C1 to s(0), so N is eventually bound to s(s(0)), i.e. 2.

In *Example* 14, in each reduction, only one program clause has a head that unifies with the goal, and the goal is always atomic goal. Absence of choices makes refuting G_0 in *Example* 14 a *deterministic* process.

When using SLD resolution, if there are more than one applicable clauses and/or the goal is conjunctive, we would then need to make choices over which predicate of the conjunctive goal to work on next and/or which one of the applicable clauses to use. As a result, alternative refutation paths exist and refutation becomes *non-deterministic*. Next we will address issues in this aspect.

2.3 Addressing Non-Deterministic Choices in SLD Resolution

We have noticed that there are two types of non-deterministic choices involved in SLD resolution: the choice of a predicate from a conjunctive goal, and the choice of a definite clause from several eligible clauses. The method of making these two types of choices are formally named as *computation rule* and *search rule*, respectively.

Definition 15 (Computation Rule)**.** Given a definite goal $\leftarrow t_1, \ldots, t_i, \ldots, t_n$ ($n > 0$), a computation rule (aka. scheduling policy) chooses a predicate t_i. A left-first computation rule (aka. stack scheduling policy) always chooses t_1. A right-first computation rule always chooses t_n.

Definition 16 (Search Rule)**.** Let t_i be the predicate chosen by a computation rule, and $C_1, \ldots, C_m$ ($m > 0$) are clauses (called *eligible* or *applicable* clauses in this report) that the head of each of them unifies with t_i, a search rule put $C_1, \ldots, C_m$ into order and chooses the most prioritized one in the order.

The definition of SLD resolution allows the two types of choices be non-deterministic. Making such decisions is also involved in implementing structural resolution.

It is proved that SLD resolution is independent from the computation rule [8]. This ensures validity of Prolog's implementers' decision on using the left-first computation rule.

Prolog search rule sorts applicable clauses according to the order in which the clauses are defined in the definite program, and chooses the earliest defined one. Such a search rule is called *sequential search*. Implementers of Prolog decided to replace non-deterministic clause choice in SLD resolution by sequential search and *back tracking*.[2]. We discuss more about this decision later. Next we informally introduce backtracking and its notation in Example 17 below.

Example 17. This example shows that the search rule determines the choice of clause for a reduction, when more than one clauses are applicable. Prolog's sequential search rule is adopted. The notions of *choice points* and *backtracking* are informally introduced. Program P_{member} is given as follows.

```
1| member(X,[X|_]).
2| member(X,[_|T]) :- member(X,T).
```

Clause 1 says for all object X, X is a member of a list whose head is X, regardless of the tail of the list. Clause 2 says for all object X and list T, if X is a member of T, then X is also the member of a list whose tail is T, regardless of the head.

We pose the query G_0: `?- member(X,[a,b])` saying that for all X, X is not a member of list `[a,b]`. The sequence of SLD resolution for G_0 is given below.

G_0: `?- member(X,[a,b])` $\quad\quad$ $©_0(\underline{1},2)$

$\rightsquigarrow G_1$: $\bot$

The chosen clause and computed unifier involved in the reduction are given as follows.

$G_0 \rightsquigarrow G_1$:

Clause 1: `member(X1,[X1|T1]).`

Unifier: θ=`{X1/a,X/a,T1/[b]}.`

Above, the copyright symbol © with subscription 0 and parameters ($\underline{1}$, 2) where 1 is underlined, marks a *choice point*—a juncture in the sequence of reduction where more than one clauses are applicable, and the parameters are a list of indices of applicable clauses, sorted in order according to the search rule. The underlined index is the index of the currently chosen clause. Copyright symbol is used because it looks like a point and the letter C in the symbol can be interpreted as "Choice". Readers should read $©_0$ as "C zero" or "choice point zero".

To reduce goal G_0 using SLD resolution, both Clause 1 and 2 are applicable, and the sequential search rule chooses Clause 1. $G_1 : \bot$ indicates successful refutation of G_0, with counter example given by θ, where X is bound to a.

Upon successful refutation, Prolog *backtracks* choice points to find alternative path of refutation. To backtrack at a choice point ©, variable bindings made due to the previously chosen clause are undone, and the next clause in the waiting list, i.e. the parameter list of choice point © is chosen.

To backtrack at ©$_0$, the bindings given as θ are undone, and the control goes back to G_0, SLD resolution is performed again, but this time Clause 2 is chosen. The new sequence of SLD resolution is given below, followed by the detail of chosen clauses and computed unifiers in each reduction.

G_0: `?- member(X,[a,b])` ©$_0$(1,2)

$\rightsquigarrow$ G_1: `?- member(X,[b])` ©$_1$(1,2)

$\rightsquigarrow$ G_2: $\bot$

$G_0 \rightsquigarrow G_1$:

Clause 2: `member(X2,[H2|T2]):-member(X2,T2).`

Unifier: θ_1=`{X2/X,H2/a,T2/[b]}.`

$G_1 \rightsquigarrow G_2$:

Clause 1: `member(X3,[X3|T3]).`

Unifier: θ_2=`{X3/b,X/b,T3/[]}.`

$G_2 : \bot$ indicates successful refutation of the query. The counter example is given by $(\theta_1\theta_2)$ which binds X to b.

Backtracking could now be performed at choice point ©$_1$, undoing θ_2 and the control goes back to G_1. The new sequence of SLD resolution is given below, followed by the detail of chosen clauses and computed unifiers in each reduction.

G_0: `?- member(X,[a,b])` ©$_0$(1,2)

$\rightsquigarrow$ G_1: `?- member(X,[b])` ©$_1$(1,2)

$\rightsquigarrow$ G_2: `?- member(X,[])`

†

$G_0 \rightsquigarrow G_1$:

Clause 2: `member(X2,[H2|T2]):-member(X2,T2).`

Unifier: θ_1=`{X2/X,H2/a,T2/[b]}.`

$G_1 \rightsquigarrow G_2$:

Clause 2: `member(X4,[H4|T4]):-member(X4,T4).`

Unifier: θ_2=`{X4/X,H4/b,T4/[]}.`

No program clause has a head that unifies with G_2, so G_2 can't be further reduced, indicating that G_2 cannot be refuted using information from P_{member}. We say SLD resolution fails, and this is marked by the dagger (†) symbol.

We have seen that Prolog maintains all choice points in a stack. When a new choice point © is created, it pushes © onto the stack. When back tracking, it pops the top-most choice point from the stack.

It is interesting to think about why random search was not used to implement the non-deterministic clause choice when implementing SLD resolution. Random search chooses arbitrary one clause from all applicable clauses for goal reduction.

The author's answer is that random search cannot traverse all possible refutation paths in as systematic a way as ordered search, making the behaviour of the program harder to predict and we could loss control over the program even when we need to control it.

At last, (we get out of the above discussion) it is assumed the readers have good understanding of *least/greatest Herbrand models* and the definition of *soundness* and *completeness* of SLD resolution.

2.4 Conclusion

We introduced definite logic programs, definitions related to SLD resolution and its implementation decisions about non-deterministic choices in terms of left-first computation rule, sequential search rule and backtracking. A similar structure of writing will be followed in later chapters, where we will define structural resolution and then discuss its implementation decisions on non-deterministic choices. We also mentioned the instantiation rule and its relevance to structural solution. Finally we assumed the reader's understanding on Herbrand models and the soundness and completeness property of an inference rule.

Chapter 3

Inductively Incomplete Structural Resolution

We define inductively incomplete structural resolution and replace the non-deterministic choices involved by a computation rule, a search rule and backtracking method, as our step towards its implementation, and the eventual implementation of inductively complete structural resolution. Hereafter within this chapter we use "structural resolution" to refer to "inductively incomplete structural resolution".

3.1 Components and Definition of Structural resolution

We defining components of structural resolution, which are *rewriting reduction* and *substitution reduction*, as follows.

Definition 18 (Rewriting Reduction)**.** Given definite goal G: $\leftarrow t_1, \ldots, t_i, \ldots, t_n$ $(n > 0)$ and definite clause $h \leftarrow s_1, \ldots, s_m$ $(m \geq 0)$ where $t_i = t_i\sigma = h\sigma$, by rewriting reduction it logically follows goal G' that

$$\leftarrow t_1, \ldots, t_{i-1}, (s_1, \ldots, s_m)\sigma, t_{i+1}, \ldots, t_n$$

Since σ is a matcher, not influencing variables in goal G, G' could be also written in the form

$$\leftarrow (t_1, \ldots, t_{i-1}, s_1, \ldots, s_m, t_{i+1}, \ldots, t_n)\sigma$$

The term "rewriting" in rewriting reduction should be understood as in the context of arithmetical expression simplification [34]. For example, the arithmetic expression $(2+2)\times 3$ can be simplified as 4×3, which is then further simplified as 12. Such simplification is formalised as *term rewriting*, where term $(2 + 2)$ is rewritten as 4, and 4×3 is rewritten as 12. Inspecting the rewriting of term $(2 + 2)$ as term 4 in more detail, it could be asserted that this process uses the rewriting rule that terms of the form $(A+B)$ can be rewritten as terms of form C, where both $A,\ B$ in our example are instantiated to 2 after $(A + B)$ matches against $(2 + 2)$, and C is instantiated to the sum of values to which $A,\ B$ are instantiated.

Rewriting reduction is a special case of SLD resolution. That the head of the chosen clause is more general than the chosen predicate from the goal, is the distinctive feature of rewriting reduction compared with other cases of SLD resolution. So it is conceptually correct to call "rewriting reduction" as "rewriting SLD resolution".

Example 19. Consider the goal `?- like(yue,ticks)` that says "Yue doesn't like ticks", and the clause `like(yue, X) :- friendly(X)` that says "for all X if X is friendly then Yue likes X". Using SLD resolution it logically follows from the goal and the clause that `?-friendly(ticks)`, which says "ticks are not friendly". The SLD resolution in this example is rewriting reduction, since the head `like(yue,X)` of the chosen clause is more general that the chosen predicate `like(yue,ticks)` from the goal.

Example 20. From P_{sum}'s clause `sum(0,N,N)` and goal `?-sum(0, s(0), C)`, using SLD resolution, the empty goal $\perp$ logically follows. The head `sum(0,N,N)` of the clause and the chosen predicate `sum(0, s(0), C)` from the goal have a unifier $\sigma = \{N/s(0), C/s(0)\}$. Unifier σ instantiates variable `C` in the goal predicate, so here SLD resolution is not qualified as rewriting reduction.

Definition 21 (Substitution Reduction)**.** Given $\leftarrow t_1, \ldots, t_i, \ldots, t_n$ $(n > 0)$ and $h \leftarrow s_1, \ldots, s_n$ $(n \geq 0)$ where $t_i\sigma = h\sigma$, using instantiation it logically follows that $\leftarrow (t_1, \ldots, t_i, \ldots, t_n)\sigma$ $(n > 0)$.

Substitution reduction is a kind of instantiation where terms that can instantiate variables in the goal are restricted to be terms from the head of the chosen clause.

Example 22. The goal G = `?-sum(0, s(0), C)` says for all C, C is not the sum of 0 and s(0) (i.e. 1). Unifying goal G with the clause H `sum(0,N,N)` gives unifier $\sigma = \{N/s(0), C/s(0)\}$. If we use instantiation rule and apply σ to instantiate G we get a logical consequence G_1 of G, where G_1 = `?-sum(0, s(0), s(0))`, saying that 1 is not the sum of 0 and 1. Since the unifier σ comes from unification of the goal and head of a clause from P_{sum}, the instantiation rule we just used is also substitution reduction of goal G with respect to clause H.

We could repeatedly apply inference rules to make a series of logical reduction. For instance, given a goal and a logic program, we could commit to using rewriting reduction to reduce the goal and any sub-goal created as a logical consequence. This way of reducing goals is used in *structural resolution*. We first define some symbolic notation as preparation, then we define structural resolution.

