

**Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596**

**NARROWING GRAMMAR: A LAZY FUNCTIONAL LOGIC
FORMALISM FOR LANGUAGE ANALYSIS**

Hau-Ming Lewis Chau

**September 1989
CSD-890056**

UNIVERSITY OF CALIFORNIA

Los Angeles

**Narrowing Grammar : A Lazy Functional Logic
Formalism for Language Analysis**

**A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science**

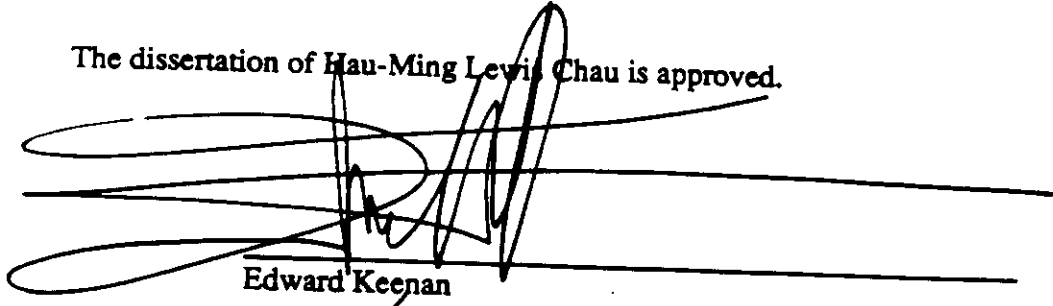
by

Hau-Ming Lewis Chau

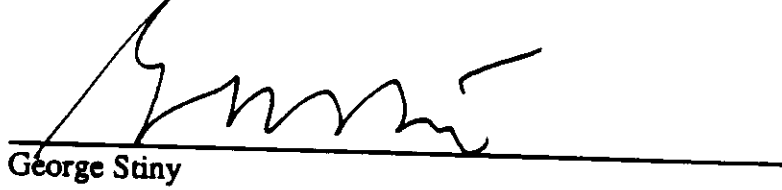
1989

**Copyright © 1989 by
Hau-Ming Lewis Chau**

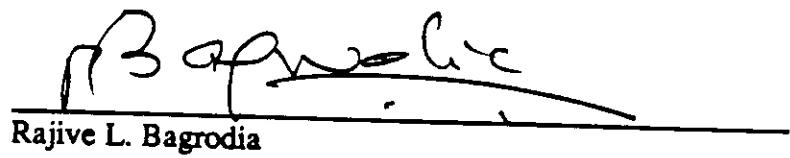
The dissertation of Hau-Ming Lewis Chau is approved.



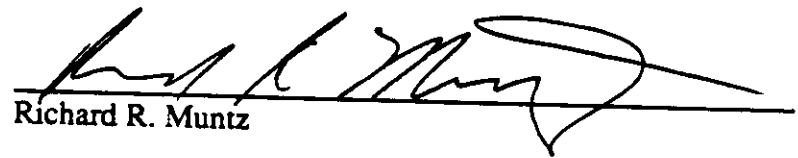
Edward Keenan



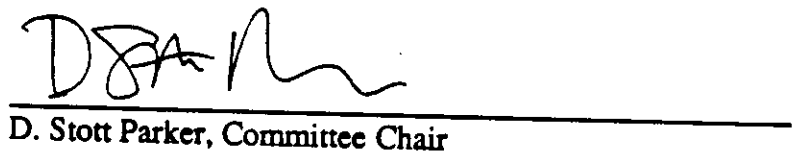
George Stiny



Rajive L. Bagrodia



Richard R. Muntz



D. Stott Parker, Committee Chair

University of California, Los Angeles

1989

To my parents

Yun-Yin Chow and Lin-Wan Chow

TABLE OF CONTENTS

Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Summary of Major Results	3
1.2.1 Narrowing Grammar	3
1.2.2 Interesting Properties of Narrowing Grammar	7
1.2.3 Compiling Narrowing Grammar into Prolog Programs	10
1.2.4 Performance Considerations	12
1.2.5 Comparison with Established Logic Grammars	13
1.2.6 Summary	20
Chapter 2 Review of Previous Work	21
2.1 Logic Programming	21
2.2 Logic Grammars	24
2.2.1 Definite Clause Grammars	25
2.2.2 Metamorphosis Grammars	26
2.2.3 Extraposition Grammars	27
2.2.4 Gapping Grammars	29
2.2.5 Other Logic Grammars	31
2.3 Rewriting and Narrowing	31
2.4 Lazy Evaluation and Logic Programming	33
2.4.1 Lazy Evaluation	33
2.4.2 Lazy Evaluation in Logic Programming	34
2.5 Integration of Logic Programming and Rewriting	34
2.5.1 Logic Programming with Equality	35
2.5.2 Narrowing	35
2.5.3 Extended Unification and Functional Logic Languages	36
2.6 Conclusions	39
Chapter 3 Narrowing Grammar	40
3.1 Introduction	40
3.2 Formalism of Narrowing Grammar	40

3.3	Specifying Patterns with Narrowing Grammar	46
3.4	What is New about Narrowing Grammar	49
3.4.1	New Model of Acceptance in Logic Grammar	50
3.4.2	Higher-order Specification, Extensibility, and Modularity	51
3.4.3	Lazy Evaluation, Stream Processing, and Coroutining	52
3.4.4	Limitations of First-order Logic Grammars	55
Chapter 4	Compilation of Narrowing Grammar to Logic Programs	57
4.1	Compilation Algorithm	57
4.2	Correctness of the Narrowing Grammar Compilation Algorithm	67
Chapter 5	Completeness Issues and Extension of Narrowing Grammar	75
5.1	Introduction	75
5.2	Narrowing-Completeness of Narrowing Grammar for Simplified Forms	75
5.3	Compilation Algorithm to Handle Duplicate LHS Variables	89
5.4	Conclusions	95
Chapter 6	Performance Considerations	97
6.1	Introduction	97
6.2	Partial Evaluation	97
6.3	Optimization of Algorithm 4.1 by Partial Evaluation	99
6.4	Efficient Classes of Narrowing Grammar	102
6.5	Partial Evaluation to Greibach Grammar	107
Chapter 7	Comparison with Other Logic Grammars	113
7.1	Introduction	113
7.2	Language Acceptance vs Generation	115
7.3	The Power of Unification	120
7.4	Higher-order Specification, Modular Composition and Lazy Evaluation	122
7.5	Transformation of Logic Grammars to Narrowing Grammar	123
7.5.1	Narrowing Grammar and Definite Clause Grammars	124
7.5.2	Narrowing Grammar and Metamorphosis Grammars	126
7.5.3	Narrowing Grammar and Extrapolation Grammars	128
7.5.4	Narrowing Grammar and Gapping Grammars	131
7.6	Narrowing Grammar can be More Expressive	134

7.7 Case Study : Natural Language Analysis	137
7.8 Summary	143
Chapter 8 Conclusions	144
References	146

ACKNOWLEDGEMENTS

I am especially grateful to my advisor, D. Stott Parker, for his valuable technical guidance and unfailing enthusiasm. I would also like to thank the members of my committee, Professors Bagrodia, Keenan, Muntz and Stiny for their participation and comments.

I would like to acknowledge the remarks of Paul Eggert, Sanjai Narain and Fernando Pereira on some of my published papers for which this dissertation is based. I also wish to acknowledge Defense Advanced Research Projects Agency, which provided continuing support during my graduate studies at UCLA (contract F29601-87-C-0072).

I am greatly indebted to Professor Chao-Chih Yang who first showed me what research is, and how to do it. I would like to thank my parents without whom this work would not have been possible.

Last but not least, I thank all brothers and sisters in Chinese Bible Church, which we have fellowship every week in the final year of my Ph.D. study. My spiritual life is enriched so that I can live with God's love.

VITA

- July 26, 1961 Born, Hong Kong
- 1984 Bachelor of Science in Computer Science
The Chinese University of Hong Kong
Hong Kong
- 1986 Master of Science in Computer Science
University of Alabama at Birmingham
Birmingham, Alabama
- 1986-1989 Post Graduate Research Engineer
University of California at Los Angeles
Los Angeles, California

PUBLICATIONS

- H. L. Chau, *Narrowing Grammar : A Comparison with Other Logic Grammars*, Proc. North American Conference on Logic Programming, MIT Press, October 1989.
- H. L. Chau, D. S. Parker, *Narrowing Grammar*, Proc. Sixth International Conference on Logic Programming, pp. 199-217, MIT Press, June 1989.
- C. C. Yang, J. J. Chen and H. L. Chau, *Algorithm for Constructing Minimal Deduction Graph*, IEEE Transactions on Software Engineering, vol. 15, no. 6, June 1989

D. S. Parker, R. R. Muntz and H. L. Chau, *The Tangram Stream Query Processing System*, Proc. Fifth International Conference on Data Engineering, Los Angeles, February 1989.

H. L. Chau, D. S. Parker, *Functional Logic Grammar : A New Scheme for Language Analysis*, UCLA Computer Science Department Technical Report CSD880097, December 1988.

H. L. Chau, D. S. Parker, *Executable Temporal Specifications with Functional Grammars*, UCLA Computer Science Department Technical Report CSD880046, June 1988.

ABSTRACT OF THE DISSERTATION

Narrowing Grammar : A Lazy Functional Logic Formalism for Language Analysis

by

Hau-Ming Lewis Chau

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1989

Professor D. Stott Parker, Chair

We present a new kind of grammar. It combines concepts from logic programming, rewriting, lazy evaluation, and logic grammar formalisms such as Definite Clause Grammars (DCGs). We call it *Narrowing Grammar*.

A Narrowing Grammar is a finite set of rewrite rules. The semantics of Narrowing Grammar is defined by a kind of special outermost lazy rewriting strategy called *NU-narrowing*. With some modest restrictions on the use of Narrowing Grammar rules, NU-narrowing is shown to be complete in the sense that whenever a term can be simplified, it can be simplified by repeatedly applying NU-narrowing.

Narrowing Grammar is directly executable, like many logic grammars. In fact, Narrowing Grammar rules can be compiled to Prolog and executed by existing Prolog interpreters as generators or acceptors. Unlike many logic grammars, Narrowing Grammar also permits higher-order specification and modular composition, and provides lazy evaluation. Lazy evaluation is important in certain language acceptance situa-

tions, such as in coroutined matching of multiple patterns against a stream.

We compare Narrowing Grammar with several established logic grammars: Definite Clause Grammars, Metamorphosis Grammars, Extraposition Grammars and Gapping Grammars, showing how these logic grammars can be transformed to Narrowing Grammar. We also investigate the versatility of Narrowing Grammar for language analysis by applying it in several examples.

Characterizing the efficiency of Narrowing Grammar is important. One important subclass of Narrowing Grammar, Greibach grammar, can be compiled to efficient Prolog code. Furthermore, Narrowing Grammar rules can, in many cases, be transformed to Greibach grammar by applying the well-known program transformation technique called *partial evaluation*. As a result, the performance of Narrowing Grammar can be comparable with first-order logic grammars such as DCGs.

Chapter 1

Introduction

1.1 Motivation

Logic programming is the use of statements of logic (Horn clauses) as a programming language. It has led to new insights into computing as well as logic. For example, one very important application of logic programming is in parsing. A *logic grammar* has rules that can be represented as Horn clauses which can then be executed for either acceptance or generation of the language specified.

Rewriting uses some set of rules to simplify objects to other objects. A set of such rules is called a rewrite rule system. Its usefulness is evident from its appearance in many branches of computer science. Examples include formal grammars and the lambda calculus [9].

Lazy evaluation is basically a computation scheme in which an expression is evaluated only when there is demand for its value. It allows certain computations to terminate more quickly and also enables computation with infinite structures. Consider the following two rules, defining how lists may be concatenated†:

† We use Prolog syntax for lists. The empty list is written as []. The head and tail of a list are components of the functor named '.', which is the dot. Thus, the list consisting of one element 'a' is $.(a,[])$ which in Prolog can also be written [a]. Alternatively, the head and tail of a list can be represented as $:(H,T)$ or $[H|T]$.

$$([], W) \Rightarrow W.$$
$$([U|V], W) \Rightarrow [U | (V, W)].$$

In one step, the term $([1, 2, 3], [4, 5, 6])$ rewrites to $[1 | ([2, 3], [4, 5, 6])]$. Only the head of the resulting concatenated list is computed. The tail, $([2, 3], [4, 5, 6])$, can then be further evaluated if this is necessary. Demand-driven computation like this is referred to as lazy evaluation.

A grammar formalism in which logic programming, rewriting, lazy evaluation and logic grammar are combined can provide considerable expressive power, going beyond the power of existing first-order logic grammars. In particular, it can afford the expressive power of both functions and relations. At the same time, lazy evaluation can be done within the eager framework of logic programming.

Many logic grammar formalisms are *first-order*. Specifically, a nonterminal symbol in these formalisms cannot be passed as an argument to some other nonterminal symbol. For example, usually Definite Clause Grammar (DCG) does not permit direct specification of grammar rules of the form

$$\text{goal}(X) \text{ --> } X.$$

This limitation was pointed out as early as [42]. We will discuss later in the dissertation why this is more than just a minor problem, as it affects the convenience of use, extensibility, and modularity of grammars. Some Prolog systems, including Quintus Prolog and Sicstus Prolog, have been extended to permit such rules. Also, very recently Abramson [3] has commented on the problem, and has addressed it by using

a new construct, *meta(X)*, to define metarules that go beyond the limit of first-order logic grammar formalisms. We propose a *lazy functional logic* approach to this problem.

Many logic grammars are also not *lazy*. Lazy evaluation can be very important in parsing. It allows coroutined recognition of multiple patterns against a stream. We are not aware of previous work connecting lazy evaluation and logic grammars, although the connection is a natural one.

We propose a new grammar formalism that rests on incorporating lazy rewriting within logic programming. This means that SLD-resolution with left-to-right goal selection, the proof procedure commonly used in Prolog, will implement this grammar formalism. That is, grammar rules are written in such a way, that when SLD-resolution interprets them, it directly simulates lazy rewriting.

1.2 Summary of Major Results

1.2.1 Narrowing Grammar

A new declarative semantics of logic grammar based on lazy narrowing is defined. The logic grammar rule system is not restricted to be terminating and permits non-determinism. We call it *Narrowing Grammar*.

Narrowing Grammar is a formalism for writing rules. It combines concepts from logic programming, rewriting, lazy evaluation, and logic grammar formalisms such as

Definite Clause Grammar. The semantics of Narrowing Grammar is defined by a special outermost lazy narrowing strategy, *NU-step*, which is different from existing logic grammar formalisms. In this dissertation, we point out a number of advantages of Narrowing Grammar for language analysis.

As a brief introductory example, let us show how easily regular expressions can be defined with Narrowing Grammar. The regular expression pattern $a^* b$ that matches sequences of zero or more copies of a followed by a b can be specified with the grammar rule

$$\text{pattern} \Rightarrow [a]^*, [b].$$

where we also define the following grammar rules:

$$(x^*) \Rightarrow [].$$
$$(x^*) \Rightarrow x, (x^*).$$
$$([], L) \Rightarrow L.$$
$$([x|L1], L2) \Rightarrow [x|(L1, L2)].$$

Here $'^*'$ is the postfix pattern operator defining the Kleene star pattern, and the rules for $'|'$ define pattern concatenation, very much like the usual Prolog rules for `append`.

For instance, the *narrowing* of $([a]^*, [b])$ to $[a, b]$ along with the rules used in each step is as follows:

Rewritten term	Rule used in rewriting
$([a]^*, [b])$	
$\rightarrow (([a], [a]^*), [b])$	$(X^*) \Rightarrow X, (X^*) .$
$\rightarrow ([a ([], [a]^*)], [b])$	$([X L1], L2) \Rightarrow [X (L1, L2)] .$
$\rightarrow [a (([], [a]^*), [b])]$	$([X L1], L2) \Rightarrow [X (L1, L2)] .$
$\rightarrow [a ([a]^*, [b])]$	$([], L) \Rightarrow L .$
$\rightarrow [a ([], [b])]$	$(X^*) \Rightarrow [] .$
$\rightarrow [a, b]$	$([], L) \Rightarrow L .$

In a similar way, all other lists matching the pattern $a^* b$ could be produced.

Narrowing Grammar can be used to produce both pattern generators and pattern acceptors. One approach for pattern acceptance is to introduce a new pair of Narrowing Grammar rules specifying pattern matching. The entire definition is the following pair of rules for `match`:

$$\text{match}([], s) \Rightarrow s .$$

$$\text{match}([X|L], [X|S]) \Rightarrow \text{match}(L, S) .$$

Grammar rules by themselves act as pattern generators, when applied with `match`, they act like parsers. Intuitively, `match` can be thought of as *applying* a pattern (the first argument) to an input stream (the second argument), in an attempt to find a prefix of the stream that the grammar defining the pattern can generate.

As a simple example, consider the derivation illustrating how `[a, b]` is accepted by the pattern $([a]^*, [b])$ with `match`:

```

match([a]*, [b]), [a,b])
  → match([a], [a]*), [b]), [a,b])
  → match([a|([], [a]*)], [b]), [a,b])
  → match([a|([], [a]*), [b]]], [a,b])
  → match([([], [a]*), [b]), [b])
  → match([a]*, [b]), [b])
  → match([], [b]), [b])
  → match([b], [b])
  → match([], [])
  → []

```

A *narrowing grammar* is a finite set of rules of the form:

$$LHS \Rightarrow RHS$$

where:

- (1) LHS is any term except a variable, and RHS is a term.
- (2) If $LHS = f(t_1, \dots, t_n)$, then each t_i is a term in normal form.

For instance, consider the following grammar:

```

s_abc => ab_c // a_bc.
ab_c => pair(a,b), [c]*.
a_bc => [a]*, pair(b,c).
pair(X,Y) => [].
pair(X,Y) => [X], pair(X,Y), [Y].

```

This grammar defines the non-context-free language $\{a^n b^n c^n \mid n \geq 0\}$ using only context-free-like constructions. The first rule for s_abc imposes simultaneous con-

straints ($a^n b^n c^*$ and $a^* b^n c^n$) on streams generated by the grammar. $//$ is a correlated pattern matching primitive that requires both argument patterns to generate or accept streams of the same length and is defined as follows:

$$\begin{aligned}
 ([X|Xs] // [Y|Ys]) &\Rightarrow [X|Xs//Ys]. \\
 ([] // []) &\Rightarrow [].
 \end{aligned}$$

The semantics of Narrowing Grammar is defined by a special outermost narrowing strategy, NU-step. The term p narrows to q in a NU-step or $p \rightarrow q$ if either ($p \Rightarrow q$) is an instance of some rule, or if the replacement of a subterm by the result of a NU-step yields q . Note that the NU-step does not permit a narrowing to begin with a variable.

A *NU-narrowing* is a narrowing p_1, p_2, \dots such that for each i , when p_i and p_{i+1} both exist, p_i narrows to p_{i+1} in a NU-step. A *simplification* is a NU-narrowing p_1, p_2, \dots, p_n if p_n is simplified and no other p_i is simplified.

For instance,

$$([a]^*, [b]) \rightarrow (([a], [a]^*), [b])$$

is a NU-step, and

$$([a]^*, [b]) \xrightarrow{*} [a \mid (([], [a]^*), [b])]$$

gives the first and last terms in a simplification.

1.2.2 Interesting Properties of Narrowing Grammar

Consider a NU-narrowing of the pattern `ab_c // a_bc` :

`ab_c // a_bc`

- `(pair([a],[b]), [c]*) // a_bc`
- `(([a], pair([a],[b]), [b]), [c]*) // a_bc`
- `([a|(pair([a],[b]), [b])], [c]*) // a_bc`
- `[a|((pair([a],[b]), [b]), [c]*)] // a_bc`
- `[a|((pair([a],[b]), [b]), [c]*)] // ([a]*, pair([b],[c]))`
- `[a|((pair([a],[b]), [b]), [c]*)] // (([a], [a]*), pair([b],[c]))`
- `[a|((pair([a],[b]), [b]), [c]*)] // ([a][a]*, pair([b],[c]))`
- `[a|((pair([a],[b]), [b]), [c]*)] // [a|([a]*, pair([b],[c]))]`
- `[a|(((pair([a],[b]), [b]), [c]*) // ([a]*, pair([b],[c])))]`
- `[a|((([]), [b]), [c]*) // ([a]*, pair([b],[c]))]`
- `[a|([[b], [c]*) // ([a]*, pair([b],[c]))]`
- `[a|([b][c]*) // ([a]*, pair([b],[c]))]`
- `[a|([b][c]*) // ([], pair([b],[c]))]`
- `[a|([b][c]*) // pair([b],[c]))]`
- `[a|([b][c]*) // ([b], pair([b],[c]), [c])]`
- `[a|([b][c]*) // [b|(pair([b],[c]), [c])]`
- `[a,b|([c]* // (pair([b],[c]), [c]))]`
- `[a,b|([(c], [c]*) // (pair([b],[c]), [c])]`
- `[a,b|([c][c]*) // (pair([b],[c]), [c])]`
- `[a,b|([c][c]*) // ([], [c])]`
- `[a,b|([c][c]*) // [c]]`
- `[a,b,c|([c]* // [])]`
- `[a,b,c|([] // [])]`
- `[a,b,c]`

Here we summarize several important features of Narrowing Grammar.

- (1) Narrowing Grammar permits higher-order specification.

Specifically, Narrowing Grammar is higher-order in the sense that patterns can be passed as input arguments to patterns, and patterns can yield patterns as outputs. For example, the enumeration pattern $(_ // _)$ is higher-order, as its arguments are patterns. The patterns a_bc and ab_c can be used as arguments to $//$, as in

$$a_bc // ab_c.$$

- (2) Narrowing Grammar supports modular composition of patterns.

For example, the pattern $(a_bc // ab_c)$ is composed of patterns $(_ // _)$, a_bc and ab_c . Each of these patterns specifies a constraint on the stream to be generated. The global constraint is the composed pattern $a_bc // ab_c$.

- (3) Narrowing Grammar permits lazy evaluation.

A specific advantage of lazy evaluation in parsing is that coroutined recognition of multiple patterns in a stream becomes accessible to the grammar writer. For example, narrowing of the stream pattern

$$ab_c // a_bc$$

interleaves the narrowing of $(_ // _)$, ab_c and a_bc . The sample narrowing of $a_bc // ab_c$ shows the actual interleaving – first ab_c is narrowed for four NU-steps, then a_bc for four NU-steps, then $(_ // _)$ for one NU-step, then ab_c for three NU-steps, then a_bc for four NU-steps, then $(_ // _)$ for one NU-step, then ab_c for two NU-steps, then a_bc for two NU-steps, then $(_$

// $_$) for one NU-step, then ab_c for one NU-step, and finally $(_ // _)$ for one NU-step. The effect of special outermost narrowing of the combined stream pattern $a_bc // ab_c$ is precisely to interleave these three narrowings.

1.2.3 Compiling Narrowing Grammar into Prolog Programs

We describe an algorithm to compile Narrowing Grammar to Prolog programs. It turns out that SLD-resolution with left-to-right goal selection, the proof procedure commonly used in Prolog, will implement NU-narrowing on these programs. The compilation of a Narrowing Grammar rule into a Prolog clause combines information about the rule and the control of NU-narrowing when interpreting that rule.

The compilation algorithm consists of two steps:

- (1) For each n -ary constructor symbol c (functional symbol that cannot be rewritten), $n \geq 0$, and for distinct Prolog variables X_1, \dots, X_n , generate the clause:

$$\text{simplify}(c(X_1, \dots, X_n), c(X_1, \dots, X_n)).$$

- (2) For each rule $f(L_1, \dots, L_m) \Rightarrow RHS$, let A_1, \dots, A_m, Out be distinct Prolog variables not occurring in the rule, and generate the clause:

$$\begin{aligned} \text{simplify}(f(A_1, \dots, A_m), Out) :- \\ \quad \text{nu_narrow}(A_1, L_1), \\ \quad \dots, \\ \quad \text{nu_narrow}(A_m, L_m), \\ \quad \text{simplify}(RHS, Out). \end{aligned}$$

For instance, given the Narrowing Grammar rules:

```
match([], S) => S.
```

```
match([X|L], [X|S]) => match(L, S).
```

the Prolog programs result from compilation are:

```
simplify(match(A, B), C) :-
```

```
    nu_narrow(A, []),
```

```
    nu_narrow(B, S),
```

```
    simplify(S, C).
```

```
simplify(match(A, B), C) :-
```

```
    nu_narrow(A, [X|L]),
```

```
    nu_narrow(B, [X|S]),
```

```
    simplify(match(L, S), C).
```

`simplify/2` guarantees its result (the second argument) will be simplified. That is, the function symbol of the result will be a constructor. Also, we can show that `simplify/2` behaves *like* NU-narrowing. We state this more formally by the following theorem:

Theorem

If X and Y are terms such that `simplify(X,Y)` has a successful Prolog-derivation, then there is a simplification from X to Y .

The converse is also true if Prolog-derivation of `simplify(X,Y)` is replaced by SLD-derivation, and we restrict the use of duplicate variables among arguments on the left

hand sides of Narrowing Grammar rules in certain ways.

With the same restriction on duplicate variables as the converse of the theorem, simplification is shown to be *narrowing-complete*, in that if a term can be simplified, it can be simplified by repeatedly applying NU-step. For instance, given the Narrowing Grammar rules:

$$\begin{aligned} a &\Rightarrow c. \\ b &\Rightarrow []. \\ c &\Rightarrow c. \\ g(x, []) &\Rightarrow []. \end{aligned}$$

$g(a, b)$ can be simplified to $[]$ by a sequence of NU-steps,

$$g(a, b) \rightarrow g(a, []) \rightarrow []$$

but the only leftmost outermost narrowing is the non-terminating narrowing

$$g(a, b), \quad g(c, b), \quad g(c, b), \quad \dots$$

1.2.4 Performance Considerations

One of the major challenges is to devise improvements for the implementation of Narrowing Grammar. Efficient implementation is possible in many cases. One possible way is to apply the well-known program transformation technique, *partial evaluation*, to optimize the Prolog programs resulting from compilation of Narrowing Grammar rules. For instance, partial evaluation alone will cause many `nu_narrow/2` subgoals to be replaced by unifications or `simplify/2` subgoals, and yields substantially faster

code.

Efficient implementation is also possible by recognizing efficient classes of Narrowing Grammar. One such class is *Greibach Grammar*, which requires the *RHS* of a rule to be simplified. Partial evaluation can, in many cases, be applied to transform Narrowing Grammar rules to Greibach form. For instance, the Narrowing Grammar rule

$$\text{pattern} \Rightarrow [a]^*, [b].$$

can be partially evaluated to

$$\text{pattern} \Rightarrow [b].$$
$$\text{pattern} \Rightarrow [a \mid \text{pattern}].$$

1.2.5 Comparison with Established Logic Grammars

Unlike many established logic grammars which rest on first-order logic, Narrowing Grammar is a new logic grammar formalism resting on lazy narrowing. This gives some extra expressive power to Narrowing Grammar and allows higher-order specification and modular composition.

We show that many established logic grammars can be transformed to Narrowing Grammar.

(1) Definite Clause Grammars

Definite Clause Grammars (DCGs) are essentially context-free grammars augmented by the language features of Prolog. DCG rules can be translated to Narrowing Grammar rules by changing all occurrences of --> to => and by

including the Narrowing Grammar definition for $\backslash, /$:

$$([], L) \Rightarrow L.$$

$$([X|L1], L2) \Rightarrow [X|(L1, L2)].$$

(2) Metamorphosis Grammars

MG [13] permits rules of the form

$$LHS, T \dashrightarrow RHS$$

where LHS is a nonterminal and T is one or more terminals. The MG rule can be read as " LHS can be expanded to RHS if T appears in the head of the input stream". We can capture the semantics of this MG rule in Narrowing Grammar by defining two more rules for $\backslash, /$ as follows (here `delete` is a constructor):

$$(\text{delete}([X|Z]), [X|Y]) \Rightarrow \text{delete}(Z), Y.$$

$$(\text{delete}([]), Y) \Rightarrow Y.$$

and transform the MG rule to

$$LHS \Rightarrow RHS, \text{delete}(T).$$

(3) Extraposition Grammars

One commonly used XG rule is of the form

$$LHS \dots T \dashrightarrow RHS.$$

Here LHS is a nonterminal symbol and T is any finite sequence of terminals or non-terminals. The XG rule can be read as " LHS can be expanded to RHS if T

appears later in the input stream". We can capture the semantics of this XG rule in Narrowing Grammar by defining three more rules for `\,` as follows (here `delete_any` is a constructor):

```
(delete_any([X|Z]), [X|Y]) => delete(Z), Y.
(delete_any([], Y)) => Y.
(delete_any(X), [Y|Z]) => [Y|(delete_any(X), Z)].
```

and transform the XG rule to

```
LHS => RHS, delete_any(T).
```

(4) Gapping Grammars

Consider a special class of GG rules [16] of the form

```
LHS, gap(X), T --> gap(X), RHS.
```

where `LHS` is a nonterminal symbol and `T` is any finite sequence of terminals or non-terminals.

We can capture the semantics of this GG rule in Narrowing Grammar by defining rules for `\,` with constructors `replace(_,_)` and `replace_any(_,_)` as follows:

$(\text{replace}([], R), Y) \Rightarrow R, Y.$
 $(\text{replace}([X|L], R), [X|Y]) \Rightarrow \text{replace}(L, R), Y.$
 $(\text{replace_any}([], R), Y) \Rightarrow R, Y.$
 $(\text{replace_any}([X|L], R), [X|Y]) \Rightarrow \text{replace}(L, R), Y.$
 $(\text{replace_any}(T, R), [X|Y]) \Rightarrow [X | (\text{replace_any}(T, R), Y)].$

and transform the GG rule to

$LHS \Rightarrow \text{replace_any}(T, RHS).$

Example : Left and Right Extrapolation

In this example, we show how Narrowing Grammar simulates the left- and right-extrapolation of GGs. The following grammar which is adapted from [17] parses sentences such as "The man is here that Jill saw".

s --> np, vp.
np --> det, n, relative.
np --> n.
np --> [term(np)].
vp --> aux, comp.
vp --> v, np.
relative --> rel_marker, s.
relative --> [].
relative, gap(G) --> gap(G), rightex.
rel_marker, gap(G), [term(np)] --> rel_pro, gap(G).
rightex --> rel_marker, s.
comp --> [here].
aux --> [is].
det --> [the].
rel_pro --> [that].
n --> [man].
n --> [jill].
v --> [saw].

The left-extrapolation GG rule†

rel_marker, gap(G), [term(np)] --> rel_pro, gap(G).

is transformed to

rel_marker => rel_pro, delete_any([term(np)]).

and the right-extrapolation GG rule

† This rule is equivalent to the XG rule
rel_marker ... [term(np)] --> rel_pro.

`relative, gap(G) --> gap(G), rightex`

is transformed to

`relative => replace_any([], rightex).`

All other rules can be translated to Narrowing Grammar rules by changing all occurrences of `-->` to `=>`. Now we illustrate how the sentence

`The man is here that Jill saw`

can be generated from the Narrowing Grammar by giving a sequence of terms that can be produced by narrowing of the Narrowing Grammar start symbol `s`:

s

- np, vp
- (det, n, relative), vp
- ([the], n, relative), vp
- [the|(n, relative)], vp
- [the|((n, relative), vp)]
- [the|((n, relative), vp)]
- [the|((n, relative), vp)]
- [the|((n, relative), vp)]
- [the,man|(relative, vp)]
- [the,man|(replace_any([], rightex), vp)]
- [the,man|(replace_any([], rightex), (aux, comp))]
- [the,man|(replace_any([], rightex), ([is], comp))]
- [the,man|(replace_any([], rightex), [is|comp])]
- [the,man,is|(replace_any([], rightex), comp)]
- [the,man,is|(replace_any([], rightex), [here])]
- [the,man,is,here|(replace_any([], rightex), [])]
- [the,man,is,here|rightex]
- [the,man,is,here|(rel_marker, s)]
- [the,man,is,here|((rel_pro, delete_any([trace])), s)]
- [the,man,is,here|([[that], delete_any([trace])), s)]
- [the,man,is,here|([that|delete_any([trace])), s)]
- [the,man,is,here,that|(delete_any([trace]), s)]
- [the,man,is,here,that|(delete_any([trace]), (np, vp))]
- [the,man,is,here,that|(delete_any([trace]), (n, vp))]
- [the,man,is,here,that|(delete_any([trace]), ([jill], vp))]
- [the,man,is,here,that|(delete_any([trace]), [jill|vp])]
- [the,man,is,here,that,jill|(delete_any([trace]), vp)]
- [the,man,is,here,that,jill|(delete_any([trace]), (v, np))]
- [the,man,is,here,that,jill|(delete_any([trace]), ([saw], np))]
- [the,man,is,here,that,jill|(delete_any([trace]), [saw|np])]

```
→ [the, man, is, here, that, jill, saw | (delete_any([trace]), np)]  
→ [the, man, is, here, that, jill, saw | (delete_any([trace]), [trace])]   
→ [the, man, is, here, that, jill, saw | (delete([]), [])]   
→ [the, man, is, here, that, jill, saw]
```

The point to be made here is that commonly used GG rules can be transformed to Narrowing Grammar rules easily.

1.2.6 Summary

Although existing first-order logic grammars have computational power equivalent to a Turing machine, they are, in general, quite limited in expressive power. A new and elegant logic grammar formalism that rests on incorporating rewriting within logic programming is proposed in this dissertation. We call it Narrowing Grammar. The expressive power of Narrowing Grammar goes beyond existing first-order logic grammar formalisms. At the same time, efficient implementation of certain classes of Narrowing Grammar using Prolog is possible now.

Chapter 2

Review of Previous Work

2.1 Logic Programming

Logic Programming began in the early 1970's as a outgrowth of earlier work in automatic theorem proving and artificial intelligence. The credit for the introduction of logic programming goes mainly to Kowalski [38] and Colmerauer [12] because they pioneered the fundamental idea that logic can be used as a programming language. The programming language Prolog [60] is an approximate implementation of logic programming. Below we summarize the basic concepts of logic programming. For a more full discussion, consult [56].

Logic programming is the use of logic statements as a programming language. The basic constructs of logic programming, terms and statements, are inherited from logic [56]. There are three basic statements: facts, clauses and queries. There is a single data structure: the logical term.

A *term* is a constant, a variable or a compound term. A *compound term* comprises a functor and a sequence of one or more terms called arguments. A *functor* is characterized by its name and its *arity* (number of arguments). Compound terms have the form

$f(t_1, t_2, \dots, t_n)$ with functor f/n . Terms are *ground* if they contain no variables; otherwise they are *nonground*.

A *substitution* is a finite set of pairs of the form $\langle X=t \rangle$, where X is a variable and t is a term, with no two pairs having the same variable left-hand side. For any substitution $\theta = \langle X_1=t_1 \rangle, \dots, \langle X_n=t_n \rangle$ and term s , the term $s\theta$ denotes the result of simultaneously replacing in s each occurrence of the variable X_i by t_i , $1 \leq i \leq n$, the term $s\theta$ is called an instance of s . A substitution θ is said to be a *unifier* of terms E and F , if $E\theta = F\theta$.

A *logic program* is a finite set of Horn clauses. A Horn clause is a logic sentence of the form

$$A \leftarrow B_1, \dots, B_n \quad n \geq 0,$$

where A and B_1, \dots, B_n are predicates and all variables in the clause are taken to be universally quantified. Such a clause is read declaratively as "A is implied by the conjunction of B_1, \dots, B_n " and is interpreted procedurally as "to answer query A, answer the conjunctive query B_1, \dots, B_n ." A and B_1, \dots, B_n are respectively called the *head* and *body* of the clause. If $n=0$, the clause is a *unit clause* or *fact*.

A *query* is a conjunction of the form

$$\leftarrow A_1, \dots, A_k.$$

where the A_1, \dots, A_k are goals. Variables in a query are existentially quantified.

A *computation* of a logic program P finds an instance of a given query logically deducible from P . A goal G is *deducible* from a program P if there is an instance A of G

where $A \leftarrow B_1, \dots, B_n$, $n \geq 0$, is a ground instance of a clause in P , and the B_1, \dots, B_n are deducible from P .

An attempt is made to find the computation using *SLD-resolution* proof procedure. SLD-resolution has been shown to be sound and complete for Horn clauses [4, 28]. Given a query $\leftarrow A_1, \dots, A_i, \dots, A_k$, $k \geq 0$, and a clause $A \leftarrow B_1, \dots, B_n$, $n \geq 0$, where A and A_i unify with most general unifier θ , the SLD-resolution proof procedure derives the new query: $\leftarrow (A_1, \dots, A_{i-1}, B_1, \dots, B_n, A_{i+1}, \dots, A_k)\theta$.

An *SLD-derivation* consists of a sequence of queries Q_0, Q_1, \dots , and a sequence of substitutions $\theta_1, \theta_2, \dots$, such that Q_{i+1} is derived from Q_i and θ_{i+1} is the most general unifier. If Q_i is empty (which is denoted by \square) for some i , the SLD-derivation is *successful*, the composition of substitutions $\theta_1, \dots, \theta_i$ is one answer to Q_0 , and the *length* of a derivation of Q_0 is i .

Consider the logic program

$$\begin{aligned} p(x, z) &\leftarrow q(x, y), p(y, z). \\ p(x, x). \\ q(a, b). \end{aligned}$$

and the goal $\leftarrow p(x, b)$. The following is an SLD-derivation of $\leftarrow p(x, b)$:

$$\begin{aligned} &\leftarrow p(x, b) \\ &\leftarrow q(x, y), p(y, b) \\ &\leftarrow p(b, b) \\ &\leftarrow \square \end{aligned}$$

The answer substitution of the goal $\leftarrow p(x, b)$ is $\{\langle x, a \rangle\}$.

Prolog-derivation is a special case of SLD-derivation. Given a query $\leftarrow A_1, \dots, A_i, \dots, A_k, k \geq 0$, instead of choosing an arbitrary goal to resolve against, Prolog's execution mechanism always chooses the leftmost one, and replaces the nondeterministic choice of a clause by sequential search and backtracking.

2.2 Logic Grammars

One very important application of Prolog and logic programming is parsing. Prolog in fact originated from attempts to use logic in expressing grammar rules and formalizing the parsing process [56]. A logic grammar has rules that can be represented as Horn clauses, and thus implemented by logic programming languages such as Prolog. These logic grammar rules are translated into Prolog clauses which can then be executed for either acceptance or generation of the language specified.

Since the development of Metamorphosis Grammar [13], the first logic grammar formalism, several variants of logic grammars have been proposed. Some are motivated by ease of implementation [47], some by a need for more general rules with more expressive power [16, 48], some by a view towards a general treatment of some natural language processing [29, 41, 55], and others by automating some part of the grammar writing process, such as the automatic construction of parse trees [2]. Generality and expressive power seem to have been the main concerns underlying all these efforts. These logic grammars are *first-order* and are not *lazy*. Unlike the logic grammar formalism based on *lazy narrowing* in this dissertation. Now we briefly review some existing logic grammars in the literature.

2.2.1 Definite Clause Grammars

The most popular approach to parsing in Prolog is *Definite Clause Grammars (DCGs)*. DCG is a special case of Colmerauer's metamorphosis grammar [13]. It extends context-free grammar in three ways:

- (1) DCG provides context dependency.
- (2) DCG allows arbitrary tree structures to be built in the process of parsing.
- (3) DCG allows extra conditions to be added to the rules, permitting auxiliary computations.

Consider the example which is taken from [47].

```
noun(N) --> [W], {rootform(W,N), is_noun(N)}.
```

`noun(N)` is a nonterminal and `[W]` is a terminal. It can be read as "a phrase identified as the noun `N` may consist of the single word `w`, where `n` is the root form of `w` and `N` is a noun".

DCGs syntax is little more than "syntactically sugared" Prolog syntax. There is a simple procedure for compiling each DCG rule to a Prolog clause. The basic idea is to add a **difference-list** argument to each nonterminal symbol, giving the input and output streams [49, 56]. Below is an example of some DCG rules and their compilation to Prolog:

```

a --> b, c.           a(S0,S) :- b(S0,S1), c(S1,S).
a --> [W].           a(S0,S) :- connects(S0,W,S).
a --> b, {d}.       s(S0,S) :- b(S0,S), call(d).

```

The definition of `connects/2` is:

```
connects([H|T],H,T).
```

Example 2.1 : Power of DCG Parameters

Consider the following DCG rules for the language $\{a^n b^n c^n \mid n \geq 0\}$:

```

s --> a(zero).
a(A) --> [a], a(succ(A)).
a(A) --> b(A), c(A).
b(succ(A)) --> [b], b(A).
b(zero) --> [].
c(succ(A)) --> [c], c(A).
c(zero) --> [].

```

The power of DCGs is beyond context-free due to its capability of passing parameters.

In this example, the parameters are used as counters.

2.2.2 Metamorphosis Grammars

Metamorphosis Grammar (MG) [13] generalizes DCGs and permits rules of the form

$$LHS, \tau \rightarrow RHS$$

where LHS is a nonterminal and τ is one or more terminals. For instance, the MG rule

`be(present), [not] --> [isnt].`

allows terminal symbol to be inserted to the head of the stream. Its translation to Prolog is:

`be(present, S0, S) :- connects(S0, isnt, S1), connects(S, not, S1).`

Example 2.2 : Context Sensitive Languages

An example of MGs for the language $\{a^n b^n c^n \mid n \geq 0\}$ is given here:

`s --> [a], b, s, [c].`

`s --> [a], [b], [c].`

`b, [a] --> [a], b.`

`b, [b] --> [b], [b].`

2.2.3 Extraposition Grammars

In spite of the power of DCGs, it is not convenient for the definition of certain constructions in natural languages such as "movement-trace" or "filler-gap" constructions in which a constituent seems to have been moved from another position in the sentence.

Extraposition Grammar (XG) [48] allows everything found in DCGs and allows, in addition, rules like the following:

`LHS ... T --> RHS.`

where *RHS* is any sequence of terminals, nonterminals, and tests, as in DCGs. The left

side of an XG rule can be a nonterminal followed by '...' and by any finite sequence of terminals or nonterminals. The example can be read as "LHS can be expanded to RHS if the category *T* appears later in the input stream".

This allows for a very natural treatment of certain filler-gap constructions. For example, relative clauses can be handled with rules like the following:

```
np --> det, n.  
np --> det, n, relative.  
np --> trace.  
  
relative --> rel_marker, s.  
  
rel_marker ... trace --> rel_pro.  
  
rel_pro --> [who].
```

Example 2.3 : Context Sensitive Languages

An example of XGs for the language $\{a^n b^n c^n \mid n \geq 0\}$ is given here:

```
s --> as, bs, cs.  
  
as --> [].  
as ... xb --> [a], as.  
  
bs --> [].  
bs ... xc --> xb, [b], bs.  
  
cs --> [].  
cs --> xc, [c], cs.
```

The implementation of XGs is similar to DCGs with two more arguments to hold the

extraposition list. For XG rule involving ..., everything on the left-hand side after the first non-terminal is stacked on the extraposition list, and the symbol is marked. Each subsequent symbol can cause a matching symbol to be popped from the extraposition list.

2.2.4 Gapping Grammars

Gapping Grammars (GGs) [16] can be viewed as a generalization of XGs. These grammars permit one to indicate where intermediate, unspecified substrings can be skipped, left unanalyzed during one part of the parse and possibly reordered by the rule's application for later analysis by other rules. Consider the following example, which is taken from [16].

$$a, \text{gap}(X), b, \text{gap}(Y), c \rightarrow \text{gap}(Y), c, b, \text{gap}(X).$$

This GG rule can be applied successfully to either of the forms a, e, f, b, d, c and a, b, d, e, f, c . Application of the rule yields d, c, b, e, f and d, e, f, c, b respectively. Therefore, the above GG rule can be viewed as a shorthand for, among others, the two rules:

$$a, e, f, b, d, c \rightarrow d, c, b, e, f$$

$$a, b, d, e, f, c \rightarrow d, e, f, c, b.$$

Example 2.4 : Context Sensitive Languages

An example of GGs for the language $\{a^n b^n c^n \mid n \geq 0\}$ is given here:

```

s --> as, bs, cs.

as --> [].

as --> xa, [a], as.

bs --> [].

xa, gap(X), bs --> gap(X), [b], bs, xb.

cs --> [].

xb, gap(X), cs --> gap(X), [c], cs.

```

Unlike GG rules, XG rules only allow the interspersing of gaps in the left hand side, and these gaps are routinely rewritten in their sequential order at the rightmost end of the rule. On the other hand, it seems very difficult to implement an efficient parser for GGs. The naive method is to implement "gap" as a predicate that allows skipping of arbitrary strings (gaps). This method can be extremely inefficient due to backtracking.

An example of a GG rule together with its compilation to Prolog taken from [31] is shown here:

```

object, [and], gap(X), object --> [and], gap(X), object.

object(S0, S) :-
    connects(S0, and, S1),
    append(Gap, S2, S1),
    object(S2, S3),
    connects(S4, term(object), S3), † add dummy nonterminal
    append(Gap, S4, S5), † add "gap"
    connects(S, and, S5). † add "and"

```

2.2.5 Other Logic Grammars

Stabler's Restricting Logic Grammars [55] extend XGs to enhance linguistic expressiveness such as natural treatment of leftward and rightward movement. Other extensions to DCGs include the automatic tree building supported in Restriction Grammars [30] and Modular Logic Grammars [41], and the semantic representation supported in Restriction Grammars, Modular Logic Grammars and Definite Clause Translation Grammars [2].

2.3 Rewriting and Narrowing

Historically, the idea of *rewriting* is an old and useful one. Rewriting uses a set of rules to transform objects to other objects. A set of such rules is called a *rewrite rule system*. Formal grammars [32] are commonly used rewrite rule systems.

In this dissertation we consider only *first-order rewrite rule systems*. A first-order rewrite rule system is a set of rules, each of the form:

$$LHS \Rightarrow RHS$$

where *LHS* and *RHS* are first-order terms. A term is either a variable, or of the form $f(t_1, \dots, t_n)$, $n \geq 0$, where *f* is an *n*-ary function symbol and t_i is a term.

Let *p*, *q* be terms where *p* is not a variable, and let *s* be a nonvariable subterm of *p* (which we write $p = r[s]$). If there exists a rule ($LHS \Rightarrow RHS$) that has no variables in common with *p*, for which there is a substitution θ such that $s = LHS \theta$, and $q = r[RHS \theta]$ (the result of replacing *s* by *RHS* and applying the substitution θ), then we say

p reduces to q.

Let p, q be terms where p is not a variable, and let s be a nonvariable subterm of p (which we write $p = r[s]$). If there exists a rule ($LHS \rightarrow RHS$) that has no variables in common with p , for which there is a most general unifier θ of LHS and s and $q = (r[RHS])\theta$ (the result of replacing s by RHS and applying the substitution θ), then we say p *narrows to* q . Thus, in reduction only variables of LHS can be bound, whereas in narrowing variables of both LHS and p can be bound.

A *narrowing* is a sequence p_0, p_1, \dots such that for all i , whenever p_i and p_{i+1} both exist, $p_i \rightarrow p_{i+1}$. $\xrightarrow{*}$ is defined to be the reflexive transitive closure of \rightarrow . If $p_0 \xrightarrow{*} p_n$, and there is no p_{n+1} such that $p_n \rightarrow p_{n+1}$, then p_n is called a *normal form* of p_0 . If whenever $p_0 \rightarrow q_1$ and $p_0 \rightarrow q_2$, there exists a term q such that $q_1 \xrightarrow{*} q$ and $q_2 \xrightarrow{*} q$, then the rewrite rule system is called *locally confluent*. The system is *confluent* if whenever $p_0 \xrightarrow{*} q_1$ and $p_0 \xrightarrow{*} q_2$, there exists q such that $q_1 \xrightarrow{*} q$ and $q_2 \xrightarrow{*} q$. If no infinite narrowing is possible, the set of rewrite rules is called *noetherian* or *terminating*. A terminating, confluent system is called *canonical*.

Given a term, there can be many narrowings starting with it. Precisely which one is generated is determined by a *narrowing strategy*. A narrowing strategy is *narrowing-complete* if for each term, each of its normal forms can be computed exclusively by use of this strategy.