Definition 23 (Right arrow ($\rightarrow$), Hook Right Arrow ($\hookrightarrow$))**.** Let rewriting reduction be denoted by right arrow ($\rightarrow$), which can be further annotated as $G \xrightarrow{C} G_1$ to mean goal G is reduced by clause C to goal G_1 using rewriting reduction. Let substitution reduction be denoted by hook right arrow ($\hookrightarrow$), which can be further annotated as $G \overset{C}{\hookrightarrow} G_1$ to mean goal G is reduced by clause C to goal G_1 using substitution reduction. For both $\rightarrow$ and $\hookrightarrow$, let $\rightarrow^n$ (similarly, $\hookrightarrow^n$) mean applying $\rightarrow$ (respectively $\hookrightarrow$) for some times as indicated by n: if n = 1,2,3,..., it means applying for *at most* n times; if n = μ, it means applying until the rule is no longer applicable.

Definition 24 (Structural Resolution)**.** Composition of $\rightarrow^{\mu}$ and $\hookrightarrow^{1}$, denoted as $\rightarrow^{\mu} \circ \hookrightarrow^{1}$, is structural resolution. It could also be called structural reduction in the context that goal G is reduced to goal G_1 using structural resolution: $G \rightarrow^{\mu} \circ \hookrightarrow^{1} G_1$.

Structural resolution is defined as committing to the use of rewriting reduction to reduce goals *and* switching to substitution reduction for *one time* at the juncture where rewriting reduction is not applicable. Let us note that structural resolution is not, as its name may suggest, pure resolution, but aggregation of resolution and instantiation. It is neither a one-step reduction as SLD resolution or instantiation, but a series of reductions chained together and regarded as a single unit, and this single unit could usually, as we will see later, be repeatedly applied to a goal given a logic program.

A series of structural reductions is of the form

$$G_0 \ (\rightarrow^{\mu} \circ \hookrightarrow^{1}) \ G_1 \ (\rightarrow^{\mu} \circ \hookrightarrow^{1}) \ G_2 \ (\rightarrow^{\mu} \circ \hookrightarrow^{1}) \ G_3 \ \cdots$$

where details of reductions in each stage from G_n to G_{n+1} is omitted for a conceptual level overview. At operational level, for instance, the structural reduction from G_0 to G_1 is performed in the form

$$G_0 \rightarrow G_0^1 \rightarrow \cdots \rightarrow G_0^x \hookrightarrow G_1$$

where there shall be no program clause whose head subsumes G_0^x because substitution reduction is used to reduce G_0^x.

It is time to see how structural resolution works with definite programs. We not yet mentioned the computation rule and search rule that we adopt to perform structural resolution, but we could still see three examples (*Example* 25, *Example* 26 and *Example* 27) where computation rule and search rule are immaterial.

Example 25. The P_{sum} program used previously is copied as follows. We pose a query G_0: `?-sum(s(0), s(0), N)` and observe the sequence of structural reduction.

```
1| sum(0, N, N).
2| sum(s(A), B, s(C)) :- sum(A, B, C).
```

G_0: `?-sum(s(0), s(0), N)`

$\hookrightarrow G_1$: `?-sum(s(0), s(0), s(C1))`

$\rightarrow G_1^1$: `?-sum(0, s(0), C1)`

$\hookrightarrow G_2$: `?-sum(0, s(0), s(0))`

$\rightarrow G_2^1$: $\bot$

The choice of clauses and computed matcher/unifier in each reduction are given below.

$G_0 \hookrightarrow G_1$:

Clause 2: `sum(s(A1), B1, s(C1)) :- sum(A1, B1, C1)`

Unifier: θ_1=`{A1/0,B1/s(0),N/s(C1)}`.

$G_1 \rightarrow G_1^1$:

Clause 2: `sum(s(A2), B2, s(C2)) :- sum(A2, B2, C2).`

Matcher: {`A2/0,B2/s(0),C2/C1`}.

$G_1^1 \hookrightarrow G_2$:
Clause 1: `sum(0, N3, N3)`

Unifier: θ_2= {`N3/s(0),C1/s(0)`}.

$G_2 \rightarrow G_2^1$:

Clause 1: `sum(0, N4, N4).`

Matcher: {`N4/s(0)`}.

Since there is no clause whose head subsumes G_0, rewriting reduction shall be applied 0 times, i.e., not applied at all. We perform substitution reduction on G_0. Only the head of Clause 2 unifies with G_0. Here we use renamed version of Clause 2: `sum(s(A1), B1, s(C1)) :- sum(A1, B1, C1)` so the unifier is θ_1={`A1/0,B1/s(0),N/s(C1)`}. $G_1 = G_0\theta_1$.

We shall reduce G_1 by rewriting reduction. Only head of Clause 2 matches against G_1. We use renamed copy of Clause 2: `sum(s(A2), B2, s(C2)) :- sum(A2, B2, C2)`. The matcher is {`A2/0,B2/s(0),C2/C1`}, the derived goal G_1^1 is `?-sum(0,s(0),C1)`.

G_1^1 can no longer be reduced by rewriting, we choose the only applicable Clause 1, being renamed as `sum(0, N3, N3)`, for substitution reduction. The unifier is θ_2= {`N3/s(0),C1/s(0)`}. The derived goal G_2 is `?-sum(0, s(0), s(0))`.

G_2 can be reduced by rewriting reduction. Clause 1 is the only clause with its head matches against G_2. With the variables renamed, Clause 1 is `sum(0, N4, N4)` and the matcher is {`N4/s(0)`}. Recall that the body of `sum(0, N4, N4)` is defined in logic to be empty, so $\perp$, the empty goal, is derived.

The empty goal indicates contradiction, so the original goal G_0, saying that for all N, N is not sum of 1 and 1, is successfully refuted. The counter example is given by composition of θ_1 and θ_2: θ_1 binds N to s(C1), θ_2 binds C1 to s(0), so N is eventually bound to s(s(0)), i.e. 2.

Example 26. In *Example* 25, rewriting reduction is performed only once in each stage of structural reduction. In this example, we see that it could be applied for more times in a single stage. The program P_{append} is:

```
1| append([],L,L).
2| append([H|A],B,[H|C]) :- append(A,B,C).
```

P_{append} is used to append two lists. Appending `[a,b]` with `[c,d]` gives `[a,b,c,d]`. Clause 1 says for all L, appending empty list `[]` to list L gives list L unchanged. Clause 2 says if append list A to list B gives list C, then prefix list A with element H, and append it to list B, gives a list that is obtained by prefixing element H to list C.

We pose query G_0: `?-append([a,b,c],[d],[a,b,c,X])`, which says for all X, appending `[a,b,c]` with `[d]` does not give `[a,b,c,X]`. The sequence of structural resolution for query G_0 is given below.

G_0: `?-append([a,b,c],[d],[a,b,c,X])`

$\rightarrow G_0^1$: `?-append([b,c],[d],[b,c,X])`

$\rightarrow G_0^2$: `?-append([c],[d],[c,X])`

$\rightarrow G_0^3$: `?-append([],[d],[X])`

$\hookrightarrow G_1$: `?-append([],[d],[d])`

$\rightarrow G_1^1$: $\bot$

The chosen clause and computed matcher/unifier involved in each reduction are given as follows.

$G_0 \rightarrow G_0^1$:

Clause 2: `append([H1|A1],B1,[H1|C1]) :- append(A1,B1,C1).`

Matcher: {`H1/a,A1/[b,c],B1/[d],C1/[b,c,X]`}.

$G_0^1 \rightarrow G_0^2$:

Clause 2: `append([H2|A2],B2,[H2|C2]) :- append(A2,B2,C2).`

Matcher: {`H2/b,A2/[c],B2/[d],C2/[c,X]`}.

$G_0^2 \rightarrow G_0^3$:
Clause 2: `append([H3|A3],B3,[H3|C3]) :- append(A3,B3,C3).`

Matcher: {`H3/c,A3/[],B3/[d],C3/[X]`}.

$G_0^3 \hookrightarrow G_1$:

Clause 1: `append([],L4,L4).`

Unifier: θ={`L4/[d], X/d`}.

$G_1 \rightarrow G_1^1$:

Clause 1: `append([],L5,L5).`

Matcher: {`L5/[d]`}.

The empty goal indicates successful refutation of the query. The counter example is given by θ where X is bound to d.

Next we see an example with the purpose to distinguish the behaviour of structural resolution from SLD resolution.

Example 27. Program P_{member} and query G_0 used previously in *Example* 17 are copied as follows.

```
1| member(X,[X|_]).
2| member(X,[_|T]) :- member(X,T).
```

G_0: `?- member(X,[a,b]).`

We have seen in *Example* 17 that SLD resolution can refute G_0 and gave counter example X = a, and it can give another counter example X = b with back tracking.

The sequence of structural resolution for G_0 is given below.

G_0: `?- member(X,[a,b])`

$\rightarrow G_0^1$: `?- member(X,[b])`

$\rightarrow G_0^2$: `?- member(X,[])`

†

The chosen clause and computed matcher involved in each reduction are given as follows.

$G_0 \rightarrow G_0^1$:

Clause 2: `member(X1,[H1|T1]) :- member(X1,T1).`

Matcher: {`X1/X,H1/a,T1/[b]`}.

$G_0^1 \rightarrow G_0^2$:

Clause 2: `member(X2,[H2|T2]) :- member(X2,T2).`

Matcher: {`X2/X,H2/b,T2/[]`}.

In each rewriting reduction step, Clause 2 is the only clause that matches the goal. In the end the goal G_0^2 can neither be reduced by rewriting nor by substitution reduction. We say structural reduction *fails* to refute the goal. A dagger symbol (†) is put at the juncture, where a non-empty goal can't be reduced, to indicate failure.

We could conclude from *Example* 27 that there exist queries that cannot be refuted by structural resolution but can be refuted by SLD resolution, with regard to a definite program. Formally, this is the contrast between *inductive incompleteness* of structural resolution and *inductive completeness* of SLD resolution. The *completeness of structural resolution* is achieved by adding extra control, formalised as *rewriting tree transition*, which we discuss later.

In *Example* 25, *Example* 26 and *Example* 27, for each reduction there is only one clause applicable, and the goals to be reduced are always atomic, therefore in those examples computation rule and search rule are immaterial and structural resolution is deterministic.

3.2 Non-Deterministic Choices in Structural Resolution

The definition of structural resolution allows two types of choices be made in a non-deterministic way: choice of a predicate from a conjunctive goal (*goal predicate choice*) and choice of a clause from several applicable clauses (*program clause choice*).

It is noticed that SLD resolution and structural resolution share same types of non-deterministic choice. SLD resolution has a successful implementation Prolog, which uses left-first computation rule instead of the non-deterministic goal predicate choice, and uses sequential search plus backtracking to replace non-deterministic program clause choice. These facts suggest that we can start with learning from Prolog and experiment with using Prolog's solution to non-deterministic choice to address non-deterministic choices involved in structural resolution.

Section 3.3 and 3.4 are devoted to describing the above mentioned experiments.

3.3 Computation Rule for Structural Resolution

We fix a search rule and explore the influence that different computation rules have on using structural resolution. Prolog's sequential search rule is used, for it is established in convention and it is simple. We compare the left-first computation rule with the right-first computation rule, by a case study.

We work on program $P_{Bit-Stream}$, given bellow. This program is chosen because it has a clause (3) whose body is conjunctive and recursive, so choice of different computation rules makes a difference.

```
1| bit(0).
2| bit(1).
3| b_str([B|S]) :- bit(B),b_str(S).
```

Clause 1 and 2 say 0 is a bit, and 1 is a bit, respectively. Clause 3 says for all B and S, if B is a bit, S is a bit stream then prefixing B to S gives a bit stream. The query is G_0: `?-b_str(L)`, saying that for all L, L is not a bit stream.

3.3.1 Using Left-First Computation Rule

The sequence of structural resolution is give below.

G_0: `?-b_str(L)`

$\hookrightarrow G_1$: `?-b_str([B1|S1])`

$\quad \rightarrow G_1^1$: `?- bit(B1),b_str(S1).` ©(1,2)

$\hookrightarrow G_2$: `?- bit(0),b_str(S1).`

$\quad \rightarrow G_2^1$: `?- b_str(S1).`

$\hookrightarrow G_3$: `?-b_str([B3|S3])`

$\quad \rightarrow G_3^1$: `?- bit(B3),b_str(S3).` ©(1,2)

$\hookrightarrow G_4$: `?- bit(0),b_str(S3).`

$\quad \rightarrow G_4^1$: `?- b_str(S3).`

$\quad \vdots$

Details of each step about choice of clause and computed unifier/matcher is given below.