For instance, given the rewrite rules:

$a \Rightarrow c.$

$b \Rightarrow [].$

$c \Rightarrow c.$

$g(x, []) \Rightarrow [].$

there is a terminating rightmost outermost narrowing sequence

$g(a, b), g(a, []), []$

but the only leftmost outermost narrowing is the non-terminating sequence

$g(a, b), g(c, b), g(c, b), \dots$

2.4 Lazy Evaluation and Logic Programming

2.4.1 Lazy Evaluation

Lazy evaluation is basically a computation scheme in which an expression is evaluated only when there is demand for its value. It has at least two advantages:

- (1) It allows certain computations to terminate more quickly.

Suppose we want to determine whether lists A and B resulting from two computations are identical. We can generate A completely, then generate B completely, and then compare their elements from left to right. However, A and B may be very long but may differ at some early position. The effort of generating A and B beyond that position would be wasted.

- (2) It allows computation with infinite structures.

Suppose A and B above are infinite. An attempt to generate A or B completely would never terminate, and we would never know that A and B differ even if they do.

Many implementations of lazy evaluation have been proposed for functional languages. However, in this section, we emphasize lazy evaluation in the logic programming context.

2.4.2 Lazy Evaluation in Logic Programming

One of the earliest proposals for implementing lazy evaluation in logic programming languages was made in IC-Prolog [10]. By annotating variables in the IC-Prolog program, the interpreter suspends proving a predication if its arguments were not sufficiently bound. A similar idea can be found in Sicstus Prolog, which provides a `freeze` predicate to suspend interpretation of some goal if certain variables are not instantiated.

An efficient technique for implementing lazy evaluation in Prolog was presented in [43,44]. In contrast to previous approaches, this technique does not require any change to the Prolog interpreter. Instead, pure logic programs are written in such a way, that when SLD-resolution interprets these programs, it simulates lazy evaluation.

2.5 Integration of Logic Programming and Rewriting

2.5.1 Logic Programming with Equality

There has always been a lot of interest in extending resolution theorem proving to theories with equality. One classical example is Robinson and Wos's [52] paramodulation. As pointed out in [6], the problem with equality is the complexity of the refutation procedure. It seems that some constraints on equality are needed to achieve the efficient computational properties of logic programs.

An important approach to implement equality in logic programming has been proposed by van Emden and Yukawa [58]. They investigate how equational rewriting can be performed within the context of logic programming. They obtain an efficient Prolog program by compiling the equations to a set of Horn clauses not involving equality. However, this approach is restricted to terminating theories. A similar approach has also been suggested by Yamamoto [62].

Log(F), an recent approach for combining logic programming, rewriting, and lazy evaluation has been suggested by Narain [45]. It rests upon subsuming within logic programming, instead of upon extending it with, rewriting, and lazy evaluation. Log(F) can be used to do lazy functional programming in logic. However, unification is not fully exploited in this system because reduction (a special case of narrowing) has been used.

2.5.2 Narrowing

Reddy [51] proposed interpreting rewrite rules using narrowing. Narrowing an

expression is applying to it the minimum substitution such that the resulting expression is reducible, and then reducing it. In narrowing, input variables can be bound, therefore, it can be used as basis for subsuming both rewriting and logic programming. For instance, with the two rules:

$$\text{append}([], z) = z.$$

$$\text{append}([X|Y], z) = [X|\text{append}(Y, z)].$$

the expression $\text{append}(A, [c]) = [a, b, c]$, cannot be reduced. However, it can be narrowed to $\text{append}([a, b], [c]) = [a, b, c]$, with substitution $A = [a, b]$.

Several proposals to use equational languages with narrowing as logic programming languages have been suggested. Some of these include [7, 25, 51, 57, 58].

2.5.3 Extended Unification and Functional Logic Languages

Let T be an equality theory interpreted as a rewrite rule system. Given terms t_1 and t_2 , if there exists some substitution θ such that $t_1\theta = t_2\theta$ is a logical consequence of T and the equality axioms, we say t_1 and t_2 T -unify.

Unification in equational theories was first studied by Plotkin [50]. A comprehensive survey was given in [20]. We can mention the works of [21, 34, 39] for those equality theories defined by a canonical term-rewriting system. Fay described in [21] a complete T -unification for equational theories T which possess a complete set of reductions as defined by Knuth & Bendix [37]. This algorithm relies basically on using narrowing defined by Lankford [39]. Hullot [34] studied the relations between narrowing

and unification and gave a new version of Fay's algorithm.

Integrating functional and logic languages amounts to modifying the unifier to move from unification in an empty theory towards unification in an equational theory defined by a term-rewriting system [24].

We briefly classify different extended unification approaches to implementing functional languages into three categories:

- (1) The extended unification algorithms use "evaluation" as procedural semantics of the functional language. Different algorithms are proposed in the literature. Some evaluate terms before unification, while others unify terms before evaluation. For instance, given the program

```
factorial(0,1) ← .  
factorial(N,N*M) ← factorial(N-1,M).
```

the goal $\leftarrow \text{factorial}(5,X)$ can succeed with the substitution $\theta = \langle X=5*M \rangle$, if *unification with delayed evaluation* is used. That is, X unifies with 5*M, and at the end of the computation, X can be evaluated to 120. On the other hand with this approach the goal $\leftarrow \text{factorial}(5,120)$ fails because the unification of 120 and 5*M fails.

- (2) The extended unification algorithms use "reduction" as procedural semantics of the functional language [45, 53, 57]. Given two ground terms t_1 and t_2 , the algorithm to determine the unifiability of t_1 and t_2 consists in computing the normal

form of t_1 and t_2 before unifying them. Unification with lazy reduction is possible. For example, consider the functional program

$$\begin{aligned} \text{conc}([], X) &\Rightarrow X. \\ \text{conc}([X|Y], Z) &\Rightarrow [X|\text{conc}(Y, Z)]. \end{aligned}$$

and the logic program

$$\text{append}(X, Y, \text{conc}(X, Y)) \leftarrow .$$

The goal $\leftarrow \text{append}([1, 2, \dots, n], [b], [c | x])$ will fail after one step of lazy reduction. $\text{conc}([1, 2, \dots, n], [b])$ yields $[1 | \text{conc}([2, \dots, n], [b])]$, which fails to unify with $[c | x]$. This unification fails without evaluating $([2, \dots, n], [b])$, and therefore, saves many unnecessary reduction steps.

- (3) The extended unification algorithms use "narrowing" as procedural semantics of the functional language [8, 19, 51]. Given two terms t_1 and t_2 (which may contain variables), the algorithm to determine the unifiability of t_1 and t_2 consists in computing the normal form of t_1 and t_2 before unifying them. Different strategies can be used to compute the normal form(s) of a given term [8, 20, 22, 23].

For instance, consider the term-rewriting system defined as follows:

$$\begin{aligned} 0 + X &\Rightarrow X. \\ s(X) + Y &\Rightarrow s(X+Y). \\ 0 * X &\Rightarrow 0. \\ s(X) * Y &\Rightarrow Y + (X * Y). \end{aligned}$$

The unification of $N + s(0)$ and $s(s(s(0)))$ will give the substitution

$\langle \mathbf{a}(0) \rangle$.

2.6 Conclusions

Many results from different areas has been made. New results can come from combining known results from different areas in some nice ways.

Chapter 3

Narrowing Grammar

3.1 Introduction

Narrowing Grammar is a clear and powerful formalism for describing languages. The semantics of Narrowing Grammar is defined by a special outermost narrowing strategy. This approach gives both a compact formal definition of Narrowing Grammar, and a logic programming implementation. In this chapter we formally define Narrowing Grammar, and illustrate some of its important features with examples.

3.2 Formalism of Narrowing Grammar

Definition 3.1

A *term* is either a variable, or an expression of the form $f(t_1, \dots, t_n)$ where f is a n -ary function symbol, $n \geq 0$, and each t_i is a term. A *ground term* is a term with no variables.

Definition 3.2

A *Narrowing Grammar* is a finite set of rules of the form:

$$LHS \Rightarrow RHS$$

where:

- (1) *LHS* is any term except a variable, and *RHS* is a term.
- (2) If $LHS = f(t_1, \dots, t_n)$, then each t_i is a term in normal form (see definition 3.4 below).

Definition 3.3

Constructor symbols are functors (function symbols) that do not appear as any rule's outermost *LHS* functor.

A *simplified term* is a term whose outermost function symbol is a constructor symbol. By convention also, every variable is taken to be a simplified term. Note that no *LHS* of any rule can be a simplified term. In this thesis we will assume the function symbols for lists (namely, the empty list `[]` and `cons [_|_]`, following Prolog syntax) are constructor symbols. Much in the way that constructors provide a notion of 'values' in a rewrite system, constructors here provide a notion of 'terminal symbols' of a grammar.

Definition 3.4

A term is said to be in *normal form* if all of its subterms are simplified. Since every variable is taken to be a simplified term, a term in normal form can be non-ground.

Definition 3.5

Let p, q be terms where p is not a variable, and let s be a nonvariable subterm of p (which we write $p = r[s]$). If there exists a rule ($LHS \Rightarrow RHS$) (which we assume has no variables in common with p), for which there is a most general unifier θ of LHS and s and $q = (r[RHS])\theta$ (the result of replacing s by RHS and applying the substitution θ), then we say p **narrow_to** q .

A *narrowing* is a sequence of terms p_1, p_2, \dots, p_n such that for each $i, 1 \leq i \leq n-1, p_i$ **narrow_to** p_{i+1} . A narrowing is *successful* if p_n is simplified.

Generally speaking, a rewrite system will specify a mechanism for *selecting* a subterm s from a given term p , to determine what to narrow. This mechanism is then used successively with the actual rewriting mechanism to implement narrowing. Below we define *NU-narrowing*, a special outermost narrowing for Narrowing Grammar.

Definition 3.6: NU-step

$p \rightarrow q$, or p narrows to q in a *NU-step* †, is defined concisely by the following clauses:

†The significance of the prefix 'NU-' in 'NU-step' comes from the fact that we use a special strategy to select a subterm for narrowing, and this strategy selects terms in an outermost, or *Normal* order, fashion. *Unification* is implicitly used by this strategy.

```

nu_step(P,Q) ← nonvar(P), (P => Q).
nu_step(P,Q) ← nonvar(P), (LHS => RHS), not Quite_unify(P,LHS),
    functor(P,F,N), functor(Q,F,N),
    subterm_nu_step(P,Q,1,N).
subterm_nu_step(P,Q,I,N) ← I < N, arg(I,P,A), arg(I,Q,A),
    plus(I,1,I1), subterm_nu_step(P,Q,I1,N).
subterm_nu_step(P,Q,I,N) ← I < N, arg(I,P,A), arg(I,Q,B),
    nu_step(A,B), unify_remaining(P,Q,I,N).
unify_remaining(_,_,N,N).
unify_remaining(P,Q,I,N) ← I < N, plus(I,1,I1), arg(I1,P,A),
    arg(I1,Q,A), unify_remaining(P,Q,I1,N).
not Quite_unify(X,Y) ← functor(X,F,N), functor(Y,F,N), ¬unify(X,Y).

```

Here \neg is negation as failure [40]. We also write $p \xrightarrow{k} q$ if p NU-narrows to q in k steps, and $p \xrightarrow{*} q$ if p NU-narrows to q in zero or more steps.

We can view NU-step as a special outermost narrowing. The term p narrows to q in a NU-step if either $(p \Rightarrow q)$ is an instance of some rule (first clause), or if the replacement of a subterm by the result of a NU-step yields q (second clause). Note that the NU-step definition does not permit a narrowing to begin with a variable.

We have used a logic program to define NU-step mainly out of interest in conciseness. Note that the definition of NU-step is nondeterministic. Nondeterminism permits NU-step to act both as an acceptor and as a generator. For instance, given the Narrowing Grammar rules:

$a \Rightarrow c.$
 $b \Rightarrow [].$
 $c \Rightarrow c.$
 $g(x, []) \Rightarrow [].$

(1) `nu_step/2` can act as an acceptor. The following two goals succeed:

$\leftarrow \text{nu_step}(g(a,b), g(a, [])). \quad \leftarrow \text{nu_step}(g(a,b), g(c,b)).$

(2) `nu_step/2` can act as a generator. With the goal

$\leftarrow \text{nu_step}(g(a,b), x).$

x can be instantiated to either $g(a, [])$ or $g(c,b)$.

Definition 3.7: NU-narrowing

A *NU-narrowing* is a narrowing p_1, p_2, \dots such that for each i , p_i narrows to p_{i+1} in a *NU-step*. Thus we can treat NU-narrowing as the reflexive transitive closure of NU-step.

$\text{nu_narrowing}(X, X).$
 $\text{nu_narrowing}(X, Y) \leftarrow \text{nu_step}(X, Z), \text{nu_narrowing}(Z, Y).$

Definition 3.8: simplification

A *simplification* is a NU-narrowing p_1, p_2, \dots, p_n if p_n is simplified and no other p_i is simplified. Again, we can treat simplification as a kind of reflexive transitive closure of NU-step.

$$\begin{aligned} \text{simplification}(X, X) &\leftarrow \text{simplified}(X). \\ \text{simplification}(X, Z) &\leftarrow \neg \text{simplified}(X), \\ &\quad \text{nu_step}(X, Y), \text{simplification}(Y, Z). \end{aligned}$$

Here \neg is negation as failure [40].

NU-narrowing is not just an outermost narrowing. For instance, given the Narrowing Grammar rules:

$$\begin{aligned} a &\Rightarrow c. \\ b &\Rightarrow []. \\ c &\Rightarrow c. \\ g(X, []) &\Rightarrow []. \end{aligned}$$

then there is a simplification

$$g(a, b) \rightarrow g(a, []) \rightarrow []$$

but the only leftmost outermost narrowing is the nonterminating narrowing:

$$g(a, b), g(c, b), g(c, b), \dots$$

For the rest of the thesis, unless explicitly stated otherwise, by a narrowing we mean a NU-narrowing.

Definition 3.9:

A *stream* is a list of ground terms. A *stream pattern* is a term that has a NU-narrowing to a stream.

3.3 Specifying Patterns with Narrowing Grammar

We illustrate how useful patterns can be developed in Narrowing Grammar with a sequence of examples.

Example 3.1 : Regular Expressions

As we suggested earlier, regular expressions can be defined easily with Narrowing Grammar rules:

```
(X+) => X.  
(X+) => X, (X+).  
(X*) => [].  
(X*) => X, (X*).  
(X;Y) => X.  
(X;Y) => Y.  
([], L) => L.  
([X|L1], L2) => [X|(L1, L2)].
```

The operators '+' and '*' define the familiar Kleene plus and Kleene star regular expressions, respectively. ';' is a disjunctive pattern operator, while ',' defines pattern concatenation, very much like the usual Prolog rules for `append`.

For example, the narrowing of `([a]+, [b])` to `[a, a, b]` along with the rules used in each step of the narrowing is as follows:

<i>Rewritten term</i>	<i>Rule used in rewriting</i>
$([a]^+, [b])$	
$\rightarrow (([a], [a]^+), [b])$	$(X^+) \Rightarrow X, (X^+).$
$\rightarrow ([a] ([], [a]^+), [b])$	$([X L1], L2) \Rightarrow [X (L1, L2)].$
$\rightarrow [a] (([], [a]^+), [b])$	$([X L1], L2) \Rightarrow [X (L1, L2)].$
$\rightarrow [a] ([a]^+, [b])$	$([], L) \Rightarrow L.$
$\rightarrow [a] ([a], [b])$	$(X^+) \Rightarrow X.$
$\rightarrow [a, a] ([], [b])$	$([X L1], L2) \Rightarrow [X (L1, L2)].$
$\rightarrow [a, a, b]$	$([], L) \Rightarrow L.$

Example 3.2 : Counting the Occurrences of a Pattern

Suppose we wish to count the number of times an uninterrupted sequence of one or more a 's is followed by a b in a stream. This pattern can be represented by the regular expression $([a]^+, [b])$, and we can count the number of its occurrences with the pattern

`number(([a]^+, [b]), Total)`

if we include the following Narrowing Grammar rules for `number`:

`number(Pattern, Total) => number(Pattern, Total, 0).`

`number(Pattern, Total, Total) => [end_of_file].`

`number(Pattern, Total, Count) =>`

`Pattern, number(Pattern, Total, plus(Count, 1)).`

Here `total` is unified with the number of occurrences of `Pattern` in a stream that is matched with the pattern `number(Pattern, Total)`, and `[end_of_file]` is a special

terminal symbol that delimits the end of stream. We assume `plus(x, 1)` yields the value of `x+1` when simplified.

From the example above it is clear that the Narrowing Grammar rules have a functional flavor. Stream operators are easily expressed using recursive functional programs. In addition, `number` is *higher-order* because it takes an arbitrary pattern as an argument. The definitions for `'+'`, `'*'`, `';'`, `'\''`, etc., above are also higher-order in that they have rules like

$$(x+) \Rightarrow x.$$

which rewrite terms to their arguments.

Example 3.3 : Coroutined Pattern Matching

Suppose we want to specify that `b` precedes `a` and `c` precedes `a` in a string, but the relative order of `b` and `c` is not important. We can use the pattern

```
precedes([b], [a]) // precedes([c], [a])
```

provided that we also include the following Narrowing Grammar rules:

```
([X|Xs] // [X|Ys]) => [X|Xs//Ys].
```

```
([] // []) => [].
```

```
precedes(X, Y) => eventually(X), eventually(Y).
```

```
eventually(X) => X.
```

```
eventually(X) => [], eventually(X).
```

The operator `//` takes two patterns as arguments, narrows them to `[X|Xs]` and

$[x|y]$ respectively, and then yields $[x|xs//ys]$. Thus `//` is a pattern matching primitive that requires both argument patterns to generate or accept streams of the same length. This example shows that multiple patterns in a stream can be simultaneously generated or accepted (i.e., coroutined) easily with `//`.

Example 3.4 : Non-Context Free Languages

Consider the following Narrowing Grammar rules†:

```
s_abc => ab_c // a_bc.
```

```
ab_c => pair([a], [b]), [c]*.
```

```
a_bc => [a]*, pair([b], [c]).
```

```
pair(x, y) => [].
```

```
pair(x, y) => x, pair(x, y), y.
```

This grammar defines the non-context-free language $\{a^n b^n c^n | n \geq 0\}$ using only context-free-like constructions. The first rule for `s_abc` imposes simultaneous (parallel) constraints ($a^n b^n c^*$ and $a^* b^n c^n$) on streams generated by the grammar.

3.4 What is New about Narrowing Grammar

In this section we summarize several important features of Narrowing Grammar. Some of these features are novel in the context of grammar formalisms, while others are not. The combination of these features is certainly new and interesting, in any event.

† Fernando Pereira suggested this example.

3.4.1 New Model of Acceptance in Logic Grammar

Previously, we have described how Narrowing Grammar rules operate as pattern generators or specifiers. In this section, we show that they can also operate as acceptors. Our approach for pattern acceptance is to introduce a new pair of Narrowing Grammar rules specifying pattern matching. The entire definition is the following pair of rules for `match`:

$$\text{match}([], S) \Rightarrow S.$$
$$\text{match}([X|L], [X|S]) \Rightarrow \text{match}(L, S).$$

`match` can take a pattern as its first argument, and an input stream as its second argument. If the pattern narrows to the empty list `[]`, `match` simply succeeds. On the other hand, if the pattern narrows to `[X|L]`, then the second argument to `match` must also narrow to `[X|S]`. Intuitively, `match` can be thought of as *applying* a pattern (the first argument) to an input stream (the second argument), in an attempt to find a prefix of the stream that the grammar defining the pattern can generate.

Pattern acceptance is requested explicitly with `match`. As a simple example, consider the following derivation illustrating how `match` accepts the stream `[a, a, b]`:

```

match(({a}+, [b]), [a, a, b])
  → match((({a}, [a}+), [b]), [a, a, b])
  → match((a|([], [a}+), [b]), [a, a, b])
  → match(a|([], [a}+), [b]), [a, a, b])
  → match((([], [a}+), [b]), [a, b])
  → match((a}+, [b]), [a, b])
  → match((a}, [b]), [a, b])
  → match(a|([], [b]), [a, b])
  → match((([], [b]), [b])
  → match(b}, [b])
  → match([], [])
  → []

```

There is a certain elegance to this; the rules of the Narrowing Grammar by themselves act as pattern generators, but when applied with `match` they act like an acceptor, or parser. This acceptance/generation duality is familiar to users of Definite Clause Grammar [47], and the ability to employ grammars both as acceptors and as generators has a number of uses [26].

3.4.2 Higher-order Specification, Extensibility, and Modularity

Narrowing Grammar is higher-order. Specifically, Narrowing Grammar is higher-order in the sense that patterns can be passed as input arguments to patterns, and patterns can yield patterns as outputs.

For example, the enumeration pattern `number(_, _)` defined in Example 3.2 is higher-order, as its first argument is a pattern. The whole pattern `([a]+, [b])` can be

used as an argument, as in:

```
number( ([a]+, [b]), Total ).
```

It is well known that a higher-order capability increases expressiveness of a language, since it makes it possible to develop generic functions that can be combined in a multitude of ways [18]. As a consequence, Narrowing Grammar rules are highly reusable and can be usefully collected in a library. In short, Narrowing Grammar is modular. Narrowing Grammar is also extensible, since it permits definition of new grammatical constructs, as the `number` and `//` examples showed earlier.

3.4.3 Lazy Evaluation, Stream Processing, and Coroutining

Leftmost outermost reduction is also called normal-order reduction. This outside-in evaluation of an expression tends to evaluate arguments of function symbols only on demand – i.e., only when the argument values are needed. That is, outside-in evaluation can be ‘lazy’.

Lazy evaluation is intimately related with a programming paradigm referred to as *stream processing* [46]. Note that in this thesis, a stream pattern is a term that will narrow to a list of ground terms. We are not aware of previous work connecting stream processing and grammars, although the connection is a natural one. Lazy evaluation and stream processing also have intimate connections with *coroutining* [27]. Coroutining is the interleaving of evaluation (here, narrowing) of two expressions. It is applicable frequently in stream processing. For example, narrowing of the

stream pattern

`match(([a]+, [b]), [a, a, b])`

interleaves the narrowing of `([a]+, [b])` with the narrowing of `match(_, [a, a, b])`. The sample narrowing of this pattern in section 3.3.1 shows the actual interleaving - first `([a]+, [b])` is narrowed for three NU-steps, then `match(_, [a, a, b])` for one NU-step, then `([a]+, [b])` for three NU-steps, then `match(_, [a, b])` for one NU-step, then `([], [b])` for one NU-step, and finally `match(_, _)` for two NU-steps. The effect of special outermost narrowing of the combined stream pattern is precisely to interleave these two narrowings. Similarly narrowing of the pattern

`([a]+, [b])`

interleaves the narrowing of `[a]+` with the narrowing of `(_, [b])`.

A specific advantage of lazy evaluation in parsing, then, is that coroutined recognition of multiple patterns in a stream becomes accessible to the grammar writer. The corouting rules

`([X|Xs] // [X|Ys]) => [X|Xs//Ys].`

`([] // []) => [].`

make explicitly coroutined pattern matching possible. Essentially `//` narrows each of its arguments, obtaining respectively `[X|Xs]` and `[X|Ys]`. Having obtained these simplified terms, it suspends narrowing of `Xs` and `Ys` until further evaluation is necessary. An immediate advantage of lazy evaluation here is reduced computation. Without lazy evaluation, both arguments would be completely simplified before pat-

term matching took place; failure to unify the heads of these completely evaluated arguments would then mean that many unnecessary narrowing steps on the tails of the arguments had been performed.

As another example, computation with infinite structures, such as stream processing, can be interpreted more elegantly in the context of lazy rewriting than in that of logic programming. Suppose we want to determine the first n elements of a stream. In Prolog we could write:

```
first(0, X, []).  
first(s(X), [U|V], [U|W]) :- first(X, V, W).  
stream([_|X]) :- stream(X).
```

Now, if we want to compute the first element of the list computed by `stream/1`, we might be tempted to use the query

```
?- stream(X), first(s(0), X, Z).
```

However, since `stream/1` might generate an infinite stream of terms, this query will not always work. By arranging the subgoals as in Parlog [11] or Concurrent Prolog [54], `stream/1` and `first/3` can be coroutined. That is, whenever a new element is generated by `stream/1`, control transfers to `first/3`. Thus by extending SLD-resolution to a concurrent form of deduction we can succeed.