$G_0 \hookrightarrow G_1$:

Clause 3: `b_str([B1|S1]) :- bit(B1),b_str(S1).`

Unifier: θ_1=`{L/[B1|S1]}`.

$G_1 \rightarrow G_1^1$:

Clause 3: `b_str([B2|S2]) :- bit(B2),b_str(S2).`

Matcher: `{B2/B1, S2/S1}`.

$G_1^1 \hookrightarrow G_2$:

Clause 1: `bit(0).`

Unifier: θ_2={`B1/0`}.

$G_2 \rightarrow G_2^1$:

Clause 1: `bit(0).`

Matcher: ϵ

$G_2^1 \hookrightarrow G_3$:

Clause 3: `b_str([B3|S3]) :- bit(B3),b_str(S3).`

Unifier: θ_3={`S1/[B3|S3]`}.

$G_3 \rightarrow G_3^1$:

Clause 3: `b_str([B4|S4]) :- bit(B4),b_str(S4).`

Matcher: {`B3/B3, S4/S3`}.

$G_3^1 \hookrightarrow G_4$:

Clause 1: `bit(0).`

Unifier: θ_4={`B3/0`}.

$G_4 \rightarrow G_4^1$:

Clause 1: `bit(0).`

Matcher: ϵ

$\vdots$

We notice that the above sequence of structural resolution does not terminate and G_0, G_2^1, G_4^1 are repeating patterns indicating a cycle that is detectable by co-SLD style goal unification based loop detection.

3.3.2 Using Right-First Computation Rule

The sequence of structural resolution is give below.

G_0: `?-b_str(L)`

$\hookrightarrow G_1$: `?-b_str([B1|S1])`

$\rightarrow G_1^1$: `?- bit(B1),b_str(S1).`

$\hookrightarrow G_2$: `?- bit(B1),b_str([B3|S3]).`

$\rightarrow G_2^1$: `?-bit(B1),bit(B3),b_str(S3).`

$\hookrightarrow G_3$: `?-bit(B1),bit(B3),b_str([B5|S5]).`

$\rightarrow G_3^1$: `?-bit(B1),bit(B3),bit(B5),b_str(S5).`

$\vdots$

Details of each step about choice of clause and computed unifier/matcher is given below.

$G_0 \hookrightarrow G_1$:

Clause 3: `b_str([B1|S1]) :- bit(B1),b_str(S1).`

Unifier: θ_1={`L/[B1|S1]`}.

$G_1 \rightarrow G_1^1$:

Clause 3: `b_str([B2|S2]) :- bit(B2),b_str(S2).`

Matcher: {`B2/B1, S2/S1`}.

$G_1^1 \hookrightarrow G_2$:

Clause 3: `b_str([B3|S3]) :- bit(B3),b_str(S3).`

Unifier: θ_2={`S1/[B3|S3]`}.

$G_2 \rightarrow G_2^1$:

Clause 3: `b_str([B4|S4]) :- bit(B4),b_str(S4).`

Matcher: {`B4/B3, S4/S3`}.

$G_2^1 \hookrightarrow G_3$:

Clause 3: `b_str([B5|S5]) :- bit(B5),b_str(S5).`

Unifier: θ_3={`S3/[B5|S5]`}.

$G_3 \rightarrow G_3^1$:

Clause 3: `b_str([B6|S6]) :- bit(B6),b_str(S6).`

Matcher: {`B6/B5, S6/S5`}.

$\vdots$

We notice that compared with using left-first computation rule, when using right first computation rule, not only the sequence of structural resolution does not terminate, but also the conjunctive goal to be reduced in each reduction is accumulating more and more predicates as reduction goes, getting larger and larger in size. If such growth in size of the goal to be reduced is regarded as a problem, then we should realize that it results from taking the right first computation rule.

3.3.3 Discussion

The problem of growing size of the goal to be reduced, when using right-first computation rule, can be easily solved if Clause 3 is modified to be left-recursive, as Clause `3'` below.

```
3'|  b_str([B|H]) :- b_str(S), bit(B).
```

In Prolog the good practice for writing recursive definite clauses is to make the recursive predicate as to the right end as possible, i.e. writing right-recursive, rather than left-recursive clauses [1]. Such good practice is determined by the design of Prolog, specifically, its left first computation rule.

The author concluded that, the influence on writing logic programs, from choice of computation rule between right-first rule and left-first rule, is merely on the "good practice" of writing recursive definite clauses. He decided to adopt

the left-first computation rule as Prolog does, so people who are used to write in Prolog don't have to change their habit when using implementation of structural resolution.

3.4 Search Rule and Backtracking for Structural Resolution

We address non-deterministic clause choice in structural resolution by exploring whether sequential search and backtracking, as done by Prolog, can be directly used or modified for structural resolution. Readers be reminded that we are now focusing on *inductively incomplete* structural resolution defined as composition of rewriting and substitution reduction.

3.4.1 Search Rule for Structural Resolution

Since structural resolution is aggregation of substitution reduction and rewriting reduction, we distinguish two types of choice points: *rewriting reduction choice points* and *substitution reduction choice points*.

Definition 28 (Choice Points)**.** When performing structural resolution, if at a juncture where rewriting reduction shall be done and there are more than one program clauses eligible for goal reduction, then this juncture constitutes a *rewriting reduction choice point*. Similarly, If at a juncture where substitution reduction has to be done and there are more than one program clauses eligible, then this juncture constitutes a *substitution reduction choice point*.

Previously in *Example* 17 we informally introduced copyright symbol © to mark choice points in a sequence of SLD resolution. Now we modify this marking scheme to accommodate the change when we start to work with structural resolution.

Definition 29 (Symbol for Choice Points)**.** The symbol ©[R] denotes a choice point for rewriting reduction. Letter R of the symbol stands for "Rewriting". The symbol ©[S] denotes a choice point for substitution reduction. Letter S of the symbol stands for "Substitution". The parameter lists of ©[R] and ©[S] are lists of indices of eligible clauses sorted in the same order as the clauses are defined in the program.

The above definitions on choice point mean we use *sequential search* for both types of reduction involved in structural resolution.

In later sections we are going to experiment with the sequential search rule, together with later newly devised backtracking method, in the context of structural resolution, and evaluate their viability as a whole solution to replace the non-deterministic clause choice.

3.4.2 Backtracking in Structural Resolution

Based on sequential search rule and two types of choice points, we are going to explore how we can do backtracking for structural resolution.

We already understand that there is just one type of choice point that Prolog needs to deal with—unification choice points. We also know that Prolog

maintains all choice points in a single stack, and uses that stack to direct backtracking.

While in our context of structural resolution, there are two distinguished types of choice point, can we still use a single stack to manage all choice points, regardless of their type?

We respond to the questions by experiments. To begin with, we study a case where there are only rewriting reduction choice points, exploring whether it is possible to perform backtracking by maintaining all such choice points in a single stack as Prolog does for its unification choice points (Section 3.4.3). Then we attempt to solve the more complexed problem of backtracking with both types of choice point present (Section 3.4.4).

3.4.3 Backtracking Case Study I

We use program P_{rail} which has the property that there are only rewriting reduction choice points but no substitution reduction choice points.

```
1| direct(aberdeen, dundee).
2| direct(dundee, edinburgh).
3| travel(A,C) :- direct(A,C).
4| travel(A,C) :- direct(A,B), travel(B,C).
```

The query is G_0: `?-travel(aberdeen, Where).`

We do sequential search and use a single stack to maintain all choice points.

Refutation Path No. 1

The sequence of structural resolution is as follows.

G_0: `?-travel(aberdeen, Where).` $\quad \copyright_0[\mathrm{R}](\underline{3},4)$

$\quad \rightarrow G_0^1$: `?-direct(aberdeen, Where).`

$\hookrightarrow G_1$: `?-direct(aberdeen, dundee).`

$\quad \rightarrow G_1^1$: $\bot$

The chosen clause and computed unifier/matcher in each reduction are as follows.

$G_0 \rightarrow G_0^1$:

Clause 3: `travel(A1,C1) :- direct(A1,C1).`

Matcher: {`A1/aberdeen, C1/Where`}.

$G_0^1 \hookrightarrow G_1$:

Clause 1: `direct(aberdeen, dundee).`

Unifier: θ_1={`Where/dundee`}.

$G_1 \rightarrow G_1^1$:

Clause 1: `direct(aberdeen, dundee).`

Matcher: ϵ

G_1^1 indicates successful refutation.

Refutation Path No. 2

We start back tracking at the only choice point $©_0$. The sequence of structural resolution is as follows.

G_0: `?-travel(aberdeen, Where).` $©_0$[R](3,4)

$\rightarrow G_0^1$: `?- direct(aberdeen, B2),travel(B2, Where).`

$\hookrightarrow G_1$: `?- direct(aberdeen, dundee),travel(dundee, Where).`

$\rightarrow G_1^1$: `?-travel(dundee, Where).` $©_1$[R](3, 4)

$\rightarrow G_1^2$: `?-direct(dundee, Where).`

$\hookrightarrow G_2$: `?-direct(dundee, edinburgh).`

$\rightarrow G_2^1$: $\perp$

The chosen clause and computed unifier/matcher in each reduction are as follows.

$G_0 \rightarrow G_0^1$:

Clause 4: `travel(A2,C2) :- direct(A2,B2),travel(B2,C2).`

Matcher: {`A2/aberdeen, C2/Where`}.

$G_0^1 \hookrightarrow G_1$:

Clause 1: `direct(aberdeen, dundee).`

Unifier: θ_2={`B2/dundee`}.

$G_1 \rightarrow G_1^1$:

Clause 1: `direct(aberdeen, dundee).`

Matcher: ϵ

$G_1^1 \rightarrow G_1^2$:

Clause 3: `travel(A3,C3) :- direct(A3,C3).`

Matcher: {`A3/dundee, C3/Where`}.

$G_1^2 \hookrightarrow G_2$:

Clause 2: `direct(dundee, edinburgh).`

Unifier: θ_3={`Where/edinburgh`}.

$G_2 \rightarrow G_2^1$:

Clause 2: `direct(dundee, edinburgh).`

Matcher: ϵ

Now the second successful path of refutation has been found.

Refutation Path No. 3

Since choices at latest choice point $©_1$ are yet exhausted, we back track at this point. The full sequence of structural resolution is as follows.

G_0: `?-travel(aberdeen, Where).` $\copyright_0$[R](3,$\underline{4}$)

$\rightarrow G_0^1$: `?- direct(aberdeen, B2),travel(B2, Where).`

$\hookrightarrow G_1$: `?- direct(aberdeen, dundee),travel(dundee, Where).`

$\rightarrow G_1^1$: `?-travel(dundee, Where).` $\copyright_1$[R](3,$\underline{4}$)

$\rightarrow G_1^2$: `?- direct(dundee,B4),travel(B4,Where).`

$\hookrightarrow G_2$: `?- direct(dundee, edinburgh),travel(edinburgh,Where).`

$\rightarrow G_2^1$: `?-travel(edinburgh,Where).` $\copyright_2$[R](3,$\underline{4}$)

$\rightarrow G_2^2$: `?-direct(edinburgh,Where).`

†

The sequence of reductions leading up to G_1^1, where choice $\copyright_1$ is located, is not changed in back tracking w.r.t Refutation Path No. 2. So here only the chosen clause and computed unifier/matcher in each reduction starting from G_1^1 are given, as follows.

$G_1^1 \rightarrow G_1^2$:

Clause 4: `travel(A4,C4) :- direct(A4,B4), travel(B4,C4).`

Matcher: {`A4/dundee, C4/Where`}.

$G_1^2 \hookrightarrow G_2$:

Clause 2: `direct(dundee, edinburgh).`

Unifier: θ_4={`B4/edinburgh`}.

$G_2 \rightarrow G_2^1$:

Clause 2: `direct(dundee, edinburgh).`

Matcher: ϵ

$G_2^1 \rightarrow G_2^2$:

Clause 3: `travel(A5,C5) :- direct(A5,C5).`

Matcher: {`A5/edinburgh, C5/Where`}.

Structural resolution fails to reduce G_2^2.

Refutation Path No. 4

We back track the latest choice point $\copyright_2$. The full sequence of structural resolution is as follows.