On the other hand, we can express this problem more elegantly with Narrowing Grammar. With the definition

`first(0,X) => [].`

`first(s(X), [U|V]) => [U | first(X,V)].`

if `stream` is defined by

`stream => [_ | stream].`

then the normal form of `first(s(0),stream)` is `[_]`. The problem is easily addressed within the framework of narrowing.

3.4.4 Limitations of First-order Logic Grammars

First-order logic grammars do not permit direct specification of grammar rules of the form:

`goal(X) --> ...,X,...`

where `x` is a variable. Therefore, it is hard to write grammars that behave like `number` given earlier. This is a basic limitation.

Abramson [3] has addressed this limitation by introducing a meta-nonterminal symbol, written `meta(X)`, where `X` may be instantiated to any terminal or nonterminal symbol. During parsing, an `X` is to be recognized at the point where `meta(X)` is used in a grammar rule. Abramson suggested two ways to implement `meta(X)`. The first method makes an interpretive metacall wherever `meta(X)` is used. (This is a special case of the approach used by the `phrase/3` metapredicate in Quintus and Sicstus Prolog.) The other approach is to preprocess the rules containing `meta(X)` so as to gen-

erate a new set of rules with no calls to *meta(X)*. However, this preprocessing can generate extra nonterminals and rules.

The same problem was pointed out in [42], where Moss proposed a special translation technique by using a single predicate name for non-terminals. For instance, Moss translates the DCG rule

```
goal(X) --> X
```

to the Prolog clause

```
nonterminal(goal(X), S0, S) :- nonterminal(X, S0, S).
```

There is one final limitation. Even with the techniques suggested by Abramson and Moss, the lazy evaluation or coroutining aspects of Narrowing Grammar are not easily attained with first-order logic grammars. To attain them, a new evaluation strategy is needed for these grammars, implemented via either a meta-interpreter or a compiler like that in next chapter.

Chapter 4

Compilation of Narrowing Grammar to Logic Programs

4.1 Compilation Algorithm

We describe an algorithm to compile Narrowing Grammar to Prolog programs. It turns out that Prolog-derivation (depth-first SLD-derivation with left-to-right goal selection and top-to-bottom clause selection with backtracking) will implement *NU-narrowing* on these programs. The compilation of a Narrowing Grammar rule into a Prolog clause combines information about the rule and the control of NU-narrowing when interpreting that rule. One interesting aspect of the compilation algorithm is its use of a suitable 'equality' predicate.

Algorithm 4.1 : Compilation of Narrowing Grammar to Prolog

- (1) For each n -ary constructor symbol c , $n \geq 0$, and for distinct Prolog variables X_1, \dots, X_n , generate the clause:

`simplify(c(X1, ..., Xn), c(X1, ..., Xn)).`

- (2) For each rule $f(L_1, \dots, L_m) \Rightarrow RHS$, let A_1, \dots, A_m, Out be distinct Prolog variables not occurring in the rule, and generate the clause:

```
simplify( $f(A_1, \dots, A_m), Out$ ) :-
    nu_narrow( $A_1, L_1$ ),
    ...,
    nu_narrow( $A_m, L_m$ ),
    simplify( $RHS, Out$ ).
```

Algorithm 4.1 does not deal with 'impure' features in Narrowing Grammar rules, such as cuts. However, it is not difficult to extend the compilation to include such features.

This compilation also requires definition of the `nu_narrow/2` predicate:

Definition 4.1: `nu_narrow(X, Y)`

```
nu_narrow(X, X) :- !.
nu_narrow(X, Y) :- simplify(X, Z),
    nu_narrow_subterms(Z, Y).

nu_narrow_subterms(X, Y) :-
    functor(X, F, N), functor(Y, F, N),
    nu_narrow_subterms(X, Y, 0, N).

nu_narrow_subterms(_, _, N, N).
nu_narrow_subterms(X, Y, I, N) :-
    I < N, plus(I, 1, I1), arg(I1, X, A),
    arg(I1, Y, B), nu_narrow(A, B),
    nu_narrow_subterms(X, Y, I1, N).
```

This definition of `nu_narrow/2` is an approximation to the NU-narrowing relation

defined in Chapter three. Note the ! (cut) is the only impure construct in this definition. When we view the compiled program as a pure logic program, and Prolog-derivation is replaced by SLD-derivation, then implicitly this ! will be removed from the definition. Note also the use of ! in the definition is solely for efficiency. Once a term (the first argument of a `nu_narrow/2` subgoal) unifies with an argument of `LHS` (the second argument of a `nu_narrow/2` subgoal), it is not necessary (under some restrictions described later) to consider any other narrowing from the first to the second argument of `nu_narrow/2`.

The table below lists some Narrowing Grammar rules together with the Prolog clauses resulting from their compilation.

Narrowing Grammar Rules	Prolog Clauses
<pre>match([], S) => S.</pre>	<pre>simplify(match(A, B), C) :- nu_narrow(A, []), nu_narrow(B, S), simplify(S, C).</pre>
<pre>match([X L], [X S]) => match(L, S).</pre>	<pre>simplify(match(A, B), C) :- nu_narrow(A, [X L]), nu_narrow(B, [X S]), simplify(match(L, S), C).</pre>
<pre>(X+) => X.</pre>	<pre>simplify((A+), B) :- nu_narrow(A, X), simplify(X, B).</pre>
<pre>(X+) => X, (X+).</pre>	<pre>simplify((A+), B) :- nu_narrow(A, X), simplify((X, (X+)), B).</pre>
<pre>([], L) => L.</pre>	<pre>simplify((A, B), C) :- nu_narrow(A, []), nu_narrow(B, L), simplify(L, C).</pre>
<pre>([X L1], L2) => [X (L1, L2)].</pre>	<pre>simplify((A, B), C) :- nu_narrow(A, [X L1]), nu_narrow(B, L2), simplify([X (L1, L2)], C).</pre>

Here we also give an example of a simplification together with one of its SLD-derivations (this SLD-derivation is also a Prolog-derivation):

Simplification:

$([a]^+, [b])$
→ $(([a], [a]^+), [b])$
→ $([a] ([], [a]^+), [b])$
→ $[a] ([], [a]^+), [b]$

SLD-derivation:

```

← simplify([a]+, [b]), Out)

← simplify([a]+, [X|L1]),
  nu_narrow([b], L2),
  simplify([X|(L1,L2)], Out)

← nu_narrow([a], Y),
  simplify([Y,Y+], [X|L1]),
  nu_narrow([b], L2),
  simplify([X|(L1,L2)], Out)

← simplify([a],[a]+), [X|L1]),
  nu_narrow([b], L2),
  simplify([X|(L1,L2)], Out)

← simplify([a], [Z|Z1]),
  nu_narrow([a]+, Z2),
  simplify([Z|(Z1,Z2)], [X|L1]),
  nu_narrow([b], L2),
  simplify([X|(L1,L2)], Out)

← nu_narrow([a]+, Z2),
  simplify([a|([],Z2)], [X|L1]),
  nu_narrow([b], L2),
  simplify([X|(L1,L2)], Out)

← simplify([a|([], [a]+)], [X|L1]),
  nu_narrow([b], L2),
  simplify([X|(L1,L2)], Out)

← nu_narrow([b], L2),
  simplify([a|([], [a]+), L2]), Out)

← simplify([a|([], [a]+), [b]], Out)

← □

```

Note that `nu_narrow/2` subgoals are replaced by `simplify/2` subgoals in this example if the second argument of `nu_narrow/2` is a simplified term with only variables as arguments.

`simplify/2` guarantees its result (the second argument) will be simplified. That is, the function symbol of the result will be a constructor. Also, we can show that `simplify/2` behaves like `simplification/2`. Consider a derivation from the goal `simplify(f(t1, ..., tm), Z)` which uses the Prolog clause

```

simplify(f(X1, ..., Xm), Out) :-
    nu_narrow(X1, Y1),
    ...,
    nu_narrow(Xm, Ym),
    simplify(RHS, Out).

```

resulting from $f(Y_1, \dots, Y_m) \Rightarrow RHS$. In left-to-right Prolog-derivation with this clause, the `nu_narrow/2` subgoals are satisfied first, each effecting either a unification (first clause of `nu_narrow/2`) or a recursive simplification (second clause of `nu_narrow/2`). These are followed by a derivation from the subgoal `simplify(RHS, Out)`, which also recursively effects a simplification of `RHS`. Concatenating these simplifications, we find that `simplify/2` effects a simplification. We state this more formally by the following theorem:

Theorem 4.1: If X and Y are terms such that `simplify(X, Y)` has a successful Prolog-derivation, then `simplification(X, Y)` has a successful SLD-derivation.

The proofs of this theorem and a limited converse appear later in this chapter. The converse of the theorem is also true *if* we replace Prolog-derivation by SLD-derivation, and we restrict the use of duplicate variables on the left hand sides of Narrowing Grammar rules in certain ways beyond the restrictions in Definition 3.2. For instance, consider the Narrowing Grammar rules:

$a \Rightarrow a.$

$a \Rightarrow [].$

The compiled Prolog code generated from Algorithm 4.1 is:

`simplify(a,X) :- simplify(a,X).`

`simplify(a,X) :- simplify([],X).`

There is a NU-narrowing $a \rightarrow []$ but the Prolog query `?- simplify(a,X)` loops. It is well known that the depth-first evaluation strategy of Prolog is not complete, i.e., sometimes fails to find a solution even though one exists. The SLD-derivation strategy, on the other hand, is complete [4, 28]. Therefore, in order to make the converse of Theorem 4.1 true, we must replace Prolog-derivation by SLD-derivation.

To gain more insight about the converse of Theorem 4.1, consider another example, given four Narrowing Grammar rules:

$a \Rightarrow c.$

$b \Rightarrow c.$

$c \Rightarrow c.$

$f(x,x) \Rightarrow [].$

with these rules there is a NU-narrowing

```
f(a,b)
  → f(c,b)
  → f(c,c)
  → []
```

With the compiled Prolog code generated from Algorithm 4.1:

```
simplify(a, Out) :- simplify(c, Out).
simplify(b, Out) :- simplify(c, Out).
simplify(c, Out) :- simplify(c, Out).
simplify(f(A,B), Out) :- nu_narrow(A, X),
                        nu_narrow(B, X),
                        simplify([], Out).
```

the Prolog query:

```
?- simplify(f(a,b), Out).
```

is reduced to the query

```
?- nu_narrow(a, X), nu_narrow(b, X), simplify([], Out).
```

The subgoal `nu_narrow(a,X)` succeeds with `x` instantiated to `a`. Now `nu_narrow(b,a)` fails and the query fails because the unification of the first subgoal `nu_narrow(a,X)` is committed (note the use of `!` in the definition of `nu_narrow/2`). The cause of the failure of the subgoal `?- nu_narrow(b,a)` is that `b` doesn't have a simplified form. The same goal still fails even if Prolog-derivation is replaced by SLD-derivation. Therefore, in order to make the converse of Theorem 4.1 true, we

must restrict the use of duplicate variables on the left hand sides of Narrowing Grammar rules in certain ways.

Definition 4.2 : Non-linear Term

A term p is *non-linear* if for some subterm $f(t_1, \dots, t_m)$ of p (including p itself) there exists a Narrowing Grammar rule $f(L_1, \dots, L_m) \Rightarrow RHS$ such that for $j = 1, \dots, m$, t_j unifies with L_j , but $f(t_1, \dots, t_m)$ does not unify with $f(L_1, \dots, L_m)$.

Definition 4.3 : Non-linear Simplification

A simplification $p_1, \dots, p_i, \dots, p_n$ is *non-linear* if for some i , $i < n$, p_i is a non-linear term.

For instance, consider the Narrowing Grammar rules:

$$a \Rightarrow c.$$

$$b \Rightarrow c.$$

$$c \Rightarrow c.$$

$$f(x, x) \Rightarrow [].$$

Both $f(a, b)$ and $f(c, b)$ are non-linear terms. The simplification

$$f(a, b) \rightarrow f(c, b) \rightarrow f(c, c) \rightarrow []$$

is non-linear. However, the goal

$$\leftarrow \text{simplify}(f(a, b), []).$$

does not have a successful SLD-derivation. Therefore, in order to make the converse

of Theorem 4.1 true, we also require no terms in the simplification to be non-linear.

Then we state the converse as follows:

Theorem 4.2 : Suppose X and Z are terms. If $\text{simplification}(X, Z)$ has a successful SLD-derivation which does not correspond to a non-linear simplification, then $\text{simplify}(X, Z)$ also has a successful SLD-derivation.

4.2 Correctness of the Narrowing Grammar Compilation Algorithm

In this section, we prove Theorems 4.1 and 4.2.

Definition 4.4

Let X and Y be two terms. We write $X \Rightarrow Y$ if there is a rule $LHS \Rightarrow RHS$ such that $X\theta = LHS\theta$ for some θ and $Y = RHS\theta$. Otherwise $X \not\Rightarrow Y$.

Theorem 4.1 : If X and Y are terms such that $\text{simplify}(X, Y)$ has a successful Prolog-derivation, then $\text{simplification}(X, Y)$ has a successful SLD-derivation.

Plan of Proof : Induction on the length of a successful Prolog-derivation of $\text{simplify}(X, Y)$. We show that in this derivation there is some subgoal $\text{simplify}(Z, Y)$, for some term Z not equal to X , such that $\text{nu_narrowing}(X, Z)$ has a successful SLD-derivation. Since $\text{simplify}(Z, Y)$ has a successful Prolog-derivation, by the induction hypothesis, $\text{simplification}(Z, Y)$ has a successful SLD-derivation. So

$\text{simplification}(X, Y)$ has a successful SLD-derivation.

Proof : By induction on the length k of a successful Prolog-derivation starting at $\text{simplify}(X, Y)$. If $k = 1$ then there is a clause

$$\text{simplify}(c(A_1, \dots, A_m), c(A_1, \dots, A_m))$$

such that $\text{simplify}(X, Y)$ unifies with this clause. Thus X is simplified and $\text{simplification}(X, Y)$ has a successful SLD-derivation.

Let $k > 1$, and assume the theorem holds for all successful Prolog-derivations of length less than k . Let $X = f(t_1, \dots, t_m)$ for some non-constructor f and terms t_1, \dots, t_m . Since $\text{simplify}(X, Y)$ succeeds, there is a clause

$$\begin{aligned} \text{simplify}(f(A_1, \dots, A_m), B) :- \\ \text{nu_narrow}(A_1, L_1), \\ \dots, \\ \text{nu_narrow}(A_m, L_m), \\ \text{simplify}(RHS, B). \end{aligned}$$

which is the compilation of a Narrowing Grammar rule

$$f(L_1, \dots, L_m) \Rightarrow RHS.$$

Moreover, $\text{simplify}(X, Y)$ unifies with the head of the above clause with m.g.u. $\tau = \langle A_1=t_1 \rangle, \dots, \langle A_m=t_m \rangle, \langle B=Y \rangle$ and $(\text{nu_narrow}(A_1, L_1), \dots, \text{nu_narrow}(A_m, L_m), \text{simplify}(RHS, B))\tau$ has a successful Prolog-derivation of length $k - 1$.

There are two possibilities:

- (1) If $X \Rightarrow RHS\tau$, then in this case the subgoals $nu_narrow(t_i, L_i)$ all succeed with just unifications, using the first clause for $nu_narrow/2$. By the induction hypothesis, $simplification(RHS\tau, Y)$ has a successful SLD-derivation. Hence $simplification(X, Y)$ has a successful SLD-derivation.
- (2) If $X \not\Rightarrow RHS\tau$, consider $nu_narrow(t_i, L_i)$ for some i .
- (a) If L_i and t_i are unifiable, then $nu_narrow(t_i, L_i)$ leads to a unification, that is, $nu_narrowing(t_i, L_i)$ has an SLD-derivation.
- (b) If L_i and t_i are not unifiable, the Prolog-derivation of $nu_narrow(t_i, L_i)$ leads immediately to the goal ($simplify(t_i, W), nu_narrow_subterms(W, L_i)$) where W is a new variable. Since the length of successful Prolog-derivation of $simplify(t_i, W)$ is less than k , by the induction hypothesis, $simplification(t_i, W)$ has a successful SLD-derivation with some binding σ . Now, consider $nu_narrow_subterms(W, L_i)\sigma$. First notice that both $W\sigma$ and $L_i\sigma$ are simplified (L_i cannot be a variable, otherwise case (a) will apply. $L_i\sigma$ cannot be non-simplified, otherwise, the goal $nu_narrow_subterms(W, L_i)$ will fail). Let $W\sigma = c(h_1, \dots, h_l)$ and $L_i\sigma = c(h_1', \dots, h_l')$ where c is a constructor. Thus Prolog reduces $nu_narrow_subterms(W, L_i)\sigma$ to ($nu_narrow(h_1, h_1')\sigma, \dots, nu_narrow(h_l, h_l')\sigma$), and each of these $nu_narrow(h_r, h_r')\sigma$ for any r either leads to a unification or recursively calls ($simplify(h_r, Out)\sigma, nu_narrow_subterms(Out, h_r')\sigma$). This recursion is finite and must terminate. Since the length of the Prolog-derivation of $simplify(h_r, Out)\sigma$ is

less than k , by the induction hypothesis, $\text{simplification}(h_r, Out)\sigma$ has a successful SLD-derivation. Similarly $\text{nu_narrow_subterms}(Out, h_r)\sigma$ reduces under Prolog-derivation to a finite number of $\text{simplify}/2$ subgoals, and each of these subgoals has a successful Prolog-derivation of length less than k . By the induction hypothesis, each corresponding $\text{simplification}/2$ has a successful SLD-derivation. The concatenation of these simplifications is a NU-narrowing, that is, $\text{nu_narrowing}(W, L_i)$ has a successful SLD-derivation. Therefore, together with the fact that $\text{simplification}(t_i, W)$ has a successful SLD-derivation, $\text{nu_narrowing}(t_i, L_i)$, has a successful SLD-derivation with accumulated binding θ_i . Since $f(t_1, \dots, t_i, \dots, t_m) \neq \text{RHS}\tau$, and L_i does not unify with t_i , therefore, $\text{nu_narrowing}(f(t_1, \dots, t_i, \dots, t_m), f(t_1, \dots, L_i, \dots, t_m))\theta_i$ has a successful SLD-derivation.

This argument can be applied to each $\text{nu_narrow}(t_j, L_j)$ subgoal, for $j \in \{1, \dots, m\}$. Applying the argument sequentially we obtain $\text{nu_narrowing}(f(t_1, \dots, t_m), f(L_1, \dots, L_m))\theta$ where θ is the accumulated set of bindings $\theta_1 \dots \theta_m$.

In both possible cases, $\text{simplify}(\text{RHS}\theta, Y)$ succeeds and the length of its Prolog-derivation is less than k . By the induction hypothesis, there is a successful SLD-derivation $\text{simplification}(\text{RHS}\theta, Y)$. Since $f(L_1, \dots, L_m)\theta \Rightarrow \text{RHS}\theta$, hence there is a successful SLD-derivation for $\text{simplification}(X, Y)$.

Q.E.D.

Theorem 4.2 : If $\text{simplification}(X, Z)$ has a successful SLD-derivation which does not correspond to a non-linear simplification, then $\text{simplify}(X, Z)$ also has a successful SLD-derivation.

Plan of Proof:

Induction on the length of $\text{simplification}(X, Z)$. We show that there is some Y_k not equal to X such that an SLD-derivation of $\text{simplify}(X, Z)$ contains the goal $\text{simplify}(Y_k, Z)$. Since $\text{simplification}(Y_k, Z)$ has a successful SLD-derivation, then by the induction hypothesis, $\text{simplify}(Y_k, Z)$ has a successful SLD-derivation. Hence $\text{simplify}(X, Z)$ has a successful SLD-derivation.

Proof:

By induction on the length n of the SLD-derivation of $\text{simplification}(X, Z)$.

If $n = 1$ then X is simplified. In particular, $X = c(t_1, \dots, t_m)$ where c is an m -ary constructor symbol, and t_1, \dots, t_m are terms. There is a clause

$$\text{simplify}(c(A_1, \dots, A_m), c(A_1, \dots, A_m))$$

where each A_i is a distinct variable, so $\text{simplify}(X, Z)$ has a successful SLD-derivation.

Let $n > 1$ and $X = f(t_1, \dots, t_m)$, where f is a non-constructor symbol and each t_i is a term. Assume the theorem holds for all simplifications of length less than n .

Here X is not simplified. Letting $X=Y_0$, we know the simplification has a correspond-

ing SLD-derivation that has the intermediate goals $\text{nu_step}(X, Y_1), \text{nu_step}(Y_1, Y_2)$
 $\dots, \text{nu_step}(Y_{k-1}, Y_k), \text{simplification}(Y_k, Z)$ where k is the least index such that
 $Y_{k-1} \Rightarrow Y_k$, i.e., $0 \leq j < k-1, Y_j \not\Rightarrow Y_{j+1}$. Hence $Y_{k-1} = f(s_1, \dots, s_m)$ for some terms
 s_1, \dots, s_m , and $\text{nu_narrowing}(t_i, s_i)$ has a successful derivation for each $i \in \{1, \dots, m\}$.
 Since $Y_{k-1} \Rightarrow Y_k$, there is some rule $f(L_1, \dots, L_m) \Rightarrow \text{RHS}$ such that Y_{k-1} and
 $f(L_1, \dots, L_m)$ unify with m.g.u. θ , and $Y_k = \text{RHS}\theta$. Therefore, s_i and L_i unify for
 each $i \in \{1, \dots, m\}$.

The rule $f(L_1, \dots, L_m) \Rightarrow \text{RHS}$ is compiled into the Horn clause

```

simplify(f(A1, ..., Am), Out) :-
    nu_narrow(A1, L1),
    ...,
    nu_narrow(Am, Lm),
    simplify(RHS, Out).
  
```

in accordance with the compilation rules stated in Algorithm 4.1.

Consider the goal $\text{simplify}(X, Z)$, i.e., $\text{simplify}(f(t_1, \dots, t_m), Z)$. It unifies with
 $\text{simplify}(f(A_1, \dots, A_m), \text{Out})$ with m.g.u. $\tau = \{ \langle A_1 = t_1 \rangle, \dots, \langle A_m = t_m \rangle, \langle \text{Out} = Z \rangle \}$ and its
 immediate descendant in a Prolog-derivation is $(\text{nu_narrow}(A_1, L_1) \dots,$
 $\text{nu_narrow}(A_m, L_m), \text{simplify}(\text{RHS}, \text{Out}))\tau$, i.e., $(\text{nu_narrow}(t_1, L_1) \dots,$
 $\text{nu_narrow}(t_m, L_m), \text{simplify}(\text{RHS}, Z))$.

Since $\text{simplification}(X, Z)$ does not correspond to a non-linear simplification, if for
 $i = 1, \dots, m$, t_i unifies with L_i , then $f(t_1, \dots, t_m)$ unifies with the left hand side of a rule,
 $f(L_1, \dots, L_m)$. Consider $\text{nu_narrow}(t_i, L_i)$ for some i .

- (1) If L_i and t_i are unifiable, $\text{nu_narrow}(t_i, L_i)$ has a successful SLD-derivation.
- (2) If L_i and t_i are not unifiable, SLD-derivation reduces $\text{nu_narrow}(t_i, L_i)$ to ($\text{simplify}(t_i, T), \text{nu_narrow_subterms}(T, L_i)$) where T is a new variable. Since $\text{nu_narrowing}(t_i, s_i)$ has a successful SLD-derivation and its length is less than k (L_i is not a variable, otherwise (1) will apply. L_i is simplified, otherwise non-linearity property holds or (1) will apply. Since L_i and s_i unify, and therefore s_i is simplified), by the induction hypothesis, $\text{simplify}(t_i, T)$ has a successful SLD-derivation with substitution σ where $T\sigma$ is simplified and $\text{nu_narrowing}(T, s_i)\sigma$ has a successful SLD-derivation. Let $T\sigma = c(h_1, \dots, h_i)$ and $L_i\sigma = c(h_1', \dots, h_i')$ where c is a constructor. Now, which SLD-derivation reduces $\text{nu_narrow_subterms}(T, L_i)\sigma$, to ($\text{nu_narrow}(h_1, h_1'), \dots, \text{nu_narrow}(h_i, h_i')$). Each of the $\text{nu_narrow}(h_r, h_r')$ subgoals either leads to a unification or recursively reduces to ($\text{simplify}(h_r, \text{Out}), \text{nu_narrow_subterms}(\text{Out}, h_r')$) in the derivation. This recursion is finite and must terminate. Since $\text{nu_narrowing}(T, L_i)\sigma$ has a successful SLD-derivation of length less than k , by the induction hypothesis, each of the $\text{simplify}/2$ subgoals or unifications derived from $\text{nu_narrow_subterms}(T, L_i)\sigma$ has a successful SLD-derivation, and therefore, $\text{nu_narrow_subterms}(T, L_i)\sigma$ has a successful SLD-derivation.

By repeating the same argument for each $\text{nu_narrow}(t_i, L_i)$, an SLD-derivation starting at $\text{simplify}(X, Z)$ contains $\text{simplify}(RHS\theta, Z)$ as a member where $\theta = \theta_1 \cdots \theta_m$. But $RHS\theta = Y_k$. Hence the SLD-derivation starting at $\text{simplify}(X, Z)$ contains $\text{simplify}(Y_k, Z)$. Since the length of the SLD-derivation of $\text{simplification}(Y_k, Z)$

is less than n , by the induction hypothesis, $\text{simplify}(Y_k, Z)$ has a successful SLD-derivation. Thus, the SLD-derivation of $\text{simplify}(X, Z)$ succeeds.

Q.E.D.

Chapter 5

Completeness Issues and Extensions of Narrowing Grammar

5.1 Introduction

We would like to prove the 'completeness' of NU-narrowing for Narrowing Grammar. That is, whenever a term can be simplified, we wish to prove it can be simplified by repeatedly applying NU-step. In chapters three and four, we defined Narrowing Grammar and proposed the compilation Algorithm 4.1 to implement it. We then showed Theorem 4.2 could be proven if we prevented simplifications from being non-linear. It turns out that a similar requirement is necessary in order to prove the Narrowing-completeness of Narrowing Grammar for simplified forms. We state and prove the completeness of Narrowing Grammar for simplified forms and extend Algorithm 4.1 to handle some non-linear simplifications.

5.2 Narrowing-Completeness of Narrowing Grammar for Simplified Forms

The choice of a narrowing strategy has an important bearing upon the issue of completeness. A strategy is *narrowing-complete* if whenever a term can be simplified to a simplified form, it can be simplified by the use of this strategy.

We first give an example showing unrestricted use of duplicate variables among arguments on the left hand side of Narrowing Grammar rules can result in "incompleteness". Given a set of Narrowing Grammar rules:

$f(x, x) \Rightarrow []$.

$a \Rightarrow b(d)$.

$c \Rightarrow b(e)$.

$b(x) \Rightarrow b(x)$.

$e \Rightarrow d$.

then $f(a, c)$ can be narrowed to $[]$:

$f(a, c)$

narrow_to $f(b(d), c)$

narrow_to $f(b(d), b(e))$

narrow_to $f(b(d), b(d))$

narrow_to $[]$.

However there is no corresponding NU-narrowing. In fact, any NU-narrowing starting at $f(a, c)$ must be non-terminating:

$f(a, c)$

→ $f(b(d), c)$

→ $f(b(d), b(e))$

→ $f(b(d), b(e))$

→ ...

→ ...

The major problem here is the unrestricted use of duplicate variables in the rule

$f(x, x) \Rightarrow []$. The term $f(b(d), b(e))$ does not unify with the left hand side of the rule $f(x, x) \Rightarrow []$. By the second clause of NU-step, $b(d)$ or $b(e)$ is selected to be NU-narrowed. Either case will lead to a non-terminating narrowing.

In order to apply the rule $f(x, x) \Rightarrow []$ for the term $f(b(d), b(e))$, the equality of $b(d)$ and $b(e)$ needs to be derived. The problem of establishing the equality of two terms is, in general, undecidable. It turns out that if we restrict the use of duplicate variables as described in Theorem 4.2, we have an important theorem regarding the narrowing-completeness of Narrowing Grammar for simplified forms. That is, whenever a term can be simplified, it can be simplified by repeatedly applying NU-step. We state this formally as follows:

Theorem 5.1: Narrowing-completeness of Narrowing Grammar for simplified forms.

Let D_0 be a term and D_0, D_1, \dots, D_n be a successful narrowing with no non-linear terms. Then there is a simplification D_0, E_1, \dots, E_m , with no non-linear terms such that E_m narrows to D_n in zero or more steps.

Example 5.1

Consider the Narrowing Grammar rules:

$f(x, y) \Rightarrow c(x, y)$.

$a \Rightarrow []$.

$b \Rightarrow []$.

$f(a, b)$ has a successful narrowing to $c([], [])$:

$f(a, b)$

narrow_to $f([], b)$

narrow_to $f([], [])$

narrow_to $c([], [])$.

Then $f(a, b)$ has a simplification with no non-linear terms

$f(a, b) \rightarrow c(a, b)$

such that

$c(a, b)$

narrow_to $c([], b)$

narrow_to $c([], [])$.

For the rest of this section, we prove Theorem 5.1. The proof here is adapted from the proof of reduction-completeness of Log(F) [45] with many important modifications and simplifications.

Definition 5.1 : $R(G, H, p, q)$

Let p, q, G, H be terms and $G \Rightarrow H$ be a Narrowing Grammar rule. The predicate $R(G, H, p, q)$ is *true* if either p and q are unifiable, or there exists a nonvariable subterm s of p (which we write $p = r[s]$) such that G unifies with s with m.g.u. θ and $q =$

$(r[H])\theta$. If multiple occurrences of s occur in p , zero or more occurrences of s are simultaneously replaced by H .

Definition 5.2

We consider sequences of integers which represent an access path in a term, with the empty sequence denoted by Λ . Concatenation of sequences is denoted by ".", and the set of finite sequences of positive integers by N^* . The elements of N^* are called *occurrences*. The set $O(t)$ of *occurrences of a term t* is defined to be:

- (1) $\Lambda \in O(t)$.
- (2) $i.u \in O(t)$ iff t is of the form $f(t_1, \dots, t_i, \dots, t_n)$, and $u \in O(t_i)$ for some $1 \leq i \leq n$.

Definition 5.3

The subterm of t at occurrence u [with $u \in O(t)$], denoted by $t@u$, is defined to be:

- (1) t if $u = \Lambda$.
- (2) $t_i@v$ if t is of the form $f(t_1, \dots, t_i, \dots, t_n)$ and $u = i.v$ with $i \in N$ and $v \in N^*$.

Definition 5.4

The replacement of the subterm at occurrence u in t by t' , denoted $t[u \leftarrow t']$, is defined to be:

- (1) t' if $u=\Lambda$.
- (2) $f(t_1, \dots, t_i[v \leftarrow t'], \dots, t_n)$ if $t=f(t_1, \dots, t_i, \dots, t_n)$ and $u=i.v$ with $i \in N$ and $v \in N^*$.

Lemma 5.1 : Let X_1, \dots, X_n be variables, $G, H, t_1, \dots, t_n, t'_1, \dots, t'_n$ be terms not containing X_1, \dots, X_n , $R(G, H, c(t_1, \dots, t_n), c(t'_1, \dots, t'_n))$ be true for some constructor c . Let $\sigma = \{\langle X_1=t_1 \rangle, \dots, \langle X_n=t_n \rangle\}$ and $\tau = \{\langle X_1=t'_1 \rangle, \dots, \langle X_n=t'_n \rangle\}$ be substitutions. Let M be a term, possibly containing variables. Then $R(G, H, M\sigma, M\tau)$ is true.

Proof : By structural induction on M .

- (1) M is a variable X_i . Then $M\sigma = t_i$ and $M\tau = t'_i$. Therefore, $R(G, H, M\sigma, M\tau)$ is true.
- (2) M is a 0-ary function symbol. Obviously, $R(G, H, M\sigma, M\tau)$ is true.
- (3) $M = f(Y_1, \dots, Y_m)$. Assume the lemma is true for Y_1, \dots, Y_m , that is, for each $i \in \{1, \dots, m\}$, $R(G, H, Y_i\sigma, Y_i\tau)$ is true. Since $f(Y_1, \dots, Y_m)\sigma = f(Y_1\sigma, \dots, Y_m\sigma)$, $f(Y_1, \dots, Y_m)\tau = f(Y_1\tau, \dots, Y_m\tau)$, and $R(G, H, c(t_1, \dots, t_n), c(t'_1, \dots, t'_n))$ is true, $R(G, H, M\sigma, M\tau)$ is true.

Q.E.D.

Lemma 5.2: Assume

- (1) $f(t_1, \dots, t_n)$ and $f(t_1', \dots, t_n')$ are terms
- (2) $R(G, H, t_i, t_i')$ is true for each $i \in \{1, \dots, n\}$
- (3) $f(L_1, \dots, L_n)$ is the head of some Narrowing Grammar rule
- (4) $f(t_1, \dots, t_n)\sigma = f(L_1, \dots, L_n)\sigma$ for some m.g.u. σ
- (5) $f(t_1', \dots, t_n')\tau = f(L_1, \dots, L_n)\tau$ for some m.g.u. τ

Then for each variable X that occurs in $f(L_1, \dots, L_n)$, if $\langle X=s \rangle \in \sigma$ and $\langle X=s' \rangle \in \tau$, then $R(G, H, s, s')$ is true.

Proof:

Consider L_i for some i .

- (1) L_i is a variable.

$\langle L_i=t_i \rangle \in \sigma$ and $\langle L_i=t_i' \rangle \in \tau$, by assumption, $R(G, H, t_i, t_i')$ is true.

- (2) L_i is not a variable.

By assumption, both t_i and t_i' unify with L_i .

(2.1) If t_i and t_i' are unifiable, then for each variable X occurring in L_i , if $\langle X=s \rangle \in \sigma$ and $\langle X=s' \rangle \in \tau$, then s and s' unify, therefore, $R(G, H, s, s')$ is true.

(2.2) If t_i and t_i' are not unifiable, since $R(G, H, t_i, t_i')$ is true, t_i contains at least one outermost non-simplified subterm that unifies with G , and $G \Rightarrow H$ is a Narrowing Grammar rule. For each such subterm $t_i@p$ of t_i , if $L_i@p$ exists, it must be a variable because it unifies with the non-simplified term $t_i@p$. Therefore, $\langle L_i@p=t_i@p \rangle \in \sigma$, $\langle L_i@p=t_i'@p \rangle \in \tau$, and $R(G, H, t_i@p, t_i'@p)$ is true.

By assumption (4) and (5), both t_j and t_j' unify with L_j for any j , therefore, the same argument can be repeated for any L_j . For each variable X occurring in $f(L_1, \dots, L_n)$, if $\langle X=s \rangle \in \theta_1$ and $\langle X=s' \rangle \in \theta_2$ then $R(G, H, s, s')$ is true.

Q.E.D.

Lemma 5.3:

If L_i is a term in normal form and $R(G, H, t_i, t_i')$ is true such that t_i' unifies with L_i but t_i does not unify with L_i and t_i does not unify with t_i' , then $\text{nu_narrowing}(t_i, L_i)$ has a SLD-derivation with some substitution θ provided that no non-linear terms will be generated.

Proof :

Since t_i does not unify with both t_i' and L_i and L_i is in normal form, there exists at least one outermost non-simplified subterm $t_i@p$ which does not unify with $L_i@p$. Since $R(G, H, t_i, t_i')$ is true, $R(G, H, t_i@p, t_i'@p)$ is true. Also since $t_i@p$ is not simplified, $L_i@p$ is in normal form, $t_i@p$ does not unify with $L_i@p$, and $t_i'@p$ unifies with $L_i@p$, then $t_i'@p$ is simplified, therefore, $t_i@p \Rightarrow t_i'@p$. That is, $\text{nu_step}(t_i, t_i[p \leftarrow t_i'@p])$ has a successful SLD-derivation.

Repeat the same argument for any outermost non-simplified subterm (say occurrence r) of $t_i[p \leftarrow t_i'@p]$ which does not unify with $L_i@r$. Since $R(G, H, t_i, t_i')$, in a finite number of steps, t_i NU-narrows to t_i' . Also, L_i unifies with t_i' , therefore, $\text{nu_narrowing}(t_i, L_i)$ has a successful SLD-derivation with some substitution θ .

Q.E.D.

Lemma 5.4:

- (1) If $f(t_1, \dots, t_{i-1}, t_i, t_{i+1}, \dots, t_n)$ is a term and $f(L_1, \dots, L_{i-1}, L_i, L_{i+1}, \dots, L_n) \Rightarrow RHS$ is a Narrowing Grammar rule such that t_i does not unify with L_i
- (2) $nu_step(t_i, t_i')$ succeeds with substitution θ .

Then $nu_step(f(t_1, \dots, t_{i-1}, t_i, t_{i+1}, \dots, t_n), f(t_1, \dots, t_{i-1}, t_i', t_{i+1}, \dots, t_n))$ succeeds with substitution θ .

Proof :

Since by assumption t_i does not unify with L_i , $f(t_1, \dots, t_{i-1}, t_i, t_{i+1}, \dots, t_n)$ does not unify with $f(L_1, \dots, L_{i-1}, L_i, L_{i+1}, \dots, L_n)$. By the second clause of NU-step, when narrowing $f(t_1, \dots, t_{i-1}, t_i, t_{i+1}, \dots, t_n)$, t_i can be selected to be NU-narrowed. Since $nu_step(t_i, t_i')$ succeeds with substitution θ , $nu_step(f(t_1, \dots, t_{i-1}, t_i, t_{i+1}, \dots, t_n), f(t_1, \dots, t_{i-1}, t_i', t_{i+1}, \dots, t_n))$ succeeds with substitution θ .

Q.E.D.

Theorem 5.0:

Assume

- (1) $R(G, H, E_1, F_1)$ is true, and

(2) $\text{nu_step}(F_1, F_2)$ succeeds.

Then there exists some term E_2 such that an NU-narrowing from E_1 to E_2 and $R(G, H, E_2, F_2)$ is true provided that no non-linear terms in an NU-narrowing from E_1 to E_2 and from F_1 to F_2 .

Proof:

$$\begin{array}{ccc}
 & \text{NU-step} & \\
 F_1 & \text{-----} & F_2 \\
 | & & \\
 | & R(G, H, E_1, F_1) & \\
 | & & \\
 E_1 & \text{-----} & E_2 \\
 & \text{NU-narrowing} &
 \end{array}$$

We have to show that $R(G, H, E_2, F_2)$ is true. We proceed by structural induction on E_1 .

Suppose E_1 is a 0-ary function symbol. If $E_1 = F_1$ then E_1, F_2 is an NU-narrowing and by setting $E_2 = F_2$ we satisfy the theorem, since then trivially $R(G, H, F_2, F_2)$ is true. If $E_1 \neq F_1$ then since $R(G, H, E_1, F_1)$ is true, $E_1 = G$ and $E_1 \Rightarrow F_1$. Thus, there is an NU-narrowing E_1, F_1, F_2 and $R(G, H, E_2, F_2)$ is true if we take $E_2 = F_2$.

Otherwise, let $E_1 = f(t_1, \dots, t_n)$. Assume the theorem is true for every proper sub-term of $f(t_1, \dots, t_n)$. If E_1 unifies with F_1 , then E_1, F_2 is an NU-narrowing and $R(G, H, E_2, F_2)$ is true if we take $E_2 = F_2$. Otherwise E_1 does not unify with F_1 . If E_1 unifies with G then since $R(G, H, E_1, F_1)$ is true, $E_1 \Rightarrow F_1$. Thus, there is an NU-narrowing E_1, F_1, F_2 , and $R(G, H, E_2, F_2)$ is true if we take $E_2 = F_2$.

We now arrive at the case where E_1 does not unify with F_1 , but G does not unify with

E_1 .

Hence since $R(G,H,E_1,F_1)$ is true, $F_1 = f(t_1', \dots, t_n')$ where for every $k \in \{1, \dots, n\}$, $R(G,H,t_k, t_k')$ is true. We consider the following cases:

(1) $F_1 \Rightarrow F_2$

There exists a Narrowing Grammar rule $f(L_1, \dots, L_n) \Rightarrow RHS$ such that F_1 unifies with $f(L_1, \dots, L_n)$ with m.g.u. τ and $F_2 = RHS\tau$. There are two cases:

(1.1) E_1 unifies with $f(L_1, \dots, L_n)$ with m.g.u. σ . Thus $E_1 \Rightarrow RHS\sigma$, so if we let

$E_2 = RHS\sigma$ then E_1, E_2 is an NU-narrowing. Of course $F_2 = RHS\tau$. By Lemma 5.2, if a variable X occurs in $f(L_1, \dots, L_n)$, if $\langle X=s \rangle \in \sigma$, and $\langle X=s' \rangle \in \tau$, then $R(G,H,s,s')$ is true. Since $R(G,H,E_1,F_1)$ is true, E_1 and F_1 do not contain any variables in $f(L_1, \dots, L_n)$, hence by Lemma 5.1, $R(G,H,E_2,F_2)$ is true.

(1.2) E_1 does not unify with $f(L_1, \dots, L_n)$. By the assumption that no terms is non-linear, there exists some i such that L_i does not unify with t_i . Summarizing what is known in this case: L_i is in normal form, $R(G,H,t_i,t_i')$ is true, and t_i' unifies with L_i but t_i does not unify with L_i . Since $R(G,H,E_1,F_1)$ and E_1 and F_1 do not unify, t_i and t_i' do not unify. By Lemma 5.3, $\text{nu_narrowing}(t_i, L_i)$ has a successful SLD-derivation with substitution θ_i . By repeatedly applying Lemma 5.4, $\text{nu_narrowing}(f(t_1, \dots, t_i, \dots, t_n), f(t_1, \dots, L_i, \dots, t_n))$ has a successful SLD-derivation with substitution θ_i .

There exists an NU-narrowing $E_1 = P_1, P_2, P_3, \dots, P_m$ such that for each j , $P_j = f(s_1, \dots, s_n)$, and for each $s_k \in \{s_1, \dots, s_n\}$, either s_k is t_k or s_k unifies with L_k . Moreover, the NU-narrowing from P_j to P_{j+1} is derived by selecting some $s_k \in \{s_1, \dots, s_n\}$ such that s_k does not unify with L_k , and $\text{nu_narrowing}(s_k, L_k)$ has a successful SLD-derivation. We also have for each j , $R(G, H, P_j, F_1)$ is true. Since n is finite, this NU-narrowing cannot be infinite and must end in P_m such that P_m unifies with $f(L_1, \dots, L_n)$ with substitution σ . Then $P_m \Rightarrow RHS\sigma$. Hence we have the NU-narrowing $P_1, \dots, P_m, RHS\sigma$. We know that $F_2 = RHS\tau$. If we take $E_2 = RHS\sigma$ and we know that $R(G, H, E_1, F_1)$ is true, E_1 and F_1 do not contain any variables in $f(L_1, \dots, L_n)$, then by Lemma 5.1, $R(G, H, E_2, F_2)$ is true.

(2) $F_1 \neq F_2$

We are given that $\text{nu_step}(F_1, F_2)$ succeeds. We now have to find an E_2 such that $\text{nu_narrowing}(E_1, E_2)$ has a successful SLD-derivation and $R(G, H, E_2, F_2)$ is true.

Since $\text{nu_step}(F_1, F_2)$ succeeds and $F_1 \neq F_2$, by the second clause of NU-step, there is a rule $f(L_1, \dots, L_i, \dots, L_n) \Rightarrow RHS$ such that F_1 does not unify with $f(L_1, \dots, L_i, \dots, L_n)$. Here $F_1 = f(t_1', \dots, t_n')$. Since no terms is non-linear, there exists some outermost subterm $t_i'@p$ of t_i' for some i , and a Narrowing Grammar rule $u \Rightarrow v$ such that: u and $t_i'@p$ unify with m.g.u. θ , $\text{nu_step}(t_i'@p, v)$ succeeds with substitution θ , and $F_2 = f(t_1', \dots, t_{i-1}', t_i'[p \leftarrow v], t_{i+1}', \dots, t_n')\theta$. We already have $R(G, H, t_i, t_i')$. Since t_i is a proper subterm of $f(t_1, \dots, t_i, \dots, t_n)$, by the induction

hypothesis there is an NU-narrowing d_1, d_2, \dots, d_r , $r \geq 1$, such that $t_i = d_1$ and $R(G, H, d_r, (t_i' [p \leftarrow v]) \theta)$. Since d_1, \dots, d_r is an NU-narrowing, by repeatedly applying Lemma 5.4, $\text{nu_narrowing}(f(t_1, \dots, t_{i-1}, d_1, t_{i+1}, \dots, t_n), f(t_1, \dots, t_{i-1}, d_r, t_{i+1}, \dots, t_n))$ has a successful SLD-derivation with substitution θ_r . Take $E_2 = f(t_1, \dots, t_{i-1}, d_r, t_{i+1}, \dots, t_n) \theta_r$. We already have $F_2 = f(t_1', \dots, t_i' [p \leftarrow v], \dots, t_n') \theta$ and for each k , $k \neq i$, $R(G, H, t_k, t_k')$ is true. Hence $R(G, H, E_2, F_2)$ is true.

Q.E.D.

Lemma 5.5

Let $R(G, H, E_0, F_0)$ be true and F_0, F_1, \dots, F_n be an NU-narrowing with no non-linear terms. Then there is an NU-narrowing $E_0, \dots, E_1, \dots, E_m$ with no non-linear terms such that $R(G, H, E_m, F_n)$ is true.

Proof :

By induction on the length n of F_0, F_1, \dots, F_n . If $n = 0$, the lemma is obvious. Suppose the lemma holds for NU-narrowings of length less than n . Since $\text{nu_step}(F_0, F_1)$ succeeds and $R(G, H, E_0, F_0)$ is true, by Theorem 5.0, there exists an NU-narrowing E_0, \dots, E_1 with no non-linear terms such that $R(G, H, E_1, F_1)$ is true. By the induction hypothesis, there exists an NU-narrowing E_1, \dots, E_m with no non-linear terms, such that $R(G, H, E_m, F_n)$ is true. Hence there exists the NU-narrowing $E_0, \dots, E_1, \dots, E_m$ with no non-linear terms such that $R(G, H, E_m, F_n)$ is true.

Q.E.D.

Theorem 5.1: Narrowing-completeness of Narrowing Grammar for simplified forms.

Let D_0 be a term and D_0, D_1, \dots, D_n be a successful narrowing with no non-linear terms. Then there is a simplification D_0, E_1, \dots, E_m , with no non-linear terms such that E_m narrows to D_n in zero or more steps.

Proof:

By induction on the length n of D_0, D_1, \dots, D_n . If $n = 0$, the theorem is obviously true.

Let $n > 0$ and assume the theorem holds for narrowings of length $< n$. Then there is a simplification D_1, F_2, \dots, F_p such that F_p narrows to D_n in zero or more steps. Since D_0 narrow_to D_1 , by Definition 3.5, there exists a Narrowing Grammar rule $G \Rightarrow H$ such that $R(G, H, D_0, D_1)$ is true. By Lemma 5.5, there is an NU-narrowing D_0, E_1, \dots, E_q such that $R(G, H, E_q, F_p)$ is true. There are now two possibilities:

(1) If E_q is simplified, take $E_m = E_q$, so D_0, E_1, \dots, E_m is a simplification. Since $R(G, H, E_m, F_p)$ is true and F_p narrows to D_n in zero or more steps, therefore E_m narrows to D_n in zero or more steps.

(2) If E_q is not simplified, since $R(G, H, E_q, F_p)$ is true and F_p is simplified, $E_q \Rightarrow F_p$. Take $E_m = F_p$, and D_0, E_1, \dots, E_m is a simplification, and therefore E_m narrows to D_n in zero or more steps.

Q.E.D.