G_0: `?-travel(aberdeen, Where).` $\copyright_0$[R](3,$\underline{4}$)

$\rightarrow G_0^1$: `?- direct(aberdeen, B2),travel(B2, Where).`

$\hookrightarrow G_1$: `?- direct(aberdeen, dundee),travel(dundee, Where).`

$\rightarrow G_1^1$: `?-travel(dundee, Where).` $\copyright_1$[R](3,$\underline{4}$)

$\rightarrow G_1^2$: `?- direct(dundee,B4),travel(B4,Where).`

$\hookrightarrow G_2$: `?- direct(dundee, edinburgh),travel(edinburgh,Where).`

$\rightarrow G_2^1$: `?-travel(edinburgh,Where).` $\textcircled{c}_2$[R](3,4)

$\rightarrow G_2^2$: `?- direct(edinburgh,B6),travel(B6,Where).`

†

With respect to Refutation Path No. 3, unchanged is the segment up to G_2^1 where $\textcircled{c}_2$ is located. The chosen clause and computed unifier/matcher in each reduction starting from G_2^1 are as follows.

$G_2^1 \rightarrow G_2^2$:

Clause 3: `travel(A6,C6) :- direct(A6,B6),travel(B6,C6).`

Matcher: {`A6/edinburgh, C6/Where`}.

So far all possible refutation paths are traversed.

Experiment Conclusion

When only rewriting reduction choice points are involved, but no substitution reduction choice points, the combination of Prolog style sequential search and back tracking for structural resolution is viable.

3.4.4 Backtracking Case Study II

The program P_{air} is given as follows, having a blend of rewriting and substitution reduction choice points.

```
1| direct(edinburgh, amsterdam).
2| direct(edinburgh, frankfurt).
3| direct(amsterdam, beijing).
4| direct(frankfurt,beijing).
5| travel(A,C) :- direct(A,C).
6| travel(A,C) :- direct(A,B), travel(B,C).
```

Query G_0: `?- travel(edinburgh, D).`

We use sequential search for both types of choice points. All choice points, regardless of their types, are maintained in the same stack as if they were of the same type, but different symbolic notation are preserved for different types of choice point.

Refutation Path No. 1

G_0: `?- travel(edinburgh, D).` $\textcircled{c}_0$[R](5,6)

$\rightarrow G_0^1$: `?- direct(edinburgh, D)` $\textcircled{c}_1$[S](1,2)

$\hookrightarrow G_1$: `?- direct(edinburgh, amsterdam).`

$\rightarrow G_1^1$: $\perp$

The details of each reduction is give below.

$G_0 \rightarrow G_0^1$:

Clause 5: `travel(A1,C1) :- direct(A1,C1).`

Matcher: {`A1/edinburgh, C1/D`}

$G_0^1 \hookrightarrow G_1$:

Clause 1: `direct(edinburgh, amsterdam).`

Unifier: $\theta_1 = \{$`D/amsterdam`$\}$.

$G_1 \rightarrow G_1^1$:

Clause 1: `direct(edinburgh, amsterdam).`

Matcher: ϵ

Refutation Path No. 2

When performing back tracking, since we are experimenting with not discriminating any one of two types of choice points, the choice points to back track shall be chosen in the reverse order as they came into being, regardless of their type. So we then back track at $©_1$. The sequence of reduction is as follows.

G_0: `?- travel(edinburgh, D).` $©_0$[R](5,6)

$\rightarrow G_0^1$: `?- direct(edinburgh, D)` $©_1$[S](1,2)

$\hookrightarrow G_1$: `?- direct(edinburgh, frankfurt).`

$\rightarrow G_1^1$: $\perp$

It is assumed that readers become familiar with structural resolution, so the details of each reduction are omitted.

Refutation Path No. 3

Back track at $©_0$. Choice point $©_1$ is exhausted and destroyed.

G_0: `?- travel(edinburgh, D).` $©_0$[R](5,6)

$\rightarrow G_0^1$: `?- direct(edinburgh, B2),travel(B2,D)` $©_2$[S](1,2)

$\hookrightarrow G_1$: `?- direct(edinburgh, amsterdam),travel(amsterdam,D).`

$\rightarrow G_1^1$: `?- travel(amsterdam,D).` $©_3$[R](5, 6)

$\rightarrow G_1^2$: `?- direct(amsterdam,D).`

$\hookrightarrow G_2$: `?- direct(amsterdam,beijing).`

$\rightarrow G_2^1$: $\perp$

Refutation Path No. 4

Backtracking at the latest choice point $©_3$.

G_0: `?- travel(edinburgh, D).` $©_0$[R](5,6)

$\rightarrow G_0^1$: `?- direct(edinburgh, B2),travel(B2,D)` $©_2$[S](1,2)

$\hookrightarrow G_1$: `?- direct(edinburgh, amsterdam),travel(amsterdam,D).`

$\rightarrow G_1^1$: `?- travel(amsterdam,D).` $\copyright_3$[R](5,6)

$\rightarrow G_1^2$: `?- direct(amsterdam,B3),travel(B3,D).`

$\hookrightarrow G_2$: `?- direct(amsterdam,beijing),travel(beijing,D).`

$\rightarrow G_2^1$: `?- travel(beijing,D).` $\copyright_4$[R](5,6)

$\rightarrow G_2^2$: `?- direct(beijing,D).`

†

Refutation Path No. 5

Backtracking at the latest choice point $\copyright_4$.

G_0: `?- travel(edinburgh, D).` $\copyright_0$[R](5,6)

$\rightarrow G_0^1$: `?- direct(edinburgh, B2),travel(B2,D)` $\copyright_2$[S](1,2)

$\hookrightarrow G_1$: `?- direct(edinburgh, amsterdam),travel(amsterdam,D).`

$\rightarrow G_1^1$: `?- travel(amsterdam,D).` $\copyright_3$[R](5,6)

$\rightarrow G_1^2$: `?- direct(amsterdam,B3),travel(B3,D).`

$\hookrightarrow G_2$: `?- direct(amsterdam,beijing),travel(beijing,D).`

$\rightarrow G_2^1$: `?- travel(beijing,D).` $\copyright_4$[R](5,6)

$\rightarrow G_2^2$: `?- direct(beijing,B4),travel(B4, D).`

†

Refutation Path No. 6

Backtracking at the latest choice point $\copyright_2$. Choice points $\copyright_3$ and $\copyright_4$ are exhausted and destroyed.

G_0: `?- travel(edinburgh, D).` $\copyright_0$[R](5,6)

$\rightarrow G_0^1$: `?- direct(edinburgh, B2),travel(B2,D)` $\copyright_2$[S](1,2)

$\hookrightarrow G_1$: `?- direct(edinburgh, frankfurt),travel(frankfurt,D).`

$\rightarrow G_1^1$: `?- travel(frankfurt,D).` $\copyright_5$[R](5,6)

$\rightarrow G_1^2$: `?- direct(frankfurt,D).`

$\hookrightarrow G_2$: `?- direct(frankfurt,beijing).`

$\rightarrow G_2^1$: $\bot$

Refutation Path No. 7

Backtracking at the latest choice point $\copyright_5$.

G_0: `?- travel(edinburgh, D).` $\copyright_0$[R](5,6)

$\rightarrow G_0^1$: `?- direct(edinburgh, B2),travel(B2,D)` $\copyright_2$[S](1,2)

$\hookrightarrow G_1$: `?- direct(edinburgh, frankfurt),travel(frankfurt,D).`

$\rightarrow G_1^1$: `?- travel(frankfurt,D).` $\copyright_5$[R](5,6)

$\rightarrow G_1^2$: `?- direct(frankfurt,B5),travel(B5, D).`

$\hookrightarrow G_2$: `?- direct(frankfurt,beijing),travel(beijing, D).`

$\rightarrow G_2^1$: `?- travel(beijing, D).` $\copyright_6[\mathrm{R}](\underline{5},6)$

$\rightarrow G_2^2$: `?- direct(beijing,D).`

†

Refutation Path No. 8

Backtracking at the latest choice point $\copyright_6$.

G_0: `?- travel(edinburgh, D).` $\copyright_0[\mathrm{R}](5,\underline{6})$

$\rightarrow G_0^1$: `?- direct(edinburgh, B2),travel(B2,D)` $\copyright_2[\mathrm{S}](1,\underline{2})$

$\hookrightarrow G_1$: `?- direct(edinburgh, frankfurt),travel(frankfurt,D).`

$\rightarrow G_1^1$: `?- travel(frankfurt,D).` $\copyright_5[\mathrm{R}](5,\underline{6})$

$\rightarrow G_1^2$: `?- direct(frankfurt,B5),travel(B5, D).`

$\hookrightarrow G_2$: `?- direct(frankfurt,beijing),travel(beijing, D).`

$\rightarrow G_2^1$: `?- travel(beijing, D).` $\copyright_6[\mathrm{R}](5,\underline{6})$

$\rightarrow G_1^2$: `?- direct(beijing,B6), travel(B6,D).`

†

So far all choice points have been exhausted and all possible refutation paths traversed. The counter examples could be found in the successful paths.

Experiment Conclusion

The backtracking method we tried in this case study works. It is viable to maintain both types of choice points in structural resolution in the same stack.

3.5 Conclusion

We introduced inductively incomplete structural resolution and saw experiments for informing the implementation decision on addressing non-deterministic choice involved in structural resolution. It turned out that left-first computation rule is adopted instead of non-deterministic goal predicate choice, and that sequential search and single stack backtracking is adopted to replace non-deterministic program clause choice.

Chapter 4

Inductively Complete Structural Resolution

We define rewriting tree and use it to model *inductively complete structural resolution* (Section 4.1). Then we explore extending the computation rule, search rule and backtracking method we developed for inductively incomplete structural resolution to inductively complete structural resolution (Section 4.2).

Sections in this chapter are logically integrated but chronologically they are separate: Section 4.1 presents concepts that were basically understood by the author before he started implementation of inductively complete structural resolution. Section 4.2 presents insight that was inspired by the implementing work and enlightened by the final actual software implementation.

4.1 Rewriting Tree

Rewriting trees are used to formalise the extra control imposed on inductively incomplete structural resolution to achieve inductive completeness. Related concepts are defined formally in [18, 19]. Here we see paraphrased version of definitions by the author. Then comes an example and discussion about comparing inductively incomplete structural resolution with the rewriting tree model.

4.1.1 Definitions about Rewriting Trees

We define *rewriting tree*, *rewriting tree substitution* and *rewriting tree transition*.

Definition 30 (Rewriting Tree)**.** A *rewriting tree* is identified by

- a clause G which could be either a definite clause or a definite goal, and
- a logic program P, and
- an auxiliary substitution σ.

The symbolic notation for a rewriting tree is rew(P, G, σ). We build rewriting tree rew(P, G, σ) using the following rules iteratively:

- The clause $G\sigma$ is the root, and each predicate of clause Gσ's body is a child of the root, and the i-th predicate of $G\sigma$'s body is the i-th child of the root.

- Any node in a rewriting tree is classified as either an *and-node* or an *or-node*. The root is classified as an or-node. Children of or-nodes are and-nodes, while children of and-nodes are or-nodes.

- Given an and-node A, it has as many children as the number of clauses in program P. The i-th child of and-node A is clause C$\theta\sigma$ if clause C is the i-th clause from P with freshly renamed variables, and C's head matches against and-node A with matcher θ; otherwise the i-th child of and-node A is a freshly named *or-node variable*.

- Given an or-node O that is not the root, if O is an or-node variable then O has no child. Otherwise O must be a definite clause, and it has as many children as the number of predicates of its body, and the i-th child of O is the i-th predicate from O's body.

An or-node variable as the i-th child of an and-node A indicates possibility of substitution reduction of A using i-th clause of program P.

Definition 31 (Rewriting Tree Substitution)**.** A rewriting tree T' results from applying substitution θ to rewriting tree T=rew(P, G, σ). The notation is $T' = T\theta$. T' is built from T and θ in the following way:

- Simultaneously apply θ to all nodes of T that are not or-node variables, resulting in tree T_1, which is not necessarily a rewriting tree.

- If T_1 satisfies Definition 30, then T_1 is a rewriting tree and $T' = T_1$.