5.3 Compilation Algorithm to Handle Duplicate LHS Variables

Both Theorems 4.2 and 5.1 assume that no NU-step in a simplification is non-linear. In this section, we extend Algorithm 4.1 to handle some cases where this assumption does not hold. One interesting aspect of the extended compilation algorithm is its use of a suitable "equality" predicate `equal(x, y)`, which succeeds when both `x` and `y` are equal under an appropriate equality theory. In this section we investigate the ordinary rewriting situation [33] where the equality predicate essentially permits different orders of narrowings among terms, but in other situations users may find it desirable to define equality in a way that has nothing directly to do with narrowing.

Example 5.2 Given the Narrowing Grammar rules:

`f(x, x) => []`.

`a => b`.

There is a NU-narrowing

`f(a, b) → f(b, b) → []`

Consider the Prolog code generated from Algorithm 4.1:

```
simplify(f(A1, A2), Out) :- nu_narrow(A1, X),
                           nu_narrow(A2, X),
                           simplify([], Out).

simplify(a, Out) :- simplify(b, Out).
```

The Prolog query

```
?- simplify(f(a,b), Out).
```

fails even though *a* and *b* are equal in the sense that *a* can be narrowed to *b*. The failure of the Prolog goal is due to the fact that the Prolog subgoals are executed from left to right. The first subgoal `nu_narrow(a, X)` effects a unification and the second subgoal `nu_narrow(b, a)` fails. If the order of execution of these two subgoals is reversed, the Prolog query will succeed.

The extended compilation algorithm translates the Narrowing Grammar rules to Prolog clauses by generating some equality predicates when duplicate variables occur on the left-hand side of the Narrowing Grammar rules. For instance, the Narrowing Grammar rule

```
f(X, X) => [].
```

is compiled into something like this:

```
simplify(f(A1, A2), Out) :- nu_narrow(A1, X1),  
                           nu_narrow(A2, X2),  
                           equal(X1, X),  
                           equal(X2, X),  
                           simplify([], Out).
```

Here the user must provide an appropriate definition for `equal/2`.

Algorithm 5.1 : Extended Compilation Algorithm to Handle Duplicate LHS Variables

- (1) For each n -ary constructor symbol c , $n \geq 0$, and for distinct Prolog variables A_1, \dots, A_n , generate the clause:

`simplify(c(A1, ..., An), c(A1, ..., An)).`

- (2) Let $f(L_1, \dots, L_m) \Rightarrow RHS$ be a Narrowing Grammar rule. Replace each of the (say n) occurrences of a Prolog variable X in $f(L_1, \dots, L_m)$ by X_1, \dots, X_n respectively where X_1, \dots, X_n are distinct Prolog variables not occurring in the rule. Also let A_1, \dots, A_m, Out , be distinct Prolog variables not occurring in the rule, and generate the clause:

`simplify(f(A1, ..., Am), Out) :-
 nu_narrow(A1, L1),
 ...,
 nu_narrow(Am, Lm),
 equal(X1, X),
 ...,
 equal(Xn, X),
 simplify(RHS, Out) .`

If there are no such variables X_1, \dots, X_n then the `equal/2` subgoals are omitted.

The user must define a predicate `equal(x, y)` that succeeds when x and y are equal under an appropriate equality theory. We propose three different definitions for `equal/2`:

(1) *Equality as unifiability.*

`equal (x, x) .`

This definition is implicit in the compilation Algorithm 4.1. That is, if equality as unifiability is used, Algorithm 5.1 reduces to Algorithm 4.1.

(2) *Equality via normal form:*

This definition is appropriate for terminating theory. That is, every term has its normal form(s). `equal (x, y)` first checks the unifiability of `x` and `y`. If this fails, it computes a normal form of `x` and `y` before unifying them.

```

equal(X,X) :- !.

equal(X,Y) :-
    normal_form(X,Z),
    normal_form(Y,Z).

normal_form(A,B) :-
    simplify(A,C),
    functor(C,F,N),
    functor(B,F,N),
    normal_form_args(N,C,B).

normal_form_args(0,A,A) :- !.

normal_form_args(I,A,B) :-
    arg(I,A,C),
    arg(I,B,D),
    normal_form(C,D),
    I1 is I-1,
    normal_form_args(I1,A,B).

```

(3) *Equality via lazy confluence:*

This definition of `equal(x,y)` tries to coroutine the narrowing of `x` and `y` until they become unifiable. In many situations, equality via lazy confluence is more efficient than equality via normal form because the computation of a normal form of a term can be very expensive.

```

equal(X,Y) :- conflate(X,Y).

conflate(X,X) :- !.

conflate(X,Y) :-
    functor(X,F,N),
    functor(Y,F,N),
    args_conflate(X,Y,0,N).

conflate(X,Y) :-
    simplify(X,X1),
    (X1=Y -> true;
     simplify(Y,Y1),
     (X=Y1 -> true;
      (X1=Y1 -> true;
       (diff(X,X1);diff(Y,Y1)) -> conflate(X1,Y1)
      )
     )
    ).

diff(X,X) :- !, fail.

diff(_,_) .

args_conflate(X,Y,N,N) .

args_conflate(X,Y,M,N) :-
    M1 is M+1,
    arg(M1,X,X1),
    arg(M1,Y,Y1),
    conflate(X1,Y1),
    args_conflate(X,Y,M1,N) .

```

Example 5.3 : Consider the following Narrowing Grammar rules:

$f(x, x) \Rightarrow []$.

$a \Rightarrow c$.

$b \Rightarrow c$.

$c \Rightarrow [c]$.

and the Prolog query

`?- simplify(f(a,b),Out).`

- (1) If equality via lazy confluence is used, the Prolog query succeeds because both a and b have the simplified form $[c]$.
- (2) If equality via normal form is used, the Prolog query will loop because neither a nor b has a normal form.
- (3) If equality via unifiability is used, the Prolog query will fail because a and b are not unifiable.

The point to be made here is that if some terms are non-linear, NU-narrowing is not complete. However, if an appropriate equality is defined, `simplify/2` has a successful Prolog-derivation.

5.4 Conclusions

The Narrowing Grammar formalism together with its implementation were presented in chapters three and four. The correctness of the compilation algorithm based on the assumption that no terms is non-linear. In other words, when we consider the extended

compilation algorithm, equality as unifiability is implicitly used in Narrowing Grammar formalism. With the same assumption, narrowing-completeness holds. However, if the basic assumption (no terms is non-linear) does not hold, the extended compilation algorithm allows the user to define some appropriate equality predicate. In this case, the choice of equality predicate has an important bearing upon two issues: completeness and efficiency.

Chapter 6

Performance Considerations

6.1 Introduction

In chapter four, we proposed Algorithm 4.1 to implement Narrowing Grammar. This implementation is simple, and is not optimized. An interesting open problem is to devise improvements for the implementation given there. Efficient implementations are possible in many cases. We first apply the popular program transformation technique called *partial evaluation* to Algorithm 4.1 in order to generate more efficient Prolog code from a given set of Narrowing Grammar rules. We then concentrate on efficient classes of Narrowing Grammar, called *Greibach Grammar* and *Tail-recursive Grammar*. Partial evaluation can, in many cases, be applied to transform Narrowing Grammar rules to Greibach Grammar rules.

6.2 Partial Evaluation

A *partial evaluator* is an *interpreter* that, with only partial information about a program's inputs, produces a specialized version of the program which exploits partial information [35, 36].

Partial evaluation of a Prolog program is accomplished mainly by instantiation of the parameters of a Prolog predicate by propagating values for top-level formal arguments through the Prolog clause (execution of unification at compile time), and reduction of the number of logical inferences by opening calls. The basis of the partial evaluation system is a pure Prolog meta-interpreter:

```
partial_eval(true,true) .  
partial_eval((G,Gs),(H,Hs)) :-  
    partial_eval(G,H), partial_eval(Gs,Hs) .  
partial_eval(G,H) :- clause(G,H1), partial_eval(H1,H) .
```

This simple partial evaluation system works correctly only for non-recursive, cut-free and deterministic clauses. The extension of the partial evaluation system to handle nondeterministic clauses is described in [59]. The major idea is to collect all the clauses and treats the resulting list as an *OR-list*. Consider an example from [59] which queries a simple database about family relationships:

```

grandparent (X, Y) :- grandmother (X, Y) .
grandparent (X, Y) :- grandfather (X, Y) .
grandmother (X, Y) :- mother (X, Z) , parent (Z, Y) .
grandfather (X, Y) :- father (X, Z) , parent (Z, Y) .
parent (X, Y) :- mother (X, Y) .
parent (X, Y) :- father (X, Y) .

mother (anna, violette) .
mother (violette, jan) .
mother (violette, stan) .
mother (henriette, henry) .
mother (henriette, stanis) .

```

Assume the tuples for the `father/2` relation are not available at compile time. Consider the query `?- grandmother(X, jan)`. The program resulting from partial evaluation is:

```

grandmother(anna, jan) :- (true; father(violette, jan)) .
grandmother(violette, jan) :- father(jan, jan) .
grandmother(violette, jan) :- father(stan, jan) .
grandmother(henriette, jan) :- father(henry, jan) .
grandmother(henriette, jan) :- father(stanis, jan) .

```

These results are accomplished mainly by forward propagation of values through the program and applying some simplifications on the resulting clauses.

6.3 Optimization of Algorithm 4.1 by Partial Evaluation

The compiled code generated from Algorithm 4.1 can be optimized considerably by elimination of some procedure calls, yielding substantially faster code. Consider Definition 4.1 which is used in Algorithm 4.1:

```

nu_narrow(X,X) :- !.
nu_narrow(X,Y) :- simplify(X,Z),
                  nu_narrow_subterms(Z,Y).

nu_narrow_subterms(X,Y) :-
                  functor(X,F,N), functor(Y,F,N),
                  nu_narrow_subterms(X,Y,0,N).

nu_narrow_subterms(_,_,N,N).
nu_narrow_subterms(X,Y,I,N) :-
                  plus(I,1,I1), arg(I1,X,A), arg(I1,Y,B),
                  nu_narrow(A,B),
                  nu_narrow_subterms(X,Y,I1,N).

```

By Definition 3.2, each argument of *LHS* is in normal form. That is, the second argument in each call to `nu_narrow/2` is initially in normal form. On the other hand, `simplify/2` guarantees its second argument is simplified. Therefore, `simplify/2` is computationally less expensive than `nu_narrow/2`.

At compile time, the `nu_narrow/2` subgoals for each argument of *LHS* are partially evaluated using the definition of `nu_narrow/2`. The idea is to check, at compile time, the types of arguments of *LHS* (the second argument of each `nu_narrow/2` subgoal). For instance, if the second argument of `nu_narrow/2` is a fresh variable, then the `nu_narrow/2` subgoal can be replaced by a unification by the first clause of

`nu_narrow/2`. Similarly if the second argument of `nu_narrow/2` is a simplified term with only variables as arguments, then `nu_narrow/2` can be replaced by `simplify/2` and will then yield substantially faster code.

Algorithm 6.1 gives an optimized version of the compilation algorithm by partial evaluation of `nu_narrow/2`, in Algorithm 4.1.

Algorithm 6.1 : Optimized Compilation Algorithm

- (1) For each n -ary constructor symbol c , $n \geq 0$, and for distinct Prolog variables X_1, \dots, X_n , generate the clause:

$$\text{simplify}(c(X_1, \dots, X_n), c(X_1, \dots, X_n)).$$

- (2) For each rule $f(L_1, \dots, L_m) \Rightarrow RHS$, let A_1, \dots, A_m, Out be distinct Prolog variables not occurring in the rule. If L_i is a variable, let Q_i be *true* and replace A_i by L_i . If L_i is simplified with only variables as arguments, let Q_i be `simplify(A_i, L_i)`. Otherwise, let Q_i be `nu_narrow(A_i, L_i)`. Generate the clause:

$$\text{simplify}(f(A_1, \dots, A_m), Out) :- Q_1, \dots, Q_m, \text{simplify}(RHS, Out).$$

We give here some examples of the partially evaluated code generated from Algorithm 6.1 and the reader can compare this with the code generated from Algorithm 4.1 in chapter four.

Narrowing Grammar Rules	Prolog Clauses
<pre>match([], S) => S.</pre>	<pre>simplify(match(A, B), C) :- simplify(A, []), simplify(B, C).</pre>
<pre>match([X L], [X S]) => match(L, S).</pre>	<pre>simplify(match(A, B), C) :- simplify(A, [D E]), simplify(B, [D F]), simplify(match(E, F), C).</pre>
<pre>(X+) => X.</pre>	<pre>simplify(A+, B) :- simplify(A, B).</pre>
<pre>(X+) => X, (X+).</pre>	<pre>simplify(A+, B) :- simplify((A, A+), B).</pre>
<pre>([], L) => L.</pre>	<pre>simplify((A, B), C) :- simplify(A, []), simplify(B, C).</pre>
<pre>([X L1], L2) => [X (L1, L2)].</pre>	<pre>simplify((A, B), C) :- simplify(A, [D E]), simplify([D (E, B)], C).</pre>

6.4 Efficient Classes of Narrowing Grammar

Efficient classes of Narrowing Grammar can be defined in terms of time and space. These classes of grammars are very important, for example, in that they can be used for analyzing streams that are arbitrarily long without concern about overflowing available stack space or using excessive amounts of time (a necessity in practical stream analysis). In this section, we define two important classes of Narrowing

Grammar, Greibach Grammar and Tail-recursive Grammar, and demonstrate how partial evaluation can, in many cases, be applied to transform Narrowing Grammar rules to Greibach form.

Definition 6.1

A Narrowing Grammar rule $LHS \Rightarrow RHS$ is *Greibach* if RHS is simplified. A Narrowing Grammar is *Greibach* if every rule is Greibach.

Example 6.1 : Consider the Narrowing Grammar rule for the Kleene Plus pattern

$$\text{pattern} \Rightarrow [a]^+, [b].$$

An equivalent Greibach Grammar is

$$\text{pattern} \Rightarrow [a,b]; [a|\text{pattern}].$$

The second grammar is computationally more efficient than the first one.

Space complexity, particularly the depth of stack, is especially important in our implementation of Narrowing Grammar. Let us consider the classical factorial example†:

$$\text{factorial}(1) \Rightarrow 1.$$
$$\text{factorial}(N) \Rightarrow N * \text{factorial}(N-1).$$

The depth of stack grows linearly with N . However, we can rewrite the definition of factorial as follows‡:

† $N-1$ yields the value of $N-1$ when simplified. + and * are defined similarly.

‡ If-then-else is defined as follows:

$$\text{if}(\text{true}, X, Y) \Rightarrow X.$$
$$\text{if}(\text{false}, X, Y) \Rightarrow Y.$$

```

factorial(N) => factorial(1,1,N).
factorial(Result,Counter,N) =>
  if((Counter>N),Result,factorial(Counter*Result,Counter+1,N)).

```

which requires constant stack space. We have applied the well-known optimization that converts tail-recursion to iteration [1]. One of the most important applications of tail-recursion optimization here is in Narrowing Grammar that specifies aggregate operations. For example, given the following Narrowing Grammar rules:

```

count(Result) => count(Result, 0).
count(Result, Result) => [end_of_file].
count(Result, Count) => [_ | count(Result, Count+1)].
sum(Result) => sum(Result, 0).
sum(Result, Result) => [end_of_file].
sum(Result, Current) => [Value | sum(Result, Current+Value)].

```

The amount of stack space required for these grammars is constant.

Definition 6.2

Let P be a term.

$$tail(P) = \begin{cases} tail(Q_2) & \text{if } P = (Q_1, Q_2) \\ \{\text{functionsymbol}(P)/\text{arity}(P)\} & \text{otherwise} \end{cases}$$

Definition 6.3

The *connection graph* for a Narrowing Grammar is a labelled graph whose vertices are the *LHS* terms of the Narrowing Grammar, and for which there exists an edge

$\langle LHS_1, LHS_2 \rangle$ if there exist rules in the Narrowing Grammar

$$LHS_1 \Rightarrow RHS$$
$$LHS_2 \Rightarrow \dots$$

where some (not necessarily proper) subterm of RHS unifies with LHS_2 . The edge is labelled with the value 0 if the subterm of RHS that unifies with LHS_2 is $tail(RHS)$, and with 1 otherwise. Note that we permit self-loops in this graph, i.e., edges from some vertex LHS_1 to itself.

A Narrowing Grammar is called *tail-recursive* if its connection graph is acyclic or all the edges in any cycles are labelled with the value 0.

Example 6.2 : First Come First Served Scheduling Policy

Consider an application to queuing systems. Suppose we wish to gather statistics on the time customers spend at specific servers. For simplicity, assume that the type of server is FCFS (First Come First Served). Arrivals and departures of customers to a specific server are captured in a stream whose items have the format:

$$a(\text{Time})$$
$$d(\text{Time})$$

For computer system performance analysis, we define *turnaround time* be the interval from the time of submission to the time of completion. We can evaluate FCFS performance criteria with Narrowing Grammar:

```

fcfs_t(Result) => fcfs_t([], [], Result).
fcfs_t(_, Result, Result) => [end_of_file].
fcfs_t(State, Current, Result) =>
    [a(T) | fcfs_t((State, [T]), Current, Result)].
fcfs_t([T0|S], Current, Result) =>
    [d(T) | fcfs_t(S, [T-T0|Current], Result)].

```

These Narrowing Grammar rules (except the first rule) are Tail-recursive Greibach Grammar.

Example 6.3 : Context-free Languages

Consider the Narrowing Grammar rules for the language $\{a^n b^n \mid n > 0\}$.

```

ab => [a], ab, [b].
ab => [a], [b].

```

The amount of stack space grows linearly with the length of the input stream. On the other hand, if we rewrite the Narrowing Grammar rules to Tail-recursive Greibach Grammar:

```

ab => a(0).
a(Count) => [a | a(plus(Count, 1))].
a(Count) => [b | b(minus(Count, 1))].
b(Count) => [b | b(minus(Count, 1))].
b(0) => [].

```

which requires only constant stack space.

6.5 Partial Evaluation to Greibach Grammar

Consider the Greibach Grammar rule

$$LHS \Rightarrow [a_1, \dots, a_m \mid RHS].$$

the prefix of $[a_1, \dots, a_m \mid RHS]$ is simplified and can be directly applied to `match` for pattern matching. That is, a grammar in Greibach form is efficiently implementable.

In this section, we apply partial evaluation to transform a given set of Narrowing Grammar rules, if possible, to Greibach form. At compile time, with some partial information about some patterns, the partial evaluator produces a specialized version of the grammar rules which exploits the partial information. The basic techniques are execution of the unification at compile time and reduction of the number of NU-steps by opening calls.

For instance, given the Narrowing Grammar rules:

$$p \Rightarrow q, r.$$

$$q \Rightarrow [a].$$

$$r \Rightarrow [b].$$

and the pattern `p`, the Narrowing grammar rules resulting from partial evaluation is:

$$p \Rightarrow [a \mid r].$$

$$r \Rightarrow [b].$$

The basic partial evaluation system described in section 6.1 works only with deterministic rules. Below we describe how Narrowing Grammar rules, in general, are

partially evaluated:

Partial Evaluation of Narrowing Grammar

- (1) The partial evaluation system takes a term, say X , as input, computes its simplified form, say Y (assume there exists unique Y). That is,

$$\text{partial_eval}(X, Y) :- \text{simplify}(X, Y).$$

The Narrowing Grammar rule resulting from partial evaluation is $X \Rightarrow Y$.

- (2) If X has more than one simplified form, the partial evaluation system computes all the simplified forms of X and treats the resulting list as an *OR-list*. This gives us the following modified partial evaluation interpreter:

$$\begin{aligned} \text{partial_eval}(X, Y) :- \\ \quad \text{findall}(T, \text{simplify}(X, T), L), \text{orlist}(L, Y). \\ \text{orlist}([A | As], (A; B)) :- \text{orlist}(As, B). \\ \text{orlist}([A], A). \end{aligned}$$

For instance, the Narrowing Grammar rule

$$\text{pattern} \Rightarrow [a]^+, [b].$$

is partially evaluated to

$$\text{pattern} \Rightarrow [a, b] ; [a | ([a]^+, [b])].$$

and can be further simplified to

$$\text{pattern} \Rightarrow [a, b] ; [a | \text{pattern}].$$

(3) Backward unification

One technique of partial evaluation consists of propagating values forward through the grammar rules by unification. However, this unification does more than instantiating input parameters of the called procedure, it also eventually gives back values for the output parameters (backward unification), which in certain cases, causes problems. Consider the following example :

`program(X) => p(X) .`

`p(a) => [d] .`

`p(b) => [e] .`

`p(X) => [c] .`

The partial evaluation system generates:

`program(a) => [d] ; [c] .`

which is not the expected outcome because `x` is instantiated to the value `'a'` by backward unification. Therefore, the rule `p(b) => [e]` cannot be applied. The proposed solution is to avoid backward unification and execute the forward unification only. We can avoid backward unification by explicitly handling unification of the arguments in the *LHS*. The corresponding changes (no constant appear in the *LHS*) are:

`program(X) => p(X) .`

`p(X) => (X = a), [d] .`

`p(X) => (X = b), [e] .`

`p(X) => [c] .`

The output of the partial evaluation system is:

```
program(X) => ((X=a), [d]) ; ((X=b), [e]) ; [c].
```

Example 6.4 : Non-Context Free Languages

Consider the Narrowing Grammar which defines the non-context-free language $\{a^n b^n c^n | n \geq 0\}$:

```
s_abc => ab_c // a_bc.
```

```
ab_c => pair([a],[b]), [c]*.
```

```
a_bc => [a]*, pair([b],[c]).
```

```
pair(X,Y) => [].
```

```
pair(X,Y) => X, pair(X,Y), Y.
```

Given the initial patterns `s_abc`, `pair([a],[b])`, `pair([b],[c])`, `[a]*` and `[c]*`, the output of the partial evaluation system is

```

s_abc => [];
      [ a | ((pair([a],[b]),[b]),[c]*) // ([a]*,pair([b],[c])) ].

pair([a],[b]) => [];
             [ a | (pair([a],[b]), [b]) ].

pair([b],[c]) => [];
             [ b | (pair([b],[c]), [c]) ].

[a]* => [];
     [ a | [a]* ].

[c]* => [];
     [ c | [c]* ].

```

Example 6.5 : Partial Ordering

Consider the Narrowing Grammar rules for `pattern`:

```

pattern => precedes([c],[a]) // precedes([b],[a]).

precedes(X,Y) => eventually(X), eventually(Y).

eventually(X) => X.

eventually(X) => [_], eventually(X).

```

which specifies the partial ordering events of `a`, `b` and `c`. Given the initial pattern `pattern`, the output of the partial evaluation system with some simplifications is:

```
pattern => [ b | pattern1];
          [ _ | pattern].

pattern => [ c | pattern2];
          [ _ | pattern].

pattern1 => [ c | eventually([a])];
           [ _ | pattern1].

pattern2 => [ b | eventually([a])];
           [ _ | pattern2].

eventually([a]) => [a];
                  [ _ | eventually([a])].
```

These Narrowing Grammar rules are Greibach and Tail-recursive.

Chapter 7

Comparison with Other Logic Grammars

7.1 Introduction

A logic grammar has rules that can be represented as Horn clauses, and thus implemented by logic programming languages such as Prolog. These logic grammar rules are translated into Prolog clauses which can then be executed for either acceptance or generation of the language specified.

Since the development of metamorphosis grammar [13], the first logic grammar formalism, several variants of logic grammars have been proposed [2, 16, 30, 41, 47, 48, 55]. Among these we must mention *Definite Clause Grammars (DCGs)*, the most popular approach to parsing in Prolog. DCG is a special case of Colmerauer's Metamorphosis Grammar. It extends context-free grammar in basically three ways:

- (1) DCG provides context dependency.
- (2) DCG allows arbitrary structures to be built in the process of parsing.

- (3) DCG allows extra conditions to be added to the rules, permitting auxiliary computations.

Consider a DCG example:

```
sentence(sentence(NP, VP)) --> noun_phrase(NP), verb_phrase(VP).
```

A sentence can take the form: a `noun_phrase` followed by a `verb_phrase`. The arguments represent the structure (the parse tree) of the sentence to be parsed. DCGs are essentially context-free grammars augmented by the language features of Prolog.