- If T_1 does not satisfy Definition 30, then modify [1] T_1 according to Definition 30 to obtain rewriting tree T_2. Let $T' = T_2$.

If variables are systematically named in a rewriting tree T = rew(P,G,σ), then $T\theta$= rew(P,G,$\sigma\theta$). For details see [19].

Definition 32 (Rewriting Tree Transition)**.** Given rewriting tree T=rew(P,G,σ) where or-node variable O is the i-th child of its parent A, transition of rewriting tree T at or-node variable O into rewriting tree T' is performed in the following way:

- If the head of i-th clause C (where variables have been freshly renamed) from program P unifies A with unifier θ, then $T' = T\theta$.

- Otherwise T' is an empty tree.

[1] Compared with definition of rewriting tree substitution in [18, 19], the paraphrased version by the author is vague when it comes to how to do the modification. When T_1 is not a rewriting tree, it is implied that there exist some or-node variable O in T_1, such that O is the i-th child of its own parent A, and the head of i-th clause C of P does match against A with matcher γ. According to [18, 19], such or-node variable as O of T_1 shall be replaced by rew(P, $C\gamma$, $\sigma\theta$).

While of all the examples the author have ever seen, rewriting tree substitution only happens when rewriting tree transition is performed, and the transition is always quite trivial (as from tree in Figure 4.1 to tree in Figure 4.4) and the vague statement suffices for performing the mentioned trivial rewriting tree substitution.

4.1.2 An Example and Discussion on Rewriting Trees

Here is an example of modelling structural resolution with rewriting trees. The purpose of the example is:

- We see, by using rewriting trees, we could naturally extend inductively incomplete structural resolution to achieve inductive completeness.
- We explore how the inductively complete structural resolution differs from inductively incomplete structural resolution in terms of goal reductions.
- We address a little bit about non-deterministic choice involved in rewriting tree transition.

Example 33. Consider program P_{happy}.

```
1| happy(X) :- likes(X,Y),likes(X,Z).
2| happy(sam).
3| like(ben, apple).
```

Query G_0: `?- happy(Who).`

This example is designed to accommodate as much interesting theoretical elements in as less number of clauses as possible. Clause 1 has *existential variables* Y and Z, because they don't occur in clause head. Clause 1 also has *conjunctive body*. Clause 1 and 2 are *overlapping clauses* because Clause 1's head subsumes Clause 2's head.

The inductively incomplete structural resolution for G_0 and P_{happy} is as follows.

G_0: `?- happy(Who).`

$\rightarrow G_0^1$: `?- likes(Who,Y1),likes(Who,Z1)`

$\hookrightarrow G_1$: `?- likes(ben,apple),likes(ben,Z1)`

$\rightarrow G_1^1$: `?-likes(ben,Z1)`

$\hookrightarrow G_2$: `?-likes(ben,apple)`

$\rightarrow G_2^1$: $\perp$

The details of each reduction are as follows, with their relation to rewriting trees T_0 (Figure 4.1), T_1 (Figure 4.2) and T_2 (Figure 4.3) explained.

$G_0 \rightarrow G_0^1$:
Clause 1: `happy(X1) :- likes(X1, Y1),likes(X1,Z1).`
Matcher: {`X1/Who`}.
$G_0 \rightarrow G_0^1$ corresponds to building rewriting tree T_0 (Figure 4.1).

$G_0^1 \hookrightarrow G_1$:
Clause 3: `likes(ben,apple).`
Unifier: θ_1={`Who/ben,Y1/apple`}.
$G_1 \rightarrow G_1^1$:
Clause 3: `likes(ben,apple).`
Matcher: ϵ
$G_0^1 \hookrightarrow G_1 \rightarrow G_1^1$ corresponds to transition from T_0 at O_5 into T_1 (Figure 4.2).

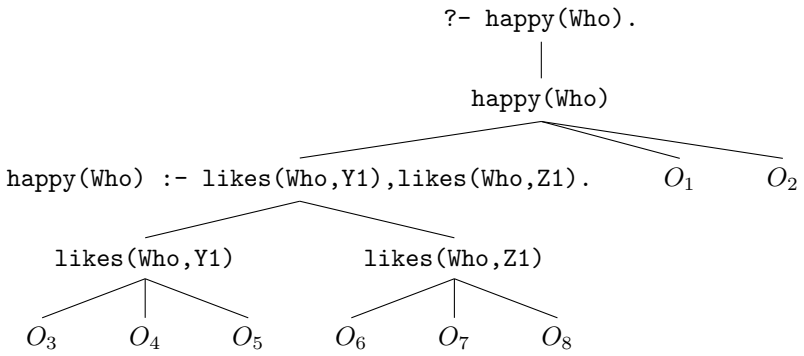

Figure 4.1: T_0=rew(P_{happy}, G_0, ϵ).

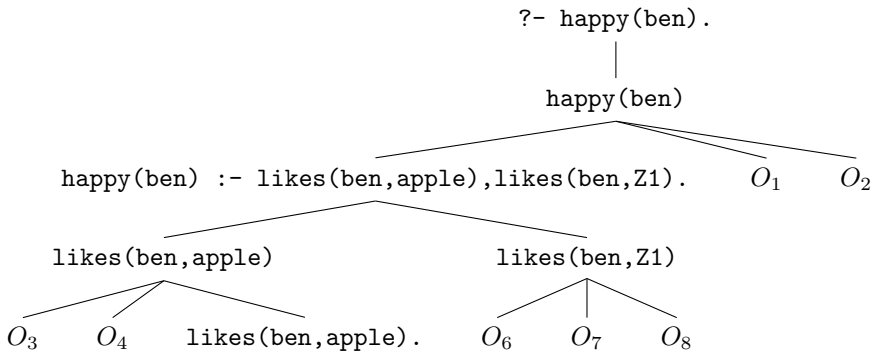

Figure 4.2: T_1=$T_0\theta_1$=rew(P_{happy}, G_0, θ_1).

$G_1^1 \hookrightarrow G_2$:
Clause 3: `likes(ben,apple).`
Unifier: θ_2={`Z1/apple`}.
$G_2 \rightarrow G_2^1$:
Clause 3: `likes(ben,apple).`
Matcher: ϵ
$G_1^1 \hookrightarrow G_2 \rightarrow G_2^1$ corresponds to transition from T_1 at O_8 into T_2 (Figure 4.3).

When performing inductively incomplete structural resolution for G_0 and P_{happy}, rewriting tree transition only happens at O_5 of T_0 and then O_8 of T_1. The definition of rewriting tree transition allows the transition happen at any or-node variable. Thus we can explore what happens if transition is performed at other or-node variables in the trees T_0, T_1 and T_2.

It is clear that transition at all or-node variables from T_2 result in empty trees. Transition at all or-node variables except O_8 of T_1 result in empty trees. Transition at all or-node variables except O_1 and O_5 of T_0 result in empty trees. Here we have adopted backtracking style to perform rewriting tree transition, where or-node variables in the latest built rewriting tree are used first as point of transition.

The or-node variable O_1 of T_0 turns out to be special: it does not make T_0

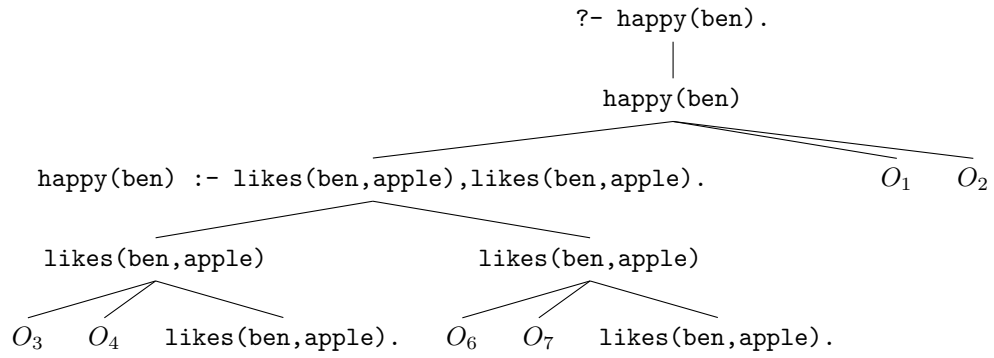

Figure 4.3: T_2=$T_1\theta_2$=rew(P_{happy}, G_0, $\theta_1\theta_2$).

transition into empty tree but it is ignored by rewriting tree transition corresponding to inductively incomplete structural resolution. We use T_3 (Figure 4.4) to refer to the rewriting tree obtained from transition of T_0 at O_1.

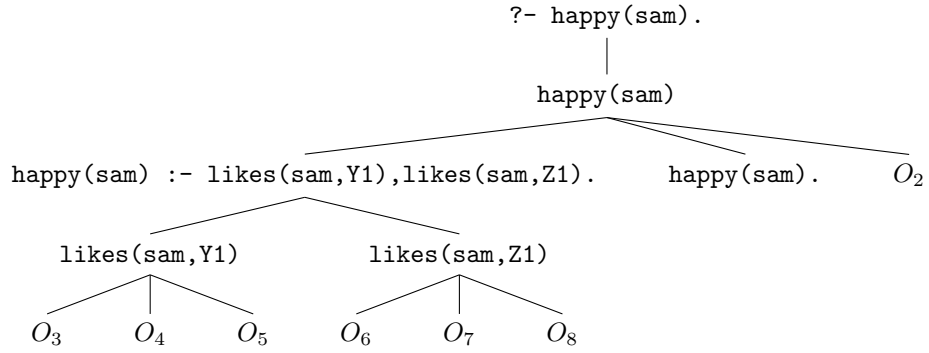

Figure 4.4: T_3=$T_0\theta_3$=rew(P_{happy}, G_0, θ_3) where θ_3={`Who/sam`}.

The unifier θ_3 computed for transition from T_0 at O_1 to T_3 gives the second counter example for the query. And further transition at or-node variables from T_3 result in empty trees.

The inductively incomplete structural resolution for G_0 and P_{happy} is able to give one counter example Who = ben. Both SLD resolution and structural resolution in terms of rewriting trees can give two counter examples Who = sam and Who = ben.

We review the way the second counter example was computed by rewriting tree transition. The transition from T_0 at O_1 into T_3 corresponds to the following reductions:

G_0: `?- happy(Who).`
$\hookrightarrow$ ${G_1}^*$: `?- happy(sam).`
$\rightarrow {G_1}^{1*}$: $\perp$

According to Definition 24 of inductively incomplete structural resolution, rewriting reduction for G_0 using Clause 1 is the only possible reduction we

can do for G_0. But when we model structural resolution with rewriting tree construction and transition, it is allowed that G_0 be reduced by Clause 2 using substitution reduction. We conclude that when using structural resolution modelled by rewriting tree construction and transition, it is allowed to perform substitution reduction on a goal for which rewriting reduction is applicable.

Rewriting tree theory allows the choice of or-node variable, at which to perform rewriting tree transition, to be non-deterministic. We made transition from T_0 at O_1 into T_3 after we have finished rewriting tree construction and transition that corresponds to inductively incomplete structural resolution, therefore the reduction of G_0 by Clause 2 using substitution reduction can be regarded as kind of backtracking. We conclude that we have introduced some kind of order to systematically perform rewriting tree transition. This point will be clearer in Section 4.2.

Recall the case studies we had in Section 3.4.3 and 3.4.4, where there are choice points for rewriting reduction. Backtracking at these rewriting choice points corresponds to building other branches within the same rewriting tree.

4.2 Linear Operational Semantics under the Rewriting Tree Model

The achievement described in this section is *not immediately* logically entailed from the previous Section 4.1. Instead, it is inspired by the implementation work done by the author.

Based on the rewriting tree model, we define choice points and backtracking method to extend the operational semantics of inductively incomplete structural resolution described in Chapter 3, resulting in linear operational semantics of inductively complete structural resolution . We demonstrate the result by examples and make comparative analysis.

4.2.1 Choice Points and Backtracking

A choice point corresponds to all children (which are or-nodes) of some and-node in a rewriting tree. We see two preparatory definitions on or-node types and clause grouping , then we define choice points.

Definition 34 (Or-node Types)**.** An or-node in a rewriting tree belongs to one of three types:

Type-1 An or-node variable O, transition at which results in an empty tree, indicating a program clause whose head does not unify with the parent of or-node variable O.