The logic grammar formalisms mentioned above are typically *first-order*, in the sense that a nonterminal symbol in these formalisms cannot be passed as an argument to some other nonterminal symbol. For example, usually DCG does not permit direct specification of grammar rules of the form

```
goal(X) --> X.
```

This problem was pointed out as early as [42]. It affects the convenience of use, extensibility, and modularity of grammars. We proposed a *lazy functional logic* approach to this problem in previous chapters.

The logic grammars mentioned above are also not *lazy*. Lazy evaluation can be very important in parsing. It allows coroutined recognition of multiple patterns against a stream. We are not aware of previous work connecting lazy evaluation and logic grammars, although the connection is a natural one.

Narrowing Grammar is a functional logic grammar formalism. It combines concepts

from logic programming, rewriting, lazy evaluation, and logic grammar formalisms such as DCGs. In particular, it simultaneously affords the expressive power of both functions and relations. The semantics of Narrowing Grammar are defined by a special outermost narrowing strategy, NU-step, which is different from existing logic grammar formalisms.

In this chapter, we point out a number of advantages of Narrowing Grammar for language analysis. We briefly summarize some interesting features of Narrowing Grammar. Then we go on to compare Narrowing Grammar with some existing logic grammars, showing how some of these logic grammars can be transformed to Narrowing Grammar. We also demonstrate the versatility of Narrowing Grammar for language analysis by illustrating its use in several natural language examples: coordination, left- and right-extrapolation.

7.2 Language Acceptance vs Generation

Many logic grammars can be straightforwardly translated to Prolog. For example, each DCG rule is translated to a Prolog clause by adding a difference-list to each non-terminal symbol giving the input and output streams.

```
sent --> np, vp.
```

```
sent(S, S0) :- np(S, S1), vp(S1, S0).
```

The grammar rule by itself acts as a language generator, but when compiled to Prolog, it acts like an acceptor.

Narrowing Grammar rules use a simple definition of `match` for language acceptance.

```
match([], S) => S.
```

```
match([X|S], [X|L]) => match(S, L).
```

```
sent => np, vp.
```

`match(sent, S)` takes an input nonterminal as the first argument, NU-narrows it to a simplified term and matches against the input stream (the second argument). For instance, with the starting pattern `sent`, Narrowing Grammar generates the sentence `[john, runs]`:

```
sent
  → np, vp
  → [john], vp
  → [john|vp]
  → [john, runs]
```

When applied with `match`, Narrowing Grammar accepts the sentence `[john, runs]`:

```
match(sent, [john, runs])
  → match((np, vp), [john, runs])
  → match(([john], vp), [john, runs])
  → match([john|vp], [john, runs])
  → match(vp, [runs])
  → match([runs], [runs])
  → match([], [])
  → []
```

Now we adapt DCGs compilation to Narrowing Grammar compilation and formally

prove that if `match` accepts a language, the compiled Narrowing Grammar accepts the same language.

Algorithm 7.1

For each Narrowing Grammar rule $f(A_1, \dots, A_m) \Rightarrow RHS$, perform the following:

- (1) If *RHS* is a variable or a simplified term, generate

$$f(A_1, \dots, A_m, S) \Rightarrow \text{match}(RHS, S).$$

- (2) If *RHS* is of the form $g(B_1, \dots, B_n)$ where *g* is a non-constructor function symbol, generate

$$f(A_1, \dots, A_m, S) \Rightarrow g(B_1, \dots, B_n, S).$$

where *S* is the input stream to be matched.

Unlike DCGs compilation, which requires arguments for the input and output streams, Narrowing Grammar compilation requires only one argument for the input stream.

Example 7.1 : Given the Narrowing Grammar rules:

`sent => np, vp.`

`([], L) => L.`

`([X|L1], L2) => [X|(L1, L2)].`

The compiled Narrowing Grammar rules are:

$\text{sent}(S) \Rightarrow ', '(np, vp, S) .$

$', '([], L, S) \Rightarrow \text{match}(L, S) .$

$', '([X|L1], L2, S) \Rightarrow \text{match}([X|(L1, L2)], S) .$

Theorem 7.1

Suppose F is a Narrowing Grammar term, and S is a stream. Let CF be the result of "compiling" F and S as follows:

$$CF = \begin{cases} f(T_1, \dots, T_m, S) & \text{if } F = f(T_1, \dots, T_m) \text{ where } f \text{ is not a constructor symbol} \\ \text{match}(F, S) & \text{if } F \text{ is a variable or a simplified term} \end{cases}$$

If $\text{match}(F, S) \xrightarrow{n} R$ where R is some suffix of S using a given set of Narrowing Grammar rules, then $CF \xrightarrow{n} R$ using the same set of Narrowing Grammar rules together with the result of compiling the Narrowing Grammar rules using Algorithm 7.1.

Proof:

By induction on n , the number of the narrowing steps of $\text{match}(F, S)$.

Basis: $n = 1$

In this case, F must be either a variable or a simplified term $[X|L]$ or $[],$ since these are only two rules for narrowing match . But then CF is just $\text{match}(F, S)$, so $CF \xrightarrow{n} R$ as claimed.

Induction step:

Let $n = k+1$, and assume the theorem is true for all $n \leq k$.

If F is simplified, then $CF = \text{match}(F, S)$. Therefore, it is trivially true that both

$\text{match}(F, S)$ and CF NU-narrow to the same term.

Let us consider the case where F is not simplified. Thus $CF = f(T_1, \dots, T_m, S)$ where $F = f(T_1, \dots, T_m)$. Here we assume $\text{match}(F, S) \rightarrow \text{match}(F', S) \xrightarrow{k} R$, so $F \rightarrow F'$.

There are several cases for F' :

- (1) F' is a variable or a simplified term.

Let $CF' = \text{match}(F', S)$ where CF' is the result of "compiling" F' and S .

Since $F \rightarrow F'$, then $L \Rightarrow H$ is in the given Narrowing Grammar, where $L = f(A_1, \dots, A_m)$, $L\theta = F\theta$ and $H\theta = F'\theta$. By (1) of Algorithm 7.1, the compiled Narrowing Grammar rule corresponding to $L \Rightarrow H$ is $f(A_1, \dots, A_m, S) \Rightarrow \text{match}(H, S)$ where $f(A_1, \dots, A_m, S)\theta = CF\theta$, and $\text{match}(H, S)\theta = CF'$.

Therefore, $CF \rightarrow CF'$, and inductively $CF' \xrightarrow{k} R$, so $CF \xrightarrow{n} R$.

- (2) $F \Rightarrow F'$ and let $F' = g(Y_1, \dots, Y_j)$ where g is an j -ary, $j \geq 0$, non-constructor function symbol.

Let $CF' = g(Y_1, \dots, Y_j, S)$ where CF' is the result of "compiling" F' and S .

Since $F \rightarrow F'$, then $L \Rightarrow H$ is in the given Narrowing Grammar where $L = f(A_1, \dots, A_m)$, $H = g(B_1, \dots, B_j)$, $L\theta = F\theta$ and $H\theta = F'\theta$. By (2) of Algorithm 7.1, the compiled grammar rule corresponding to $L \Rightarrow H$ is $f(A_1, \dots, A_m, S) \Rightarrow g(B_1, \dots, B_j, S)$ where $f(A_1, \dots, A_m, S)\theta = CF\theta$, and $g(B_1, \dots, B_j, S)\theta = CF'\theta$.

Therefore, $CF \rightarrow CF'$, and inductively $CF' \xrightarrow{k} R$, so $CF \xrightarrow{n} R$.

- (3) $F \neq \Rightarrow F'$ and let $F' = f(T_1, \dots, T_{i-1}, Y_i, T_{i+1}, \dots, T_m)\theta$ where $T_i \rightarrow Y_i$ with binding θ for some i . Let $CF' = f(T_1, \dots, T_{i-1}, Y_i, T_{i+1}, \dots, T_m, S)\theta$ where CF' is the

result of "compiling" F' and S .

Since $F \rightarrow F'$, and $T_i \rightarrow Y_i$, therefore, $CF \rightarrow CF'$, and inductively $CF' \xrightarrow{k} R$, so $CF \xrightarrow{n} R$.

Therefore, both $\text{match}(F, S)$ and CF NU-narrow to R in $k+1$ steps. By induction, if $\text{match}(F, S) \xrightarrow{n} R$ using a given set of Narrowing Grammar rules, then $CF \xrightarrow{n} R$ using the same set of Narrowing Grammar rules together with the compiled Narrowing Grammar.

Q.E.D.

We have adapted DCGs compilation to Narrowing Grammar compilation for the language acceptance by adding an extra argument for input stream.

7.3 The power of unification

Unification arises naturally in parsing. Narrowing Grammar captures the power of unification, permitting arguments of terms in rewrite rules to be used not only as inputs, but also as grammar outputs [49]. Like many logic grammars, Narrowing Grammar allows arbitrary structures to be built in the process of parsing.

As a simple example, we can even parse English-like sentences using these Narrowing Grammar rules. Consider the following Narrowing Grammar directly adapted from [14].


```

s(P) => np(X,P1,P), vp(X,P1).

np(X,P,P) => propernoun(X).
np(X,S,P) => article(X,R,S,P1), noun(X,L,R),
             complements(L,P1,P).

vp(X,P) => verb(X,L,P1), complements(L,P1,P).

complements([],P,P) => [].
complements([K-X|L],P1,P) => case(K), np(X,P1,P2),
                               complements(L,P2,P).

case(object) => [].
case(from) => [from].
case(of) => [of].

article(X,R,S,a(X,R,S)) => [a].
article(X,R,S,the(X,R,S)) => [the].
article(X,R,S,each(X,R,S)) => [each].

noun(X,[],disk(X)) => [disk].
noun(X,[],cpu(X)) => [cpu_123].
noun(Y,[of-X],failureof(X,Y)) => [failure].

propernoun(cpu) => [cpu].
propernoun(disk) => [disk].

verb(X,[object-Y],signaled(X,Y)) => [signaled].

([],L) => L.
([X|L1],L2) => [X|(L1,L2)].

```

With this Narrowing Grammar and the definition of `match`, the goal

```
?- simplify( match( s(P), [cpu,signaled,the,failure,of,disk] ), []).
```

yields the resulting parse tree

$P = \text{the}(X, \text{failureof}(\text{disk}, X), \text{signaled}(\text{cpu}, X))$.

7.4 Higher-order Specification, Modular Composition and Lazy Evaluation

Consider the familiar example which specifies the non-context-free language $\{a^n b^n c^n | n \geq 0\}$.

```
s_abc => ab_c // a_bc.  
ab_c => pair([a], [b]), [c]*.  
a_bc => [a]*, pair([b], [c]).  
pair(X, Y) => [].  
pair(X, Y) => X, pair(X, Y), Y.
```

Narrowing Grammar is higher-order. Specifically, Narrowing Grammar is higher-order in the sense that patterns can be passed as input arguments to patterns, and patterns can yield patterns as outputs. The enumeration pattern $a_bc // ab_c$ is higher-order, as its arguments are patterns. The patterns a_bc and ab_c can be used as arguments to $//$.

Narrowing Grammar supports modular composition of patterns. The enumeration pattern $a_bc // ab_c$ is composed of patterns a_bc and ab_c . Each of these patterns specifies a constraint ($a^n b^n c^*$ or $a^* b^n c^n$) on the stream to be generated.

Lazy evaluation is intimately related with a programming paradigm referred to as *stream processing* [46]. Note that in this paper, a stream pattern is a term that will yield a list of ground terms under lazy evaluation. We are not aware of previous work

connecting stream processing and grammars, although the connection is a natural one. Lazy evaluation and stream processing also have intimate connections with *coroutining* [27]. Coroutining is the interleaving of evaluation (here, narrowing) of two expressions. It is applied frequently in stream processing. For example, narrowing of the stream pattern

$$ab_c // a_bc$$

interleaves the narrowing of $(_ // _)$, ab_c and a_bc .

A specific advantage of lazy evaluation in parsing, then, is that coroutined recognition of multiple patterns in a stream becomes accessible to the grammar writer. Given two constraints $a^n b^n c^*$ and $a^* b^n c^n$ which are represented by two patterns ab_c and a_bc , the specification of these simultaneously constraints are $ab_c // a_bc$.

7.5 Transformation of Logic Grammars to Narrowing Grammar

In this section, we show how to transform four established logic grammars to Narrowing Grammar. These are Definite Clause Grammars (DCGs), Metamorphosis Grammars (MGs), Extraposition Grammars (XGs) and Gapping Grammars (GGs). The latter three are extensions of DCGs which provide some context-sensitive constructs. There is a straightforward transformation from DCGs to Narrowing Grammar. With only a few predefined Narrowing Grammar rules, Narrowing Grammar can simulate the context-sensitive constructs of MGs, XGs and GGs.

7.5.1 Narrowing Grammar and Definite Clause Grammars

We show how a pure Definite Clause Grammar (DCG) rule can be translated into a Narrowing Grammar rule so that both describe the same language.

- (1) Essentially, DCG rules can be translated to Narrowing Grammar rules by changing all occurrences of $-->$ to $=>$ and by including the Narrowing Grammar definition for $\backslash, /$.

$$([], L) => L.$$

$$([X|L1], L2) => [X|(L1, L2)].$$

- (2) $\backslash, /$ of DCGs can be defined at the grammatical level in Narrowing Grammar.

$$(X; Y) => X.$$

$$(X; Y) => Y.$$

Example 7.2 : Non-Context-Free Languages

Consider the following DCG rules for the language $\{a^n b^n c^n \mid n \geq 0\}$ and their translation to Narrowing Grammar rules:

$$s \text{ --> } a(\text{zero}).$$

$$a(A) \text{ --> } [a], a(\text{succ}(A)).$$

$$a(A) \text{ --> } b(A), c(A).$$

$$b(\text{succ}(A)) \text{ --> } [b], b(A).$$

$$b(\text{zero}) \text{ --> } [].$$

$$c(\text{succ}(A)) \text{ --> } [c], c(A).$$

$$c(\text{zero}) \text{ --> } [].$$

$$s \text{ => } a(\text{zero}).$$

$$a(A) \text{ => } [a], a(\text{succ}(A)).$$

$$a(A) \text{ => } b(A), c(A).$$

$$b(\text{succ}(A)) \text{ => } [b], b(A).$$

$$b(\text{zero}) \text{ => } [].$$

$$c(\text{succ}(A)) \text{ => } [c], c(A).$$

$$c(\text{zero}) \text{ => } [].$$

A NU-narrowing showing how the stream $[a, b, c]$ is generated:

s

- a(zero)
- [a], a(succ(zero))
- [a | a(succ(zero))]
- [a | (b(succ(zero)), c(succ(zero)))]
- [a | ([b], b(zero)), c(succ(zero))]
- [a | ([b | b(zero)], c(succ(zero)))]
- [a, b | (b(zero), c(succ(zero)))]
- [a, b | ([], c(succ(zero)))]
- [a, b | c(succ(zero))]
- [a, b | ([c], c(zero))]
- [a, b, c | c(zero)]
- [a, b, c]

A NU-narrowing showing how the stream $[a, b, c]$ is accepted:

```

match(s, [a,b,c])
  → match(a(zero), [a,b,c])
  → match([a], a(succ(zero))), [a,b,c])
  → match([a|a(succ(zero))], [a,b,c])
  → match(a(succ(zero)), [b,c])
  → match((b(succ(zero)), c(succ(zero))), [b,c])
  → match(((b], b(zero)), c(succ(zero))), [b,c])
  → match([b|b(zero)], c(succ(zero))), [b,c])
  → match([b|(b(zero), c(succ(zero))]), [b,c])
  → match((b(zero), c(succ(zero))), [c])
  → match([], c(succ(zero))), [c])
  → match(c(succ(zero)), [c])
  → match([c], c(zero)), [c])
  → match([c|c(zero)], [c])
  → match(c(zero), [])
  → match([], [])
  → []

```

7.5.2 Narrowing Grammar and Metamorphosis Grammars

MG [13] permits rules of the form

$$LHS, T \dashrightarrow RHS$$

where *LHS* is a nonterminal and *T* is one or more terminals. The MG rule can be read as "*LHS* can be expanded to *RHS* if *T* appears in the head of the input stream after *RHS* is parsed". We can capture the semantics of this MG rule in Narrowing Grammar by defining two more rules for `\`, `'` as follows (here `delete` is a constructor):

$(\text{delete}([X|Z]), [X|Y]) \Rightarrow \text{delete}(Z), Y.$
 $(\text{delete}([]), Y) \Rightarrow Y.$

and transform the MG rule to

$LHS \Rightarrow RHS, \text{delete}(T).$

Note, however, that with NG T can be *any* pattern, not just a stream of terminals.

Example 7.3 Consider the following MG and Narrowing Grammar rules which accept all strings of $[a]$'s and $[b]$'s which have an equal number of $[a]$'s and $[b]$'s.

$ns \rightarrow [].$	$ns \Rightarrow [].$
$ns \rightarrow na, ns, nb.$	$ns \Rightarrow na, ns, nb.$
$na \rightarrow [a].$	$na \Rightarrow [a].$
$na, [\text{term}(nb)] \rightarrow nb, na.$	$na \Rightarrow nb, na, \text{delete}(nb).$
$nb \rightarrow [b].$	$nb \Rightarrow [b].$
$nb \rightarrow [\text{term}(nb)].$	

The $[\text{term}(nb)]$ 'terminal' permits the MGs to treat the nonterminal nb temporarily as a terminal. However, the grammar will recognize/generate a superset of the original language. Note that this artifice is not needed with Narrowing Grammar.

A derivation showing one possible stream is generated from the start symbol ns :

ns

→ na, ns, nb
→ (nb, na, delete(nb)), ns, nb
→ ([b], na, delete(nb)), ns, nb
→ [b|(na, delete(nb))], ns, nb
→ [b|((na, delete(nb)), ns, nb)]
→ [b|([a], delete(nb)), ns, nb]
→ [b|([a|delete(nb)], ns, nb)]
→ [b,a|(delete(nb), ns, nb)]
→ [b,a|(delete([b]), ns, nb)]
→ [b,a|(delete([b]), [], nb)]
→ [b,a|(delete([b]), nb)]
→ [b,a|(delete([b]), [b])]
→ [b,a|(delete([]), [])]
→ [b,a]

7.5.3 Narrowing Grammar and Extraposition Grammars

One commonly used XG rule is of the form

$$LHS \dots T \dashrightarrow RHS.$$

Here *LHS* is a nonterminal symbol and *T* is any finite sequence of terminals or non-terminals. The XG rule can be read as "*LHS* can be expanded to *RHS* if *T* appears later in the input stream". We can capture the semantics of this XG rule in Narrowing Grammar by defining three more rules for `\`, `'` as follows (here `delete_any` is a constructor):


```

(delete_any([X|Z]), [X|Y]) => delete(Z), Y.
(delete_any([], Y) => Y.
(delete_any(X), [Y|Z]) => [Y|(delete_any(X), Z)].

```

and transform the XG rule to

$$LHS \Rightarrow RHS, \text{delete_any}(T).$$

As compared to MGs, one extra rule (the last rule) for `' , '` is needed due to ... in the *LHS* of an XG rule. It is because the pattern `τ` can appear *anywhere* later in the input stream.

Example 7.4 : Left-Extrapolation

Pereira pointed out that relative clauses can be handled with rules like the following:

```

s --> np, vp.
np --> det, n, optional_relative.
np --> [john].
np --> [trace].
vp --> v.
vp --> v, np.
v --> [write].
optional_relative --> [].
optional_relative --> relative.
relative --> rel_marker, s.
rel_marker ... [trace] --> rel_pro.
rel_pro --> [that].

```

The left-extrapolation XG rule

rel_marker ... [trace] --> rel_pro.

is transformed to

rel_marker => rel_pro, delete_any([trace]).

A NU-narrowing showing how the relative clause ([that, john, wrote]) can be generated:

relative

→ rel_marker, s
→ (rel_pro, delete_any([trace])), s
→ ([that], delete_any([trace])), s
→ [that|delete_any([trace])], s
→ [that|(delete_any([trace]), s)]
→ [that|(delete_any([trace]), (np, vp))]
→ [that|(delete_any([trace]), ([john], vp))]
→ [that|(delete_any([trace]), [john|vp])]
→ [that, john|(delete_any([trace]), vp)]
→ [that, john|(delete_any([trace]), (v, np))]
→ [that, john|(delete_any([trace]), ([wrote], np))]
→ [that, john|(delete_any([trace]), [wrote|np])]
→ [that, john, wrote|(delete_any([trace]), np)]
→ [that, john, wrote|(delete_any([trace]), [trace])]
→ [that, john, wrote|(delete([]), [])]
→ [that, john, wrote]

The point to be made here is that commonly used XG rules can be transformed to Narrowing Grammar rules easily.

7.5.4 Narrowing Grammar and Gapping Grammars

In the previous section, we described how Narrowing Grammar simulates left-extrapolation of XGs, here we show how Narrowing Grammar simulates right-extrapolation of GGs. Consider a special class of GG rules [16] of the form

$$LHS, gap(X), T \rightarrow gap(X), RHS.$$

where *LHS* is a nonterminal symbol and *T* is any finite sequence of terminals or non-terminals. This rule implements right-extrapolation in linguistic theory.

We can capture the semantics of this GG rule in Narrowing Grammar by defining rules for `\`, `'` with constructors `replace(_,_)` and `replace_any(_,_)` as follows:

```
(replace([],R), Y) => R, Y.  
(replace([X|L],R), [X|Y]) => replace(L,R), Y.  
(replace_any([],R), Y) => R, Y.  
(replace_any([X|L],R), [X|Y]) => replace(L,R), Y.  
(replace_any(T,R), [X|Y]) => [X|(replace_any(T,R), Y)].
```

and transform the GG rule to

$$LHS \rightarrow \text{replace_any}(T, RHS).$$

Example 7.5 : Left- and Right-Extrapolation

In this example, we show how Narrowing Grammar simulates the left- and right-extrapolation of GGs. The following grammar which is adapted from [17] parses sen-

tences such as "The man is here that Jill saw".

```
s --> np, vp.
np --> det, n, relative.
np --> n.
np --> [term(np)].
vp --> aux, comp.
vp --> v, np.
relative --> rel_marker, s.
relative --> [].
relative, gap(G) --> gap(G), rightex.
rel_marker, gap(G), [term(np)] --> rel_pro, gap(G).
rightex --> rel_marker, s.
comp --> [here].
aux --> [is].
det --> [the].
rel_pro --> [that].
n --> [man].
n --> [jill].
v --> [saw].
```

The left-extrapolation GG rule†

```
rel_marker, gap(G), [term(np)] --> rel_pro, gap(G).
```

is transformed to

† This rule is equivalent to the XG rule

```
rel_marker ... [term(np)] --> rel_pro.
```

`rel_marker => rel_pro, delete_any([term(np)]).`

and the right-extraposition GG rule

`relative, gap(G) --> gap(G), rightex`

is transformed to

`relative => replace_any([], rightex).`

All other rules can be transformed to Narrowing Grammar rules by changing all occurrences of `-->` to `=>`. Now we illustrate how the sentence

`The man is here that Jill saw`

can be generated from the Narrowing Grammar by giving a sequence of terms that can be produced by NU-narrowing of the Narrowing Grammar start symbol `s`:

s

- np, vp
- (det, n, relative), vp
- ([the], n, relative), vp
- [the|(n, relative)], vp
- [the|((n, relative), vp)]
- [the|([[man], relative), vp]]
- [the|([man|relative], vp)]
- [the,man|(relative, vp)]
- [the,man|(replace_any([], rightex), vp)]
- [the,man|(replace_any([], rightex), (aux, comp))]
- [the,man|(replace_any([], rightex), ([is], comp))]
- [the,man|(replace_any([], rightex), [is|comp])]
- [the,man,is|(replace_any([], rightex), comp)]
- [the,man,is|(replace_any([], rightex), [here])]
- [the,man,is,here|(replace_any([], rightex), [])]
- [the,man,is,here|rightex]

The non-terminal `rightex` generates the stream `[that,jill,saw]` as illustrated in Example 7.4. The point to be made here is that commonly used GG rules can be transformed to Narrowing Grammar rules easily.

7.6 Narrowing Grammar can be More Expressive

In previous sections, we described how Narrowing Grammar simulates other logic grammars. Here we demonstrate the versatility of Narrowing Grammar in specification by illustrating its use in several examples. The key to many of the

demonstrations is the definition of a simple but powerful primitive, `//`, that coroutines grammatical expressions in the generation of a single stream. This cannot be done easily with some other logic grammar formalisms such as DCGs.