Type-2 An or-node variable O', transition at which results in an non-empty rewriting tree, indicating a program clause that could be used to perform substitution reduction with the parent of or-node variable O'.

Type-3 An or-node N that is not an or-node variable, being an instance of a program clause that could be used for rewriting reduction for the parent of N.

Definition 35 (Clause Grouping)**.** Given a goal G with a selected predicate A to reduce using structural resolution, the clauses in program P can be classified into three groups:

Group-1 Clauses whose heads do not unify with the selected predicate A from goal G.

Group-2 Clauses whose heads unify with, but more specifically, not match against the selected predicate A from goal G.

Group-3 Clauses whose heads unify with, and more specifically, match against the selected predicate A from goal G.

For mnemonic purpose, Group-1 is named not-unifying, Group-2 is named *unifying-but-matching*, Group-3 is named *unifying-and-matching*.

By definition of rewriting tree and its transition, a Group-i clause is related to a Type-i or-node for i=1,2,3.

Example 36. Consider program P_{likes}:

```
1| likes(tom, juice).
2| likes(X, food).
3| likes(Y,Z).
4| hates(tom, ticks).
5| hates(john, chilli).
```

Given query `?-likes(tom, What).`, clause 4 and 5 are in Group-1 (not-unifying), clause 1 and 2 are in Group-2 (unifying-not-matching) while clause 3 is in Group-3 (unifying-and-matching).

Prolog ignores Group-1 (not-unifying) clauses and only searches for Group-2 (unifying-not-matching) and Group-3 (unifying-and-matching) clauses without distinguishing them. The structural resolution implementation is based on Prolog meta-interpreter, therefore Group-1 (not-unifying) clauses are also ignored.

To enable backtracking, the marking scheme of *choice points* is introduced to indicate presence of choices at a juncture, and the choices that have been or yet to be tried.

Definition 37 (Choice Points)**.** Given a goal G with a selected predicate *Pred* to reduce using inductively complete structural resolution w.r.t. program P, a *choice point* is created for goal G only if there are more than one program clause whose head *unifies* with goal predicate *Pred*. The following properties hold for all choice points.

1. A choice point is symbolically represented in the form

   $$©(\ r_1, r_2, \ldots, r_m \mid s_1, s_2, \ldots, s_n\ ),\ \text{where } m+n>1.$$

   The argument list of © is divided by a vertical bar into two fields: the left field and the right field.

2. The left field whose elements are $r_1, r_2, \ldots, r_m$ is a list of program clause indices sorted in ascending order, referring to Group-3 (unifying-and-matching) clauses in the context of goal G, program P and goal predicate *Pred*. If this field is empty, i.e. if $m = 0$, mark the choice point by

   $$©(-\mid\ s_1, s_2, \ldots, s_n\ ),\ \text{where } n>1.$$

3. The right field whose elements are $s_1, s_2, \ldots, s_n$ is a list of program clause indices sorted in ascending order, referring to Group-2 (unifying-not-matching) clauses in the context of goal G, program P and goal predicate *Pred*. If this field is empty, i.e. if $n = 0$, mark the choice point by

$$©(\ r_1, r_2, \ldots, r_m\ |-),\ \text{where } m > 1.$$

4. The initially selected index is the first (left-most) index of the argument list, irrespective of the vertical bar. Only the selected index is underlined. If the selected index is in the left field of the argument list of ©, rewriting reduction shall be performed. Otherwise substitution reduction be performed.

The property that $m + n > 1$ in Definition 37 is based on the practice that a choice point is created only if the number $(m + n)$ of program clauses, whose head unify with the goal predicate, is more than one.

The way we define choice points above indicates that sequential search is used. By the way, we continue to use left-first computation rule. We will later see that these decisions are still viable in the context of inductively complete structural resolution.

Definition 38 (Backtracking)**.** All choice points are maintained in a single stack to direct backtracking. When back tracking, the choice point at the top of the stack is popped, then the index to the right of the previously selected index is chosen, irrespective of the vertical bar.

Putting indices of Group-3 (unifying-and-matching) clauses in the left field of ©'s parameter list and selecting indices from left to right ensures that rewriting is performed as far as possible, implementing "rewriting-first" principle ($\rightarrow^{\mu}$) in structural resolution.

The vertical bar of ©'s parameter list does not prevent selecting a Group-2 (unifying-not-matching) clause's index from the right field when all indices in the left field has been exhausted, so substitution reduction can be performed on predicates for which all possible ways of rewriting reduction have been done, implementing rewriting tree transition.

Definition 39 (Linear Operational Semantics of Inductively Complete Structural Resolution)**.** The operational semantics of inductively complete structural resolution is structural resolution (Definition 24) with backtracking (Definition 38) at choice points (Definition 37).

The above defined operational semantics is linear in the same sense as SLD resolution is linear, that refutation of a goal is done by a linear sequence of (goal, clause) pairs.

In the next section we see some examples on the linear operational semantics.

4.2.2 Examples

We see examples of linear operational semantics of inductively complete structural resolution, which refutes goals without directly using the form of rewriting trees but as a series of structural reductions with control (backtracking).

Both examples given in this section re-use logic programs that are used in previous sections, where the programs were used to demonstrate either the rewriting tree semantics or the inductively incomplete structural resolution, or SLD resolution. Re-using the logic programs, therefore, is for the purpose of comparison between the different operational semantics used to process the program and query.

Example 40. The program P_{happy} is copied here.

P_{happy} is previously used in *Example* 33 to associate the rewriting tree semantics with inductively incomplete structural resolution, and to demonstrate how rewriting tree semantics extends the inductively incomplete structural resolution to achieve inductive completeness.

```
1| happy(X) :- likes(X,Y),likes(X,Z).
2| happy(sam).
3| like(ben, apple).
```

Query G_0: `?- happy(Who).`

The trace of refuting query G_0 using linear inductively complete structural resolution is given as follows.

G_0: `?- happy(Who).` $\quad \copyright_0(\ \underline{1}\ |\ 2\ )$

$\quad \rightarrow G_0^1$: `?- likes(Who,Y1),likes(Who,Z1).`

$\hookrightarrow G_1$: `?- likes(ben,apple),likes(ben,Z1).`

$\quad \rightarrow G_1^1$: `?- likes(ben,Z1).`

$\hookrightarrow G_2$: `?- likes(ben,apple).`

$\quad \rightarrow G_2^1$: $\bot$

; (backtracking at $\copyright_0$)

G_0: `?- happy(Who).` $\quad \copyright_0(\ 1\ |\ \underline{2}\ )$

$\hookrightarrow G_1$: `?- happy(sam).`

$\quad \rightarrow G_1^1$: $\bot$

In the above sequence of reduction we used semi-colon to indicate start of backtracking.

The initially selected parameter of $\copyright_0$ is 1, which is in the left field, so rewriting reduction is performed for G_0.

When backtracking at $\copyright_0$, the selected parameter is 2, being in the right field, so substitution reduction is done for G_0 in backtracking.

The computed substitutions can be inferred from the trace. The first answer is Who = ben; the second answer is Who = sam.

Example 41. Program P_{member} and query G_0 are copied as follows.

They (P_{member} and G_0) are used previously in *Example* 17 to demonstrate SLD resolution and in *Example* 27 to demonstrate inductively incomplete structural resolution and to compare inductively incomplete structural resolution with SLD resolution.

```
1| member(X,[X|_]).
2| member(X,[_|T]) :- member(X,T).
```

G_0: `?- member(X,[a,b]).`

The inductively complete structural resolution is as follows. For clarity, when backtracking, we present the full sequence of reduction from the query, rather than from the goal to which the backtracked choice point is attached.

G_0: `?- member(X,[a,b]).` $\quad ©_0(\ \underline{2} \mid 1\ )$

$\quad \rightarrow G_0^1$: `?- member(X,[b]).` $\quad ©_1(\ \underline{2} \mid 1\ )$

$\quad \rightarrow G_0^2$: `?- member(X,[]).`

$\quad \dagger$

; (Backtracking at $©_1$)

G_0: `?- member(X,[a,b]).` $\quad ©_0(\ \underline{2} \mid 1\ )$

$\quad \rightarrow G_0^1$: `?- member(X,[b]).` $\quad ©_1(\ 2 \mid \underline{1}\ )$

$\hookrightarrow G_1$: `?- member(b,[b]).` $\quad ©_2(\ \underline{1}, 2 \mid -)$

$\quad \rightarrow G_1^1$: $\bot$

; (Backtracking at $©_2$)

G_0: `?- member(X,[a,b]).` $\quad ©_0(\ \underline{2} \mid 1\ )$

$\quad \rightarrow G_0^1$: `?- member(X,[b]).` $\quad ©_1(\ 2 \mid \underline{1}\ )$

$\hookrightarrow G_1$: `?- member(b,[b]).` $\quad ©_2(\ 1, \underline{2} \mid -)$

$\quad \rightarrow G_1^1$: `?- member(b,[]).`

$\quad \dagger$

; (Backtracking at $©_0$, destroy $©_2$ and $©_1$)

G_0: `?- member(X,[a,b]).` $\quad ©_0(\ 2 \mid \underline{1}\ )$

$\hookrightarrow G_1$: `?- member(a,[a,b]).` $\quad ©_3(\ \underline{1}, 2 \mid -)$

$\quad \rightarrow G_1^1$: $\bot$

; (Backtracking at $©_3$)

G_0: `?- member(X,[a,b]).` $\quad ©_0(\ 2 \mid \underline{1}\ )$

$\hookrightarrow G_1$: `?- member(a,[a,b]).` $\quad ©_3(\ 1, \underline{2} \mid -)$

$\quad \rightarrow G_1^1$: `?- member(a,[b]).`

$\quad \rightarrow G_1^2$: `?- member(a,[]).`

$\quad \dagger$

(No more solutions, destroy $©_3$ and $©_0$)

The first counter example is X = b, the second X = a. These counter examples are given in the reverse order as given by SLD resolution, due to a rewriting-first approach.

4.2.3 Discussion

We compare inductively incomplete semantics with inductively complete semantics of structural resolution.

Recall that choice points (Section 3.4) in the operational semantics of inductively incomplete structural resolution are defined in the form of ©$[R](a, b, c \ldots)$, for rewriting reduction choice points, and ©$[S](h, i, j \ldots)$, for substitution reduction choice points.

Using our new notion of choice points (Section 4.2.1), ©$[R](a, b, c \ldots)$ and ©$[S](h, i, j \ldots)$ can be mapped into ©*(a,b,c...* | ~~*h,i,j...*~~*)* and ©$(-|\ h, i, j \ldots)$, respectively, where all indices of Group-2 (unifying-but-matching) clauses are ignored if there are any Group-3 (unifying-and-matching) clauses.

Example 42. We have seen that inductively incomplete structural resolution cannot refute G_0: `?- member(X,[a,b]).` w.r.t P_{member} in *Example* 27. The reason is explained in this example using our new notion of choice points.

G_0: `?- member(X,[a,b]).` ©$_0(\ \underline{2}\ |\ \lightning\ )$

$\rightarrow G_0^1$: `?- member(X,[b]).` ©$_1(\ \underline{2}\ |\ \lightning\ )$

$\rightarrow G_0^2$: `?- member(X,[]).`

†

(Backtracking is not applicable, no more solutions.)

©$_0$ and ©$_1$ are not regarded as choice points in inductively incomplete structural resolution because after the Group-2 (unifying-but-matching) clauses (Clause 1) are ignored, there is only one applicable clause for each juncture. After reductions fails at G_0^2 no backtracking can be made, therefore not successful refutation is given.

The programs and queries we used in case studies (Section 3.4.3 and 3.4.4) are special in terms that whenever there are choice of alternative clauses for rewriting reduction, there happen to be no Group-2 (unifying-but-matching) clauses.

In other words, all choice points in the case studies are either of the form ©$(\ r_1, r_2, \ldots, r_m\ |-)$, where $m > 1$, or of the form ©$(-|\ s_1, s_2, \ldots, s_n\ )$, where $n > 1$. There was no choice point of the form ©$(\ r_1, r_2, \ldots, r_m\ |\ s_1, s_2, \ldots, s_n\ )$, where $m > 0, n > 0$.

So in those case studies, inductively incomplete semantics does not make a difference from inductively complete semantics, and our maintaining ©$[R]$'s and ©$[S]$'s in the same stack previously is a special case of our maintaining all ©$(\ r_1, r_2, \ldots, r_m\ |\ s_1, s_2, \ldots, s_n\ )$, where $m + m > 1$, in the same stack.

4.3 Conclusion

We defined rewriting trees to model structural resolution and to achieve inductive completeness. With the help of the implementing work, a linear operational semantics of inductively complete structural resolution is discovered. Reflecting about the linear operational semantics, the author finds that structural resolution, when being modelled by rewriting trees, is as if defined in the from $\rightarrow^n \circ \hookrightarrow^1 \circ \rightarrow^\mu \circ \hookrightarrow^1$, where $n \geq 0$, and $\rightarrow^n \circ \hookrightarrow^1$ achieves the inductive completeness and $\rightarrow^\mu$ reserves observational productivity.

Chapter 5

The Meta-Interpreter for Structural Resolution

We see how structural resolution is implemented in Prolog. *Approximation* is the development strategy successfully used: starting from existing solution to related tasks, modifying the code bit by bit to solve more closely related tasks, and the last modification produced the code for the target task.

Approximation strategy is described in more detail in G. Polya's 1950s book *How to Solve It* [35] and it is used also in 1990s for the development of Warren's Abstract Machine [36], the implementation of Prolog language.

Each section of this chapter introduces a meta-interpreter that represents a major step of approximation. We indicate how to learn these meta-interpreters, and how they came into being. The detailed codes are attached in appendix.

5.1 SLD Vanilla

Although much effort has been devoted by the author to learn Prolog and many exercise programs written, the most relevant *existing* program to the task is meta-interpreters for SLD resolution.

The simplest form of meta-interpreter written in Prolog, implementing SLD resolution, is nicknamed *vanilla* [2, 4]. We call them SLD vanilla in this report. Depending on the data type used to store logic programs, SLD vanilla is classified into two styles:

Normal style, which uses the Prolog built in predicate `clause/2` to access clauses, and has a tree structure. The advantage of this style is that

- A proof tree could be built after finishing refuting a goal.
- A trail stack can be added for loop detection, and the stack is easy to maintain—proved goal predicates can be automatically removed from the trail stack.

Continuation style, which stores clauses as customised terms, and uses a stack to store goal predicates that are yet proved. This style has simpler code structure but it is short of the advantage of normal style SLD vanilla.

Variable Value Sharing in Prolog

To understand meta-interpreters, it is important to understand the *variable value sharing* mechanism in Prolog:

- Variables of the same name in the same term share value.
- Two variables of different names share value after they are unified with each other.

Example 43. We see the following step by step analysis of an interactive Prolog session.

1. We turn on the interactive interpreter of SWI-Prolog[1], and posing the conjunctive query Q:

   ```
   ?- (G,Gs)=(direct(A,B),direct(B,C),travel(A,B,C)), G=
   direct(dundee, edinburgh).
   ```

   The first goal G_1 of the conjunctive query Q is

   ```
   (G,Gs) = (direct(A,B),direct(B,C),travel(A,B,C))
   ```

   where the term `(direct(A,B),direct(B,C),travel(A,B,C))`, which can be regarded as a typical conjunctive body of some definite clause, is to be unified with term `(G,Gs)`.

   The second goal G_2 of the conjunctive query Q is

   ```
   G= direct(dundee, edinburgh)
   ```

   where term `G` is to be unified with term `direct(dundee, edinburgh)`.

2. The first goal G_1 is selected under the left-first computation rule of Prolog. Given the right associativity of comma operator, variable `G` in term `(G,Gs)` is bound to term `direct(A,B)`, variable `Gs` is bound to term `(direct(B,C),travel(A,B,C))`.

3. Due to variable value sharing rule , the variable `G` in the second goal G_2 of the conjunctive query Q shares value with the variable `G` in the first goal G_1, so after `G` in G_1 is bound to term `direct(A,B)`, `G` in G_2 is also bound to term `direct(A,B)`.

4. When proving the second goal G_2 of query Q, unification is done between `direct(A,B)`, which is the value to which variable `G` has been instantiated (bound) to, and `direct(dundee, edinburgh)`. The result is that `A` be bound to `dundee`, `B` to `edinburgh`.

5. The occurrence of `A,B` in term `(direct(B,C),travel(A,B,C))`, which has been bound to `Gs`, are also instantiated accordingly, as result of value sharing.

6. The result of query Q is:

   ```
   G = direct(dundee, edinburgh),
   Gs = (direct(edinburgh, C), travel(dundee, edinburgh, C)),
   A = dundee,
   B = edinburgh.
   ```

[1]SWI-Prolog version 7.2.3 is used by the author on his Windows 10 PC.

5.2 Term Matching Vanilla

The first modification on SLD vanilla to approximate our target was coding of a term-matching vanilla, which uses only rewriting reduction to solve queries. We explain why term-matching vanilla comes immediately after SLD vanilla.

The most distinctive feature of structural resolution, compared with SLD resolution, is use of rewriting reduction as the basic inference rule. It is useful for the target task to implement rewriting reduction.

We have seen in previous sections that rewriting reduction is a special case of SLD reduction. The author found that it is possible to modify SLD resolution implementation for rewriting reduction implementation.

Implementation

SLD vanilla, both in normal style and in continuation style, has a clause, which we call "core clause". The "core clause" takes a goal predicate `G` and finds a program clause in the form[2] `Head :- Body`, whose head unifies with goal predicate `G`. The "core clause" unifies `G` with `Head`, and returns `Body` with an optional reference number of the chosen program clause. Then the "core clause" passes `Body` to a recursive procedure call for further SLD reduction.

The above mentioned "core clause" can be modified to choose a program clause, the head of which not only unifies with, but also *subsumes* the selected goal predicate. This is achieved by inserting "subsuming-test" before passing the `Body` to recursive procedure call for further reduction. If the "subsuming-test" fails, backtracking is forced to find the next program clause whose head unifies with the goal predicate.

The result of the above modification is Term-Matching Vanilla.

5.3 Matching-First Vanilla

Moving on from term-matching vanilla, we could get SLD resolution with alternative search rule. We have seen how SLD resolution works with sequential search. Given that structural resolution classifies clauses into three groups (Section 4.2.1), and adopts "rewriting-first" principle, we could imagine using a similar search rule for SLD resolution, where Group-3 (unifying-and-matching) clauses are used sequentially in prior to Group-2 (unifying-but-matching) clauses, which are also used sequentially. Doing this we are getting one step further from term matching vanilla and mean while this one step also helps us move closer to structural resolution.

Implementation

We have introduced that term matching vanilla has the mechanism to select only Group-3 (unifying-and-matching) clauses w.r.t. the goal predicate for SLD resolution, implemented as doing "subsuming test" after a clause is found applicable for SLD reduction. This mechanism have a dual, which selects only Group-2 (unifying-but-matching) clauses w.r.t. the goal predicate for SLD reduction, by

[2]Recall that a Prolog fact `Fact.` is treated as `Fact :- true`.

doing "non-subsuming test" after a clause is found applicable for SLD reduction. "Non-subsuming test" is implemented as negation of the predicate used for "subsuming test".

Now adding the above mentioned new mechanism to term matching vanilla, we get implementation of SLD resolution with the same search rule as structural resolution. We call this version of SLD resolution implementation Matching-First Vanilla.

Comparison

Compared with SLD vanilla, matching-first vanilla gives answers in reversed order as to SLD vanilla in some cases, e.g. consider the P_{member} program used in *Example* 41, term matching vanilla chooses Clause 2 in prior to Clause 1 while SLD vanilla would do the contrast.

Another consequence of adopting the matching-first search rule for SLD resolution is that some programs that rely on rule-order to function with SLD vanilla now no longer works. For example consider P_{nat} in arithmetic form:

```
1| nat(0).
2| nat(Y) :- nat(X), Y is X + 1.
```

This program can be used in Prolog to generate natural numbers. We pose query `?- nat(N).` for the matching-first vanilla and it will not terminate, because Clause 2 is a Group-3 (unifying-and-matching) clause and it is always chosen in prior to Clause 1 which is a Group-2 (unifying-but-matching) clause.

5.4 Vanilla for Structural Resolution

Starting from matching-first vanilla, we get the code for structural resolution by modifying the mechanism for finding Group-2 (unifying-but-matching) clauses. In matching first vanilla, after a Group-2 clause is found, the instantiated *body* of the clause is passed for further reduction. We change it to be that after a Group-2 clause is found, the instantiated *head* of the clause is passed for further reduction.

The execution model of the structural resolution vanilla has been studied and is described in detail in Section 4.2.

5.5 Test

All implementation of meta-interpreters mentioned in this chapter have been tested.

Strength

Particularly, the implementation of structural resolution is tested with about 50 logic programs written in Prolog. These test programs are exercise programs written by the author when he learnt Prolog following [1, 2]. Therefore the test programs are regarded as a comprehensive collection of basic inductive Prolog programs. They all worked well with structural resolution vanilla, in terms that given a program and some possible ways to query the program, if Prolog

can given correct answer, then structural resolution can also give the same correct answer, except possible change in the order of answers and understandable duplication of answers.

The only two exceptions are the P_{nat} program mentioned in Section 5.3 and `numbervar/3` from Program 10.8 of [2], that with Prolog they work well but with structural resolution, the former does not terminate and the latter gives correct answers then wrong answers in backtracking. However, the problem with these programs lies within the programs themselves rather than the meta-interpreter. For instance for P_{nat} it is expected to non-terminate with structural resolution and for `numbervar/3` the ambiguity in its definition is tolerated by SLD resolution but not by structural resolution, and by making its definition more precise it works equally fine with structural resolution.

Weakness

As common limitation of vanilla meta-interpreter, structural resolution in the form of a meta-interpreter requires extra clauses to support library predicates and built-in predicates. The cut is not supported. The author also found there is a lack of support for meta-call predicates [5], which may cause "interpreter leak" (a term coined by the author) because meta-call predicates are classified as built-in predicates and passed to Prolog system for evaluation, even when the meta-variable is bound to a predicate that is supposed to be evaluated by the meta-interpreter.

5.6 Conclusion

We have went through implementation notes of structural resolution and saw how the structural resolution meta-interpreter was coded by approximation. The strength and weakness of the meta-interpreter is discussed.

Chapter 6

Report Conclusion

We conclude the report by talking about the achievement by the author from doing the reported work, the highlight of the work, what can be improved, and further work.

6.1 Achievement

The inductively complete structural resolution has been implemented. The author increased his skill in Prolog programming and understood more about logic programming theories. He had his reading and comprehension skill practised, as well as logical thinking practised by programming exercises.

More generally, he practised his ability to manage his own studies, setting and experimenting with practical provisional short term goals and targets in response to the ultimate longer term goal. Better time management was developed to ensure enough working hours and fulfilment of a researcher's responsibility.

6.2 Highlight

Development of Prolog programming skill was emphasized, particularly after the author realised that no matter how much he understands the theory, his task will eventually require him to write programs. Surprisingly, his devotion to Prolog programming exercise, his limited understanding of the theory and his approximation strategy helped him write the implementation which inversely improved his understanding of the theory.

The author agrees with his supervisor's opinion that doing some implementing could help one learn more than just reading about theories, because the latter (theory study) is only part of the things one needs to do in order to get the former (implementation) worked out—the other things one needs to do for implementation are programming skill development, learning a development strategy (e.g. approximation), and experiment.

6.3 Improvement

The author is satisfied with his work and his time arrangement. Perhaps more work can be done to enhance the support to meta-call predicates by the meta-interpreter, w.r.t. the test result. It also worth studying about how "cut" can be defined for structural resolution.

6.4 Further work

Further work will be on understanding existing co-SLD implementation and exploring adding co-SLD loop detection to inductively complete structural resolution. In longer term, using coinductive logic programming to solve problems in type inference is of interest.

Appendix A

Code for Meta-interpreters Mentioned in the Report

SWI-Prolog [5] version 7.2.3 is used for the work in this report.