Example 7.6 : Non-Context Free Languages

Consider the following Narrowing Grammar which defines the non-context-free language $\{a^n b^m c^n d^m \mid m, n \geq 0\}$:

```
abcd => a_c // b_d.
a_c => triple([a],[b],[c]), [d]*.
b_d => [a]*, triple([b],[c],[d]).
triple(X,Y,Z) => Y*.
triple(X,Y,Z) => X, triple(X,Y,Z), Z.
```

It is obvious that Narrowing Grammar rules are modular. The first rule imposes simultaneous constraints $a^n b^* c^n d^*$ and $a^* b^m c^* d^m$.

Example 7.7 : Lazy Narrowing and Coroutined Pattern Matching

In this example, we consider a basic problem for testing a certain set of events that satisfies certain precedence (partial ordering) constraints. A directed acyclic graph (DAG) is well suited for representing these constraints. In a DAG whose nodes represent events, an edge $X \rightarrow Y$ represents the ordering constraint that event X occurs before event Y , which can also be specified by `precedes(x,y)`:

`precedes(X, Y) => eventually(X), eventually(Y).`

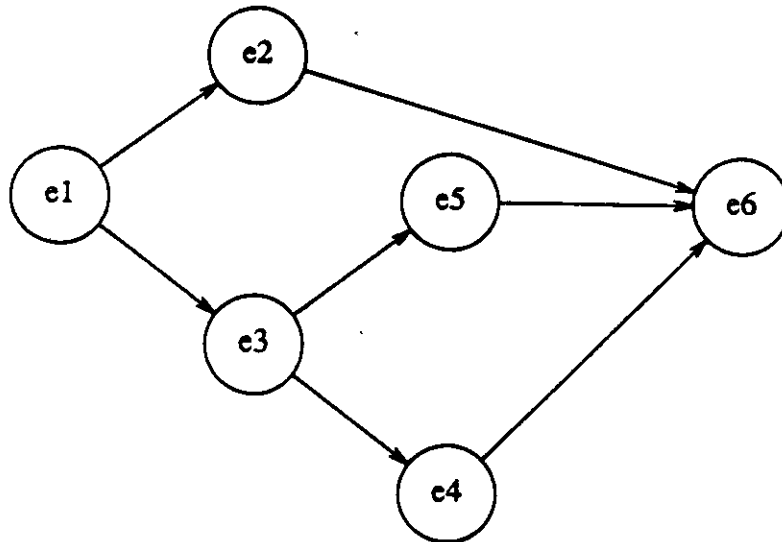
`eventually(X) => X.`

`eventually(X) => [_], eventually(X).`

Now, we can directly represent an ordering of any two events by a pattern as follows:

`... // precedes(X, Y) // ...`

Each `precedes(x, y)` corresponds to an edge $X \rightarrow Y$ in the DAG. For instance, suppose we are given a partial ordering of events which is represented by the following DAG:



The pattern that describes the partial ordering is

`precedes(e1, e2) // precedes(e1, e3) // precedes(e3, e5) //`
`precedes(e3, e4) // precedes(e2, e6) // precedes(e5, e6) //`
`precedes(e4, e6).`

Here `e1, ..., e6` are some event patterns.

7.7 Case Study : Natural Language Analysis

Coordination (grammatical construction with the conjunctions 'and', 'or', 'but') has long been one of the most difficult natural language phenomena to handle. The reduced coordinate constructions are of the form

$A X \text{ and } Y B,$

for example,

John drove the car through and completely demolished a window.
A X Y B

where the unreduced deep structure corresponds to

$A X B \text{ and } A Y B.$

The SYSCONJ facility for Augmented transition networks (ATNs) [5, 61] was one of the most general and powerful metagrammatical device for handling coordination in computational linguistics. Two logic grammar formalisms, Modifier Structure Grammars (MSGs) [15] and Gapping Grammars [16, 17], were proposed later to handle coordination in natural language.

In previous sections, we define some context-sensitive constructs in Narrowing Grammar to simulate MGs, XGs and GGs. In this case study, we apply the same predefined context-sensitive rules to specify natural language phenomena such as coordination, left- and right-extrapolation. Let's first summarize all the context-sensitive Narrowing Grammar rules here:

$(\text{delete}([X|Z]), [X|Y]) \Rightarrow \text{delete}(Z), Y.$
 $(\text{delete}([]), Y) \Rightarrow Y.$
 $(\text{delete_any}([X|Z]), [X|Y]) \Rightarrow \text{delete}(Z), Y.$
 $(\text{delete_any}([]), Y) \Rightarrow Y.$
 $(\text{delete_any}(X), [Y|Z]) \Rightarrow [Y|(\text{delete_any}(X), Z)].$
 $(\text{replace}([], R), Y) \Rightarrow R, Y.$
 $(\text{replace}([X|L], R), [X|Y]) \Rightarrow \text{replace}(L, R), Y.$
 $(\text{replace_any}([], R), Y) \Rightarrow R, Y.$
 $(\text{replace_any}([X|L], R), [X|Y]) \Rightarrow \text{replace}(L, R), Y.$
 $(\text{replace_any}(T, R), [X|Y]) \Rightarrow [X|(\text{replace_any}(T, R), Y)].$

Consider the following Narrowing Grammar rules for natural language analysis with coordination, left- and right-extrapolation:

```

sent => s.
sent => s, [and], sent.

s => np, vp.
s => [term(s)].

np => n.
np => det, n, comp, relative.
np => [and], vp, delete(([and],[term(s)])) .
np => [and], s, delete(([and],[term(s)])) .
np => [term(np)].

relative => [].
relative => rel_marker, sent.
relative => replace_any([], rightex).

rel_marker => rel_pro, delete_any([term(np)]).

rightex => rel_marker, sent.

vp => advl, v, comp.

advl => [].
advl => adv.

comp => [].
comp => np.
comp => prep, np.

rel_pro => [that].

```

A NU-narrowing showing how the sentence with coordination "John drove the car through and completely demolished a window" is generated:

sent

- s, [and], sent.
- (np, vp), [and], sent.
- * → [john|(vp, [and], sent)].
- [john|((advl, v, comp), [and], sent)].
- [john|([[], v, comp), [and], sent)].
- [john|((v, comp), [and], sent)].
- [john|([[drove], comp), [and], sent)].
- * → [john,drove|(comp, [and], sent)].
- [john,drove|(np, [and], sent)].
- [john,drove|((det, n, comp, relative), [and], sent)].
- * → [john,drove,the|((n, comp, relative), [and], sent)].
- * → [john,drove,the,car|((comp, relative), [and], sent)].
- [john,drove,the,car|(((prep, np), relative), [and], sent)].
- * → [john,drove,the,car,through|((np, relative), [and], sent)].
- [john,drove,the,car,through|((([and], vp, delete([and], [term(s)]))), relative), [and], sent)].
- * → [john,drove,the,car,through,and|(((vp, delete([and], [term(s)]))), relative), [and], sent)].
- [john,drove,the,car,through,and|(((advl, v, comp), delete([and], [term(s)]))), relative), [and], sent)].
- * → [john,drove,the,car,through,and,completely|(((v, comp), delete([and], [term(s)]))), relative), [and], sent)].
- * → [john,drove,the,car,through,and,completely,demolished|(((comp, delete([and], [term(s)]))), relative), [and], sent)].
- [john,drove,the,car,through,and,completely,demolished|(((np, delete([and], [term(s)]))), relative), [and], sent)].
- * → [john,drove,the,car,through,and,completely,demolished,a>window|((delete([and], [term(s)]))), relative), [and], sent)].
- [john,drove,the,car,through,and,completely,demolished,a>window|((delete([and],[term(s)]))), relative), [and], sent)].
- * → [john,drove,the,car,through,and,completely,demolished,a>window|(delete([and],[term(s)])), [and], sent)].

- [john, drove, the, car, through, and, completely, demolished, a, window |
(delete([and|{term(s)}]), [and|sent])].
- [john, drove, the, car, through, and, completely, demolished, a, window |
(delete({term(s)}), sent)].
- [john, drove, the, car, through, and, completely, demolished, a, window |
(delete({term(s)}), s)].
- [john, drove, the, car, through, and, completely, demolished, a, window |
(delete({term(s)}), [term(s)])].
- [john, drove, the, car, through, and, completely, demolished, a, window |
(delete([], []))].
- [john, drove, the, car, through, and, completely, demolished, a, window]

Similarly the sentence with coordination, left- and right-extrapolation "The girl saw the man that Mary saw and Bill heard" can also be generated:

sent

- s
- np, vp
- (dat, n, comp, relative), vp
- * → [the, girl | (relative, vp)]
- [the, girl | (replace_any([], rightex), vp)]
- [the, girl | (replace_any([], rightex), (advl, v, comp))]
- * → [the, girl | (replace_any([], rightex), [saw | comp])]
- [the, girl, saw | (replace_any([], rightex), comp)]
- [the, girl, saw | (replace_any([], rightex), np)]
- [the, girl, saw | (replace_any([], rightex), (dat, n, comp, relative))]
- [the, girl, saw | (replace_any([], rightex), [the | (n, comp, relative)])]
- [the, girl, saw, the | (replace_any([], rightex), (n, comp, relative))]
- * → [the, girl, saw, the | (replace_any([], rightex), [man | (comp, relative)])]
- [the, girl, saw, the, man | (replace_any([], rightex), (comp, relative))]
- [the, girl, saw, the, man | (rightex, comp, relative)]
- [the, girl, saw, the, man | ((rel_marker, sent), comp, relative)]
- * → [the, girl, saw, the, man, that | (sent, comp, relative)]
- [the, girl, saw, the, man, that | ((s, [and], sent), comp, relative)]
- [the, girl, saw, the, man, that | (((np, vp), [and], sent), comp, relative)]
- * → [the, girl, saw, the, man, that, mary | ((vp, [and], sent), comp, relative)]
- [the, girl, saw, the, man, that, mary |
((advl, v, comp), [and], sent), comp, relative]
- * → [the, girl, saw, the, man, that, mary, saw |
((comp, [and], sent), comp, relative)]
- [the, girl, saw, the, man, that, mary, saw |
((np, [and], sent), comp, relative)]
- [the, girl, saw, the, man, that, mary, saw |
((([and], s, delete([and], [term(s)])), [and], sent), comp, relative)]
- * → [the, girl, saw, the, man, that, mary, saw, and |
(((s, delete([and], [term(s)])), [and], sent), comp, relative)]
- [the, girl, saw, the, man, that, mary, saw, and |

```

      (((np, vp), delete([[and], [term(s)]]), [and], sent), comp, relative)]
* → [the, girl, saw, the, man, that, mary, saw, and, bill|
      ((vp, delete([[and], [term(s)]]), [and], sent), comp, relative)]
* → [the, girl, saw, the, man, that, mary, saw, and, bill, heard|
      (((delete([[and], [term(s)]]), [and], sent), comp, relative)]
* → [the, girl, saw, the, man, that, mary, saw, and, bill, heard|
      (((delete([and, term(s)]), [and|sent]), comp, relative)]
* → [the, girl, saw, the, man, that, mary, saw, and, bill, heard|
      ((delete([term(s)]), sent), comp, relative)]
→ [the, girl, saw, the, man, that, mary, saw, and, bill, heard|
      ((delete([term(s)]), s), comp, relative)]
→ [the, girl, saw, the, man, that, mary, saw, and, bill, heard|
      ((delete([term(s)]), [term(s)]), comp, relative)]
→ [the, girl, saw, the, man, that, mary, saw, and, bill, heard|
      ((delete([], [])), comp, relative)]
* → [the, girl, saw, the, man, that, mary, saw, and, bill, heard]

```

7.8 Summary

Narrowing Grammar compares favorably in expressive power and generality with other logic grammar formalisms. In this chapter we have demonstrated how to transform several established logic grammars to Narrowing Grammar. Like many logic grammars, Narrowing Grammar can be straightforwardly translated to Prolog, so the full power of unification is exploited. Unlike many logic grammars, Narrowing Grammar combines concepts from logic programming, rewriting, and lazy evaluation.

Chapter 8

Conclusions

In this dissertation, we have shown how Narrowing Grammar comprises a new formalism for language analysis. Narrowing Grammar combines concepts from logic programming, rewriting, lazy evaluation, and logic grammar formalisms, such as Definite Clause Grammars. The semantics of Narrowing Grammar are defined by a kind of special outermost lazy narrowing called NU-narrowing. With some modest restrictions on the use of duplicate variables on left-hand side of Narrowing Grammar rules, NU-narrowing is shown to be complete in the sense that when a term can be simplified, it can be simplified by a sequence of NU-steps.

The rules of Narrowing Grammar by themselves act as pattern generators, but when applied with `match` they act like an acceptor, or parser. All Narrowing Grammar rules can be compiled to logic programs in such a way that, when SLD-resolution interprets them, it directly simulates NU-narrowing.

Narrowing Grammar is modular, extensible and highly reusable, so saving rules in a library makes sense. These grammars extend the expressive power of first-order logic grammars, by permitting patterns to be passed as arguments to the grammar rules. As

a consequence, some complex patterns can be specified more easily. Narrowing Grammar also provides lazy evaluation. Lazy evaluation is important in certain language acceptance situations, such as in coroutined matching of multiple patterns against a stream.

Narrowing Grammar compares favorably in expressive power and generality with other logic grammar formalisms such as Definite Clause Grammars, Metamorphosis Grammars, Extraposition Grammars and Gapping Grammars. We have demonstrated how to translate different logic grammars into Narrowing Grammar. We have also pointed out some limitations of first-order logic grammars.

Although narrowing is in general difficult to implement efficiently, Algorithm 4.1 is a relatively efficient way to implement Narrowing Grammar. Further improvements for the implementation are possible in many cases. Chapter 6 presented several approaches, including compiling Narrowing Grammar rules with partial evaluation so that the right-hand side of any rule is simplified. Another approach is to optimize the Prolog code generated from Algorithm 4.1 by eliminating some redundant `nu_narrow/2` predicates.

Very good results can come from convincing people to stand on others' shoulders. In this dissertation, we proposed a new view of logic grammars. It links many good results from different areas such as logic programming, rewriting, lazy evaluation, logic grammars and parsing.

References

1. Abelson, H. and G. Sussman, *The Structure and Analysis of Computer Programs*, MIT Press, Boston, MA (1985).
2. Abramson, H., "Definite Clause Translation Grammars," *Proc. First Logic Programming Symposium*, pp. 233-240 IEEE Computer Society, (1984).
3. Abramson, H., "Metarules and an Approach to Conjunction in Definite Clause Translation Grammars: Some Aspects of Grammatical Metaprogramming," *Proc. Fifth International Conference and Symposium on Logic Programming*, pp. 233-248 MIT Press, (1988).
4. Apt, K.R. and M.H. vanEmden, "Contributions to the theory of logic programming," *JACM* 29(3)(July 1982).
5. Bates, M., "The Theory and Practice of Augmented Transition Network Grammars," *In Bolc, L., Ed., Natural Language Communication with Computers*, pp. 191-259 Springer-Verlag, (1978).
6. Bellia, M. and G. Levi, "The Relation between Logic and Functional Languages: A Survey," *J. Logic Programming*, pp. 217-236 (1986).
7. Bosco, P.G., E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi, "A Complete Semantic Characterization of K-LEAF, a Logic Language with Partial Functions," *Proc. 1987 Symp. on Logic Programming*, pp. 318-327 IEEE Computer Society, (1987).

8. Chau, H.L. and D.S. Parker, "Narrowing Grammar," *Proc. Sixth International Conference on Logic Programming*, pp. 199-217 MIT Press, (June 1989).
9. Church, A., "The Calculi of lambda-conversion," *Annals of Mathematics Studies Number 6*, Princeton University Press, Princeton (1941).
10. Clark, K., "Predicate Logic as a Computational Formalism," *Research Monograph*, Imperial College, University of London (1980).
11. Clark, K. and S. Gregory, "PARLOG: Parallel Programming in Logic," *ACM Transactions on Programming Languages and Systems* 8(1)(1986).
12. Colmerauer, A., H. Kanoui, P. Roussel, and R. Pasero, "Un Systeme de Communication Homme-Machine en Francais,," *Groupe de Recherche en Intelligence Artificielle*, Universite d'Aix-Marseille (1973).
13. Colmerauer, A., "Metamorphosis Grammars," in *Natural Language Communication with Computers, LNCS 63*, Springer (1978).
14. Colmerauer, A., "An Interesting Subset of Natural Language," in *Logic Programming*, ed. K. Clark, S.-A. Tarnlund, Academic Press, New York (1982).
15. Dahl, V. and M.C. McCord, "Treating Coordination in Logic Grammars," *American Journal of Computational Linguistics* 9(2) pp. 69-91 (April-June 1983).
16. Dahl, V. and H. Abramson, "On Gapping Grammars," *Proc. Second Intl. Logic Programming Conf.*, pp. 77-88 (1984).

17. Dahl, V., "More on Gapping Grammars," *Proc. Intl. Conf. on Fifth Generation Computer Systems*, (1984).
18. Darlington, J., A.J. Field, and H. Pull, "The Unification of Functional and Logic Languages," *Logic Programming: Functions, Relations, and Equations*, Prentice-Hall, (1986).
19. Dershowitz, N. and D.A. Plaisted, "Logic Programming cum Applicative Programming," *Proc. of the IEEE International Symposium on Logic Programming*, pp. 54-66 (July 1985).
20. Dincbas, M. and P.V. Hentenryck, "Extended Unification Algorithms for the Integration of Functional Programming into Logic Programming," *Journal of Logic Programming* 4(1987).
21. Fay, M., "First Order Unification in an Equational Theory," *Proc. of the 4th Conference on Automated Deduction*, (1979).
22. Fribourg, L., "Oriented Equational Clauses as a Programming Language," *Journal of Logic Programming* 1(2) pp. 165-177 (August 1984).
23. Fribourg, L., "SLOG: A Logic Programming Language Interpreter based on Clausal Superposition and Rewriting," *Proc. of the IEEE International Symposium on Logic Programming*, pp. 172-184 (July 1985).
24. Gallier, J.H. and S. Raatz, "Extending SLD Resolution to Equational Horn Clauses Using E-Unification," *Journal of Logic Programming* 3 pp. 3-43 (1989).

25. Goguen, J.A. and J. Meseguer, "Equality, Types, Modules and (Why Not?) Generics for Logic Programming," *J. Logic Programming* 1 pp. 179-210 (1984).
26. Gorlick, M.D., C. Kesselman, D. Marotta, and D.S. Parker, "Mockingbird: A Logical Methodology for Testing," Technical Report, The Aerospace Corporation, P.O. Box 92957, Los Angeles, CA 90009-2957 (May 1987). To appear, *Journal of Logic Programming*, 1989.
27. Henderson, P., *Functional Programming: Application and Implementation*, Prentice/Hall International (1980).
28. Hill, R., "LUSH Resolution and its completeness," DCL Memo 78, Dept. of Computer Science, University of Edinburgh (1974).
29. Hirschman, L. and K. Puder, "Restriction Grammars in Prolog," *Proc. First Intl. Conf. on Logic Programming*, pp. 85-90 (1982).
30. Hirschman, L. and K. Puder, "Restriction Grammar: A Prolog Implementation," *Logic Programming and its Applications*, (1985).
31. Hirschman, L., "Tutorial: Natural Language and Logic Programming," *IEEE Symposium on Logic Programming*, (1987).
32. Hopcroft, J.E. and J. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, Menlo Park, CA (1979).
33. Huet, G., "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems," *Journal of the ACM* 27 pp. 797-821 (1980).

34. Hullot, J.M., "Canonical Forms and Unification," *Proc. of the 5th Conference on Automated Deduction*, (1980).
35. Kahn, K., "Partial Evaluation, Programming Methodology, and Artificial Intelligence," *The AI Magazine* 5(1)(Spring 1984).
36. Kahn, K. and M. Carlsson, "The Compilation of Prolog Programs without the Use of a Prolog Compiler," *Proc. of the Intl. Conf. on Fifth Generation Computer System*, pp. 348-355 ICOT, (1984).
37. Knuth, D. and P. Bendix, "Simple Word Problems in Universal Algebras," *Computational Problems in Abstract Algebra*, pp. 263-297 Pergamon Press, (1970).
38. Kowalski, R.A., "Predicate Logic as a Programming Language," *Information Processing 74*, pp. 569-574 North Holland, (1974).
39. Lankford, D.S., "Canonical Inference," ATP-32, Dept. of Mathematics and Computer Science, Univ. of Texas (December 1975).
40. Lloyd, J.W., *Foundations of Logic Programming*, Springer-Verlag (1988).
41. McCord, M.C., "Modular Logic Grammars," *Proc. 23rd Annual Meeting of the Association for Computational Linguistics*, pp. 104-117 (1985).
42. Moss, C.D.S., "The Formal Description of Programming Languages using Predicate Logic," Ph.D. Dissertation, Imperial College, London (1981).
43. Narain, S., "A Technique for Doing Lazy Evaluation in Logic," *Proc. Symp. on Logic Programming*, pp. 261-269 IEEE Computer Society, (1985).

44. Narain, S., "A Technique for Doing Lazy Evaluation in Logic," *J. Logic Programming* 3(3) pp. 259-276 (October 1986).
45. Narain, S., "LOG(F): An Optimal Combination of Logic Programming, Rewrite Rules and Lazy Evaluation," Ph.D. Dissertation, UCLA Computer Science Dept., Los Angeles, CA 90024-1596 (1988).
46. Parker, D.S., "Stream Data Analysis in Prolog," Technical Report CSD-890004, UCLA Computer Science Dept., Los Angeles, CA 90024-1596 (January 1989).
47. Pereira, F.C.N. and D.H.D. Warren, "Definite Clause Grammars for Language Analysis," *Artificial Intelligence* 13 pp. 231-278 (1980).
48. Pereira, F.C.N., "Extrapolation Grammars," *American Journal for Computational Linguistics* 7(1981).
49. Pereira, F.C.N. and S.M. Shieber, *Prolog and Natural-Language Analysis*, CSLI Stanford (1987).
50. Plotkin, G., "Building-in Equational Theories," *Machine Intelligence* 7 pp. 73-90 (1972).
51. Reddy, U.S., "Narrowing as the Operational Semantics of Functional Languages," *Proc. of the Symposium on Logic Programming*, (1985).
52. Robinson, G. and L. Wos, "Paramodulation and theorem proving in first order theories with equality," *Machine Intelligence* 4, (1969).
53. Robinson, J.A. and E.E. Sibert, "LOGLISP an Alternative to Prolog," *Machine Intelligence*, pp. 399-419 (1982).

54. Shapiro, E.Y., "A Subset of Concurrent Prolog and its Interpreter," Technical Report TR-003, ICOT, Tokyo (February 1983).
55. Stabler, E.P., "Restricting Logic Grammars with Government-Binding Theory," *Computational Linguistics* 13(1-2)(1987).
56. Sterling, L. and E. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, MA (1986).
57. Subrahmanyam, P.A. and J-H. You, "FUNLOG=Functions+Logic: A Computational Model Integrating Functional and Logic Programming," *Proc. International Symp. on Logic Programming*, pp. 144-153 IEEE Computer Society, (1984).
58. vanEmden, M.H. and K. Yukawa, "Logic Programming with Equations," *Journal of Logic Programming* 4(4)(1987).
59. Venken, R., "A Prolog Meta-Interpreter for Partial Evaluation and its Application to Source to Source Transformation and Query-Optimization," *Proc. ECAI 84*, pp. 91-104 Elsevier Sci. Publ., North-Holland, (1984).
60. Warren, D.H.D., L.M. Pereira, and F.C.N. Pereira, "Prolog - the language and its implementation compared with Lisp," *Proc. Symp. on AI and Programming Languages*, (1977).
61. Woods, W.A., "An Experimental Parsing System for Transition Network Grammars," *In Rustin, R., Ed., Natural Language Processing*, pp. 145-149 Algorithmics Press, (1973).

62. Yamamoto, A., "A Theoretical Combination of SLD-Resolution and Narrowing," *Proc. of the Fourth International Conference on Logic Programming*, (1987).