A.1 Term-Matching Vanilla

```
% choose clauses by term matching instead of unification
% *- matching is checked  by using built_in predicate subsumes_term/2.
% *- Built-in  copy_term/2 is used to  make a copy  of a goal to use to
%    search  for a matching rule  without instantiating variables from
%    the goal. This predicate has some limitation regarding the ground
%    arguments  of the goal,  which is shared between the goal and its
%    copy, meaning that if a ground argument in the copy is forcefully
%    modified using setarg/3 then such modification will be synchro-
%    nised to the goal and vice versa.

solve(Goal) :- solve(Goal,[]).
solve([],[]).
solve([],[G|Goals]) :- solve(G, Goals).
solve([A|Bs], Goals) :-
           append(Bs, Goals, Goals1),
           solve(A, Goals1).
solve(A,Goals) :-
          matching_rule(A, Body),
          solve(Body, Goals).

matching_rule(A, Body) :-
       copy_term(A,A_copy),
       rule(A_copy,Body,Ref),
       rule(A1,_,Ref),
       subsumes_term(A1,A).

% rule(Head, Body, Ref).
% Object program clauses shall be represented in the above way.
```

```
% This echoes built_in predicate clause/3  which, given a clause
% head,  returns the  body as  well  as an unique identifier for
% that  clause.  When the  rules are not represented in default
% way  using  ':-'  operator,  the identifier shall be provided
% explicitly.

% sample object program


rule(member(X,[X|_]),          [], ref_1).
rule(member(X,[_|Y]), member(X,Y), ref_2).
rule(member(a,[a,_,_]),        [], ref_3).
rule(member(a,[_,a,_]),        [], ref_4).
rule(member(a,[_,_,a]),        [], ref_5).

rule(related(abraham, issac),   [], ref_6).
rule(related(abraham, joseph),  [], ref_7).
rule(related(abraham, _),       [], ref_8).
rule(related(_, tom),           [], ref_9).
rule(related(_, _),             [], ref_10).
```

A.2 Matching-First Vanilla

```
% Of all unifying clauses, matching ones are used first.
% Therefore, unifying  clauses are sorted not only by their order of
% definition, but also whether they unify by matching or not.
% *- matching is checked by using built-in predicate subsumes_term/2,
%    which does not instantiate variables.
% *- The continuation style of the meta-interpreter is inherited from
%    the  term  matching meta-interpreter. This style does not support
%    correct loop detection nor proof tree building.


solve(Goal) :- solve(Goal,[]).
solve([],[]) :- !.                  % Note 1
solve([],[G|Goals]) :- !, solve(G, Goals).
solve([A|Bs], Goals) :- !,
           append(Bs, Goals, Goals1),
           solve(A, Goals1).
solve(A,Goals) :-
          unifying_and_matching_rule(A, Body),
          solve(Body, Goals).
solve(A,Goals) :-
          unifying_but_matching_rule(A, Body),
          solve(Body, Goals).

% choose clauses whose heads unifies with the goal, and specifically,
% matches the goal.
```

```
unifying_and_matching_rule(A, Body) :-
       copy_term(A,A_copy),          % Note 2
       rule(A_copy,_,Ref),           % Note 3
       rule(A1,_,Ref),               % Note 4
       subsumes_term(A1,A),          % Note 5
       rule(A,Body,Ref).             % Note 6

% choose clauses whose head unifies with the goal, and specifically,
% does not match the goal
unifying_but_matching_rule(A, Body) :-
      copy_term(A,A_copy),
      rule(A_copy,_,Ref),
      rule(A1,_,Ref),
      \+ subsumes_term(A1,A),
      rule(A,Body,Ref).

% Note 1: Clauses deal with mutually exclusive cases, hence the cuts
% Note 2: At run time variable A is bound to the current goal, A_copy
%         then is a variant of A
% Note 3: At run time, this procedure finds some clause whose head uni-
%         fies with a variant  of the  current goal and get the clause
%         reference number. Let's assume that the '_' is replaced by a
%         named  variable  ``Body'',   and  that  the  last  procedure
%         ``rule(A,Body,Ref)'' is removed. Then  this will cause error
%         in cases when the chosen clause  indeed matches the goal and
%         in  this  case if  some variables  in the body of the chosen
%         clause share value with variables from the variant of current
%         goal to which A_copy is bound, further instantiation for the
%         term bound  to Body  will not influence term bound to A, and
%         this is not the desired behaviour.  For this reason the body
%         of the  chosen clause is discarded, as by '_' in the proce-
%         dure.  And ``rule(A,Body,Ref)'' is  added  as  the last step
%         to make sure the goal is reduced by correct sub-goals.
% Note 4: Get a copy of the chosen applicable rule.
% Note 5: Check whether the head also matches the goal, any binding made
%         for checking will be undone by implementation of subsumes_term/2.
% Note 6: If matches, use this rule to reduce the goal.

% sample object program

rule(direct(aberdeen, dundee),[],ref_1).
rule(direct(dundee,edinburgh),[],ref_2).
rule(direct(edinburgh,london),[],ref_3).

rule(travel(A,B), direct(A,B),                ref_4).
rule(travel(A,B), [direct(A,C),travel(C,B)],  ref_5).
```

A.3 Vanilla for Structural Resolution

```
clause_tree(true) :- !.
clause_tree((G,R)) :-
   !,
   clause_tree(G),
   clause_tree(R).

clause_tree(G) :-
  (predicate_property(G, built_in);   % for built-in's
   predicate_property(G, nodebug) ),  % for std library preds
  !,
  call(G).

clause_tree(A) :-                     % rewriting reduction
            unifying_and_matching_rule(A, Body),
            clause_tree(Body).
clause_tree(A) :-                     % substitution reduction.
            unifying_but_matching_rule(A, _),        % Note 1
            clause_tree(A).                          % Note 1


unifying_and_matching_rule(A, Body) :-
         copy_term(A,A_copy),
         clause(A_copy,_,Ref),
         clause(A1,_,Ref),
         subsumes_term(A1,A),
         clause(A,Body,Ref).

unifying_but_matching_rule(A, Body) :-
        copy_term(A,A_copy),
        clause(A_copy,_,Ref),
        clause(A1,_,Ref),
        \+ subsumes_term(A1,A),
        clause(A,Body,Ref).

% Note 1: These  are the only  parts  changed compared with the matching
%         first  vanilla of continuation style. Here, after an unifying-
%         but-matching clause is found, what to evaluate next is not the
%         instantiated body of  the  clause but the instantiated head of
%         the clause. This corresponds to substitution reduction.

% sample object programs:
% Logic programs used in examples of this report.
```

Bibliography

[1] Patrick Blackburn, Johan Bos and Kristina Striegnitz. *Learn Prolog Now !* `http://www.learnprolognow.org/index.php`

[2] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, MA, 2nd edition, 1994.

[3] Danny Crookes, *Introduction to Programming in Prolog*, Prentice Hall International (UK) Lttd., 1988.

[4] John R. Fisher, *Prolog :- Tutorial*, `http://www.cpp.edu/~jrfisher/www/prolog_tutorial/contents.html`

[5] SWI Prolog website: `http://www.swi-prolog.org/`

[6] The GNU Prolog Website: `http://gprolog.org/`

[7] YAProlog website: `http://www.dcc.fc.up.pt/~vsc/Yap/`

[8] John Wylie Lloyd, *Foundations of Logic Programming* Springer-Verlag New York, Inc. Secaucus, NJ, USA. 2nd edition, 1993.

[9] Robert Kowalski, *Predicate Logic as a Programming Language* Proceedings IFIP Congress, Stockholm, North Holland Publishing Co., 1974, pp. 569-574.

[10] Maarten van Emden and Robert Kowalski, *The Semantics of Predicate Logic as a Programming Language*, Journalofthe AssooaUonforComputing-Machinery, Vol 23, No 4, October1976, pp 733-742.

[11] Keith L. Clark, *Predicate logic as a computational formalism*. Queen Mary University of London, UK 1980

[12] Maarten van Emden and M.A. Nait Abdallah, *Top-down semantics of fair computations of logic programs*, The Journal of Logic Programming, Volume 2, Issue 1, April 1985, Pages 67-75.

[13] André Arnold and Maurice Nivat, *Metric Interpretations of Infinite Trees and Semantics of Non-Deterministic Recursive programs*, Theoretical Computer Science 11 (1980) 181-205, North-Holland Publishing Company.

[14] Luke Simon at.al., *Co-Logic Programming: Extending Logic Programming with Coinduction*, Proceedings of the 34th International Colloquium on Automata, Languages and Programming, Wroclaw, Poland, July 9–13, 2007.

[15] Davide Ancona and Agostino Dovier, *A theoretical perspective of Coinductive Logic Programming*, Fundamenta Informaticae, vol. 140, no. 3-4, pp. 221-246, 2015.

[16] Gopal Gupta et. al., *Infinite Computation, Co-induction and Computational Logic*, Proceedings of the 4th International Conference on Algebra and Coalgebra in Computer Science, pp. 40-54, Winchester, UK, August 30 – September 02, 2011.

[17] Davide Ancona, *Regular Corecursion in Prolog*, SAC'12, March 25-29, 2012, Riva del Garda, Italy.

[18] Ekaterina Komendantskaya and Patricia Johann. *Structural Resolution: a Framework for Coinductive Proof Search and Proof Construction in Horn Clause Logic*. Submitted to ACM Transactions in Computational Logic. 36 pages. November 2015.

[19] Patricia Johann, Ekaterina Komendantskaya and Vladimir Komendantskiy. *Structural Resolution for Logic Programming*. ICLP 2015 Technical Communications

[20] Ekaterina Komendantskaya, Patricia Johann and Martin Schmidt, *A Productivity Checker for Logic Programming*, Pre-proceedings paper presented at the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016), Edinburgh, Scotland UK, 6-8 September 2016.

[21] František Farka, Ekaterina Komendantskaya and Kevin Hammond, *Coinductive Soundness of Corecursive Type Class Resolution*, Pre-proceedings paper presented at the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016), Edinburgh, Scotland UK, 6-8 September 2016.

[22] Luca Franceschini, Davide Ancona and Ekaterina Komendantskaya. *Structural Resolution for Abstract Compilation of Object-Oriented Languages*. Workshop on Coalgebra, Horn Clause Logic Programming and Types, 28-29 November, Edinburgh, full version appeared as MSc thesis of L.Franceschini, U.Genova, 2016.

[23] Martin Schmidt and Ekaterina Komendantskaya, *Prototype Implementation of Coalgebraic Logic Programming*,
`https://github.com/coalp/coalp`

[24] Ulf Nilsson and Jan Maluszynski, *Logic, Programming and Prolog*, 2nd edition,
`http://www.ida.liu.se/~ulfni53/lpp/index.shtml`

[25] John C. Mitchell, *Foundations for Programming Languages*, The MIT Press, Cambridge, MA, London, England, 1996.

[26] Benjamin C. Pierce, *Types and Programming Languages*, The MIT Press, Cambridge, MA, London, England, 2002.

[27] Alfred Tarski, *A Lattice-Theoretical Fixpoint Theory and Its Applications*, Pacific J. Math. 5 (1955), 285-309

[28] George Grätzer, *Lattice Theory: Foundation*, Springer Basel AG, 2011.

[29] D. E. Rutherford, *Introduction to Lattice Theory*, Oliver and Boyd Ltd., Edinburgh and London, 1966.

[30] Luke Simon, *Extending Logic Programming with Coinduction*, Ph.D. Thesis, UT Dallas, July 2006.

[31] Stanford Encyclopedia of Philosophy, Online resource, `https://plato.stanford.edu/index.html`

[32] Abraham A. Fraenkel, *Abstract Set Theory*, 3rd, revised edition, North-Hollannd Publishing Company, Amsterdam, 1966.

[33] Jon Barwise and Lawrence Moss, *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*, CSLI Publications, Stanford, U.S.A., 1996.

[34] Paul Klint, *Quick Introduction to Term Rewriting*, Meta-Environment, 2007, `http://www.meta-environment.org/doc/books/extraction-transformation/term-rewriting/term-rewriting.html`

[35] George Polya, *How to Solve It: A New Aspect of Mathematical Method*, 2nd Edition, Princeton University Press, 1957.

[36] Hassan Ait-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*, Free online version, 1999. `http://wambook.sourceforge.net/`